March 9, 2024

Analiza FERNANDES

candidate number: 9514

Center Number: 51411

Tetris Education Application

ANALIZA FERNANDES

Contents

[Analysis 3](#_Toc161082582)

[Overview 3](#_Toc161082583)

[Problem identification 3](#_Toc161082584)

[Background 3](#_Toc161082585)

[User Interview 4](#_Toc161082586)

[Audience Identification 5](#_Toc161082587)

[Description of Current Systems 5](#_Toc161082588)

[Tetris AI Player 5](#_Toc161082589)

[Tetris 99 6](#_Toc161082590)

[User Needs and Limitations 7](#_Toc161082591)

[Modelling the Problem 7](#_Toc161082592)

[Implementation of the system at a high level 7](#_Toc161082593)

[Algorithms and Data Structures 8](#_Toc161082594)

[Ideas for programming software 8](#_Toc161082595)

[Objectives 9](#_Toc161082596)

[UI and User Navigation 9](#_Toc161082597)

[Single Player Mode 10](#_Toc161082598)

[Educational mode 11](#_Toc161082599)

[Local Multiplayer 11](#_Toc161082600)

[Documented Design 11](#_Toc161082601)

[Overview 11](#_Toc161082602)

[Program structure 12](#_Toc161082603)

[Prototyping 13](#_Toc161082604)

[System Flowcharts 26](#_Toc161082605)

[Start Screen System 26](#_Toc161082606)

[Basic Gameplay Loops 27](#_Toc161082607)

[Class Diagrams 29](#_Toc161082608)

[Data Flow Diagram 30](#_Toc161082609)

[Database Design 31](#_Toc161082610)

[Database queries 31](#_Toc161082611)

[Key Algorithms 33](#_Toc161082612)

[MD5 Algorithm 33](#_Toc161082613)

[Rotation Matrix 36](#_Toc161082614)

[Human Computer Interface 38](#_Toc161082615)

[Technical Solutions 41](#_Toc161082616)

[Overview 41](#_Toc161082617)

[Techniques used 41](#_Toc161082618)

[Scripts 42](#_Toc161082619)

[Base Game 42](#_Toc161082620)

[Local Multiplayer 63](#_Toc161082621)

[Program UI 65](#_Toc161082622)

[Registration and Login 76](#_Toc161082623)

[Testing Overview 94](#_Toc161082624)

[Overview 94](#_Toc161082625)

[Test plan and outcome 94](#_Toc161082626)

[Evaluation 110](#_Toc161082627)

[Overview 110](#_Toc161082628)

[Meeting Objectives 110](#_Toc161082629)

[UI and User Navigation 110](#_Toc161082630)

[Single Player Mode 113](#_Toc161082631)

[Educational mode 115](#_Toc161082632)

[Local Multiplayer 115](#_Toc161082633)

[Modifications 117](#_Toc161082634)

[Challenges and Difficulties 117](#_Toc161082635)

[Feedback 118](#_Toc161082636)

[Future Development 118](#_Toc161082637)

[Bibliography 120](#_Toc161082638)

# Analysis

## Overview

In this section, I will be giving a general overview of the problem I am attempting to solve and the research I have done to identify a solution.

## Problem identification

Tetris, a game with a history spanning over three decades, has seen the collaborative nature of the internet and video game communities giving rise to a wealth of resources. These resources range from tutorials that offer insights into the most efficient piece-stacking strategies to dedicated websites focused on teaching players the art of 'openers' – techniques for arranging the first few tetrominoes for optimal gameplay.

However, the abundance of information can often be confusing to new players who may not yet understand the correct terms or techniques. So inexperienced players oftentimes must struggle through several difficult games of Tetris before achieving success. This led me to produce the idea of creating a program specifically tailored to streamline this learning process.

## Background

In June 1985, Alexey Pazhitnov created Tetris on an Electronica 60 while working at the Dorodnitsyn Computing Centre of the Soviet Academy of Sciences. Inspired by the wooden puzzle game “pentominoes” which involves wooden blocks, consisting of 5 cubes, being put together to make shapes. Pazhitnov took the principles of “pentominoes” and created “Tetris.

The gameplay of Tetris involves, randomly selected tetrominoes[[1]](#footnote-2) (shapes consisting of four squares) fall from the top of a 10 x 20 playfield, one at a time. Each tetromino enters the playfield with a given orientation and colour, dependent on its shape, and moves downward through the playfield at a predetermined speed. Whilst the tetromino is falling the player can shift the piece right or left and rotate it in intervals of 90 degrees. Once the tetromino lands on the floor of the playfield or on other blocks the player will no longer be able to control the piece. When a tetromino locks and by doing so fills all empty spaces within one or more rows of the playfield, those full rows will clear (Wiki, 2020b)[[2]](#footnote-3). If the playfield has not filled up with blocks, the next piece enters.

## User Interview

During the research stage of my Tetris program, I was fortunate enough to be put in contact with a competitive-level Tetris player, Alexis Brightman, who is currently struggling to find the right teaching materials to run a club for beginners in Tetris.

***Q: How helpful would a new program be in either teaching a person how to play Tetris or for general entertainment purposes?***

A: “As far as I know there is no direct Learning Game for Tetris, at least not for guideline Tetris, which is the problem that I was facing with my club. For the most part, you have to pull your knowledge from websites with images like four.lol (*FOUR*, n.d.)[[3]](#footnote-4) or hardrop.com (*Hard Drop Tetris Wiki*, n.d.)[[4]](#footnote-5). As you can imagine trying to collect resources for teaching can be quite the hassle. It could also add value in terms of entertainment. By incorporating various game modes, challenges, and perhaps even a multiplayer component, it could provide hours of enjoyment for both newcomers and seasoned players. So, I do think this kind or program would be helpful”

***Q: What would the interface/graphics ideally look like?***

A: “Ideally the graphics would follow the traditional look of Tetris, following the guidelines available on the official Tetris wiki (*Tetris Guideline - TetrisWiki*, n.d.)[[5]](#footnote-6), since these are the guidelines used across almost all modern Tetris Games. Using these would mean that the skills that my club members would be able to apply the skills learned in your program in any officially produced Tetris game.”

***Q: What features should the program contain to ensure ease of use for the end user?***

A: “As I said in the previous question following the guidelines from the Tetris Wiki would probably be the best practice. But aside from that using a good colour scheme and having the features from tradition Tetris Games, such as the hard drops would be sufficient.”

***Q: What acceptable limitations would you expect there to be in the project?***

A: “If you're planning to make a public or widely available game, you should follow most of the guidelines. The fan made games can usually divert a bit by excluding things like Line-Clear Delay or a custom Combo System, like in Tetr.io (Tetr.Io, n.d.)[[6]](#footnote-7).”

***Q: Do you think a program like this could be used as a learning tool?***

A: “If well made it could prove as a very good learning tool, as nothing of the sort exists as yet. Most of the knowledge on things like Stacking, Openers, T-Spins and other are mostly found on different websites, and so if you managed to find a way to compile all that information and present it in a particularly engaging manner, I think that it would be incredibly useful.”

## Audience Identification

In theory, this type of program should be usable by most audiences, however due to the nature of my client’s request, I will be focusing on an audience of 10 – 13-year-olds, as this is the audience that my client would like to use the program with.

## Description of Current Systems

### Tetris AI Player

#### A screenshot of a game Description automatically generated with medium confidenceSummary given by the creator

Figure 1 A screenshot from the program

“This application primarily demonstrates the performance of an AI algorithm, but you can play and try to compete against it too.

While playing, a continuous statistical analysis of the play field and the set blocks is done.

A range of histograms and metrics are offered to assess player performance and to provide insight into playstyle and the game itself.” (*Tetris AI Player, Tetris-hp*)[[7]](#footnote-8)

A screenshot of a game

Description automatically generated with medium confidence

Figure 2 A screenshot showing the various statistics shown by the program

#### Pros + Cons

One of the notable pros of this game is its implementation of a Minmax algorithm with alpha-beta pruning, which enhances the AI's gameplay and offers a challenging experience for solo players. Additionally, the capability to view various in-depth statistics and the ability to step through each move can be beneficial for those looking to improve their strategic skills. However, there are some cons to consider. The user interface is quite basic and may require users to be well-versed in keyboard shortcuts for efficient navigation. The excessive focus on statistics could potentially distract players from focusing on their actual gameplay technique. Furthermore, the screen can feel cluttered with an overload of information. Lastly, the game offers only a single-player mode, lacking cooperative or competitive multiplayer options.

### Tetris 99

#### Summary

*Tetris 99* is a puzzle game in which 99 players compete against each other online to be the last player remaining. The game retains the basic gameplay of previous *Tetris* games, in which players must rotate and place Tetrominoes so that they form complete lines to clear them from the screen. By clearing multiple lines, the player will send garbage blocks to other players. Players can use the right control stick to choose between focusing on attacking players who have more KOs, defending against attackers, attacking players who have mostly cleared boards, or targeting random players; alternatively, the player can use the left control stick to manually target another player.” (*Tetris 99® for Nintendo Switch*, n.d.)[[8]](#footnote-9)

Figure 3 A screenshot from the game

#### Pros + Cons

This game has some noteworthy advantages, including the presence of multiple gameplay modes to keep players engaged and a player progression tree that motivates them to enhance their skills over time. However, it falls short in terms of its educational value, as it does not effectively assist players in improving their gameplay. The game also assumes a certain level of familiarity with its mechanics, which may alienate less experienced players. On the visual side, the retro gaming aesthetic pays homage to the game's origins but can make several features less readable and includes some questionable-quality assets. Additionally, there is a downside in the form of several features hidden behind a paywall, including the versus computer mode and LAN multiplayer, which could limit the overall gaming experience for those not willing to make additional purchases.

## User Needs and Limitations

The interview and research into other software highlighted several user needs and preferences for a program designed to teach Tetris and provide entertainment. First, the program must offer a valuable educational resource, particularly for beginners. This calls for comprehensive tutorials, guides, and resources on Tetris strategies and techniques, helping users grasp the complexities of the game. An essential requirement is to create a user-friendly interface that caters to individuals of all skill levels, ensuring that both newcomers and seasoned players can easily navigate the program.

In addition to its educational value, the program should also be an engaging source of entertainment. The inclusion of various game modes, challenges, and potentially a multiplayer component is proposed, allowing users to enjoy Tetris in a playful and competitive manner. Furthermore, the interview emphasized the importance of following the official Tetris guidelines for the program's interface and graphics. This adherence to standards ensures that the skills acquired in the program can be seamlessly applied to any officially produced Tetris game, underlining the program's potential to serve as a bridge between learning and practical gameplay.

Lastly, it was suggested that the program should consolidate Tetris-related resources, alleviating the need for users to search multiple websites for learning materials. This comprehensive approach aims to offer a comprehensive and convenient solution for Tetris education and entertainment. Moreover, flexibility is desired in terms of adherence to guidelines, allowing for slight divergence to align with user preferences, while also contemplating the potential for wide public availability.

## Modelling the Problem

### Implementation of the system at a high level

In order to implement the system effectively, several key components must be developed and integrated seamlessly. These components encompass user interaction, data storage, and the graphical presentation of the application's interface.

The system will require robust mechanisms for capturing user input, facilitating interactions such as keyboard and mouse inputs. This functionality is pivotal for user engagement and interaction within the application.

For the storage of persistent user data, a reliable and efficient solution is imperative. Utilizing a database management system, such as SQLite, offers a structured approach to storing user data securely across sessions. Additionally, temporary user data needs to be managed during runtime, possibly through in-memory storage or temporary files, to ensure smooth operation of the application.

A graphical user interface (GUI) is essential to provide users with an intuitive and visually appealing interface. To achieve this, Unity will serve as the primary development platform due to its comprehensive game development capabilities, including the creation of sophisticated GUI systems.

### Algorithms and Data Structures

In terms of algorithms, a hashing algorithm will be implemented for the login and registration system to securely store and verify user passwords. Additionally, a rotation matrix algorithm will be utilized to enable the rotation of tetrominoes within the game, enhancing gameplay mechanics.

Data structures play a crucial role in managing game elements efficiently. Dictionaries will be employed to store the attributes of different types of tetrominoes effectively, allowing for easy retrieval and manipulation of game elements. Arrays will store all tetrominoes, providing a structured approach to managing game elements, while 2D arrays (vectors) will represent positional arrays, enabling precise movement of tetrominoes within the play area.

By meticulously planning the implementation approach and selecting appropriate technologies, algorithms, and data structures, the system will be well-equipped to handle user interactions, manage data effectively, and provide an immersive and engaging gaming experience.

### Ideas for programming software

In order to create my program, I have decided to write the code for my project in Visual Studio, using the Unity framework to create the graphics of the program. Unity is an incredibly reliable and robust framework and is still being used to create games to this very day, so it will be a reliable framework to support my project.

Due to the use of Unity, I am greatly limited in the number of languages that I am able to code my program in and so my first idea for the programming language was C#. Having studied the language in Year 12, I am fairly confident in how to construct my program in this language and the languages has a lot of structures built into it that might allow me to complete more complex operations such as enumeration or polymorphism. Also due to the Object-Oriented nature of the language it might allow me to write my code in a more modular manner, allowing for greater maintainability.

Another option that I had previously considered was to use a different framework such as Godot, to allow me to code the program in C++, as it is a language that I am also fairly familiar with having used it to create small scale projects in the past. However, I decided against this idea as I could not find a framework that used C++ that allowed for the level of control that I needed for my project, as most of the C++ game making frameworks are built at a more abstract level to allow those without much programming knowledge to build small scale games

I am also planning to include a relational database system for Login/Registration and Leaderboard systems. In order to do this, I am likely to use SQL to formalise CRUD (Create Read Update Delete) principles in my program so I can create and maintain user profiles inside the program.

## Objectives

### UI and User Navigation

1. The user should be able to navigate around the GUI (Graphical User Interface) of the application with the available buttons and menus. The UI will be displayed using the Unity UI modules
   1. User should be provided with the Start Menu UI
      1. Create button to direct the user to the Registration UI
      2. Create button to direct the user to the Login UI
      3. Create button to direct the user to allow the user to exit the program
   2. When the user selects the option to register then the Registration UI should be presented
      1. The available input fields should be email address, username, password, confirm password
   3. Create a function that checks all of the specified input fields
      1. The authorise function should check if the inputs entered in the password and confirm password fields are the same
      2. The authorise function should check whether a valid email address has been entered in the email address field
      3. Create button to register the user
   4. When the user selects the register button
      1. The register function should encrypt the user’s entered password using a hashing algorithm
      2. The encrypted password should be stored in the database
      3. The username should be stored in the database
      4. The email address should be stored in the database
      5. The user should be brought to the Start Menu UI
   5. When the user selects the option to login then the Login UI should be presented
      1. The available input fields should be username, password
      2. Create a login button that checks the user’s input
   6. When the user selects the login button
      1. Check the database if the given username exists
      2. Run the given user password through the hashing algorithm
      3. Retrieve the hashed password from the database and compare the two passwords to see if they match
      4. If the login is successful the user should be brought to the Game Menu UI
   7. When the user is brought to the Game Menu UI
      1. Create button to direct the user to the Single Player UI
      2. Create button to direct the user to the Multiplayer UI
      3. Create button to direct the user to the Education UI

### Single Player Mode

1. The Single Player mode should allow the user to play a standard single player version of Tetris
   1. Create a 10 x 20-unit rectangular grid which forms the bounds of the play area
   2. Design a suitable gameplay loop
      1. A randomly selected tetromino should appear at the top-centre of the play area
      2. The tetromino should start moving through the play area one unit at a time downwards
      3. When the player clicks the correct button the tetromino should rotate 90 degrees clockwise
      4. When the player clicks the correct button the tetromino should rotate 90 degrees anticlockwise
      5. When the player clicks the correct button the tetromino should move one unit left
      6. When the player clicks the correct button the tetromino should move one unit right
      7. When the player places a piece that completes a line, that line should clear from the playfield
      8. When the player places a piece that completes multiple lines, all completed lines should clear from the playfield
   3. While the gameplay loop is running, create a display for the user’s current score
   4. When the player places a piece outside of the playfield, due to the playfield being full, the gameplay loop should end.
   5. Once the game ends a game over screen should appear
      1. Create a button to allow the player to restart the game
      2. Create a button to return to the game menu
   6. The player should be able to pause the game, either through clicking an on-screen button or pressing the Escape key
   7. When the player pauses the game the appropriate pause screen menu should be displayed
      1. Create a button to allow the player to resume the game
      2. Create a button to allow the player to return to the main menu

### Educational mode

1. The Educational mode should allow the user to view various different pieces of information which includes controls, terminology and technique

### Local Multiplayer

1. The LAN Multiplayer mode should allow two users to play a 2-player version of Tetris on the same device
   1. The users should be presented with two 10 x 20-unit rectangular grid which forms the bounds of both play areas
   2. The suitable gameplay loop should be started
      1. A randomly selected tetromino should appear at the top-centre of both play areas
      2. The tetromino should start moving through both play areas one unit at a time downwards
      3. When the player clicks the correct button the tetromino should rotate 90 degrees clockwise
      4. When the player clicks the correct button the tetromino should rotate 90 degrees anticlockwise
      5. When the player clicks the correct button the tetromino should move one unit left
      6. When the player clicks the correct button the tetromino should move one unit right
      7. When the player places a piece that completes a line, that line should clear from the playfield
      8. When the player places a piece that completes multiple lines, all completed lines should clear from the playfield
      9. While the gameplay loop is running, the program should display both users’ current scores
      10. When the player places a piece outside of the playfield, due to the playfield being full, the gameplay loop should end.
   3. Once the game ends a game over screen should appear
      1. Create a button to allow the player to restart the game
      2. Create a button to allow the player to return to the game menu
      3. Display which player won

# Documented Design

## Overview

As mentioned previously the aim of this program is to provide an easy and accessible way to learn how to play Tetris. To achieve this the player will be able to read through various Tetris methods (Objective 3) and be able to play different variations of Tetris that involve all of the mechanics of a true Tetris Game. The first version is a single player version of Tetris (Objective 2) that records the players score as the game is played (Objective 2.3), a score that may then be added to a player leaderboard. The second version is a multiplayer version that introduces a fun competitive element that allows two players to play the game against each other (Objective 4).

## Program structure

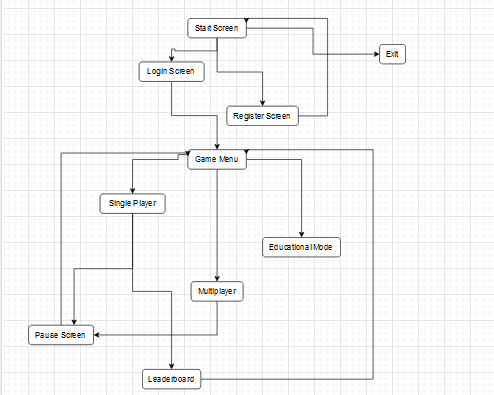
Figure 4 displays the hierarchy chart for my program, showing how the system will be set out. As shown in the chart the user will first be presented with a Start Screen, from which they can access the Login and Register screens, and also choose to exit the program. If the user chooses to register, they will be prompted to enter their details and then will be routed back to the Start Screen to login using the same details. Once the user has logged in, they will be able to access any of the various game modes. In the Single Player and Multiplayer modes, they will be able to access the Pause Screen which allows them to return to the game menu.

