Final Year Project Report

Full Unit – Final Report

Huckers with a Twist

David Playfoot

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science and Mathematics

Supervisor: Professor Adrian Johnstone

Department of Computer Science

Royal Holloway, University of London

April 11, 2024

Declaration

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 Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 15,000

Student Name: David Playfoot

Date of Submission: 12th April 2024

Signature: [illegible]

Table of Contents

Introduction

Huckers the game – History and Rules [1], [3]

Huckers (also commonly known as Uckers) is a board game derived from the classic game of Ludo. It consists of four players, each player paired up with the player diagonally opposite to form two teams. Each player takes it in turns to roll the two dice and strategically move their counters. The ultimate aim is to get all four of your coloured counters on the winning squares in the middle, followed by all four of your partner's counters in order to win the game as a team. Unlike a classic game of Ludo, Huckers introduces an element of strategy that has never been seen before in the Ludo world! The key twist in this game from the original Huckers is the idea that every player is on their own, and your 'partner' can actually be a hindrance in that both counters then go back to the beginning instead of one.

Although the first official version of Huckers was not seen until 1946, the Royal Navy servicemen used to play this game throughout the course of World War 2. It is believed that Huckers was introduced to many countries' armed forces throughout the commonwealth [1]. In my family, Huckers has been a board game passed down the maternal side of my family, starting with my maternal grandfather, who was a Naval serviceman. Today, Uckers is still a game that is widely played amongst the Royal Marines and Army Air Corps in the UK [2]. Throughout the course of the report, a full list of rules will be provided for the twist on Huckers that is provided, however, it is important to note that all variations of Huckers are different from one Naval unit to another.

3D Noughts & Crosses (Qubic) [3], [4]

Qubic is a brand name for a 4x4x4 Noughts & Crosses (3D Tic-Tac-Toe) grid. Traditional Noughts & Crosses consists of a flat plane with a 3x3 grid. Qubic is a two-player game whereby the players either choose 'X' or 'O', much like a regular game of 2D Noughts & Crosses (2D Tic-tac-toe). Aside from this, the rules remain similar, you will have to get four in a line, thus completing a winning line and therefore win the game.

In Qubic, there are 76 possible winning lines [3]. A player is only required to fill one of the winning lines in order to fill the game. The lines can either stem from each 4x4 layer, or passing through each layer either diagonally or vertically.

In 1976 3D tic-tac-toe games, such as Qubic, the existence of a winning strategy was proven by Eugene Mahalko without actually presenting an example of such strategy. However, in 1980, the problem was better solved using examples by Oren Patashnik. Patashnik used a computer-based proof to attempt to solve the problem of 3D tic-tac-toe, this had a total computation time of 1500 hours! Over time, there has been multiple implementations of 3D tic-tac-toe, one of which consisted of a graphical version, released by Atari in 1978 for the Atari 2600 console. Another famous implementation of a similar program to Qubic was created as part of Microsoft Windows Entertainment Pack in the 1990s. In 2010, Microsoft then made this game available on the Xbox 360 console. [3]

Referencing (The Syntax)

Throughout this report, some of the sentences will be concluded with a number in square brackets, a number that has a font size slightly smaller. Where this is the case, references are provided and these numbers will correspond to the number in the bibliography. The research document, or website, where the information came from is referenced next to the corresponding number in the bibliography. References can apply to that specific sentence and any preceding sentences in that paragraph.

Where the reference is used throughout the whole sub-section, the sub-section headings will include the reference instead (in the same way as outlined above).

Project Specification and Revised Plan

The specification, main objectives and goals are a very important part of the development of any project. Furthermore, breaking the main objectives down into goals also aids in the implementation of each feature, therefore meeting the initial project objectives. In this project, all of the objectives are listed alongside their respective goals.

In addition to the project objectives, a project plan is always an essential part of creating an organised strategy to tackle the problem at hand. Therefore, based on the importance of a well-structured project plan, taken together with the advice given in the feedback from the project plan, I have decided to include the Revised project plan which demonstrates a more accurate representation of a timeline of when tasks will be completed.

Technologies

A list of technologies is a useful tool to have when referring to all of the different technologies. In order to implement a good strategy to ensure working code is present (Code that is readable and passes the tests), stating the strategies to tackle the Software Engineering of a project is crucial. For this project, the use of Unity C#, taken together, with the use of a Version Control System was crucial in aiding the development of the project, ensuring that the risks (outlined in Chapter 5) are mitigated.

In addition, allocating time to experiment with the different new technologies is also a really important learning mechanism for the implementation of any software on a new programming language or User-Interface based structure.