Figure 4 System Hierarchy Chart

## Prototyping

In order to figure out the basic operations of the game section of the program, I decided to create a prototype of my program that implemented the logic that would be performed during the running of the single player game. The code I used in the prototype is provided below:

**using** UnityEngine;

**using** System.Collections;

**public** **class** Tetramino : MonoBehaviour

{

**float** fall = 0;

**public** **float** fallSpeed = 1;

*// Update is called once per frame*

**void** Update()

{

CheckUserInput(); *// Check for user input every frame*

}

**void** CheckUserInput()

{

*// Check for user input*

**if** (Input.GetKeyDown(KeyCode.RightArrow)) *// User presses the right arrow key*

{

transform.position += new Vector3(1, 0, 0); *// Move tetramino right by 1 unit*

**if** (!CheckIfPositionValid())

{

transform.position += new Vector3(-1, 0, 0); *// Undo movement if position is not valid*

}

**else**

{

FindObjectOfType<Game>().UpdateGrid(**this**); *// Update the game grid if position is valid*

}

}

**else** **if** (Input.GetKeyDown(KeyCode.LeftArrow)) *// User presses the left arrow key*

{

transform.position += new Vector3(-1, 0, 0); *// Move tetramino left by 1 unit*

**if** (!CheckIfPositionValid())

{

transform.position += new Vector3(1, 0, 0); *// Undo movement if position is not valid*

}

**else**

{

FindObjectOfType<Game>().UpdateGrid(**this**); *// Update the game grid if position is valid*

}

}

**else** **if** (Input.GetKeyDown(KeyCode.UpArrow)) *// User presses the up arrow key*

{

transform.Rotate(0, 0, 90); *// Rotate tetramino 90 degrees clockwise*

**if** (!CheckIfPositionValid())

{

transform.Rotate(0, 0, -90); *// Undo rotation if position is not valid*

}

**else**

{

FindObjectOfType<Game>().UpdateGrid(**this**); *// Update the game grid if position is valid*

}

}

**else** **if** (Input.GetKeyDown(KeyCode.DownArrow) || Time.time - fall >= fallSpeed)

{

*// Move tetramino down or apply gravity*

transform.position += new Vector3(0, -1, 0);

**if** (!CheckIfPositionValid())

{

transform.position += new Vector3(0, 1, 0); *// Undo movement if position is not valid*

FindObjectOfType<Game>().DeleteMinoRow(); *// Delete filled rows*

enabled = **false**; *// Disable the script component*

FindObjectOfType<Game>().SpawnNewTetramino(); *// Spawn a new tetramino*

}

**else**

{

FindObjectOfType<Game>().UpdateGrid(**this**); *// Update the game grid if position is valid*

}

fall = Time.time; *// Reset fall time*

}

}

**bool** CheckIfPositionValid()

{

*// Check if the current position of the tetramino is valid*

**foreach** (Transform mino **in** transform)

{

Vector2 pos = FindObjectOfType<Game>().Rounding(mino.position);

*// Check if position is within the game grid*

**if** (!FindObjectOfType<Game>().CheckIsInTheGrid(pos))

{

**return** **false**;

}

*// Check if position is occupied by another tetramino*

**if** (FindObjectOfType<Game>().GetTransformedAtGridPosition(pos) != **null** && FindObjectOfType<Game>().GetTransformedAtGridPosition(pos).parent != transform)

{

**return** **false**;

}

}

**return** **true**; *// Position is valid*

}

}

**using** System.Collections;

**using** System.Collections.Generic;

**using** UnityEngine;

**public** **class** Game : MonoBehaviour

{

**public** **static** **int** gridWidth = 11;

**public** **static** **int** gridHeight = 20;

**public** **static** Transform[,] grid = new Transform[gridWidth, gridHeight];

**void** Start()

{

SpawnNewTetramino (); *// Start the game by spawning a new tetramino*

}

**public** **bool** IsRowFull(**int** y)

{

*// Check if a row is full*

**for** (**int** x = 0; x < gridWidth; ++x)

{

**if** (grid[x, y] == **null**)

{

**return** **false**; *// Row is not full*

}

}

**return** **true**; *// Row is full*

}

**public** **void** DeleteLine(**int** y)

{

*// Delete a filled row*

**for** (**int** x = 0; x < gridWidth; ++x)

{

Destroy(grid[x, y].gameObject); *// Destroy the game object in the grid cell*

grid[x, y] = **null**; *// Clear the grid cell*

}

}

**public** **void** MoveMinoRowDown(**int** y)

{

*// Move a row of minos down*

**for** (**int** x = 0; x < gridWidth; ++x)

{

**if** (grid[x, y] != **null**)

{

grid[x, y - 1] = grid[x, y]; *// Move the mino down*

grid[x, y] = **null**; *// Clear the original grid cell*

grid[x, y - 1].position += new Vector3(0, -1, 0); *// Update the mino's position*

}

}

}

**public** **void** MoveAllMinoRowsDown(**int** y)

{

*// Move all rows above a certain row down*

**for** (**int** j = y; j < gridHeight; ++j)

{

MoveMinoRowDown(j); *// Move each row down*

}

}

**public** **void** DeleteMinoRow()

{

*// Delete filled rows*

**for** (**int** i = 0; i < gridHeight; ++i)

{

**if** (IsRowFull(i))

{

DeleteLine(i); *// Delete the filled row*

MoveAllMinoRowsDown(i + 1); *// Move all rows above the deleted row down*

--i; *// Decrease the index to recheck the same row*

}

}

}

**public** **void** UpdateGrid(Tetramino tetramino)

{

*// Update the game grid with the current tetramino*

**for** (**int** j = 0; j < gridHeight; j++)

{

**for** (**int** i = 0; i < gridWidth; i++)

{

**if** (grid[i, j] != **null**)

{

**if** (grid[i, j].parent == tetramino.transform)

{

grid[i, j] = **null**; *// Clear the grid cell if it belongs to the tetramino*

}

}

}

}

*// Update the grid with the minos of the tetramino*

**foreach** (Transform mino **in** tetramino.transform)

{

Vector2 pos = Rounding(mino.position);

**if** (pos.y < gridHeight)

{

grid[(**int**)pos.x, (**int**)pos.y] = mino; *// Update the grid cell with the mino*

}

}

}

**public** **void** SpawnNewTetramino()

{

*// Spawn a new tetramino*

GameObject newTetramino = (GameObject)Instantiate(Resources.Load(GetNextTetramino(), typeof(GameObject)), new Vector2(5.0f, 20.0f), Quaternion.identity);

}

**public** **bool** CheckIsInTheGrid(Vector2 pos)

{

*// Check if a position is within the game grid*

**return** ((**int**)pos.x <= gridWidth && (**int**)pos.y > 0 && (**int**)pos.x > 0);

}

**public** Vector2 Rounding(Vector2 pos)

{

*// Round a position to the nearest integer*

**return** new Vector2(Mathf.Round(pos.x), Mathf.Round(pos.y));

}

**public** Transform GetTransformedAtGridPosition(Vector2 pos)

{

*// Get the transform at a specific grid position*

**if** (pos.y > gridHeight - 1)

{

**return** **null**; *// Return null if the position is out of bounds*

}

**else**

{

**return** grid[(**int**)pos.x, (**int**)pos.y]; *// Return the transform at the specified position*

}

}

**string** GetNextTetramino()

{

*// Get the name of the next tetramino*

**int** nextTetramino = Random.Range(0, 8);

**string** nextTetraminoName = "J Tetramino";

*// Determine the name of the next tetramino based on random selection*

**switch** (nextTetramino)

{

**case** 0:

nextTetraminoName = "I Tetramino";

**break**;

**case** 1:

nextTetraminoName = "J Tetramino";

**break**;

**case** 2:

nextTetraminoName = "L Tetramino";

**break**;

**case** 3:

nextTetraminoName = "O Tetramino";

**break**;

**case** 4:

nextTetraminoName = "S Tetramino";

**break**;

**case** 5:

nextTetraminoName = "T Tetramino";

**break**;

**case** 6:

nextTetraminoName = "Z Tetramino";

**break**;

}

**return** nextTetraminoName; *// Return the name of the next tetramino*

}

}

After completing the prototype, several issues became apparent that I aim to address in the final project. One notable limitation was the use of pre-created tetramino sprites, which restricted my ability to fully control the rotation of each tetramino. Consequently, I found it challenging to implement advanced features such as the Super Rotation System, which requires precise control over rotation and the ability to perform wall kicks and restricted rotations for certain pieces. Additionally, the absence of a hard drop feature hindered the gameplay experience, as players were unable to quickly drop pieces to the bottom of the playfield. Another issue encountered was occasional instances where pieces would stop or hang in the middle of the playfield due to array errors, indicating the need for more robust error handling and boundary checks in the game logic.

I then created a second prototype as shown below:

**using** System.Collections;  
**using** System.Collections.Generic;  
**using** UnityEngine;  
**using** UnityEngine.SceneManagement; *// Required for scene management*  
   
**public** **class** **Group** : MonoBehaviour  
{  
 **float** lastFall = 0; *// Tracks the time of the last fall*  
 **float** fallSpeed = 1.0f; *// Speed at which the tetramino falls*  
   
 *// Function to move the tetramino in a specified direction*  
 **void** Move(Vector3 direction)  
 {  
 transform.position += direction; *// Move the tetramino in the specified direction*  
   
 *// Check if the new position is valid*  
 **if** (isValidGridPos())  
 updateGrid(); *// Update the grid if the position is valid*  
 **else**  
 transform.position -= direction; *// Revert the move if the position is not valid*  
 }  
   
 *// Function to rotate the tetramino*  
 **void** Rotate()  
 {  
 transform.Rotate(0, 0, -90); *// Rotate the tetramino clockwise by 90 degrees*  
   
 *// Check if the new rotation is valid*  
 **if** (isValidGridPos())  
 updateGrid(); *// Update the grid if the rotation is valid*  
 **else**  
 transform.Rotate(0, 0, 90); *// Revert the rotation if it's not valid*  
 }  
   
 *// Function to move the tetramino downwards*  
 **void** MoveDown()  
 {  
 transform.position += new Vector3(0, -1, 0); *// Move the tetramino downwards*  
   
 *// Check if the new position is valid*  
 **if** (isValidGridPos())  
 {  
 updateGrid(); *// Update the grid if the position is valid*  
 }  
 **else**  
 {  
 transform.position += new Vector3(0, 1, 0); *// Revert the move if the position is not valid*  
 Playfield.deleteFullRows(); *// Check and delete any full rows in the playfield*  
 FindObjectOfType<Spawner>().spawnNext(); *// Spawn the next tetramino*  
 enabled = **false**; *// Disable this tetramino script component*  
 }  
   
 lastFall = Time.time; *// Update the last fall time*  
 }  
   
 *// Function to move the tetramino downwards rapidly*  
 **void** MoveDownFast()  
 {  
 transform.position += new Vector3(0, -1, 0); *// Move the tetramino downwards*  
   
 *// Check if the new position is valid*  
 **if** (isValidGridPos())  
 {  
 updateGrid(); *// Update the grid if the position is valid*  
 }  
 **else**  
 {  
 transform.position += new Vector3(0, 1, 0); *// Revert the move if the position is not valid*  
 Playfield.deleteFullRows(); *// Check and delete any full rows in the playfield*  
 FindObjectOfType<Spawner>().spawnNext(); *// Spawn the next tetramino*  
 enabled = **false**; *// Disable this tetramino script component*  
 }  
   
 lastFall = Time.time; *// Update the last fall time*  
 }  
   
 *// Function to check if the current grid position of the tetramino is valid*  
 **bool** isValidGridPos()  
 {  
 **foreach** (Transform child **in** transform)  
 {  
 Vector2 vec = Playfield.roundVec2(child.position); *// Round the position to integers*  
   
 *// Check if the position is inside the playfield and not occupied by another tetramino*  
 **if** (!Playfield.insideBorder(vec) || (Playfield.grid[(**int**)vec.x, (**int**)vec.y] != **null** && Playfield.grid[(**int**)vec.x, (**int**)vec.y].parent != transform))  
 **return** **false**;  
 }  
 **return** **true**; *// Return true if all positions are valid*  
 }  
   
 *// Function to update the grid with the tetramino's current position*  
 **void** updateGrid()  
 {  
 **for** (**int** y = 0; y < Playfield.height; y++)  
 **for** (**int** x = 0; x < Playfield.width; x++)  
 **if** (Playfield.grid[x, y] != **null** && Playfield.grid[x, y].parent == transform)  
 Playfield.grid[x, y] = **null**; *// Clear the grid cell if it belongs to this tetramino*  
   
 **foreach** (Transform child **in** transform)  
 {  
 Vector2 v = Playfield.roundVec2(child.position); *// Round the position to integers*  
 Playfield.grid[(**int**)v.x, (**int**)v.y] = child; *// Update the grid with the tetramino's child transforms*  
 }  
 }  
   
 *// Function called every frame*  
 **void** Update()  
 {  
 **if** (!PauseMenuUI.isPaused) *// Check if the game is not paused*  
 {  
 *// Check for user input to move or rotate the tetramino*  
 **if** (Input.GetKeyDown(KeyCode.LeftArrow)) Move(new Vector3(-1, 0, 0));  
 **else** **if** (Input.GetKeyDown(KeyCode.RightArrow)) Move(new Vector3(1, 0, 0));  
 **else** **if** (Input.GetKeyDown(KeyCode.UpArrow)) Rotate();  
 **else** **if** (Input.GetKeyDown(KeyCode.DownArrow) || Time.time - lastFall >= fallSpeed) MoveDown();  
 **else** **if** (Input.GetKey(KeyCode.RightShift) && Input.GetKey(KeyCode.DownArrow) || Time.time - lastFall >= fallSpeed) MoveDown();  
 }  
 }  
   
 *// Function called when the object is initialized*  
 **void** Start()  
 {  
 **if** (!isValidGridPos()) *// Check if the initial position of the tetramino is valid*  
 {  
 Destroy(gameObject); *// Destroy the tetramino object*  
 SceneManager.LoadScene("Scene8 - Game Over"); *// Load the game over scene*  
 }  
 }  
}  
**using** System.Collections;  
**using** System.Collections.Generic;  
**using** UnityEngine;  
   
**public** **class** Manager : MonoBehaviour  
{  
 **public** **static** **int** score = 0;  
 **public** **static** **int** linesCleared = 0;  
   
 *// Reset the score to zero*  
 **public** **static** **void** ResetScore()  
 {  
 score = 0;  
 }  
}  
**using** System.Collections;  
**using** System.Collections.Generic;  
**using** UnityEngine;  
   
**public** **class** Playfield : MonoBehaviour  
{  
 **public** **static** **int** width = 10;  
 **public** **static** **int** height = 20;  
 **public** **static** Transform[,] grid = new Transform[width, height]; *// 2D array representing the game grid*  
   
 *// Rounds a Vector2 to the nearest integer values*  
 **public** **static** Vector2 roundVec2(Vector2 v)  
 {  
 **return** new Vector2(Mathf.Round(v.x), Mathf.Round(v.y));  
 }  
   
 *// Checks if a given position is inside the game grid*  
 **public** **static** **bool** insideBorder(Vector2 pos)  
 {  
 **return** ((**int**)pos.x >= 0 && (**int**)pos.x < width && (**int**)pos.y >= 0);  
 }  
   
 *// Deletes a row from the grid*  
 **public** **static** **void** deleteRow(**int** y)  
 {  
 **for** (**int** x = 0; x < width; x++)  
 {  
 Destroy(grid[x, y].gameObject); *// Destroys the game object in the grid cell*  
 grid[x, y] = **null**; *// Clears the grid cell*  
 }  
 }  
   
 *// Moves all blocks of a row down by one cell*  
 **public** **static** **void** decreaseRow(**int** y)  
 {  
 **for** (**int** x = 0; x < width; x++)  
 {  
 **if** (grid[x, y] != **null**)  
 {  
 grid[x, y - 1] = grid[x, y]; *// Moves the block reference down*  
 grid[x, y] = **null**; *// Clears the current grid cell*  
 grid[x, y - 1].position += new Vector3(0, -1, 0); *// Moves the block in the game world*  
 }  
 }  
 }  
   
 *// Decreases all rows above the given row by one cell*  
 **public** **static** **void** decreaseRowsAbove(**int** y)  
 {  
 **for** (**int** i = y; i < height; i++)  
 decreaseRow(i);  
 }  
   
 *// Checks if a row is full*  
 **public** **static** **bool** isRowFull(**int** y)  
 {  
 **for** (**int** x = 0; x < width; x++)  
 **if** (grid[x, y] == **null**)  
 **return** **false**;  
 **return** **true**;  
 }  
   
 *// Deletes all full rows from the grid*  
 **public** **static** **void** deleteFullRows()  
 {  
 **for** (**int** y = 0; y < height; y++)  
 {  
 **if** (isRowFull(y))  
 {  
 deleteRow(y); *// Deletes the full row*  
 decreaseRowsAbove(y + 1); *// Moves down all rows above*  
 y--; *// Decrements y to recheck the current row*  
 Manager.score += 100; *// Increases the score*  
 Manager.linesCleared++; *// Increases the number of lines cleared*  
 }  
 }  
 }  
}  
**using** System.Collections;  
**using** System.Collections.Generic;  
**using** TMPro;  
**using** UnityEngine;  
   
**public** **class** ScoreDisplay : MonoBehaviour  
{  
 **public** TMP\_Text scoreText;  
   
 *// Update is called once per frame*  
 **void** Update()  
 {  
 *// Update the score displayed on the UI with the current score value from the Manager class*  
 scoreText.text = "" + Manager.score;  
 }  
}  
**using** System.Collections;  
**using** System.Collections.Generic;  
**using** UnityEngine;  
   
**public** **class** Spawner : MonoBehaviour  
{  
 **public** GameObject[] tetraminos; *// Array to hold different tetramino prefabs*  
 **public** GameObject[] nextTetraminos = new GameObject[3]; *// Array to store the next three tetraminos*  
 **public** Transform queueDisplayPosition; *// Position for displaying the next tetraminos*  
   
 **void** Start()  
 {  
 *// Start the game with the first tetramino*  
 spawnNext();  
 }  
   
 *// Spawns the next tetramino in the game*  
 **public** **void** spawnNext()  
 {  
 *// Instantiate a random tetramino prefab at the spawner's position*  
 GameObject currentTetramino = Instantiate(tetraminos[Random.Range(0, tetraminos.Length)], transform.position, Quaternion.identity);  
 Debug.Log("Spawned at " + transform.position);  
 }  
}

This prototype also had similar issues as the first prototype lacking controlled rotation, however it also came with its own set of issues. For example, the code lacks object-oriented design principles. Functions related to tetramino movement and grid management are scattered across different scripts without a clear hierarchy or organization. To address this issue, implementing a more structured and modular design using object-oriented principles would be beneficial. Creating separate classes for entities such as Tetramino, Grid, and GameManager, each responsible for specific functionalities, would improve code organization, reusability, and maintainability.

Another area of concern was the inefficient collision detection mechanism used in the Tetramino class. The current approach relies on iterating through all the minos in the tetramino and checking their positions against the game grid. This approach may become inefficient as the size of the grid or the number of minos increases, leading to performance issues, especially on lower-end devices. To optimize collision detection, implementing a more optimized grid-based collision detection method would be beneficial.

More importantly this version did improve on some of the feature of the first prototype. Namely this protype includes hard-drop movement and a score manager that allowed me to display the score onscreen.

These insights from the prototype phase will guide my efforts to improve the game mechanics and overall user experience in the final project.

## System Flowcharts

### Start Screen System

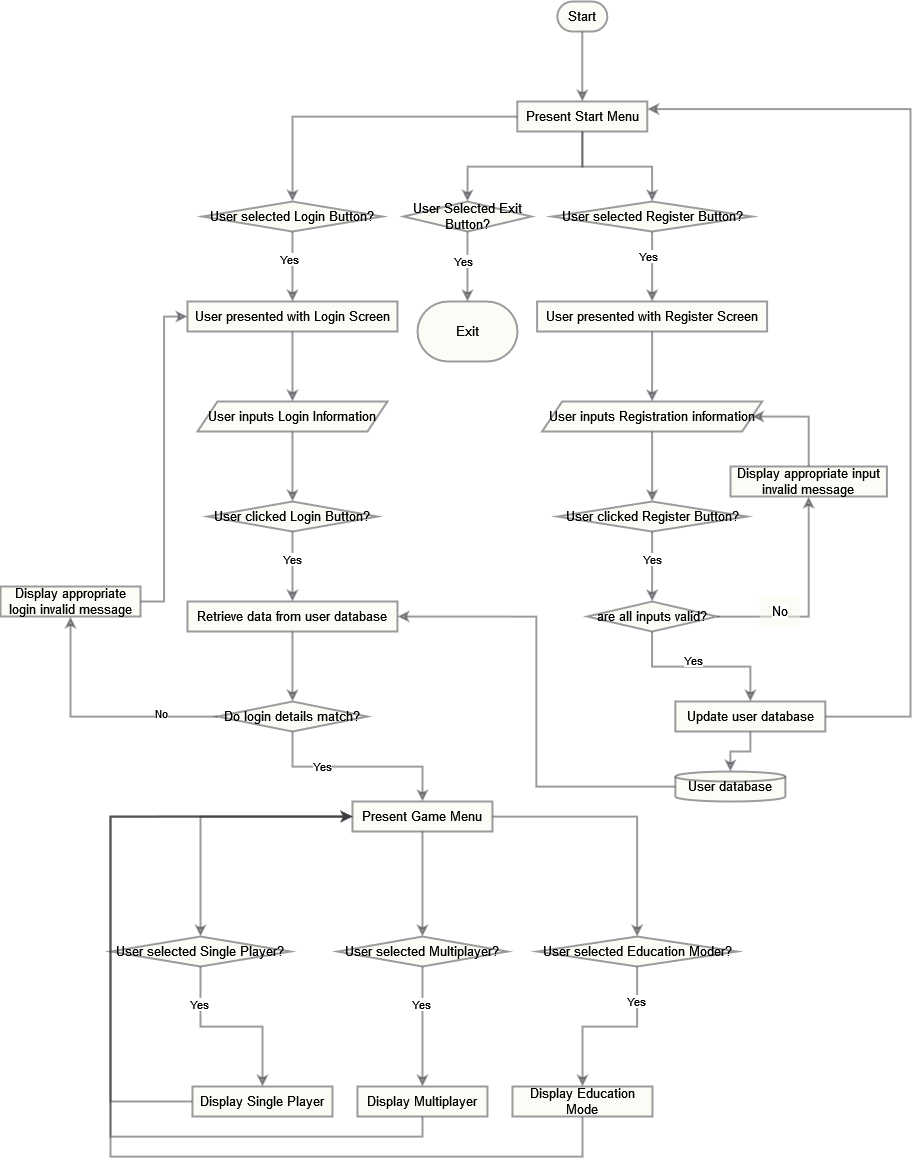
Figure 5 show the basic logic system for the Start Screen. When the start screen is displayed it will present three options: Login, Register, and Exit. Selecting the Login Button will present them with the Login screen where they can enter their user information and progress to the Game screen. Selecting the Register button will present them with the Registration Screen where they can enter their registration data which will then be stored in the database and the user redirected back to the start menu. Selecting the exit button will exit the program

Figure 5 Start Screen System

### Basic Gameplay Loops

#### Tetromino movement

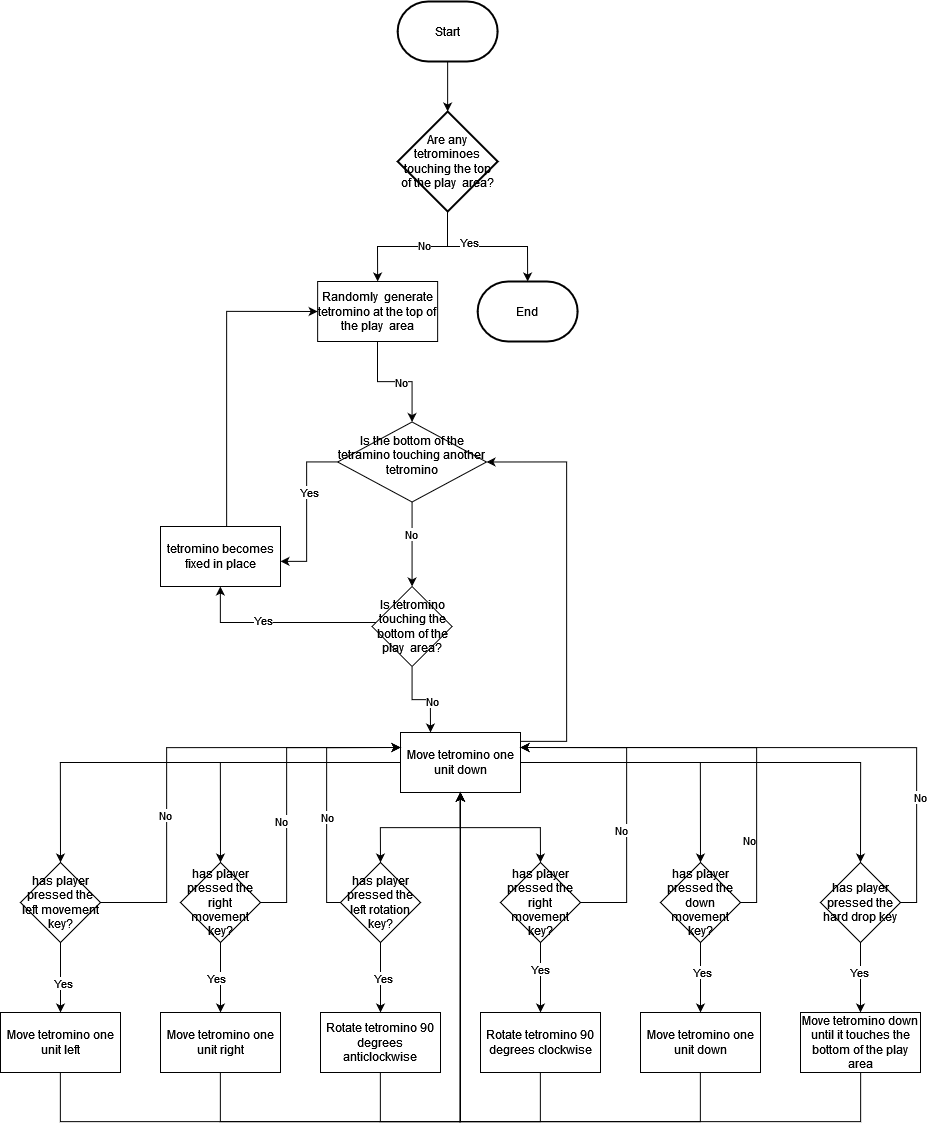


Figure 6 A flowchart of the basic movement of a tetromino through the playfield

Figure 6 shows the basic movement of a tetromino through the playfield. Randomly selected tetrominoes fall from the top of a 10 x 20 playfield, one at a time. Each tetromino moves downward through the playfield at a predetermined speed. Whilst the tetromino is falling the player can shift the piece down, right or left and rotate it in intervals of 90 degrees. Once the tetromino lands on the floor of the playfield or on other blocks the player will no longer be able to control the piece.

#### Line clear check

Figure 7 is a visualisation of how the program may check which lines to clear from the playfield, due to the line being full. It shows a more optimised method where instead of constantly checking if every line in the play area is full, it checks from the bottom up until the grid is filled.

Figure 7 Line clear flowchart

## Class Diagrams

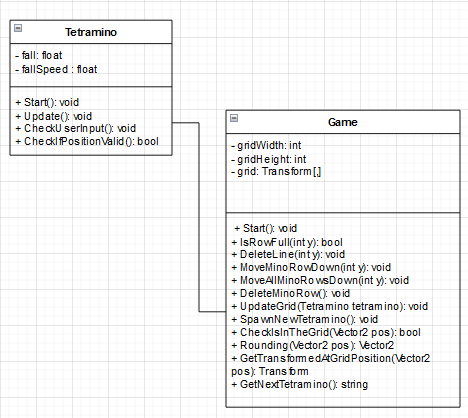


Figure 8 Class Diagram

The class diagram in Figure 8 displays a basic idea of what the classes in the base game may look like. The Tetramino class represents the individual tetramino objects in the game, handling their movement and position validation. The Game class manages the game grid, including spawning tetraminos, updating the grid, and handling row clearing. The Tetramino class interacts with the Game class through method calls. For example, in the CheckIfPositionValid() method of the Tetramino class, it calls methods of the Game class such as Rounding() and CheckIsInTheGrid(). Similarly, the Game class interacts with the Tetramino class through method calls, like UpdateGrid() and SpawnNewTetramino().

## Data Flow Diagram

Figure 9 Depicts what a potential data flow diagram of my program might look like. It’s entities and functions are detailed below:

External entities:

* User input: this represents any input provided by the user, such as keyboard inputs (arrow keys) to control the game.

Processes:

* Game logic: this process manages the overall game logic, including user input handling, updating the game grid, spawning new tetraminos, checking for completed rows, and scoring.
* Tetramino: this process represents the behavior and movement of tetraminos within the game, including rotation, movement, and checking validity of positions.
* Playfield: this process manages the game grid, including updating the grid with tetramino positions, checking for completed rows, and handling row deletion and shifting.

Figure 9A Data Flow Diagram showing the flow of data in my program using Gane and Sarson Notation

* Manager: this process handles game-level management tasks, such as resetting the score.

Data stores:

* Grid: this data store represents the grid of the game where tetraminos are placed. It holds information about the current state of the game board.
* Score: this data store stores the current score of the game.

Data flows:

* User input -> game logic: represents the flow of user input to the game logic process for controlling tetraminos.
* Game logic -> tetramino: flow of control and data from the game logic process to the tetramino process for updating tetramino positions and rotations.
* Tetramino -> game logic: feedback from the tetramino process to the game logic process after tetramino movements or rotations.
* Game logic -> playfield: transfer of data from the game logic process to the playfield process for updating the game grid.
* Playfield -> game logic: notification from the playfield process to the game logic process about completed rows or other events.
* Playfield -> manager: data flow from the playfield process to the manager process for updating the score.
* Manager -> game logic: communication from the manager process to the game logic process, such as resetting the score.

## Database Design

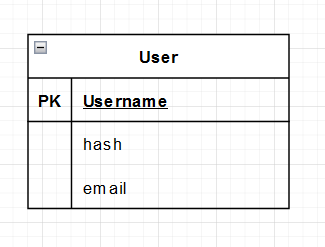
Figure 9 shows the basic structure of the database that will be present in the program.

Figure 10 Data Base Diagram

The User Table holds user-specific information, including usernames, hashed passwords, email addresses, and timestamps of creation.

Each user's username serves as a unique identifier, acting as the primary key for the table.

The schema ensures data integrity by using appropriate data types for each field.

### Database queries

I am planning to have the following queries:

SELECT \* FROM User

* Selects all the data from the User table.

DROP TABLE IF EXISTS User

* Deletes all the data for the User table

SELECT COALESCE(MAX(id)+1, 0) FROM User

* It counts how many rows there are in the table User, allowing for a quick snapshot of how many users have been registered to the program

INSERT INTO USER (user, hash, email) VALUES (‘ user ', ' hash ', ' email ‘)

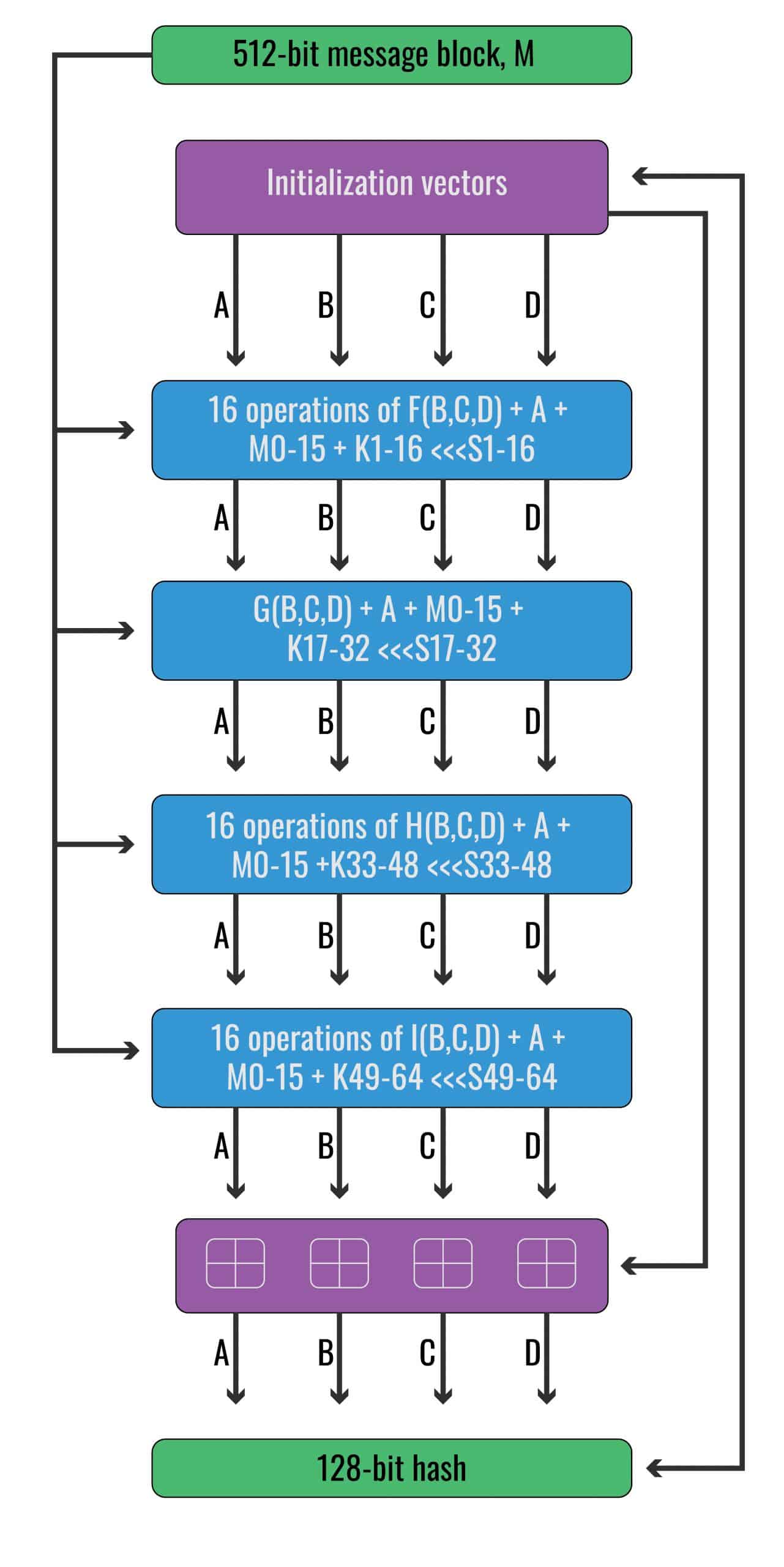
* Inserts a new entry into the User table, specifying values for the user, hash, and email fields

SELECT \* FROM USER WHERE KEY\_USER = ' user '

* Selects the record from the table User where the user field matches the provided user string

## Key Algorithms

Figure 11 The MD5 Algorithm

MD5 Algorithm  
The MD5 algorithm functions by converting an input string, of variable length, into a fixed-length 128-bit hash value, commonly referred to as a fingerprint. Subsequently, for a given input string, the MD5 algorithm consistently produces the same hash output. Primarily, MD5 hashes find application in scenarios involving the storage of sensitive information, such as passwords or credit card numbers, within databases like MySQL. This encoding method facilitates data integrity verification, particularly in file management contexts, where users compare the MD5 hash of a source file with that of a destination file to confirm the latter's integrity and authenticity.

It is essential to note that MD5 hashing differs fundamentally from encryption. Unlike encryption, which allows for reversible transformations, MD5 hashing represents a one-way process. Consequently, it is exceedingly difficult, if not practically impossible, to reverse-engineer an MD5 hash to retrieve the original input string.[[9]](#footnote-10)

How it works (image opposite)[[10]](#footnote-11):

Padding: The input message is padded to ensure its length is congruent to 448 modulo 512 bits. This padding involves appending a single '1' bit followed by '0' bits until the length satisfies the padding condition. Additionally, the length of the original message is appended as a 64-bit little-endian integer to the padded message.

Initialization: MD5 initializes four 32-bit registers (A, B, C, D) with predetermined constant values. These registers serve as accumulators for the hash computation.

Message Processing: The padded message is divided into 512-bit (64-byte) blocks. Each block is further divided into 32-bit (4-byte) words, forming a total of 16 words per block.

Round Function: MD5 employs a series of rounds (64 rounds in total), each consisting of four main operations: F, G, H, and I functions, along with bitwise rotations and additions. These operations are applied to each 512-bit block of the message iteratively.

F Function: Performs a bitwise combination of the B and C registers using AND, OR, and XOR operations.

G Function: Performs a bitwise combination of the B and D registers using AND, OR, and XOR operations.

H Function: Performs a bitwise combination of the B, C, and D registers using AND, OR, and XOR operations.

I Function: Performs a bitwise combination of the C and (not D) registers using AND, OR, and XOR operations.

Rotation and Addition: Within each round, the values of the registers are rotated left by a fixed number of bits and then added to the current word of the message block, along with a predetermined constant value derived from the sine function.

Update Registers: After applying the round function to each block, the values in the A, B, C, and D registers are updated based on the results of the round.

Final Hash: Once all blocks have been processed, the final hash value is generated by concatenating the values of the four registers (A, B, C, D) in little-endian byte order. The resulting 128-bit hash value represents a unique fingerprint of the input message.

Pseudocode

function MD5(string message):

// Step 1: Initialization

constants = InitializeConstants()

shifts = InitializeShifts()

s = InitializeS()

A = 0x67452301 // Initialize variables

B = 0xefcdab89

C = 0x98badcfe

D = 0x10325476

// Step 2: Padding

message = AppendPadding(message)

// Step 3: Process Message in 512-bit Blocks

for each 512-bit block M in message:

// Step 4: Initialize hash values for this chunk

AA = A

BB = B

CC = C

DD = D

// Step 5: Main Loop

for i from 0 to 63:

if 0 ≤ i ≤ 15 then

F = (B AND C) OR ((NOT B) AND D)

g = i

else if 16 ≤ i ≤ 31 then

F = (D AND B) OR ((NOT D) AND C)

g = (5\*i + 1) MOD 16

else if 32 ≤ i ≤ 47 then

F = B XOR C XOR D

g = (3\*i + 5) MOD 16

else if 48 ≤ i ≤ 63 then

F = C XOR (B OR (NOT D))

g = (7\*i) MOD 16

dTemp = D

D = C

C = B

B = B + leftrotate((A + F + constants[i] + M[g]), shifts[i])

A = dTemp

// Step 6: Update hash values

A = A + AA

B = B + BB

C = C + CC

D = D + DD

// Step 7: Produce the final hash value

hash = concatenate(A, B, C, D)

return hash

function InitializeConstants():

constants = array of 64 constants

for i from 0 to 63:

constants[i] = floor(abs(sin(i + 1)) × (2^32))

return constants

function InitializeShifts():

shifts = array of 64 shifts

for i from 0 to 63:

shifts[i] = (7\*i) MOD 32

return shifts

function InitializeS():

s = array of 64 shifts

for i from 0 to 63:

s[i] = i

return s

function leftrotate(x, c):

return (x << c) OR (x >> (32 - c))

function AppendPadding(message):

length = length of message in bits

message = message + bit '1'

while length + 1 mod 512 ≠ 448:

message = message + bit '0'

message = message + length in binary representation

return message

### Rotation Matrix

A rotation matrix algorithm is a mathematical operation used to rotate points or vectors in a coordinate system. It's particularly useful in computer graphics, robotics, and physics simulations where objects need to be rotated around an axis.

The algorithm involves multiplying the coordinates of each point by a specific matrix to achieve the desired rotation. The matrix used depends on the angle of rotation and the axis around which the rotation occurs.

For example, in a 2D rotation matrix, a point (x, y) can be rotated counterclockwise around the origin by an angle θ using the following formulas:

new\_x = x \* cos(θ) - y \* sin(θ)

new\_y = x \* sin(θ) + y \* cos(θ)

Here, cos(θ) and sin(θ) are the cosine and sine of the angle θ, respectively.

Similarly, in a 3D rotation matrix, a point (x, y, z) can be rotated around one of the coordinate axes (x-axis, y-axis, or z-axis) using a 3x3 matrix. The formulas for each axis's rotation matrix depend on the axis of rotation and the angle of rotation.

In essence, the rotation matrix algorithm calculates the new coordinates of a point after rotating it by a specified angle around a particular axis. This process is crucial for various applications, including rendering 3D graphics, controlling robotic arms, and simulating physical phenomena.

In the program, I will utilize a rotation matrix to dynamically rotate the tetrominoes as they fall onto the playfield. By applying the rotation matrix algorithm to the current tetramino’s coordinates, the program will update its position to reflect a clockwise or counterclockwise rotation. This operation ensures that the tetrominoes align properly with the existing blocks on the playfield, allowing players to strategically manoeuvre and fit them into gaps. As the tetromino descends, it will continuously calculate and update its rotated position, providing a smooth and responsive gameplay experience.

Pseuodocode

function ApplyRotationMatrix(direction):

matrix = Data.RotationMatrix

for each cell in cells:

cell\_x = cell.x

cell\_y = cell.y

if data.tetromino is "I" or "O":

cell\_x -= 0.5

cell\_y -= 0.5

x = CeilToInt(cell\_x \* matrix[0] \* direction + cell\_y \* matrix[1] \* direction)

y = CeilToInt(cell\_x \* matrix[2] \* direction + cell\_y \* matrix[3] \* direction)

else:

x = RoundToInt(cell\_x \* matrix[0] \* direction + cell\_y \* matrix[1] \* direction)

y = RoundToInt(cell\_x \* matrix[2] \* direction + cell\_y \* matrix[3] \* direction)

cells[i] = Vector3Int(x, y, 0)

## Human Computer Interface

I have sketched out below some of the UI scenes and elements that I plan to incorporate into my program.

Figure 11 depicts what the opening screen of the program might look like, displaying the Title and overall aesthetic of the program. It should include a Login button to access the login screen and a Register button to access the registration page. It will also include an Exit button to allow the user to leave the program.

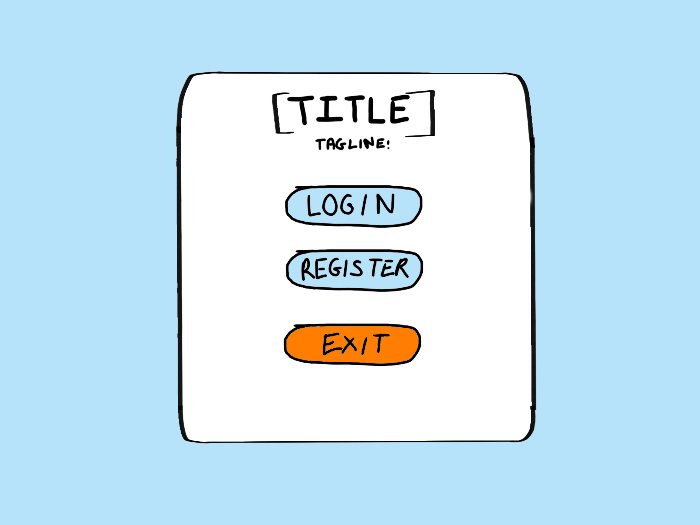


Figure 12 Start Menu Screen

Figure 12 shows the user interface for a potential login screen. It includes two input fields for the user’s username and password, and also a button to allow the user to login and access the rest of the program.