Abstract & Reading – Critical Analysis

Board Game

Board games date back thousands of years. Despite the rapid increase in Computer technology during recent years, board games still remain a much-loved family favourite.

In addition to the vast amount of history that surrounds the board game world, they can also be used for educational purposes. It has been widely believed for many years that board games can become a distraction in the education world, however, after a study conducted by Siu Yun Cheung and Kai Yin Ng into the effectiveness of games in education, it became apparent that board games can aid a multi-sensory environment that enhances learning [11]. Furthermore, board games in general can enhance critical thinking skills once a strategic element is introduced.

Games of Luck Vs. Games of Strategy

Most board games have an element of luck, wherever the pieces end up depends on a combination of different features, rules and elements of the game. Most commonly, a dice is thrown and the position from the previous turn can determine the next moves. One examples of a game involving solely luck-based features is classic Ludo, the dice roll determines exactly where you end up! In Huckers, the rule set (outlined in the appendix) allows for a more strategic approach to solving this problem.

Significance of ‘Huckers with a Twist’

There are many reasons why this project is significant. In the past recent years, it has been very difficult to get your hands on a 3D tic-tac-toe board and, upon searching on many search engines such as Bing, Google and Yahoo Search, it has been difficult to find an online, computerised version of 3D Noughts & Crosses that is easy-to-use and has an online computer player, indicating that the problem of 3D Noughts & Crosses has not been a common problem that many people have been beginning to solve. As time has progressed, more modern games have come out. This project brings back an old traditional board game, whilst maintaining a computerised presence, hence, finding a middle ground between modern-day games and enabling people to play traditional board games through the use of a computerised platform.

The combination of the two games enables families with ancestors from the armed forces to either learn a game which could have played a role in the history of their family or learn a unique twist on a familiar game. In addition, for those servicemen who are still alive, there is the chance for them to play an inter-generational family board game with the younger members of their family (or even the community as a whole).

The significance of the ‘Computerised Player’ is a key part of the strategic elements of the game. Focusing on one mode for the computer means that the difficulty level is raised. The computer will always think about the most optimal location to place itself (where a so-called “optimal position” exists, please see Chapter 3 for more details which describes the notion of a ‘perfect player’).

Chapter 1: **Tasks, Goals and map to completion**

Primary Objectives

The main objectives that the project is required to fulfill are outlined in the bullet points below. Once the main goals were outlined, I was able to break these down in to subsections of more manageable tasks, these tasks are labelled as 'goals'. These goals are then further broken down during the implementation into smaller, more manageable tasks that work towards either that specific goal or the primary objective as a whole.

1. Create a Huckers twist game, combining rules of the standard Huckers game whilst adding a unique twist on the rules.
2. Creating 3D Noughts & Crosses using a 4x4x4 grid.
3. Implementing a computerised player for both of these games.
4. Figuring out a 'twist' to the Huckers Twist game which allows for the two games to be combined into one larger game.

Goals

The goals are outlined below, and where applicable, the goals will be further broken down into more manageable set tasks or handy tips that aid in a final implementation. After this section of the report, the goals may be referenced with the word "Goal" followed by the number of the primary objective and the number of the goal. For example, the first goal under primary objective 1 may be referred to as "Goal 1.1".

Primary Objective 1:

1. Experimenting with the tools provided in Unity 3D, do some background research into implementations of Ludo in Unity and write a one-page document explaining how this will aid in the development of Huckers, which is played on the same kind of board.

   - Experiment with Unity tools, planes, Cubes, and Buttons. How the location of these objects can move around the board.

2. Import the image of the basic Ludo board, the images for the different counters and ensure that each counter has its own unique label (this could be its name).

3. Implement a random number generator twice, one for each dice, ensuring that the number can only go between 1 and 6 in value. Make sure this number is clearly displayed and the numbers can be generated on the click of a button.

4. Implement a turn counter that only moves on to the next player once the current player's turn is over.

5. Attempt to move the counters around the board based on the dice numbers and the moves that are available to the computer.

6. Link the counters to each individual player, so the counters that are not in play at that moment cannot be played, meaning that only the counters that belong to the player who is taking their turn, can be moved. In addition to this, the computer will also be able to move spaces to knock off another counter (if the board allows, based on any given current stage.

7. Implement the basic Huckers Twist rules onto the board game to create a functioning Huckers game.
   - Start with one rule, ensure that the rule is working and then implement the others, one at a time. Make sure that all rules are tested using White Box Testing, and, any obvious code smells generated are quickly identified and ruled out to ensure that the code remains readable.