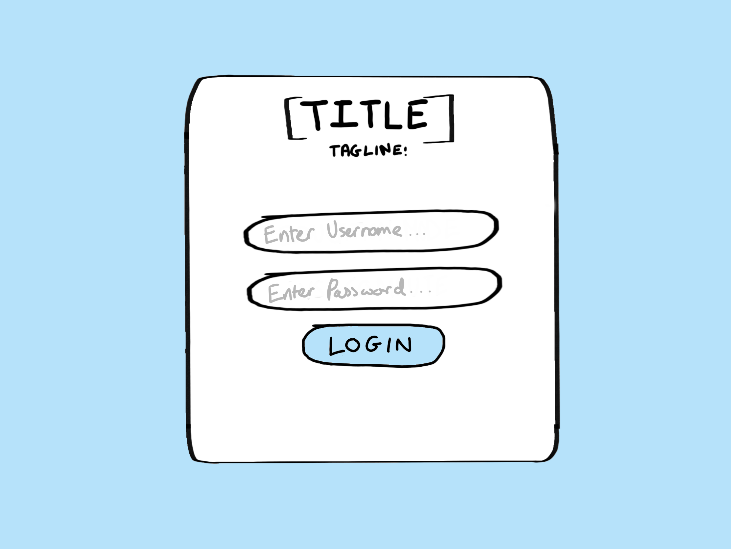


Figure 13 Login Screen

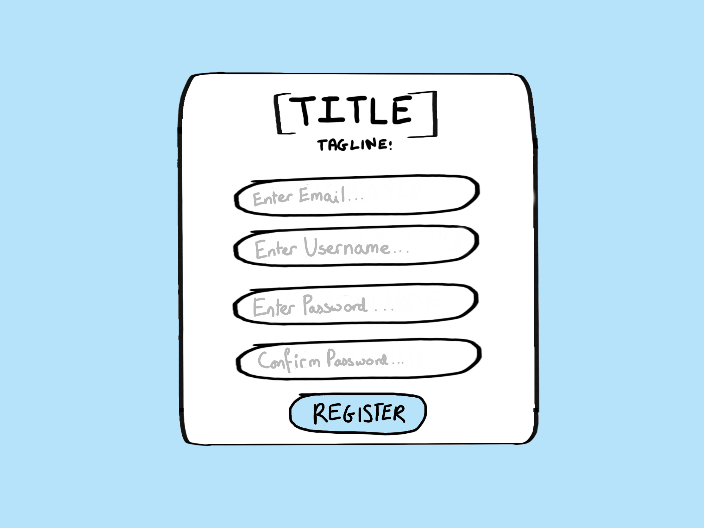
Figure 14 shows the registration screen of the program. It will include four input fields: email, username, password and a password confirmation field. It will also include a button that will allow the user to register their details and redirect them back to the start menu to login with those details.

Figure 14 Game menu

Figure 15 Registration Screen

Figure 15 shows the game menu for the program. It will display three buttons that allow the user to access each of the modes available: a single player mode, a multiplayer mode and a education mode.

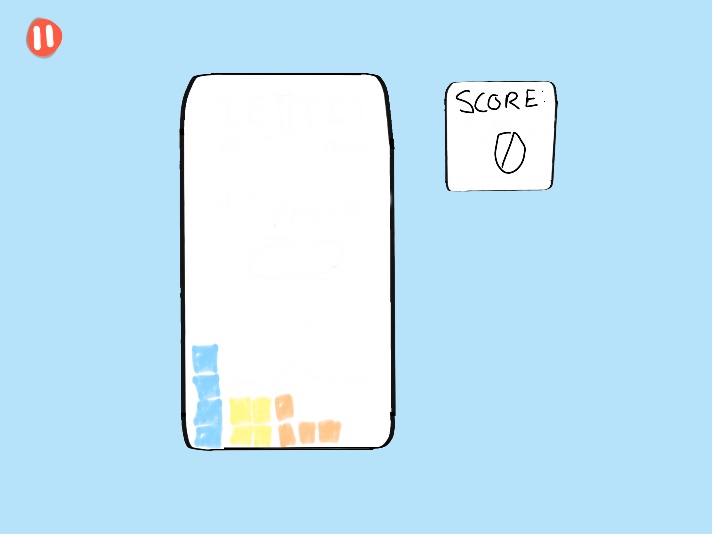


Figure 15 shows the proposed User Interface for the Single Player Mode of the program. This includes the playfield in which the tetrominoes will spawn, a display of the users current score and a pause button that lets the user pause the current game.

Figure 16 Single player

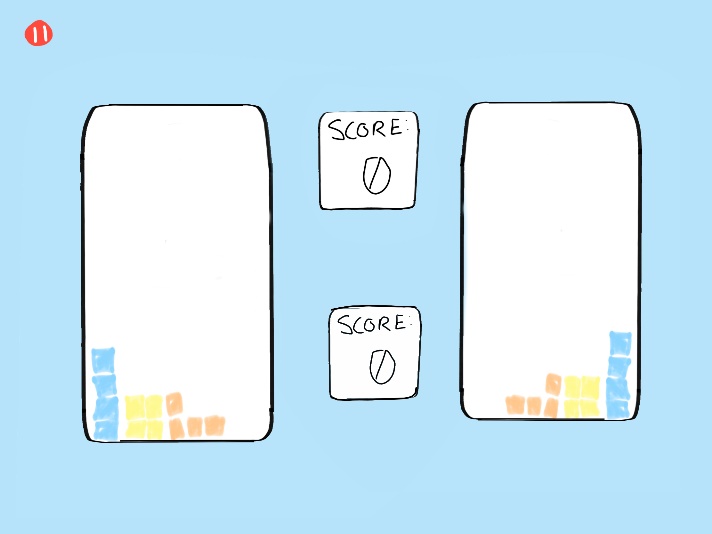


Figure 16 shows the proposed deign for the Multiplayer mode of the program. This mode includes two separate play areas where each of the players tetrominoes will spawn and two separate score displays for each of the players.

Figure 17 Multiplayer mode

Hardware Requirements

Ideally, as the program is designed for education purposes the program should be executable on most general-purpose laptops, as these will be the kind of devices that the club members will typically use.

# Technical Solutions

## Overview

In this section I have provided an annotated copy of all the code used in this program, along with a description at the start of each section of what each script is designed to do. This section will also go over all of the complex techniques that I have used in my code pointing out exactly where they are used.

## Techniques used

|  |  |
| --- | --- |
| **Instruction** | **Page Number** |
| OOP | Used Extensively |
| Classes |  |
| Struct |  |
| Enum |  |
| Inheritance |  |
| Polymorphism |  |
| Procedures and Functions | Used Extensively |
| Unity 3D or 2D simulation | Used Extensively |
| Arrays |  |
| Graphs/Trees/neural networks |  |
| Queues |  |
| Stacks |  |
| Lists |  |
| Sets |  |
| Vectors |  |
| Dictionary |  |
| Records |  |
| Sorts |  |

## Scripts

### Base Game

Board: Manages the game board, including spawning pieces, detecting valid positions, clearing lines, calculating scores, and handling game over conditions.

Piece: Represents a Tetromino piece in the game. It handles movement (including rotation and dropping), collision detection with the board, locking pieces in place, and updating the score manager.

**Board.cs Techniques:**

**Classes:**

Board

**Procedures and Functions:**

Awake()

Start()

SpawnPiece()

GameOver()

Set(Piece piece)

Clear(Piece piece)

IsValidPosition(Piece piece, Vector3Int position)

ClearLines()

CalculateScore(int clearedLines)

IsLineFull(int row)

LineClear(int row)

**Arrays:**

tetrominoes

**Vectors:**

Vector2Int boardSize

Vector3Int spawnPosition

Vector2Int position

Ghost: Displays a ghost piece that shows where the current piece will land if dropped. It uses a similar mechanism to the main piece but only visualizes the drop position without affecting the actual game state.

ScoreManager: Manages the player's score and updates the UI to display the current score.

Maths: Provides utility functions for mathematical operations like rounding integers.

Data: Contains static data structures defining Tetromino shapes, their rotation matrices, and wall kick offsets for rotation adjustments.

#### Boards.cs

**using** UnityEngine;

**using** UnityEngine.Tilemaps;

**using** UnityEngine.SceneManagement;

**public** **class** Board : MonoBehaviour

{

*// Reference to the tilemap component*

**public** Tilemap tilemap { **get**; **private** **set**; }

*// Reference to the currently active piece on the board*

**public** Piece activePiece { **get**; **private** **set**; }

*// Reference to the ScoreManager for updating scores*

**public** ScoreManager scoreManager;

*// Variables to track game-specific data*

**private** **int** score = 0;

**private** **int** comboCount = 0;

*// Array of TetrominoData representing different types of tetrominoes*

**public** TetrominoData[] tetrominoes;

*// Size of the game board*

**public** Vector2Int boardSize = new Vector2Int(10, 20);

*// Initial spawn position for pieces*

**public** Vector3Int spawnPosition = new Vector3Int(-1, 8, 0);

*// Property to calculate and return the current level based on the score*

**public** **int** Level

{

**get** { **return** Mathf.FloorToInt(score / 1000) + 1; }

}

*// Property to define the boundaries of the game board*

**public** RectInt Bounds

{

**get**

{

Vector2Int position = new Vector2Int(-boardSize.x / 2, -boardSize.y / 2);

**return** new RectInt(position, boardSize);

}

}

*// Awake is called when the script instance is being loaded*

**private** **void** Awake()

{

*// Getting reference to the Tilemap component*

tilemap = GetComponentInChildren<Tilemap>();

*// Getting reference to the active piece on the board*

activePiece = GetComponentInChildren<Piece>();

*// Initializing all TetrominoData objects*

**for** (**int** i = 0; i < tetrominoes.Length; i++) {

tetrominoes[i].Initialize();

}

}

*// Start is called before the first frame update*

**private** **void** Start()

{

*// Spawn the first piece when the game starts*

SpawnPiece();

}

*// Method to spawn a new piece onto the board*

**public** **void** SpawnPiece()

{

*// Randomly select a TetrominoData object*

**int** random = Random.Range(0, tetrominoes.Length);

TetrominoData data = tetrominoes[random];

*// Initialize the active piece with the selected TetrominoData*

activePiece.Initialize(**this**, spawnPosition, data);

*// If the initial position of the piece is valid, place it on the board*

**if** (IsValidPosition(activePiece, spawnPosition)) {

**Set**(activePiece);

} **else** {

*// Otherwise, end the game*

GameOver();

}

}

*// Method to handle game over condition*

**public** **void** GameOver()

{

*// Clear all tiles on the board*

tilemap.ClearAllTiles();

*// Get the currently active scene*

Scene currentScene = SceneManager.GetActiveScene();

*// Check if the game type is SinglePlayer*

**if** (currentScene.name == "Scene11 - SinglePlayer")

{

*// Save the score and load the game over scene*

PlayerPrefs.SetInt("LastScore", score);

PlayerPrefs.Save();

SceneManager.LoadScene("Scene8 - Game Over");

}

*// Check if the game type is Local Multiplayer*

**else** **if** (currentScene.name == "Scene10 - Local Multiplayer Screen")

{

*// Save player scores and load the game over scene*

FindObjectOfType<SavePlayerScores>().Save();

SceneManager.LoadScene("Scene5 - Local Multiplayer Game Over");

}

}

*// Method to set tiles on the board based on the piece's position*

**public** **void** **Set**(Piece piece)

{

**for** (**int** i = 0; i < piece.cells.Length; i++)

{

Vector3Int tilePosition = piece.cells[i] + piece.position;

tilemap.SetTile(tilePosition, piece.data.tile);

}

}

*// Method to clear tiles occupied by a piece on the board*

**public** **void** Clear(Piece piece)

{

**for** (**int** i = 0; i < piece.cells.Length; i++)

{

Vector3Int tilePosition = piece.cells[i] + piece.position;

tilemap.SetTile(tilePosition, **null**);

}

}

*// Method to check if a position is valid for placing a piece*

**public** **bool** IsValidPosition(Piece piece, Vector3Int position)

{

RectInt bounds = Bounds;

*// Check each cell of the piece*

**for** (**int** i = 0; i < piece.cells.Length; i++)

{

Vector3Int tilePosition = piece.cells[i] + position;

*// Check if the position is within the board boundaries*

**if** (!bounds.Contains((Vector2Int)tilePosition)) {

**return** **false**;

}

*// Check if the position is already occupied by another tile*

**if** (tilemap.HasTile(tilePosition)) {

**return** **false**;

}

}

**return** **true**;

}

*// Method to clear filled lines on the board*

**public** **void** ClearLines()

{

RectInt bounds = Bounds;

**int** row = bounds.yMin;

**int** clearedLines = 0;

*// Clear from bottom to top*

**while** (row < bounds.yMax)

{

*// Only clear the line if it's full*

**if** (IsLineFull(row)) {

LineClear(row);

clearedLines++;

} **else** {

row++;

}

}

*// Calculate and update the score based on the number of cleared lines*

CalculateScore(clearedLines);

}

*// Method to calculate score based on the number of cleared lines*

**private** **void** CalculateScore(**int** clearedLines)

{

**if** (clearedLines == 0) **return**;

**bool** isTetris = clearedLines == 4;

**bool** isBackToBack = clearedLines >= 1 && clearedLines <= 4;

**int** BaseScore = 0;

*// Determine base score based on the number of cleared lines*

**switch** (clearedLines)

{

**case** 1:

BaseScore = 100;

**break**;

**case** 2:

BaseScore = 300;

**break**;

**case** 3:

BaseScore = 500;

**break**;

**case** 4 :

BaseScore = 800;

**break**;

}

*// Calculate line clear score*

**int** lineClearScore = BaseScore \* Level;

*// Apply back-to-back bonus*

**if** (isBackToBack)

{

lineClearScore = Mathf.FloorToInt(lineClearScore \* 1.5f);

}

*// Apply combo bonus*

**if**(comboCount > 0)

{

**int** comboBonus = 50 \* comboCount \* Level;

lineClearScore += comboBonus;

}

*// Update the total score*

score += lineClearScore;

*// Increment combo count if lines were cleared, otherwise reset combo count*

comboCount = clearedLines > 0 ? comboCount + 1 : 0;

*// Update the score displayed using ScoreManager*

scoreManager.UpdateScore(score);

}

*// Method to check if a specific row on the board is full*

**public** **bool** IsLineFull(**int** row)

{

RectInt bounds = Bounds;

**for** (**int** col = bounds.xMin; col < bounds.xMax; col++)

{

Vector3Int position = new Vector3Int(col, row, 0);

*// If any cell in the row is empty, return false*

**if** (!tilemap.HasTile(position)) {

**return** **false**;

}

}

*// All cells in the row are filled*

**return** **true**;

}

*// Method to clear a filled line on the board*

**public** **void** LineClear(**int** row)

{

RectInt bounds = Bounds;

*// Clear all tiles in the row*

**for** (**int** col = bounds.xMin; col < bounds.xMax; col++)

{

Vector3Int position = new Vector3Int(col, row, 0);

tilemap.SetTile(position, **null**);

}

*// Shift all rows above the cleared row down by one*

**while** (row < bounds.yMax)

{

**for** (**int** col = bounds.xMin; col < bounds.xMax; col++)

{

Vector3Int position = new Vector3Int(col, row + 1, 0);

**Data.cs Techniques:**

**Classes:**

Data

**Arrays:**

RotationMatrix

Cells

WallKicksI

WallKicksJLOSTZ

**Vectors:**

Vector2Int

Vector2Int[]

Vector2Int[,]

**Dictionary:**

Cells

WallKicks

TileBase above = tilemap.GetTile(position);

position = new Vector3Int(col, row, 0);

tilemap.SetTile(position, above);

}

row++;

}

}

}

#### Data.cs

**using** UnityEngine;

*/// <summary>*

*/// Static class containing data related to tetrominoes, such as their cell positions and wall kick data.*

*/// </summary>*

**public** **static** **class** Data

{

*// Constants for rotation calculations*

**private** **static** **readonly** **float** cos = Mathf.Cos(Mathf.PI / 2f);

**private** **static** **readonly** **float** sin = Mathf.Sin(Mathf.PI / 2f);

**private** **static** **readonly** **float**[] RotationMatrix = new **float**[] { cos, sin, -sin, cos };

*// Dictionary mapping tetromino types to their respective cell positions*

**public** **static** **readonly** Dictionary<Tetromino, Vector2Int[]> Cells = new Dictionary<Tetromino, Vector2Int[]>()

{

{ Tetromino.I, new Vector2Int[] { new Vector2Int(-1, 1), new Vector2Int( 0, 1), new Vector2Int( 1, 1), new Vector2Int( 2, 1) } },

{ Tetromino.J, new Vector2Int[] { new Vector2Int(-1, 1), new Vector2Int(-1, 0), new Vector2Int( 0, 0), new Vector2Int( 1, 0) } },

{ Tetromino.L, new Vector2Int[] { new Vector2Int( 1, 1), new Vector2Int(-1, 0), new Vector2Int( 0, 0), new Vector2Int( 1, 0) } },

{ Tetromino.O, new Vector2Int[] { new Vector2Int( 0, 1), new Vector2Int( 1, 1), new Vector2Int( 0, 0), new Vector2Int( 1, 0) } },

{ Tetromino.S, new Vector2Int[] { new Vector2Int( 0, 1), new Vector2Int( 1, 1), new Vector2Int(-1, 0), new Vector2Int( 0, 0) } },

{ Tetromino.T, new Vector2Int[] { new Vector2Int( 0, 1), new Vector2Int(-1, 0), new Vector2Int( 0, 0), new Vector2Int( 1, 0) } },

{ Tetromino.Z, new Vector2Int[] { new Vector2Int(-1, 1), new Vector2Int( 0, 1), new Vector2Int( 0, 0), new Vector2Int( 1, 0) } },

};

*// Dictionary mapping tetromino types to their respective wall kick data*

**private** **static** **readonly** Vector2Int[,] WallKicksI = new Vector2Int[,] {

{ new Vector2Int(0, 0), new Vector2Int(-2, 0), new Vector2Int( 1, 0), new Vector2Int(-2,-1), new Vector2Int( 1, 2) },

{ new Vector2Int(0, 0), new Vector2Int( 2, 0), new Vector2Int(-1, 0), new Vector2Int( 2, 1), new Vector2Int(-1,-2) },

{ new Vector2Int(0, 0), new Vector2Int(-1, 0), new Vector2Int( 2, 0), new Vector2Int(-1, 2), new Vector2Int( 2,-1) },

{ new Vector2Int(0, 0), new Vector2Int( 1, 0), new Vector2Int(-2, 0), new Vector2Int( 1,-2), new Vector2Int(-2, 1) },

{ new Vector2Int(0, 0), new Vector2Int( 2, 0), new Vector2Int(-1, 0), new Vector2Int( 2, 1), new Vector2Int(-1,-2) },

{ new Vector2Int(0, 0), new Vector2Int(-2, 0), new Vector2Int( 1, 0), new Vector2Int(-2,-1), new Vector2Int( 1, 2) },

{ new Vector2Int(0, 0), new Vector2Int( 1, 0), new Vector2Int(-2, 0), new Vector2Int( 1,-2), new Vector2Int(-2, 1) },

{ new Vector2Int(0, 0), new Vector2Int(-1, 0), new Vector2Int( 2, 0), new Vector2Int(-1, 2), new Vector2Int( 2,-1) },

};

**private** **static** **readonly** Vector2Int[,] WallKicksJLOSTZ = new Vector2Int[,] {

{ new Vector2Int(0, 0), new Vector2Int(-1, 0), new Vector2Int(-1, 1), new Vector2Int(0,-2), new Vector2Int(-1,-2) },

{ new Vector2Int(0, 0), new Vector2Int( 1, 0), new Vector2Int( 1,-1), new Vector2Int(0, 2), new Vector2Int( 1, 2) },

{ new Vector2Int(0, 0), new Vector2Int( 1, 0), new Vector2Int( 1,-1), new Vector2Int(0, 2), new Vector2Int( 1, 2) },

{ new Vector2Int(0, 0), new Vector2Int(-1, 0), new Vector2Int(-1, 1), new Vector2Int(0,-2), new Vector2Int(-1,-2) },

{ new Vector2Int(0, 0), new Vector2Int( 1, 0), new Vector2Int( 1, 1), new Vector2Int(0,-2), new Vector2Int( 1,-2) },

{ new Vector2Int(0, 0), new Vector2Int(-1, 0), new Vector2Int(-1,-1), new Vector2Int(0, 2), new Vector2Int(-1, 2) },

{ new Vector2Int(0, 0), new Vector2Int(-1, 0), new Vector2Int(-1,-1), new Vector2Int(0, 2), new Vector2Int(-1, 2) },

{ new Vector2Int(0, 0), new Vector2Int( 1, 0), new Vector2Int( 1, 1), new Vector2Int(0,-2), new Vector2Int( 1,-2) },

};

*// Dictionary mapping tetromino types to their respective wall kick data*

**public** **static** **readonly** Dictionary<Tetromino, Vector2Int[,]> WallKicks = new Dictionary<Tetromino, Vector2Int[,]>()

{

{ Tetromino.I, WallKicksI },

**Ghost.cs Techniques:**

**Classes:**

Ghost

**Procedures and Functions:**

Awake()

LateUpdate()

Clear()

Copy()

Drop()

Set()

**Arrays:**

cells

**Vectors:**

Vector3Int position

{ Tetromino.J, WallKicksJLOSTZ },

{ Tetromino.L, WallKicksJLOSTZ },

{ Tetromino.O, WallKicksJLOSTZ },

{ Tetromino.S, WallKicksJLOSTZ },

{ Tetromino.T, WallKicksJLOSTZ },

{ Tetromino.Z, WallKicksJLOSTZ },

};

}

#### Ghost.cs

**using** UnityEngine;

**using** UnityEngine.Tilemaps;

**public** **class** Ghost : MonoBehaviour

{

**public** Tile tile; *// The tile to represent the ghost piece*

**public** Board mainBoard; *// Reference to the main game board*

**public** Piece trackingPiece; *// The piece that the ghost follows*

**public** Tilemap tilemap { **get**; **private** **set**; } *// The tilemap component attached to this GameObject*

**public** Vector3Int[] cells { **get**; **private** **set**; } *// Array to hold the cell positions of the ghost piece*

**public** Vector3Int position { **get**; **private** **set**; } *// Current position of the ghost piece*

**private** **void** Awake()

{

tilemap = GetComponentInChildren<Tilemap>(); *// Get the Tilemap component attached to this GameObject*

cells = new Vector3Int[4]; *// Initialize the array to hold cell positions*

}

**private** **void** LateUpdate()

{

Clear(); *// Clear the previous ghost piece*

Copy(); *// Copy the cell positions from the tracking piece*

Drop(); *// Calculate the position where the ghost piece should drop*

**Set**(); *// Set the ghost piece on the tilemap*

}

**private** **void** Clear()

{

*// Clear previous ghost piece from the tilemap*

**for** (**int** i = 0; i < cells.Length; i++)

{

Vector3Int tilePosition = cells[i] + position;

tilemap.SetTile(tilePosition, **null**);

}

}

**private** **void** Copy()

{

*// Copy cell positions from the tracking piece*

**for** (**int** i = 0; i < cells.Length; i++)

{

cells[i] = trackingPiece.cells[i];

}

}

**private** **void** Drop()

{

Vector3Int position = trackingPiece.position;

**int** current = position.y;

**int** bottom = -mainBoard.boardSize.y / 2 - 1;

mainBoard.Clear(trackingPiece);

*// Calculate the position where the ghost piece should drop*

**for** (**int** row = current; row >= bottom; row--)

{

position.y = row;

**if** (mainBoard.IsValidPosition(trackingPiece, position))

{

**this**.position = position;

}

**else**

{

**break**;

}

}

mainBoard.**Set**(trackingPiece);

}

**private** **void** **Set**()

{

*// Set the ghost piece on the tilemap*

**for** (**int** i = 0; i < cells.Length; i++)

{

Vector3Int tilePosition = cells[i] + position;

tilemap.SetTile(tilePosition, tile);

}

}

}

#### Math.cs

**Math.cs Techniques:**

**Procedures and Functions:**

CeilToInt(float value)

FloorToInt(float value)

RoundToInt(float value)

**using** UnityEngine;

**public** **class** Maths : MonoBehaviour

{

*/// <summary>*

*/// Rounds the specified float value up to the nearest integer.*

*/// </summary>*

*/// <param name="value">The float value to round.</param>*

*/// <returns>The rounded integer value.</returns>*

**public** **static** **int** CeilToInt(**float** **value**)

{

**int** intValue = (**int**)**value**;

**return** **value** > intValue ? intValue + 1 : intValue;

}

*/// <summary>*

*/// Rounds the specified float value down to the nearest integer.*

*/// </summary>*

*/// <param name="value">The float value to round.</param>*

*/// <returns>The rounded integer value.</returns>*

**public** **static** **int** FloorToInt(**float** **value**)

{

**int** intValue = (**int**)**value**;

**return** **value** < 0 ? (**value** - intValue < 0 ? intValue - 1 : intValue) : intValue;

}

*/// <summary>*

*/// Rounds the specified float value to the nearest integer.*

*/// </summary>*

*/// <param name="value">The float value to round.</param>*

*/// <returns>The rounded integer value.</returns>*

**public** **static** **int** RoundToInt(**float** **value**)

{

**int** intValue = (**int**)**value**;

**float** fractionalPart = **value** - intValue;

**if** (fractionalPart >= 0.5f)

{

**return** intValue + 1;

}

**else** **if** (fractionalPart <= -0.5f)

{

**return** intValue - 1;

}

**else**

{

**return** intValue;

}

}

}

#### Piece.cs

**using** UnityEngine;

**using** UnityEngine.SceneManagement;

**Piece.cs Techniques:**

**Classes:**

Piece

**Procedures and Functions:**

Initialize(Board board, Vector3Int position, TetrominoData data)

Update()

HandleMoveInputs()

Step()

HardDrop()

Lock()

Move(Vector2Int translation)

Rotate(int direction)

ApplyRotationMatrix(int direction)

TestWallKicks(int rotationIndex, int rotationDirection)

GetWallKickIndex(int rotationIndex, int rotationDirection)

Wrap(int input, int min, int max)

**Arrays:**

cells

**Vectors:**

Vector3Int position

**public** **class** Piece : MonoBehaviour

{

*// Reference to the game board*

**public** Board board { **get**; **private** **set**; }

*// Data of the tetromino piece*

**public** TetrominoData data { **get**; **private** **set**; }

*// Array of cells representing the piece's shape*

**public** Vector3Int[] cells { **get**; **private** **set**; }

*// Current position of the piece on the game board*

**public** Vector3Int position { **get**; **private** **set**; }

*// Index representing the current rotation state of the piece*

**public** **int** rotationIndex { **get**; **private** **set**; }

*// Delay before the piece automatically moves down*

**public** **float** stepDelay = 1f;

*// Delay between lateral movements of the piece*

**public** **float** moveDelay = 0.1f;

*// Delay before the piece locks in place*

**public** **float** lockDelay = 0.5f;

*// Timer to control the automatic downward movement of the piece*

**private** **float** stepTime;

*// Timer to control lateral movement of the piece*

**private** **float** moveTime;

*// Timer to control locking of the piece*

**private** **float** lockTime;

*// Key bindings for controlling the piece's movement and rotation*

**public** KeyCode moveLeft = KeyCode.LeftArrow;

**public** KeyCode moveRight = KeyCode.RightArrow;

**public** KeyCode moveDown = KeyCode.DownArrow;

**public** KeyCode rotateClockwise = KeyCode.PageDown;

**public** KeyCode rotateCounterClockwise = KeyCode.UpArrow;

**public** KeyCode hardDrop = KeyCode.Space;

*// Reference to the score manager for updating score*

**public** ScoreManager scoremanager;

*// Initializes the piece with the specified board, position, and tetromino data*

**public** **void** Initialize(Board board, Vector3Int position, TetrominoData data)

{

**this**.data = data;

**this**.board = board;

**this**.position = position;

rotationIndex = 0;

stepTime = Time.time + stepDelay;

moveTime = Time.time + moveDelay;

lockTime = 0f;

*// Initialize cells array*

**if** (cells == **null**)

{

cells = new Vector3Int[data.cells.Length];

}

*// Copy cell positions from tetromino data*

**for** (**int** i = 0; i < cells.Length; i++)

{

cells[i] = (Vector3Int)data.cells[i];

}

}

*// Update is called once per frame*

**private** **void** Update()

{

*// Clear the piece from the board before processing movement*

board.Clear(**this**);

*// Increment lock time*

lockTime += Time.deltaTime;

*// Handle rotation input*

**if** (Input.GetKeyDown(rotateCounterClockwise))

{

Rotate(-1);

}

**else** **if** (Input.GetKeyDown(rotateClockwise))

{

Rotate(1);

}

*// Handle hard drop input*

**if** (Input.GetKeyDown(hardDrop))

{

HardDrop();

}

*// Process lateral movement input if enough time has passed since last move*

**if** (Time.time > moveTime)

{

HandleMoveInputs();

}

*// Move the piece downward automatically at regular intervals*

**if** (Time.time > stepTime)

{

Step();

}

*// Update the board with the current piece position and shape*

board.**Set**(**this**);

}

*// Handles lateral movement input*

**private** **void** HandleMoveInputs()

{

*// Process soft drop movement*

**if** (Input.GetKey(moveDown))

{

**if** (Move(Vector2Int.down))

{

*// Update the step time to prevent double movement*

stepTime = Time.time + stepDelay;

}

*// Update score for soft drop*

scoremanager.UpdateScore(1);

}

*// Process left/right movement*

**if** (Input.GetKey(moveLeft))

{

Move(Vector2Int.left);

}

**else** **if** (Input.GetKey(moveRight))

{

Move(Vector2Int.right);

}

}

*// Move the piece downward*

**private** **void** Step()

{

stepTime = Time.time + stepDelay;

*// Attempt to move the piece down*

Move(Vector2Int.down);

*// Lock the piece if it has been inactive for too long*

**if** (lockTime >= lockDelay)

{

**Lock**();

}

}

*// Performs hard drop, dropping the piece to the lowest possible position instantly*

**private** **void** HardDrop()

{

**while** (Move(Vector2Int.down))

{

**continue**;

}

*// Update score for hard drop*

scoremanager.UpdateScore(2);

**Lock**();

}

*// Locks the piece in place on the board*

**private** **void** **Lock**()

{

*// Set the piece on the board*

board.**Set**(**this**);

*// Clear filled lines and spawn a new piece*

board.ClearLines();

board.SpawnPiece();

}

*// Attempt to move the piece in the specified direction*

**private** **bool** Move(Vector2Int translation)

{

*// Calculate the new position after movement*

Vector3Int newPosition = position + translation;

*// Check if the new position is valid*

**bool** valid = board.IsValidPosition(**this**, newPosition);

*// If the new position is valid, update the piece's position and reset move time*

**if** (valid)

{

position = newPosition;

moveTime = Time.time + moveDelay;

lockTime = 0f; *// reset lock time*

}

**return** valid;

}

*// Rotate the piece clockwise or counterclockwise based on the direction*

**private** **void** Rotate(**int** direction)

{

*// Store the current rotation index for potential reversion*

**int** originalRotation = rotationIndex;

*// Rotate the piece's cells*

rotationIndex = Wrap(rotationIndex + direction, 0, 4);

ApplyRotationMatrix(direction);

*// Revert the rotation if it results in an invalid position*

**if** (!TestWallKicks(rotationIndex, direction))

{

rotationIndex = originalRotation;

ApplyRotationMatrix(-direction);

}

}

*// Apply rotation matrix to the piece's cells*

**private** **void** ApplyRotationMatrix(**int** direction)

{

**float**[] matrix = Data.RotationMatrix;

*// Rotate each cell using the rotation matrix*

**for** (**int** i = 0; i < cells.Length; i++)

{

Vector3 cell = cells[i];

**int** x, y;

**switch** (data.tetromino)

{

**case** Tetromino.I:

**case** Tetromino.O:

cell.x -= 0.5f;

cell.y -= 0.5f;

x = Maths.CeilToInt((**float**)(cell.x \* matrix[0] \* direction) + (**float**)(cell.y \* matrix[1] \* direction));

y = Maths.CeilToInt((**float**)(cell.x \* matrix[2] \* direction) + (**float**)(cell.y \* matrix[3] \* direction));

**break**;

**default**:

x = Maths.RoundToInt((**float**)(cell.x \* matrix[0] \* direction) + (**float**)(cell.y \* matrix[1] \* direction));

y = Maths.RoundToInt((**float**)(cell.x \* matrix[2] \* direction) + (**float**)(cell.y \* matrix[3] \* direction));

**break**;

}

cells[i] = new Vector3Int(x, y, 0);

}

}

*// Test wall kicks to resolve collision after rotation*

**private** **bool** TestWallKicks(**int** rotationIndex, **int** rotationDirection)

{

**int** wallKickIndex = GetWallKickIndex(rotationIndex, rotationDirection);

*// Test each wall kick translation*

**for** (**int** i = 0; i < data.wallKicks.GetLength(1); i++)

{

Vector2Int translation = data.wallKicks[wallKickIndex, i];

*// If successful, move the piece and return true*

**if** (Move(translation))

{

**return** **true**;

}

}

**return** **false**;

}

*// Get the appropriate wall kick index based on rotation index and direction*

**private** **int** GetWallKickIndex(**int** rotationIndex, **int** rotationDirection)

{

**int** wallKickIndex = rotationIndex \* 2;

**if** (rotationDirection < 0)

{

wallKickIndex--;

}

**return** Wrap(wallKickIndex, 0, data.wallKicks.GetLength(0));

}

*// Helper function to wrap a value between a minimum and maximum*

**private** **int** Wrap(**int** input, **int** min, **int** max)

{

**if** (input < min)

{

**return** max - (min - input) % (max - min);

}

**else**

{

**return** min + (input - min) % (max - min);

}

}

}

#### ScoreManger.cs

**ScoreManager.cs Techniques:**

**Classes:**

ScoreManager

**Procedures and Functions:**

UpdateScore(int newScore)

**using** System.Collections;

**using** System.Collections.Generic;

**using** UnityEngine;

**using** TMPro;

**public** **class** ScoreManager : MonoBehaviour

{

*// Reference to the TextMeshProUGUI component for displaying the score*

**public** TextMeshProUGUI scoreText;

*// Variable to hold the current score*

**private** **long** score = 0;

*// Method to update the score with a new value and display it*

**public** **void** UpdateScore(**int** newScore)

{

*// Add the new score to the current score*

score = score + newScore;

*// Update the score displayed in the UI*

UpdateScoreUI();

}

*// Method to update the score displayed in the UI*

**private** **void** UpdateScoreUI()

{

*// Check if the scoreText reference is assigned*

**if** (scoreText != **null**)

{

*// Update the text of the scoreText component to show the current score*

scoreText.text = score.ToString();

}

}

}

#### Tetromino.cs

**using** UnityEngine;

**using** UnityEngine.Tilemaps;

*// Enumeration representing different types of Tetromino pieces*

**public** **enum** Tetromino

**Tetromino.cs Techniques:**

**Enum:**

Tetromino

**Struct:**

TetrominoData

**Methods:**

Initialize()

**Arrays:**

cells

wallKicks

**Vectors:**

Vector2Int arrays (used within the struct)

{

I, *// Straight Tetromino*

J, *// J Tetromino*

L, *// L Tetromino*

O, *// Square Tetromino*

S, *// S Tetromino*

T, *// T Tetromino*

Z *// Z Tetromino*

}

*// Struct containing data for a Tetromino piece*

[System.Serializable]

**public** **struct** TetrominoData

{

**public** Tile tile; *// Tile representing the Tetromino piece*

**public** Tetromino tetromino; *// Type of Tetromino*

*// Array of relative cell positions composing the Tetromino*

**public** Vector2Int[] cells { **get**; **private** **set**; }

*// Array representing possible wall kick translations for Tetromino rotation*

**public** Vector2Int[,] wallKicks { **get**; **private** **set**; }

*// Method to initialize the Tetromino data*

**public** **void** Initialize()

{

*// Retrieve cell positions and wall kicks from a central data source*

cells = Data.Cells[tetromino];

wallKicks = Data.WallKicks[tetromino];

}

}

### Local Multiplayer

LoadPlayerScores: is responsible for loading saved scores from PlayerPrefs, updating the UI with the loaded scores, and determining the winner or if it's a tie based on the scores.

SavePlayerScores: is responsible for saving player scores entered by the players to PlayerPrefs.

#### LocalPlayerScore.cs

**using** System.Collections;

**LoadPlayerScores.cs Techniques:**

**Procedures and Functions:**

Start()

**using** System.Collections.Generic;

**using** UnityEngine;

**using** TMPro;

**public** **class** LoadPlayerScores : MonoBehaviour

{

*// References to TextMeshProUGUI objects for displaying scores and tagline*

**public** TextMeshProUGUI Score1;

**public** TextMeshProUGUI Score2;

**public** TextMeshProUGUI Tagline;

**void** Start()

{

*// Retrieve the saved scores for Player 1 and Player 2 from PlayerPrefs*

**string** Player1 = PlayerPrefs.GetString("Player1Score");

**int** Player1Value = **int**.Parse(Player1);

*// Log Player 1 score to console (for debugging)*

Debug.Log("Player 1 Score:" + Player1);

*// Update TextMeshProUGUI objects with loaded Player 1 score*

Score1.text = Player1;

*// Retrieve and parse the saved score for Player 2*

**string** Player2 = PlayerPrefs.GetString("Player2Score");

**int** Player2Value = **int**.Parse(Player2);

*// Log Player 2 score to console (for debugging)*

Debug.Log("Player 2 Score:" + Player2);

*// Update TextMeshProUGUI objects with loaded Player 2 score*

Score2.text = Player2;

*// Determine the winner or if it's a tie and update the tagline accordingly*

**if** (Player1Value > Player2Value)

{

Tagline.text = "PLAYER 1 IS THE WINNER!";

}

**else** **if** (Player1Value == Player2Value)

{

Tagline.text = "IT'S A TIE";

}

**else**

{

Tagline.text = "PLAYER 2 IS THE WINNER!";

}

}

}

#### SavePlayerScores.cs

**SavePlayerScores.cs Techniques:**

**Classes:**

SavePlayerScores

**Procedures and Functions:**

Save()

**using** System.Collections;

**using** System.Collections.Generic;

**using** TMPro;

**using** UnityEngine;

**public** **class** SavePlayerScores : MonoBehaviour

{

*// References to TextMeshProUGUI objects for Player 1 and Player 2 scores*

**public** TextMeshProUGUI Score1;

**public** TextMeshProUGUI Score2;

*// Method to save player scores to PlayerPrefs*

**public** **void** Save()

{

*// Get the value (score) from the TextMeshProUGUI objects*

**string** Player1 = Score1.text;

**string** Player2 = Score2.text;

*// Save the scores to PlayerPrefs using specific keys*

PlayerPrefs.SetString("Player1Score", Player1);

PlayerPrefs.SetString("Player2Score", Player2);

*// Save PlayerPrefs*

PlayerPrefs.Save();

}

}

### Program UI

Backbutton.cs: Defines a script for handling a back button in Unity. It contains a method backButton() that loads the previous scene specified in the previousScene variable using SceneManager.LoadScene().

EducationSceneUI.cs: Manages UI interactions for an education scene. It contains methods to load specific education scenes (Openers(), Methods(), etc.), as well as a method TheBasicsButton() to handle clicks on "The Basics" scene button. It saves a boolean flag indicating whether "The Basics" scene was clicked using PlayerPrefs and then loads the scene accordingly.

EventManager.cs: Manages events and scene transitions based on button clicks. It contains an array to store scene names as options and a method Button(int index) to handle button clicks and load the corresponding scene using SceneManager.LoadScene().

GameMenuUI.cs: Handles UI interactions for the game menu. It contains methods to load different game scenes (SinglePlayerButton(), MultiplayerButton(), etc.) and a method EducationModeButton() to determine whether to load a specific education screen or the full education menu based on a boolean flag stored in PlayerPrefs.

GameOverUI.cs: Manages UI elements for the game over screen. It retrieves the final score from PlayerPrefs and displays it in a UI text element. It also contains methods to return to the game menu scene or reset the game by reloading the single-player scene.

LMGameOver.cs: Handles game over actions in a local multiplayer game. It contains methods to return to the main menu scene or reset the game by reloading the local multiplayer screen.

MultiplayerMenu.cs: Manages UI interactions for the multiplayer menu. It contains methods to initiate hosting or joining a lobby, load the local multiplayer screen, and navigate back to the main game menu.

PauseMenu.cs: Controls the pause menu functionality. It toggles the pause menu UI and pauses/resumes the game when the "Escape" key is pressed. It also contains methods to go back to the game menu scene or quit the game.

StartMenuUI.cs: Handles UI interactions for the start menu. It contains methods to load the login screen, register screen, and quit the game.