Primary Objective 2:

1. Conduct background research into 3D tic-tac-toe and its implementations. In the conclusion, explain how the research has helped to gauge ideas of how this can be implemented in the 3D Tic-tac-toe game that will be built as part of this project.

2. Create test runs of the method for a 2-player game using Java. Include this in the deliverables as the proof-of-concept programs to show a test run of the program working.

3. Create 3D planes in Unity to mark out the different layers of the board.
   – Ensure that Unity tools have been experimented with prior to this implementation, as outlined in Goal 1.1.

4. Create the different Cubes and place them in the correct positions using the 3D layers (planes) that have been created in the previous goal.

5. Create the scripts that change the colour based upon who has a turn at that moment.

6. Using a separate game manager script, implement a working 2-player 3D tic-tac-toe.
   - Use the proof-of-concept program for this, all details have been provided in that program.
   - Create a method within the game manager, or even a separate script, to make sure that a space that has already been taken cannot be "overwritten" by another counter, even if it is the same player.

Primary Objective 3:

1. Using the literature readings and research into the different implementations of Ludo & 3D tic-tac-toe, think of a possible strategy that can be used to tackle the problem of 3D tic-tac-toe.

2. For Huckers Twist, write a detailed description of what the computer player will need to do in order to get the counters round the board.

3. Where a proof-of-concept program is easily created (For 3D tic-tac-toe), implement the strategies on a proof-of-concept program.
   – Create a computerised player class which controls all moves that are available. Make sure it analyses the board in its current state, and makes an informed decision based on the status of the board at that given moment.

4. Begin the implementations of the 3D tic-tac-toe computerised player.
5. Start off using the strategies implemented, see if the proof-of-concept program works and then implement on the main game.

Revised Plan

In conclusion, taken together with the feedback that was received for the project plan. During the course of the first and second weeks, I chose to rewrite the project plan and change the structure of how things were going to be researched, programmed and implemented. The creation of the revised project plan was the best way to go about this.

Term 1

Week	**Tasks to Complete**	**Completed?**
Week 1	Creation of a project plan, research into the background of Huckers, games of Luck Vs. games of strategy.	Yes
Week 2	Creation of a revised project plan, make changes to the initial plan where necessary.	Yes
Week 3	Start creating a proof-of-concept program for the 3D tic-tac-toe implementation – A simple Java project which will calculate all possible wins.	Yes
Week 4	Complete the proof-of-concept Java program.	Yes
Week 5	Begin researching implementations of 3D tic-tac-toe and research into Unity C#.	Yes
Week 6	Conclude research into implementations of 3D-tic-tac-toe in Unity C#, conclude with how Unity C# will aid the project development as a whole.	Yes
Week 7	Begin creating a strategy for the Huckers Twist perfect player. Explain an outline of the 'perfect player' for both games and how this can be implemented.	Yes
Week 8	Conclude research into the 'perfect player' and how this gives rise to the point scoring system and the 'Rules of 3-2-1' *(See technical implementation and Strategy to achieving computerised player section).*	Yes
Week 9	Begin writing the interim report and code refactoring.	Yes
Week 10	Prepare & rehearse the interim presentation.	Yes
Week 11	Continue writing the interim report and submit the interim deliverables.	Yes

Term 2

Week	**Tasks to Complete**	**Completed?**
Week 1	Create a 3D-tic-tac-toe game for 2-players in Unity C#. Start experimenting with the 'Rules of 3-2-1'	Yes
Week 2	Create the class for the computerised player – Implementing the rules of 3-2-1	Yes
Week 3	Create the Huckers Twist board, begin with implementation of the basic rules of Ludo.	Yes
Week 4	Start implementing the rules (extending from the basic Ludo game) for Huckers Twist.	Yes
Week 5	Create the class for the 3D computerised player, initialize the class and create the main Huckers Twist game.	Yes
Week 6	Continue implementation of the Huckers Twist game. Implement the Huckers Twist computerised player (Allowing for a strategy to be identified against a set of rules).	Yes
Week 7	Continue implementation of the Huckers Twist computerised player (Ensuring testing is regularly carried out and documented).	Yes
Week 8	Continue implementation of the Huckers Twist computerised player (as outlined in Week 7).	Yes
Week 9	Code refactoring, ensuring everything is in place for final submission. Begin working on final report.	Yes
Week 10	Begin working on the final report (working towards the final submission).	Yes
Week 11	Finish off working on the final report, hand in the final submission.	Yes

In the revised project plan above, all of the Term 1 tasks have either been completed or require some more work. In comparison to the Interim submission, the revised plan now has a Term 2 completion column, meaning that the tasks from Term 2 have all been completed and the goals have been achieved.