#### Backbutton.cs

**Backbutton.cs Techniques:**

**Classes:**

Backbutton

**Procedures and Functions:**

backButton()

**using** System.Collections;

**using** System.Collections.Generic;

**using** UnityEngine;

**using** UnityEngine.SceneManagement;

**public** **class** Backbutton : MonoBehaviour

{

[SerializeField] **public** **string** previousScene; *// Serialized field to specify the previous scene to load*

*// Method called when the back button is pressed*

**public** **void** backButton()

{

*// Load the previous scene specified in the 'previousScene' variable*

SceneManager.LoadScene(previousScene);

}

}

#### EducationSceneUI.cs

**using** UnityEngine;

**EducationSceneUI.cs Techniques:**

**Classes:**

EducationSceneUI

**Procedures and Functions:**

BackButton()

Openers()

Methods()

Mid\_Game()

Perf\_Clear()

SRS()

Stacking()

Vocab()

TheBasicsButton()

**using** UnityEngine.SceneManagement;

**public** **class** EducationSceneUI : MonoBehaviour

{

*// Reference to the canvas containing arrows for navigation*

**public** Canvas Arrows;

*// Method to go back to the main game menu scene*

**public** **void** BackButton()

{

SceneManager.LoadScene("Scene4 - Game Menu");

}

*// Methods to load specific education scenes*

**public** **void** Openers()

{

SceneManager.LoadScene("Scene15 - Openers");

}

**public** **void** Methods()

{

SceneManager.LoadScene("Scene17 - Methods");

}

**public** **void** Mid\_Game()

{

SceneManager.LoadScene("Scene18 - Mid Game");

}

**public** **void** Perf\_Clear()

{

SceneManager.LoadScene("Scene19 - Perfect Clears");

}

**public** **void** SRS()

{

SceneManager.LoadScene("Scene20 - SRS");

}

**public** **void** Stacking()

{

SceneManager.LoadScene("Scene21 - Stacking");

}

**public** **void** Vocab()

{

SceneManager.LoadScene("Scene22 - Vocabulary");

}

*// Method to handle button click for "The Basics" scene*

**public** **void** TheBasicsButton()

{

*// Set a boolean flag indicating that "The Basics" scene was clicked*

**bool** TheBasicsClicked = **true**;

*// Destroy the canvas containing arrows (if it exists)*

Destroy(Arrows);

*// Save the boolean flag using PlayerPrefs*

PlayerPrefs.SetInt("Basics", TheBasicsClicked ? 1 : 0);

*// Load the "The Basics" scene*

SceneManager.LoadScene("Scene13 - The Basics");

*// Save PlayerPrefs*

PlayerPrefs.Save();

}

}

#### EventManager.cs

**using** System.Collections;

**using** System.Collections.Generic;

**using** UnityEngine;

**using** UnityEngine.SceneManagement;

**EventManager.cs Techniques:**

**Arrays:**

options array

**Procedures and Functions:**

Button(int index)

**Exception handling:**

Error logging using Debug.LogError()

**public** **class** EventManager : MonoBehaviour

{

*// An array to store scene names as options*

[SerializeField] **public** **string**[] options;

*// Method to handle button clicks, takes index of the selected option*

**public** **void** Button(**int** index)

{

*// Check if the index is within the bounds of the options array*

**if** (index >= 0 && index < options.Length)

{

*// Load the scene corresponding to the selected index*

SceneManager.LoadScene(options[index]);

}

**else**

{

*// Log an error if the index is out of range*

Debug.LogError("Index out of range!");

}

}

}

#### GameMenuUI.cs

**using** System.Collections;

**GameMenuUI.cs Techniques: Classes:**

GameMenuUI

**Procedures and Functions:**

SinglePlayerButton()

MultiplayerButton()

EducationModeButton()

vsComputerButton()

ExitButton()

**using** System.Collections.Generic;

**using** UnityEngine;

**using** UnityEngine.SceneManagement;

**public** **class** GameMenuUI : MonoBehaviour

{

*// Method to load the single-player game scene*

**public** **void** SinglePlayerButton()

{

SceneManager.LoadScene("Scene11 - SinglePlayer");

}

*// Method to load the multiplayer lobby scene*

**public** **void** MultiplayerButton()

{

SceneManager.LoadScene("Scene9 - Multiplayer Lobby");

}

*// Method to handle the education mode button*

**public** **void** EducationModeButton ()

{

*// Check if the "Basics" key is set in PlayerPrefs*

**int** whichScene = PlayerPrefs.GetInt("Basics");

**if** (whichScene == 1)

{

*// Load the education screen if the value is 1*

SceneManager.LoadScene("Scene12 - Education Screen");

}

**else**

{

*// Load the full education menu if the value is not 1*

SceneManager.LoadScene("Scene16 - Education Full Menu");

}

}

*// Method to handle the vsComputerButton*

**public** **void** vsComputerButton()

{

*// Load the scene with index 0*

SceneManager.LoadScene(0);

}

*// Method to handle the exit button, quits the application*

**public** **void** ExitButton()

{

Application.Quit();

}

}

#### GameOverUI.cs

**using** UnityEngine;

**using** UnityEngine.SceneManagement;

**using** TMPro;

**GameOverUI.cs Techniques:**

**Classes:**

GameOverUI

**Procedures and Functions:**

Start()

ReturnToMenu()

ResetGame()

**public** **class** GameOverUI : MonoBehaviour

{

*// Reference to the UI text element displaying the final score*

**public** TextMeshProUGUI FinalScore;

**void** Start()

{

*// Retrieve the score from PlayerPrefs with a default value of 0*

**int** lastScore = PlayerPrefs.GetInt("LastScore", 0);

*// Display the score in a UI text element*

Debug.Log("Score:" + lastScore); *// Log the score to console (for debugging)*

FinalScore.text = "" + lastScore; *// Update the UI text element with the score*

}

*// Method to return to the game menu scene*

**public** **void** ReturnToMenu()

{

SceneManager.LoadScene("Scene4 - Game Menu");

}

*// Method to reset the game by reloading the single player scene*

**public** **void** ResetGame()

{

SceneManager.LoadScene("Scene11 - SinglePlayer");

}

}

#### LMGameOver.cs

**using** System.Collections;

**using** System.Collections.Generic;

**LMGameOver.cs Techniques:**

**Classes:**

LMGameOver

**Procedures and Functions:**

ReturnToMenu()

ResetGame()

**using** UnityEngine;

**using** UnityEngine.SceneManagement;

**public** **class** LMGameOver : MonoBehaviour

{

*// Method to return to the main menu scene*

**public** **void** ReturnToMenu()

{

*// Load the "Scene4 - Game Menu" scene*

SceneManager.LoadScene("Scene4 - Game Menu");

}

*// Method to reset the game (reload local multiplayer screen)*

**public** **void** ResetGame()

{

*// Load the "Scene10 - Local Multiplayer Screen" scene*

SceneManager.LoadScene("Scene10 - Local Multiplayer Screen");

}

}

#### MultiplayerMenu.cs

**using** System.Collections;

**MultiplayerMenu.cs Techniques: Classes:**

MultiplayerMenuUI

**Procedures and Functions:**

HostLobby()

JoinLobby()

LoadLocal()

Back()

**using** System.Collections.Generic;

**using** UnityEngine;

**using** UnityEngine.SceneManagement;

**public** **class** MultiplayerMenuUI : MonoBehaviour

{

*// Method to initiate hosting a lobby*

**public** **void** HostLobby()

{

*// Add logic here to start hosting a multiplayer lobby*

}

*// Method to initiate joining a lobby*

**public** **void** JoinLobby()

{

*// Add logic here to join an existing multiplayer lobby*

}

*// Method to load the local multiplayer screen*

**public** **void** LoadLocal()

{

*// Load the scene for local multiplayer gameplay*

SceneManager.LoadScene("Scene10 - Local Multiplayer Screen");

}

*// Method to navigate back to the main game menu*

**public** **void** Back()

{

*// Load the main game menu scene*

SceneManager.LoadScene("Scene4 - Game Menu");

}

}

#### PauseMenu.cs

**using** System.Collections;

**using** System.Collections.Generic;

**using** UnityEngine;

**using** UnityEngine.SceneManagement;

**public** **class** PauseMenuUI : MonoBehaviour

{

**PauseMenu.cs Techniques:**

**Classes:**

PauseMenuUI

**Procedures and Functions:**

Start()

Update()

PauseGame()

ResumeGame()

GoToGameMenu()

QuitGame()

*// Reference to the pause menu GameObject*

**public** GameObject pauseMenu;

*// Variable to track whether the game is paused*

**public** **static** **bool** isPaused;

*// Start is called before the first frame update*

**void** Start()

{

*// Deactivate the pause menu at the start of the game*

pauseMenu.SetActive(**false**);

}

*// Update is called once per frame*

**void** Update()

{

*// Check for the "Escape" key press to toggle pause*

**if** (Input.GetKeyDown(KeyCode.Escape))

{

**if** (isPaused)

{

*// If game is already paused, resume it*

ResumeGame();

}

**else**

{

*// If game is not paused, pause it*

PauseGame();

}

}

}

*// Method to pause the game*

**public** **void** PauseGame()

{

*// Activate the pause menu UI*

pauseMenu.SetActive(**true**);

*// Stop time in the game, effectively pausing it*

Time.timeScale = 0f;

*// Update the isPaused flag*

isPaused = **true**;

}

*// Method to resume the game*

**public** **void** ResumeGame()

{

*// Deactivate the pause menu UI*

pauseMenu.SetActive(**false**);

*// Resume time in the game*

Time.timeScale = 1f;

*// Update the isPaused flag*

isPaused = **false**;

}

*// Method to go back to the game menu scene*

**public** **void** GoToGameMenu()

{

*// Load the game menu scene*

SceneManager.LoadScene("Scene4 - Game Menu");

*// Ensure time resumes when returning to the menu*

Time.timeScale = 1f;

}

*// Method to quit the game*

**public** **void** QuitGame()

{

*// Quit the application*

Application.Quit();

}

}

#### StartMenuUI.cs

**using** System.Collections;

**using** System.Collections.Generic;

**StartMenuUI.cs Techniques:**

**Classes:**

StartMenuUI

**Procedures and Functions:**

LoginButton()

RegisterButton()

QuitGame()

**using** UnityEngine;

**using** UnityEngine.SceneManagement;

**public** **class** StartMenuUI : MonoBehaviour

{

*// Method to handle the login button click event*

**public** **void** LoginButton()

{

*// Load the login screen scene*

SceneManager.LoadScene("Scene2 - Login Screen");

}

*// Method to handle the register button click event*

**public** **void** RegisterButton()

{

*// Load the register scene*

SceneManager.LoadScene("Scene3 - Register Scene");

}

*// Method to handle quitting the game*

**public** **void** QuitGame()

{

*// Quit the application*

Application.Quit();

}

}

### Registration and Login

LoginUI.cs: Manages the user interface for logging in, interacts with the database to validate user credentials, and loads the game menu upon successful authentication.

MD5Script.cs: Implements a method to generate MD5 hashes for passwords, used for hashing user passwords before storing them in the database.

RegisterUI.cs: Handles user registration, including password hashing and database entry creation, along with reading entries from the user database.

RegistrationManager.cs: Validates user input for registration, including email format, username format, password strength, and password confirmation, enabling registration button interaction accordingly.

SqliteHelper.cs: Provides a helper class for SQLite database operations, including connection management, data retrieval, deletion, and table creation.

UserDb.cs: Defines methods to interact with the user database, such as adding user data, retrieving data by username, deleting data, and fetching the latest timestamp entry.

UserDbTest.cs: Facilitates reading from the user database, adding new user entries with error handling for duplicate usernames, and checking for existing usernames in the database.

UserEntity.cs: Represents a user entity with properties for username, hashed password, email, and creation date, along with a method to generate a fake user for testing purposes.

#### LoginUI.cs

**using** System.Collections;

**using** System.Collections.Generic;

**using** DataBank;

**using** TMPro;

**using** UnityEngine;

**using** UnityEngine.SceneManagement;

**LoginUI.cs Techniques:**

**Classes:**

LoginUI

MD5Script

UserDbTest

**Procedures and Functions:**

CheckDataBase()

LoadInGM()

**public** **class** LoginUI : MonoBehaviour

{

*// Reference to the MD5Script component for password hashing*

**public** MD5Script mD5Script;

*// Reference to the UserDbTest component for database operations*

**public** UserDbTest userDbTest;

*// Reference to the input field for username*

**public** TMP\_InputField userInputField;

*// Reference to the input field for password*

**public** TMP\_InputField passwordField;

*// Reference to the text element for error messages*

**public** TMP\_Text errorMessage;

*// Method to check the user credentials in the database*

**public** **void** CheckDataBase()

{

*// Get the username input from the user*

**string** username = userInputField.text;

*// Create an instance of UserDb to interact with the database*

UserDb userDb = new UserDb();

*// Retrieve user data from the database based on the provided username*

System.Data.IDataReader reader = userDb.getDataByString(username);

*// Check if data is retrieved from the database*

**if** (reader != **null**)

{

*// Check if there is data to read*

**if** (reader.Read())

{

*// Retrieve the hashed password stored in the database*

**string** hashedPasswordFromDb = reader[1].ToString();

*// Hash the password provided by the user*

**string** hashedPasswordProvidedByUser = mD5Script.GenerateMD5Hash();

*// Compare the hashed passwords*

**if** (hashedPasswordFromDb.Equals(hashedPasswordProvidedByUser))

{

*// Passwords match, user authentication successful*

Debug.Log("User authenticated successfully");

LoadInGM(); *// Load the game menu scene*

}

**else**

{

*// Passwords don't match, authentication failed*

errorMessage.text = "Incorrect password";

}

}

**else**

{

*// User not found in the database*

errorMessage.text = "User does not exist";

}

}

}

*// Method to load the game menu scene*

**public** **void** LoadInGM()

{

SceneManager.LoadScene("Scene4 - Game Menu");

}

}

#### MD5Script.cs

**using** System;

**using** System.Text;

**using** UnityEngine;

**using** TMPro;

**MD5Script.cs Techniques:**

**Classes:**

MD5Script

**Procedures and Functions:**

GenerateMD5Hash()

CalculateMD5Hash(string input)

RotateLeft(uint x, int n)

**Arrays:**

inputBytes

paddedBytes

w

hash

tempBytes

**public** **class** MD5Script : MonoBehaviour

{

*// Reference to the input field where the password is entered*

**public** TMP\_InputField inputField;

*// Method to generate MD5 hash based on the input password*

**public** **string** GenerateMD5Hash()

{

**string** inputText = inputField.text;

*// Check if input text is empty*

**if** (**string**.IsNullOrEmpty(inputText))

{

Debug.LogError("Input field is empty!");

**return** **null**;

}

*// Calculate MD5 hash based on the input text*

**string** md5Hash = CalculateMD5Hash(inputText);

**return** md5Hash;

}

**private** **string** CalculateMD5Hash(**string** input)

{

*// Step 1: Convert input string to byte array*

**byte**[] inputBytes = Encoding.UTF8.GetBytes(input);

*// Step 2: Initialize MD5 variables*

**uint**[] r = new **uint**[] {

0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476

};

**uint**[] k = new **uint**[] {

*// Constants defined for MD5 algorithm*

*// These constants are used in the hash calculation process*

0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee,

0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501,

0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be,

0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821,

0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa,

0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8,

0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed,

0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a,

0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c,

0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70,

0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05,

0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665,

0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039,

0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1,

0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1,

0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391

};

*// Step 3: Append padding to the input byte array*

*// (Padding is required as per MD5 specification)*

**int** initialLength = inputBytes.Length;

**int** paddedLength = (initialLength + 8 + 63) / 64 \* 64;

**byte**[] paddedBytes = new **byte**[paddedLength];

**for** (**int** i = 0; i < initialLength; i++)

{

paddedBytes[i] = inputBytes[i];

}

paddedBytes[initialLength] = 0x80;

**ulong** bitLength = (**ulong**)initialLength \* 8;

**for** (**int** i = 0; i < 8; i++)

{

paddedBytes[paddedLength - 8 + i] = (**byte**)(bitLength >> (8 \* i));

}

*// Step 4: Process message in 512-bit chunks*

**for** (**int** chunkStart = 0; chunkStart < paddedLength; chunkStart += 64)

{

**uint**[] w = new **uint**[16];

**for** (**int** i = 0; i < 16; i++)

{

w[i] = (**uint**)(

(paddedBytes[chunkStart + i \* 4 + 0] << 0) |

(paddedBytes[chunkStart + i \* 4 + 1] << 8) |

(paddedBytes[chunkStart + i \* 4 + 2] << 16) |

(paddedBytes[chunkStart + i \* 4 + 3] << 24)

);

}

**uint**[] hash = (**uint**[])r.Clone();

**for** (**int** i = 0; i < 64; i++)

{

**uint** f, g;

**if** (i < 16)

{

f = (hash[1] & hash[2]) | (~hash[1] & hash[3]);

g = (**uint**)i;

}

**else** **if** (i < 32)

{

f = (hash[3] & hash[1]) | (~hash[3] & hash[2]);

g = (**uint**)((5 \* i + 1) % 16);

}

**else** **if** (i < 48)

{

f = hash[1] ^ hash[2] ^ hash[3];

g = (**uint**)((3 \* i + 5) % 16);

}

**else**

{

f = hash[2] ^ (hash[1] | ~hash[3]);

g = (**uint**)((7 \* i) % 16);

}

**uint** temp = hash[3];

hash[3] = hash[2];

hash[2] = hash[1];

hash[1] = hash[1] + RotateLeft((hash[0] + f + k[i] + w[g]), (**int**)i);

hash[0] = temp;

}

**for** (**int** i = 0; i < 4; i++)

{

r[i] += hash[i];

}

}

*// Step 5: Format hash as string*

StringBuilder sb = new StringBuilder();

**for** (**int** i = 0; i < 4; i++)

{

**byte**[] tempBytes = BitConverter.GetBytes(r[i]);

**for** (**int** j = 0; j < 4; j++)

{

sb.Append(tempBytes[j].ToString("x2"));

}

}

**return** sb.ToString();

}

*// Method to perform circular left shift operation on a 32-bit integer*

**private** **uint** RotateLeft(**uint** x, **int** n)

{

**return** (x << n) | (x >> (32 - n));

}

}

#### RegisterUI.cs

**using** TMPro;

**using** UnityEngine;

**using** UnityEngine.SceneManagement;

**RegisterUI.cs Techniques:**

**Classes:**

RegisterUI

**Procedures and Functions:**

RegisterButton()

**public** **class** RegisterUI : MonoBehaviour

{

*// Reference to the MD5Script component for password hashing*

**public** MD5Script md5Script;

*// Reference to the UserDbTest component for database operations*

**public** UserDbTest userDbTest;

*// Input field for username*

**public** TMP\_InputField userInputField;

*// Input field for password*

**public** TMP\_InputField passwordField;

*// Input field for email*

**public** TMP\_InputField emailInputField;

*// Method triggered by the register button to register a new user*

**public** **void** RegisterButton()

{

*// Get inputs from the user*

**string** username = userInputField.text;

*// Hash the password using the MD5Script*

**string** password = md5Script.GenerateMD5Hash();

*// Get email input from the user*

**string** email = emailInputField.text;

*// Add the user entry to the database*

userDbTest.AddEntryUserDb(username, password, email);

*// Read all entries from the user database*

userDbTest.ReadUserDb();

}

}

#### RegistrationManager.cs

**using** System.Collections;

**using** System.Collections.Generic;

**using** System.Text.RegularExpressions;

**using** UnityEngine;

**using** TMPro;

**using** UnityEngine.UI;

**public** **class** RegistrationManager : MonoBehaviour

{

*// References to input fields and UI elements*

**public** TMP\_InputField userInputField; *// Input field for username*

**public** TMP\_InputField passwordField; *// Input field for password*

**RegisterManager.cs Techniques:**

**Classes:**

RegistrationManager

**Procedures and Functions:**

Start()

Update()

KeepTagline(bool Valid)

ChangeTagline(string tagline)

ValidateInput(bool valid, string input, string pattern, string tagline)

ValidateEmail(string input)

ValidateUsername(string input)

ValidatePassword(string input)

ValidateConfirmPass(string input)

**public** TMP\_InputField confirmPassField; *// Input field for confirming password*

**public** TMP\_InputField emailInputField; *// Input field for email*

**public** TMP\_Text validationText; *// Text for displaying validation messages*

**public** TMP\_Text Password; *// Text for password validation*

**public** TMP\_Text confirmPassword; *// Text for confirming password validation*

**public** Button RegisterButton; *// Button for user registration*

*// Boolean flags to track input field validation status*

**private** **bool** EmailValid = **false**; *// Flag indicating if email input is valid*

**private** **bool** UserValid = **false**; *// Flag indicating if username input is valid*

**private** **bool** PassValid = **false**; *// Flag indicating if password input is valid*

**private** **bool** ConfirmPassValid = **false**; *// Flag indicating if confirm password input is valid*

**private** **void** Start()

{

*// Initialize register button as not interactable*

RegisterButton.interactable = **false**;

*// Subscribe to input field value changed events for validation*

emailInputField.onValueChanged.AddListener(ValidateEmail);

userInputField.onValueChanged.AddListener(ValidateUsername);

passwordField.onValueChanged.AddListener(ValidatePassword);

confirmPassField.onValueChanged.AddListener(ValidateConfirmPass);

}

**private** **void** Update()

{

*// Check if all input fields are valid to enable register button*

**if** (EmailValid && UserValid && PassValid && ConfirmPassValid)

{

RegisterButton.interactable = **true**;

}

}

*// Method to display a validation message when input is valid*

**private** **void** KeepTagline(**bool** Valid)

{

validationText.text = "Looking Good!";

validationText.color = Color.black;

}

*// Method to display a validation message when input is invalid*

**private** **void** ChangeTagline(**string** tagline)

{

validationText.text = tagline;

validationText.color = Color.red;

}

*// Method to validate input against a regular expression pattern*

**private** **void** ValidateInput(**bool** valid,**string** input,**string** pattern, **string** tagline)

{

*// Check if input matches the specified pattern*

**bool** isValid = Regex.IsMatch(input, pattern);

**if** (isValid)

{

KeepTagline(valid); *// Display validation message for valid input*

valid = **true**;

}

**else**

{

ChangeTagline(tagline); *// Display validation message for invalid input*

}

}

*// Method to validate email input*

**private** **void** ValidateEmail(**string** input)

{

*// Regular expression pattern for a valid email address*

**string** pattern = @"^(([^<>()\[\]**\\**.,;:\s@\""]+(**\.**[^<>()**\[\]\\**.,;:**\s**@**\"**"]+)\*)|(\"".+\""))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$";

ValidateInput(EmailValid ,input, pattern, "Invalid Email");

}

// Method to validate username input

private void ValidateUsername (string input)

{

// Regular expression pattern for a valid username

string pattern = @"^[a-z0-9\_\-()]{3,20}$";

ValidateInput(UserValid,input, pattern, "Username must only contain 3-20 alphanumeric characters or '\_-()'");

}

// Method to validate password input

private void ValidatePassword(string input)

{

// Regular expression pattern for a valid password

string pattern = @"^[a-zA-Z0-9\_\-()!@#$%^&\*]{8,20}$"; // Example pattern for a more secure password

ValidateInput(PassValid, input, pattern, "Password must be 8-20 characters long and can contain alphanumeric characters or '\_-()!@#$%^&\*'");

}

*// Method to validate confirm password input*

**private** **void** ValidateConfirmPass (**string** input)

{

*// Check if confirm password matches password*

**if** (passwordField.text != confirmPassField.text)

{

ChangeTagline("Passwords must match"); *// Display validation message for mismatched passwords*

ConfirmPassValid = **true**;

}

**else**

{

KeepTagline(ConfirmPassValid); *// Display validation message for matched passwords*

RegisterButton.interactable = **true**; *// Enable register button when all fields are valid*

}

}

}

#### SqliteHelper.cs

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**SqliteHelper.cs Techniques:**

**Classes:**

SqliteHelper

**Procedures and Functions:**

SqliteHelper() (Constructor)

~SqliteHelper() (Destructor)

getAllData(string table\_name)

deleteAllData(string table\_name)

getNumOfRows(string table\_name)

close()

**using** System.Text;

**using** Mono.Data.Sqlite;

**using** UnityEngine;

**using** System.Data;

**namespace** DataBank

{

*/// <summary>*

*/// A helper class for interacting with SQLite database.*

*/// </summary>*

**public** **class** SqliteHelper

{

**private** **const** **string** CodistanTag = "Tetra: SqliteHelper:**\t**";

**private** **const** **string** database\_name = "tetra\_db";

**public** **string** db\_connection\_string;

**public** IDbConnection db\_connection;

*/// <summary>*

*/// Constructor for SqliteHelper. Opens a connection to the SQLite database.*

*/// </summary>*

**public** SqliteHelper()

{

*// Construct the connection string*

db\_connection\_string = "URI=file:" + Application.persistentDataPath + "/" + database\_name;

Debug.Log("db\_connection\_string" + db\_connection\_string);

*// Open a connection to the database*

db\_connection = new SqliteConnection(db\_connection\_string);

db\_connection.Open();

}

*/// <summary>*

*/// Destructor for SqliteHelper. Closes the connection to the SQLite database.*

*/// </summary>*

~SqliteHelper()

{

*// Close the connection to the database*

db\_connection.Close();

}

*// Virtual functions*

*// These methods are placeholders and will be overridden by subclasses.*

*// Helper functions*

*/// <summary>*

*/// Retrieves all data from a specified table in the database.*

*/// </summary>*

**public** IDataReader getAllData(**string** table\_name)

{

*// Create a command to execute SQL query*

IDbCommand dbcmd = db\_connection.CreateCommand();

dbcmd.CommandText = "SELECT \* FROM " + table\_name;

*// Execute the query and return the result set*

IDataReader reader = dbcmd.ExecuteReader();

**return** reader;

}

*/// <summary>*

*/// Deletes all data from a specified table in the database.*

*/// </summary>*

**public** **void** deleteAllData(**string** table\_name)

{

*// Create a command to execute SQL query*

IDbCommand dbcmd = db\_connection.CreateCommand();

*// Execute the SQL command to drop the table*

dbcmd.CommandText = "DROP TABLE IF EXISTS " + table\_name;

dbcmd.ExecuteNonQuery();

}

*/// <summary>*

*/// Retrieves the number of rows in a specified table in the database.*

*/// </summary>*

**public** IDataReader getNumOfRows(**string** table\_name)

{

*// Create a command to execute SQL query*

IDbCommand dbcmd = db\_connection.CreateCommand();

dbcmd.CommandText = "SELECT COALESCE(MAX(id)+1, 0) FROM " + table\_name;

*// Execute the query and return the result set*

IDataReader reader = dbcmd.ExecuteReader();

**return** reader;

}

*/// <summary>*

*/// Closes the connection to the SQLite database.*

*/// </summary>*

**public** **void** close()

{

*// Close the connection to the database*

db\_connection.Close();

}

}

}

**UserDb.cs Techniques:**

**Classes:**

UserDb

**Inheritance:**

Inherits from SqliteHelper

**Procedures and Functions:**

UserDb() constructor

addData(UserEntity user)

getDataByString(string str)

deleteDataByString(string id)

deleteAllData()

getAllData()

getLatestTimeStamp()

**Arrays:**

COLUMNS

#### UserDb.cs

**using** System;

**using** System.Collections.Generic;

**using** System.Data;

**using** UnityEngine;

**namespace** DataBank

{

*// This class represents a database helper specifically designed for managing user-related data.*

**public** **class** UserDb : SqliteHelper

{

**private** **const** **String** Tag = "Tetra: UserDb:**\t**";

**private** **const** **String** TABLE\_NAME = "User";

**private** **const** **String** KEY\_USER = "user";

**private** **const** **String** KEY\_HASH = "hash";

**private** **const** **String** KEY\_EMAIL = "email";

**private** **const** **String** KEY\_DATE = "date";

**private** **String**[] COLUMNS = new **String**[] { KEY\_USER, KEY\_HASH, KEY\_EMAIL, KEY\_DATE };

*// Constructor for initializing the UserDb class.*

**public** UserDb() : **base**()

{

*// Create the User table if it doesn't exist already.*

IDbCommand dbcmd = getDbCommand();

dbcmd.CommandText = "CREATE TABLE IF NOT EXISTS " + TABLE\_NAME + " ( " +

KEY\_USER + " TEXT, " +

KEY\_HASH + " TEXT, " +

KEY\_EMAIL + " TEXT, " +

KEY\_DATE + " DATETIME DEFAULT CURRENT\_TIMESTAMP )";

dbcmd.ExecuteNonQuery();

}

*// Adds a new user entry to the database.*

**public** **void** addData(UserEntity user)

{

IDbCommand dbcmd = getDbCommand();

dbcmd.CommandText =

"INSERT INTO " + TABLE\_NAME

+ " ( "

+ KEY\_USER + ", "

+ KEY\_HASH + ", "

+ KEY\_EMAIL + " ) "

+ "VALUES ( '"

+ user.\_user + "', '"

+ user.\_hash + "', '"

+ user.\_email + "' )";

dbcmd.ExecuteNonQuery();

}

*// Retrieves user data based on username.*

**public** **override** IDataReader getDataByString(**string** str)

{

Debug.Log(Tag + "Getting User: " + str);

IDbCommand dbcmd = getDbCommand();

dbcmd.CommandText =

"SELECT \* FROM " + TABLE\_NAME + " WHERE " + KEY\_USER + " = '" + str + "'";

**return** dbcmd.ExecuteReader();

}

*// Deletes user data based on username.*

**public** **override** **void** deleteDataByString(**string** id)

{

Debug.Log(Tag + "Deleting User: " + id);

IDbCommand dbcmd = getDbCommand();

dbcmd.CommandText =

"DELETE FROM " + TABLE\_NAME + " WHERE " + KEY\_USER + " = '" + id + "'";

dbcmd.ExecuteNonQuery();

}

*// Deletes all user data from the table.*

**public** **override** **void** deleteAllData()

{

Debug.Log(Tag + "Deleting Table");

**base**.deleteAllData(TABLE\_NAME);

}

*// Retrieves all user data from the database.*

**public** **override** IDataReader getAllData()

{

**return** **base**.getAllData(TABLE\_NAME);

}

*// Retrieves the user data with the latest timestamp.*

**public** IDataReader getLatestTimeStamp()

{

IDbCommand dbcmd = getDbCommand();

dbcmd.CommandText =

"SELECT \* FROM " + TABLE\_NAME + " ORDER BY " + KEY\_DATE + " DESC LIMIT 1";

**return** dbcmd.ExecuteReader();

}

}

}

#### UserDbTest.cs

**using** System.Collections;

**using** System.Collections.Generic;

**using** UnityEngine;

**using** DataBank;

**using** TMPro;

**using** UnityEngine.SceneManagement;

**public** **class** UserDbTest : MonoBehaviour

{

*// Reference to display error messages to the user*

**public** TMP\_Text errorMessage;

**UserDbTest.cs Techniques:**

**Classes:**

UserDbTest

UserEntity

**Procedures and Functions:**

ReadUserDb()

AddEntryUserDb(string user, string hash, string email)

IsUsernameExists(string username)

**Arrays:**

myList (List<UserEntity>)

*// Method to read all data from the User database*

**public** **void** ReadUserDb()

{

*// Instantiate a UserDb object to access database operations*

UserDb mUserDb = new UserDb();

*// Retrieve all data from the User database*

System.Data.IDataReader reader = mUserDb.getAllData();

*// Get the number of fields in the database*

**int** fieldCount = reader.FieldCount;

*// List to store UserEntity objects retrieved from the database*

List<UserEntity> myList = new List<UserEntity>();

*// Iterate through each row of data retrieved from the database*

**while** (reader.Read())

{

*// Create a new UserEntity object with data from the current row*

UserEntity entity = new UserEntity(

reader[0].ToString(),

reader[1].ToString(),

reader[2].ToString(),

reader[3].ToString()

);

*// Log the retrieved data to the console (for debugging)*

Debug.Log(entity.\_user + " " + entity.\_hash + " " + entity.\_email + " " + entity.\_dateCreated);

*// Add the UserEntity object to the list*

myList.**Add**(entity);

}

}

*// Method to add a new entry to the User database*

**public** **void** AddEntryUserDb(**string** user, **string** hash, **string** email)

{

*// Instantiate a UserDb object to access database operations*

UserDb mUserDb = new UserDb();

*// Check if the provided username already exists in the database*

**if** (IsUsernameExists(user))

{

*// Display an error message to prompt the user to enter a new username*

errorMessage.text = "Username already exists. Please enter a new username.";

errorMessage.color = Color.red;

**return**;

}

*// Add a new entry to the User database with the provided user information*

mUserDb.addData(new UserEntity(user, hash, email));

*// Log a success message to the console (for debugging)*

Debug.Log("User added successfully.");

*// Load the start screen scene*

SceneManager.LoadScene("Scene1 - Start Screen");

}

*// Method to check if a username already exists in the User database*

**private** **bool** IsUsernameExists(**string** username)

{

*// Instantiate a UserDb object to access database operations*

UserDb mUserDb = new UserDb();

*// Retrieve data from the User database for the provided username*

System.Data.IDataReader reader = mUserDb.getDataByString(username);

*// If any data is returned, the username already exists in the database*

**return** reader.Read();

}

}

# Testing Overview

## Overview

In this section I will be testing my program against each of the objectives that I originally set out in the Analysis (Objectives) section.

## Test plan and outcome

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test number | Test description | Input Data | Expected Result | Actual Result | Pass/Fail | |
| 1 | UI and User Navigation | | | | |
| 1.1 | 1.1|User should be provided with the Start Menu UI | n/a | Start Menu UI scene will open | Start Menu UI scene opens |  | |
| 1.1.1|Create button to direct the user to the Registration UI | n/a | Registration button will be created | Registration button is created |  | |
| 1.1.2|Create button to direct the user to the Login UI | n/a | Login button will be created | Login button is created |  | |
| 1.1.3|Create button to direct the user to allow the user to exit the program | Button press | Exit button will be created and when clicked will exit the program | Exit button is created and exits the program when clicked |  | |
| 1.2 | 1.2|When the user selects the option to register then the Registration UI should be presented | Button press | Registration UI scene will open | Registration UI scene opens |  | |
| 1.2.1|The available input fields should be email address, username, password, confirm password | User text input | Four input fields will be displayed that the user will be able to type into | Four input fields are displayed and the user is able to type into each of them |  | |
| 1.3 | 1.3|Create a function that checks all of the specified input fields | Email, username, password and confirm password strings | Continuously checks all the input fields for valid inputs. If any inputs invalid it displays the appropriate message | Continuously checks all the input fields for valid inputs. If any inputs invalid it displays the appropriate message |  | |
| 1.3.1|The authorise function should check if the inputs entered in the password and confirm password fields are the same | password:  “3asts1d3”  confirm password: “3asts1d3” | Correct data – should be accepted | Input accepted |  | |
| password:  “3asts1d3”  confirm password:  “wests1de” | Erroneous data – passwords do not match -should be rejected  ‘Passwords do not match’ message displayed | Input rejected |  | |
| 1.3.2|The authorise function should check whether a valid email address has been entered in the email address field | Email: “example@gmail.com” | Correct Data – input should be accepted | Input accepted |  | |
| Email: “thisisnotanemail” | Erroneous data – is not a valid email - should be rejected  “Please enter a valid email address” message displayed | Input rejected |  | |
| 1.3.3|Create button to register the user | n/a | Register button will be created. | Register button was created. |  | |
| 1.4 | 1.4|When the user selects the register button | Button press | When pressed the button will run the register function. | When pressed the register function ran |  | |
| 1.4.1|The register function should encrypt the user’s entered password using a hashing algorithm | Password: “Happy19” | Expected hash:” 32f6a59a89b786512c068b0e5ec62a6c” | Actual Hash: “32f6a59a89b786512c068b0e5ec62a6c” |  | |
| 1.4.2|The encrypted password should be stored in the database | Password String: “Happy19” | Debug Log should display a message containing the email, username, hashed password and time stamp of the registered user. | Debug Log Message: “NewUser12 32f6a59a89b786512c068b0e5ec62a6c example24@exampleemail.com 06/03/2024 10:19:33” |  | |
| 1.4.3|The username should be stored in the database | Username string:“NewUser12” | Debug Log should display a message containing the email, username, hashed password and time stamp of the registered user. | Debug Log Message: “NewUser12 32f6a59a89b786512c068b0e5ec62a6c example24@exampleemail.com 06/03/2024 10:19:33” |  | |
| 1.4.4|The email address should be stored in the database | Email string:”example24@exampleemail.com” | Debug Log should display a message containing the email, username, hashed password and time stamp of the registered user. | Debug Log Message: “NewUser12 32f6a59a89b786512c068b0e5ec62a6c example24@exampleemail.com 06/03/2024 10:19:33” |  | |
| 1.4.5|The user should be brought to the Start Menu UI | n/a | Start Menu scene will open | Start Menu scene opens |  | |
| 1.5 | 1.5|When the user selects the option to login then the Login UI should be presented  1.5.1|The available input fields should be username, password  1.5.2|Create a login button that checks the user’s input | Button press | Login UI scene will open | Login UI scene opens |  | |
| User text input | Two input fields will be displayed that the user will be able to type into | Two input fields are displayed and the user is able to type into each of them |  | |
| n/a | A login button will be created | A login button was created |  | |
| 1.6 | 1.6|When the user selects the login button | Button press | Login function will run | Login functions run |  | |
| 1.6.1|Check the database if the given username exists | Username string: “NewUser12” | Correct Data – User that exists -  Debug Log will display that it is trying to fetch user | Debug Log Message:“ Tetra: UserDb: Getting User: NewUser12” |  | |
| Username string: “NotNewUser” | Erroneous Data – User that doesn’t exists -  Debug Log will display that it is trying to fetch user, will fail and then display that user does not exist. | Debug Log Message:“ Tetra: UserDb: Getting User: NotNewUser” |  | |
| 1.6.2|Run the given user password through the hashing algorithm | Password string: “Happy19” | Expected hash:” 32f6a59a89b786512c068b0e5ec62a6c” | Actual Hash: “32f6a59a89b786512c068b0e5ec62a6c” |  | |
| 1.6.3|Retrieve the hashed password from the database and compare the two passwords to see if they match | Password string: “Happy19” | Correct Data – User exists and password hash matches – Debug Log should deplay an appropriate authantication message | Debug Log Message:“ User authenticated successfully” |  | |
| Password string: “NotPassword” | Erroneous Data – User exists and password hash does not match – Display ‘Incorrect Password’ error message | ‘Incorrect Password’ error message displayed |  | |
| 1.6.4|If the login is successful the user should be brought to the Game Menu UI | Username string: “NewUser12”  Password string: “Happy19” | Correct Data – User exists and password hash matches – Game Menu should be loaded | Game menu loaded |  | |
| Username string: “NotNewUser”  Password string: “NotPassword” | Erroneous Data – User does not exist and password hash does not match – Game Menu should not be loaded, ‘User does not exists’ message should be displayed | Game menu not loaded  ‘User does not exists’ message displayed |  | |
| Username string: “NotNewUser”  Password string: “Happy19” | Erroneous Data – User does not exist and password hash does not match – Game Menu should not be loaded, ‘User does not exists’ message should be displayed | Game menu not loaded  ‘User does not exists’ message displayed |  | |
| Username string: “NewUser12”  Password string: “NotPassword” | Erroneous Data – User exists and password hash does not match – Game Menu should not be loaded, ‘Incorrect Password’ message should be displayed | Game menu not loaded  ‘Incorrect Password’ message displayed |  | |
| 1.7 | 1.7|When the user is brought to the Game Menu UI | n/a | Game Menu scene will open | Game Menu scene opens |  | |
| 1.7.1|Create button to direct the user to the Single Player UI | n/a | Single Player button will be created when clicked it will direct the user to the Single Player scene | Single Player button is created  When clicked it loaded the Single Player scene |  | |
| 1.7.2|Create button to direct the user to the Multiplayer UI | n/a | Multiplayer button will be created when clicked it will direct the user to the Multiplayer scene | Multiplayer button is created  When clicked it loaded the Multiplayer scene |  | |
| 1.7.3|Create button to direct the user to the Education UI | n/a | Education button will be created when clicked it will direct the user to the Education scene | Education button is created  When clicked it loaded the Education scene |  | |
| 2 | Single Player Mode | | | | |
| 2.1 | 2.1|Create a 10 x 20-unit rectangular grid which forms the bounds of the play area | n/a | A 10 x 20 cell grid will be displayed | A 10 x 20 cell grid is displayed |  | |
| 2.2 | 2.2|Design a suitable gameplay loop | n/a | The gameplay loop should run until the game over condition is reached | The gameplay loop runs until the game over condition is reached |  | |
| 2.2.1|A randomly selected tetromino should appear at the top-centre of the play area | n/a | Either an I,J,O,L,S,Z or T piece will spawn in the top-center of the grid | An I piece spawned at the top-center of the grid |  | |
| 2.2.2|The tetromino should start moving through the play area one unit at a time downwards | n/a | The piece that spawns will move move down the grid until it hits the bottom or another tetromino | The piece that spawns moved down the grid until it hit the bottom |  | |
| 2.2.3|When the player clicks the correct button the tetromino should rotate 90 degrees clockwise | Page down button | The piece should rotate 90 degrees clockwise | The piece rotated 90 degrees clockwise |  | |
| 2.2.4|When the player clicks the correct button the tetromino should rotate 90 degrees anticlockwise | Page up button | The piece should rotate 90 degrees anticlockwise | The piece rotated 90 degrees anticlockwise |  | |
| 2.2.5|When the player clicks the correct button the tetromino should move one unit left | Left arrow key | The piece should move one cell left | The piece moved one cell left |  | |
| 2.2.6|When the player clicks the correct button the tetromino should move one unit right | Right arrow key | The piece should move one cell right | The piece moved one cell right |  | |
| 2.2.7|When the player places a piece that completes a line, that line should clear from the playfield  2.2.8|When the player places a piece that completes multiple lines, all completed lines should clear from the playfield | n/a | The line(s) should clear from the grid | The line(s) cleared from the grid |  | |
| 2.3 | 2.3|While the gameplay loop is running, create a display for the user’s current score | Score data from | Score data is displayed and constantly updated every time a line clears and other scoring events | The score was constantly updated with the final score being |  | |
| 2.4 | 2.4|When the player places a piece outside of the playfield, due to the playfield being full, the gameplay loop should end. | n/a | The gameplay loop ends | The gameplay loop ended |  | |
| 2.5 | 2.5|Once the game ends a game over screen should appear | n/a | Once the game over condition is reached the Game Over scene is loaded in and displays the user’s score | The game over condition was reached and the Game Over Screen was loaded in with a score of “” |  | |
| 2.5.1|Create a button to allow the player to restart the game | Button Press | Restart button will be created when clicked it will direct the user back to the Singleplayer scene | Restart button is created  When clicked it loaded the Singleplayer scene |  | |
| 2.5.2|Create a button to return to the game menu | Button Press | Game menu button will be created when clicked it will direct the user to the Game Menu scene | Game Menu .  button is created  When clicked it loaded the Game Menu scene |  | |
| 2.6 | 2.6|The player should be able to pause the game, either through clicking an on-screen button or pressing the Escape key  2.7|When the player pauses the game the appropriate pause screen menu should be displayed | Button press | The pause screen menu should appear and all movement on the gameplay screen should stop | The pause screen menu appeared  All movement on the gameplay screen stopped |  | |
| Escape Key | The pause screen menu should appear and all movement on the gameplay screen should stop | The pause screen menu appeared  All movement on the gameplay screen stopped |
| 2.7 |  | |
| 2.7.1|Create a button to allow the player to resume the game | Button Press | Resume button will be created when clicked it will return the user to the gameplay scene | Resume button is created  When clicked it returns the user back to the gameplay scene |  | |
| 2.7.2|Create a button to allow the player to return to the main menu | Button Press | Game menu button will be created when clicked it will direct the user to the Game Menu scene | Game Menu .  button is created  When clicked it loaded the Game Menu scene |  | |
| 3 | Educational Mode | | | | |
|  | 3|The Educational mode should allow the user to view various different pieces of information which includes controls, terminology and technique | n/a | The Education mode should display various different menus and buttons that allows the user to access information | The Education mode displays various different menus and buttons that allows the user to access information |  | |
| 4 | Local Multiplayer | | | | |
| 4.1 | 4.1|The users should be presented with two 10 x 20-unit rectangular grid which forms the bounds of both play areas | n/a | The gameplay loop should run until the game over condition is reached | The gameplay loop runs until the game over condition is reached |  | |
| 4.2 | 4.2|The suitable gameplay loop should be started | n/a | Either an I,J,O,L,S,Z or T piece will spawn in the top-center of the grid | An I piece spawned at the top-center of the grid |  | |
| 4.2.1|A randomly selected tetromino should appear at the top-centre of both play areas | n/a | The piece that spawns will move move down the grid until it hits the bottom or another tetromino | The piece that spawned moved down the grid until it hits the bottom |  | |
| 4.2.2|The tetromino should start moving through both play areas one unit at a time downwards | Page down button | The piece should rotate 90 degrees clockwise | The piece should rotate 90 degrees clockwise |  | |
| 4.2.3|When the player clicks the correct button the tetromino should rotate 90 degrees clockwise | Page up button | The piece should rotate 90 degrees anticlockwise | The piece should rotate 90 degrees anticlockwise |  | |
| 4.2.4|When the player clicks the correct button the tetromino should rotate 90 degrees anticlockwise | Left arrow key | The piece should move one cell left | The piece should move one cell left |  | |
| 4.2.5|When the player clicks the correct button the tetromino should move one unit left | Right arrow key | The piece should move one cell right | The piece should move one cell right |  | |
| 4.2.6|When the player clicks the correct button the tetromino should move one unit right | Right arrow key | The piece should move one cell right | The piece moved one cell right |  | |
| 4.2.7|When the player places a piece that completes a line, that line should clear from the playfield  4.2.8|When the player places a piece that completes multiple lines, all completed lines should clear from the playfield | n/a | The line(s) should clear from the grid | The line(s) cleared from the grid |  | |
| 4.2.9|While the gameplay loop is running, the program should display both users’ current scores | Score data from | Score data is displayed and constantly updated every time a line clears and other scoring events | The score was constantly updated with the final score being |  | |
| 4.2.10|When the player places a piece outside of the playfield, due to the playfield being full, both the gameplay loops should end. | n/a | Both of the gameplay loops should end | Both of the gameplay loops ended |  | |
| 4.3 | 4.3|Once the game ends a game over screen should appear | n/a | Once the game over condition is reached the Game Over scene is loaded in and displays both the user’s score | The game over condition was reached and the Game Over Screen was loaded in with scores of “” and “” |  | |
| 4.3.1|Create a button to allow the player to restart the game | Button Press | Restart button will be created when clicked it will direct the user back to the Singleplayer scene | Restart button is created  When clicked it loaded the Singleplayer scene |  | |
| 4.3.2|Create a button to allow the player to return to the game menu | Button Press | Game menu button will be created when clicked it will direct the user to the Game Menu scene | Game Menu .  button is created  When clicked it loaded the Game Menu scene |  | |
| 4.3.3|Display which player won | n/a | The Game Over scene will have a message displaying who won | Message displayed on Game Over Scene: “” |  | |