Chapter 2: **Technologies, Techniques, Testing and Implementations**

Unity

Since its creation in 2005, Unity has been a very popular game engine. The easy 3D implementations of different games, combined with its straight-forward nature make it a straight-forward user interface to create new game objects, C# Scripts and the ease of the link between the two. For example, creating a simple C# script requires you to select ‘Assets’ from the menu bar then click ‘Create’ followed by ‘C# Script’. In order to link these together, you drag and drop the Script into the ‘Add component’ section of the object or a parent object (Which links the Script to all child objects as well). [5a]

C# is the programming language that Unity makes use of. The fact that C# is an object-based programming language makes it much more straight forward to assign objects to a particular variable, thus having a knock-on effect of how easy this makes the game plan.

C# - Conventions and Good Practices

Much like all programming languages, C# has specific conventions and good practices that should be adhered to when writing a program, these conventions and good practices come under the rather large category of coding standards, an essential rule to follow when developing any program, making the code readable to other developers and collaborators.

Access Modifiers [6]

It is always a good idea to use the correct access level modifiers. The different access levels in C# can be found below, alongside a description of their access levels:

Modifier	**Description**
`Public`	The code has unlimited access to any part of the code in the program.
`Private`	The variable can only be accessed by methods in the same class.
`Protected`	The variable can only be accessed by methods in the same class or classes that are inherited by it.
`Internal`	The variable can only be accessed by classes and methods within its own assembly, but not from outside the assembly.

Any code that does not have an access modifier will default to being built as a `private` variable. This can generate code smells and cause many different access issues with regards to the interaction between different classes and assemblies.

Naming Conventions [6]

Naming conventions refer to a set of standards that are followed by a team of developers or collaborators. They aid in ensuring the readability of the code through making the code clearer and, in some senses, more concise and easier to understand. The main benefit of the naming conventions are to ensure that developers and collaborators do not waste time in finding out where a colleague has left the code, and to ensure that the code is understood by the whole team.

A few important things to note, camelCase and PascalCase are widely used for naming conventions. Different programming languages have different naming conventions and, as such, they will have different methods of using the different cases. Pascal Case refers to the notion of every word in the name declaration starting with an uppercase letter.

In contrast, Camel Case refers to the first letter of the first word being lowercase, followed by any other words in the name declaration using an uppercase letter at the beginning.

Some of the popular naming conventions for C# include:

Naming Convention	**Example**	**Description**
Classes begin with a capital letter for each word in the class name definition.	`Public class MyNewClass() { }`	Every class name requires a capital letter which clearly displays each word that is defined in the class name. No spaces are allowed to be used, so the next best thing is for capital letters to separate different words upon reading the code.
Methods begin with a capital letter for each word in the name definition.	`Public void MyNewMethod() { }`	Every method name requires a capital letter for the same reason as stated above for classes.
Non-static fields should be camelCase.	`Private int myInteger = 0;`	Camel Case refers to the notion of the first word having a lowercase letter at the beginning. The rest of the words given in the definition of that particular field follow the same convention as the methods and classes (Pascal Case).
Static fields should be Pascal Case.	`Private static int MyInteger = 0;`	Pascal Case is used for any static fields that are defined.

During the course of this chapter, a lot of sentences were referenced to one particular number in the bibliography. This was number [8].

Software Engineering Techniques

Test Driven Development

Test Driven Development (TDD) is a vital strategy to ensure that the code is working properly (Readable and passes the tests). In this project, the test-driven development strategy has been adjusted and put into the form of a written table which logs the vast majority of the tests that have been completed.

Examples of where TDD has been used

Creation of any new methods in the method writing required the use of TDD. The following maps display two particular examples of the paths taken to pass a test with fully working code, along with the desired results.

Code	**Explanation**

Step 1:
`NewPlayerTurn = game.IncrementTurnCounter();`
 	
Step 2:
`public void IncrementTurnCounter() {`
`}`
 	
Step 3:
`Debug.Log(TurnCounter);`
 	
Step 4:
`public void IncrementTurnCounter() {`
`    TurnCounter++;`
`}`
 	
Step 5:
`NewPlayerTurn = game.IncrementTurnCounter();`
`Debug.Log(TurnCounter);`
	Step 1: Initially, this test would fail due to the fact that the `IncrementTurnCounter()` did not yet exist.
	
	Step 2: To pass the test the method was created.
	
	Step 3: Print the value of the value of the field to see if the value `TurnCounter` had been updated upon calling the method. This next test then fails because there was no content to the `IncrementTurnCounter()` method.
	