# Evaluation

## Overview

In this section, I'll assess how well my project met the objectives outlined in the Analysis section. I'll also talk about any changes I made from the original plan, the challenges I faced during development, and my plans for the project's future development.

## Meeting Objectives

### UI and User Navigation

1. The user should be able to navigate around the GUI (Graphical User Interface) of the application with the available buttons and menus. The UI will be displayed using the Unity UI modules
   1. User should be provided with the Start Menu UI

This was successful as when the application opens the Start Menu is the first scene. From this menu the user is able to go to the Login and Registration sections of the program.

* + 1. Create button to direct the user to the Registration UI

This was successful as a button is created in the scene to direct the user to the Registration Scene.

* + 1. Create button to direct the user to the Login UI

This was successful as a button is created in the scene to direct the user to the Registration Scene.

* + 1. Create button to direct the user to allow the user to exit the program

This was successful as a button is created in the scene to allow the user to completely exit and close the program.

* 1. When the user selects the option to register then the Registration UI should be presented

This was successful as when the user selects the option to register, the Registration UI is presented with all the appropriate input fields and buttons.

* + 1. The available input fields should be email address, username, password, confirm password

This was successful as input fields for email address, username, password, and confirm password are available in the Registration UI.

* 1. Create a function that check all of the specified input fields

This was successful as a function (ValidateInput()) is available to check all specified input fields for validity

* + 1. The authorise function should check is the inputs entered in the password and confirm password fields are the same

This was successful as there is a function (ValidateConfirmPass()) that successfully checks if the inputs entered in the password and confirm password fields match.

* + 1. The authorise function should check whether a valid email address has been entered in the email address field

This was successful as there is a function (ValidateEmail()) that effectively checks whether a valid email address has been entered in the email address field.

* + 1. Create button to register the user

This was successful as a button has been created to allow users to register their account.

* 1. When the user selects the register button
     1. The register function should encrypt the user’s entered password using a hashing algorithm

This was successful as the RegisterUI script calls the GenerateMD5Hash() to generate a hash of the password string.

* + 1. The encrypted password should be stored in the database

This was successful as after encryption, the hashed password is stored securely in the database.

* + 1. The username should be stored in the database

This was successful as the user's chosen username is stored in the database.

* + 1. The email address should be stored in the database

This was successful as the user's chosen email is stored in the database.

* + 1. The user should be brought to the Start Menu UI

This was successful as upon successful registration, the user is directed to the Start Menu UI

* 1. When the user selects the option to login then the Login UI should be presented

This was successful as when the user selects the option to login, the Login UI is presented with all the appropriate input fields and buttons.

* + 1. The available input fields should be username, password

This was successful as the Login UI provides input fields for the username and password

* + 1. Create a login button that checks the user’s input

This was successful as a login button has been created to initiate the login process, which verifies the user's input.

* 1. When the user selects the login button
     1. Check the database if the given username exists

This was successful as the system checks the database to confirm the existence of the entered username.

* + 1. Run the given user password through the hashing algorithm

This was successful as the user's entered password is hashed using the designated algorithm.

* + 1. Retrieve the hashed password from the database and compare the two passwords to see if they match

This was successful as the system retrieves the hashed password associated with the provided username and compares it with the entered password to authenticate the user.

* + 1. The user should be brought to the Game Menu UI

This was successful as upon successful login, the user is directed to the Game Menu UI.

* 1. When the user is brought to the Game Menu UI
     1. Create button to direct the user to the Single Player UI

This was successful as a button has been created to allow users to access the Single Player UI.

* + 1. Create button to direct the user to the Multiplayer UI

This was successful as a button has been created to allow users to access the Single Player UI.

* + 1. Create button to direct the user to the Education UI

This was successful as a button has been created to allow users to access the Single Player UI.

### Single Player Mode

1. The Single Player mode should allow the user to play a standard single player version of Tetris
   1. Create a 10 x 20-unit rectangular grid which forms the bounds of the play area

This was successful as a 10 x 20-unit rectangular grid was created, defining the boundaries of the play area.

* 1. Design a suitable gameplay loop
     1. A randomly selected tetromino should appear at the top-centre of the play area

This was successful as a randomly selected tetromino appears at the top-center of the play area to initiate gameplay.

* + 1. The tetromino should start moving through the play area one unit at a time downwards

This was successful as the tetromino starts moving downwards through the play area at a steady pace.

* + 1. When the player clicks the correct button the tetromino should rotate 90 degrees clockwise

This was successful as the tetromino rotates 90 degrees clockwise when the player clicks the corresponding button.

* + 1. When the player clicks the correct button the tetromino should rotate 90 degrees anticlockwise

This was successful as the tetromino rotates 90 degrees anticlockwise when the player clicks the corresponding button.

* + 1. When the player clicks the correct button the tetromino should move one unit left

This was successful as the tetromino moves one unit left when the player clicks the corresponding button.

* + 1. When the player clicks the correct button the tetromino should move one unit right

This was successful as the tetromino moves one unit right when the player clicks the corresponding button.

* + 1. When the player places a piece that completes a line, that line should clear from the playfield

This was successful as completed lines are cleared from the playfield when the player places a piece that completes them.

* + 1. When the player places a piece that completes multiple lines, all completed lines should clear from the playfield

This was successful as all completed lines are cleared from the playfield when the player places a piece that completes multiple lines

* 1. While the gameplay loop is running, create a display for the user’s current score

This was successful as a display for the user’s current score was created and is visible during gameplay.

* 1. When the player places a piece outside of the playfield, due to the playfield being full, the gameplay loop should end.

This was successful as the gameplay loop ends when the player places a piece outside of the playfield, indicating game over.

* 1. Once the game ends a game over screen should appear

This was successful as a game over screen appears once the game ends.

* + 1. Create a button to allow the player to restart the game

This was successful as a button allowing the player to restart the game was created on the game over screen.

* + 1. Create a button to return to the game menu

This was successful as a button allowing the player to return to the game menu was created on the game over screen.

* 1. The player should be able to pause the game, either through clicking an on-screen button or pressing the Escape key

This was successful as the player can pause the game either by clicking an on-screen button or pressing the Escape key.

* 1. When the player pauses the game the appropriate pause screen menu should be displayed

This was successful as the appropriate pause screen menu is displayed when the player pauses the game.

* + 1. Create a button to allow the player to reset the game

This was successful as a button allowing the player to reset the game was created on the pause screen menu.

* + 1. Create a button to allow the player to return to the main menu

This was successful as a button allowing the player to return to the main menu was created on the pause screen menu.

### Educational mode

1. The Educational mode should allow the user to view various different pieces of information which includes controls, terminology and technique

### Local Multiplayer

1. The LAN Multiplayer mode should allow two users to play a 2-player version of Tetris on the same device
   1. The users should be presented with two 10 x 20-unit rectangular grid which forms the bounds of both play areas

This was successful as the game interface displays two separate grids, each representing the play area for one player.

* 1. The suitable gameplay loop should be started
     1. A randomly selected tetromino should appear at the top-centre of both play areas

This was successful as random tetrominoes are generated and displayed at the designated starting positions for both players.

* + 1. The tetromino should start moving through both play areas one unit at a time downwards

This was successful as the tetrominoes descend in both play areas at a consistent rate.

* + 1. When the player clicks the correct button the tetromino should rotate 90 degrees clockwise

This was successful as players can rotate tetrominoes clockwise by pressing the appropriate button.

* + 1. When the player clicks the correct button the tetromino should rotate 90 degrees anticlockwise

This was successful as players can rotate tetrominoes counterclockwise by pressing the appropriate button.

* + 1. When the player clicks the correct button the tetromino should move one unit left

This was successful as players can move tetrominoes left by pressing the appropriate button.

* + 1. When the player clicks the correct button the tetromino should move one unit right

This was successful as players can move tetrominoes right by pressing the appropriate button.

* + 1. When the player places a piece that completes a line, that line should clear from the playfield

This was successful as completed lines are cleared from the playfield when either player fills an entire row.

* + 1. When the player places a piece that completes multiple lines, all completed lines should clear from the playfield

This was successful as all completed lines are cleared from the playfield when a player fills multiple rows simultaneously.

* + 1. While the gameplay loop is running, the program should display both users’ current scores

This was successful as the game interface displays the scores of both players during gameplay.

* + 1. When the player places a piece outside of the playfield, due to the playfield being full, the gameplay loop should end.

This was successful as the game ends when a player is unable to place a tetromino within the playfield boundaries.

* 1. Once the game ends a game over screen should appear
     1. Create a button to allow the player to restart the game

This was successful as a button is provided on the game over screen to restart the game.

* + 1. Create a button to allow the player to return to the game menu

This was successful as a button is provided on the game over screen to return to the game menu.

* + 1. Display which player won

This was successful as the game over screen displays which player achieved the highest score and won the game.

## Modifications

Over the course of development, I implemented several beneficial changes to the original plan for my program. One significant enhancement was the addition of the full Tetris Super Rotation System, which incorporated features such as wall kicks and restricted rotation. This addition greatly improved the gameplay experience, providing players with more flexibility and strategic options when manipulating Tetris pieces. Additionally, it allowed for a more educational gameplay experience as it was more representative than the average Tetris game

Another notable change involved the implementation of regular expressions for input validation instead of relying on Unity's built-in input validation system. By leveraging regular expressions, I was able to apply more robust and customizable input validation logic, ensuring that user inputs met specific criteria with greater accuracy and efficiency.

In addition to this, I made the decision to utilize SQLite as the database solution instead of opting for a full SQL database. This choice was motivated by SQLite's portability and the relatively small scale of my project. By using SQLite, I streamlined the data management process while still maintaining the necessary functionality for storing and retrieving information within the application.

Furthermore, employing SQLite ensured that the program remained entirely local and operated independently of the device it was run on. This provided users with a seamless experience, free from reliance on external servers or network connectivity. Overall, these changes contributed to the overall success and effectiveness of the program, enhancing its functionality and usability for end-users.

## Challenges and Difficulties

During the project's development, I encountered various challenges and undertook efforts to address them. Initially, learning how to utilize the Unity framework proved essential. This hurdle was overcome through the completion of multiple smaller Unity projects, prototyping my project, and extensive study of Unity's documentation.

Furthermore, I faced difficulties with aspects such as the MD5 hashing algorithm, array errors, padding up to 512 bits, and rotation systems. These challenges were addressed through iterative processes and extensive testing.

Another significant learning curve involved mastering the usage of SQLite and the Unity library necessary for database communication. I found valuable guidance through resources like a Medium article titled ”Saving Data in Unity3D Using SQLite” [[11]](#footnote-12)and by studying examples provided by other developers.

## Feedback

After finishing development, I held a discussion with my interviewee, Alexis Brightman, about the Tetris project, and they were happy withthat progress had been made in providing novice players with helpful information and understanding of the game. The user interface improvements, such as clear controls and tutorials, were praised for making the game more accessible. Additionally, the full gameplay experience was effective in helping beginners learn how to play and improve their skills. One feature that Alexis enjoyed, in particular, was the competitive mode as it allowed for their clubmates to compete against each other in mini tournaments that allowed them to develop their skills quickly in a fun and amiable manner

However, it was noted that there is still room for improvement. It was suggested that adding a glossary of Tetris terms that was accessible from the pause menu could further aid players in understanding the game. Additionally, incorporating interactive challenges or mini-games focusing on specific techniques, in the tutorials, could encourage players to explore advanced strategies.

## Future Development

In terms of future development, several ideas are under consideration for implementation in the project. One potential idea involves introducing a client-server based registration system. This expansion would broaden the scope and client base of the project beyond its original requirements, potentially facilitating a more robust and scalable user registration process.

Another idea for future development is the incorporation of a “versus computer” mode. This feature could involve implementing a neural network or algorithm as a computer player, offering users the option to challenge AI opponents. However, this may pose challenges due to Tetris' random and infinite nature, requiring careful consideration and implementation to ensure an engaging and fair gameplay experience.

Additionally, plans include the integration of LAN and online multiplayer functionality. This would enable users to engage in multiplayer matches either locally, on the same network (LAN mode), or remotely, across different networks, such as over the internet. Achieving this goal could involve leveraging user-created Unity libraries like Mirror Networking and others to facilitate networked gameplay and communication between players.

Overall, these future development ideas aim to enhance the project's features and capabilities, providing users with a more dynamic and engaging gaming experience while expanding its reach and accessibility across different platforms and networks.

## Bibliography

Dan's Tools (n.d.). *MD5 Hash Generator*. [online] www.md5hashgenerator.com. Available at: http://www.md5hashgenerator.com/ [Accessed 6 Jan. 2024].

Dominic Frei, D. (2022). *Saving Data in Unity3D Using SQLite | MongoDB*. [online] www.mongodb.com. Available at: https://www.mongodb.com/developer/code-examples/csharp/saving-data-in-unity3d-using-sqlite/.

FOUR. (n.d.). *FOUR*. [online] Available at: https://four.lol/.

harddrop.com. (n.d.). *Hard Drop Tetris Wiki*. [online] Available at: https://harddrop.com/wiki [Accessed 6 Mar. 2024].

Lake, J. (2021). *md5-no-2fix-02-scaled*. [online] Comparitech.com. Available at: https://cdn.comparitech.com/wp-content/uploads/2021/06/md5-no-2fix-02-scaled.webp.

Tepples (2006). *Gameplay of Tetris - TetrisWiki*. [online] tetris.wiki. Available at: https://tetris.wiki/Gameplay\_of\_Tetris.

TETR.IO (n.d.). *TETR.IO*. [online] tetr.io. Available at: https://tetr.io/.

tetris.wiki. (n.d.). *Tetris Guideline - TetrisWiki*. [online] Available at: https://tetris.wiki/Tetris\_Guideline.

1. Alternative spelling: ‘Tetraminos’, ‘Tetraminoes, ‘Tetrominoes’ [↑](#footnote-ref-2)
2. *Gameplay overview - Tetris Wiki*, <https://tetris.miraheze.org/wiki/Gameplay_overview> [↑](#footnote-ref-3)
3. FOUR. (n.d.). FOUR. <https://four.lol/> [↑](#footnote-ref-4)
4. Hard Drop Tetris Wiki. (n.d.). <https://harddrop.com/wiki> [↑](#footnote-ref-5)
5. Tetris Guideline - TetrisWiki. (n.d.). <https://tetris.wiki/Tetris_Guideline> [↑](#footnote-ref-6)
6. Tetr.Io. (n.d.). TETR.IO. <https://tetr.io/> [↑](#footnote-ref-7)
7. *Tetris AI Player by tetris-hp - Itch.io*, https://tetris-hp.itch.io/tetris-ai-player. [↑](#footnote-ref-8)
8. Tetris 99® for Nintendo Switch. (n.d.). <https://www.nintendo.com/store/products/tetris-99-switch/> [↑](#footnote-ref-9)
9. Dan's Tools (n.d.). MD5 Hash Generator. [online] [www.md5hashgenerator.com](http://www.md5hashgenerator.com) [↑](#footnote-ref-10)
10. Lake, J. (2021). md5-no-2fix-02-scaled. webp. Available at: <https://cdn.comparitech.com/wp-content/uploads/2021/06/md5-no-2fix-02-scaled.webp> [Accessed 21 Feb. 2024].

    ‌ [↑](#footnote-ref-11)
11. Dominic Frei, D. (2022). Saving Data in Unity3D Using SQLite | MongoDB. [online] www.mongodb.com. Available at: <https://www.mongodb.com/developer/code-examples/csharp/saving-data-in-unity3d-using-sqlite/> [↑](#footnote-ref-12)