	Step 4: Content was added to ensure that the `TurnCounter` method was incrementing the `TurnCounter`. To pass the test, in the original method, a new line was added to ensure that the code was printing the correct `TurnCounter` value after calling the method.
	
	A visual comparison between the current value of `TurnCounter` and the previous value of `TurnCounter` was sufficient evidence to conclude that this piece of code was functioning in the way it should.

Step 1: `    public static void SetBoard(int x, int y, int z, char Contents)`	Step 1: `SetBoard(x, y, z, 'R').` This test initially fails due to the fact that the method did not yet exist. By creating the method, including the relevant parameters, this

```
    {
    }
```
 Step 2:
```
 public static void SetBoard(int
x, int y, int z, char Contents)
    {
        Board[x, y, z] =
Contents;
    }
```
	test aimed to ensure that no exceptions were thrown when the method was run, therefore, this test passed.
	
	Step 2: The next test ensured that value of the board at the location (x,y,z) was set to the value of 'R' (as shown in the previous test). The method was run as shown `SetBoard(x, y, z, 'R')` with the value of the index (x,y,z) being set to 'R'. This test then failed due to the fact that no values were being set in the method `SetBoard`. The code was then adjusted to ensure that the test passed when running the code `Board[x,y,z] == 'R'`.

Step 1:
```
public GameObject[]
GetBoardLocations()
    {
        return StartingBoard;
    }
```
 Step 2:
```
public GameObject[]
GetBoardLocations()
    {
        return BoardLocations;
    }
```
	Step 1: `game.GetBoardLocations()` was used in a line of code to retrieve the Board in it's initial state. Initially, this test failed as the method did not exist. This test only passed when returning a blank starting board.
	
	Step 2: Checking a value using `game.GetBoardLocations()[x,y,z]` found that the incorrect value of the Board at the point (x,y,z) was being provided. This was then updated and the BoardLocations in their current state were returned.

White Box Testing

The notion of White Box Testing is essentially testing the code to check its functionality with some prior knowledge / background knowledge of the program code. This can be particularly useful for refactoring the code as one can determine how much refactoring is required in order to give rise to 'Working code', this can work by determining how much code contains code smells and just how much refactoring is required to perform the same functionality.

Code	**Explanations**

`public void DecrementCountersOut(char Player)`
`    {`
`        switch (Player){ //Switch-case statement to switch to ensure that counters taken off have a decremented CountersOut label.`
`            case 'R': { RedCountersOut--; break; }`
`            case 'B': { BlueCountersOut--; break; }`
`            case 'G': { GreenCountersOut--; break; }`
`            case 'Y': { YellowCountersOut--; break; }`
`        }`
`    }`
	Taking the method to the left, initially this method was comprised of many individual comparison if-statements. This introduced many different code smells as there was multiple lines of repeated code in multiple different methods. By researching the functionality and implementation of a switch-case statement [19], it was safe to conclude that the switch-case statement as a refactor was a good solution to remove the repetitive if comparison statements.

`Debug.Log(turnCounter);`	Debug.Log() is a very useful tool when White Box Testing. As the programme is going along, ensuring that the right values exist is critical. The easiest way to do this was writing to the Debug Log. Once the test was had passed, there was then a

Use of a Version Control System (VCS)

As outlined in the initial 'Risks & Mitigations' document, Git as a Version Control System holds all of the required features that are needed in order to maintain working code, and regular backups to another machine (in this case a centralized server). This allows for the computer to work efficiently. In order to access Git as a whole, I will be using GitLab to do this. GitLab has multiple different features including a UI, two-factor authentication (aiding in a more secure development environment), as well as room for easy addition of extensions (E.g., the Visual Studio Code extension) [8].

Throughout this project. GitLab was used as provided by the Royal Holloway University Computing, Information Security and Mathematics (CIM) team.

Method of Software Engineering

Every software-based product requires a method of software engineering, in this particular project I have chosen to use the Waterfall methodology based on the time frame, the scale of the project, taken together with the fact that there is no specific end user to the game. This allowed for the project development to become more efficient as it linked nicely with the deliverable deadlines that are required for the project as a whole.

Design Patterns and Styling

Design patterns are incredibly important to use in the development of any project. They enable the programmer to make the code readable as well as adhering to a certain set of standards which contributes towards working code.

One of the major structural design patterns that was used in the creation of multiple classes, specifically the classes linked to Unity was the Singleton design pattern. By using static methods and classes, this meant that only once instance of a specific class could be created. By generating one specific class for the game, it meant that instantiation was not required in order to maintain the optimum level of working code.

Through the use of Unity, the Model-View-Controller (MVC) design pattern has also been used. The Unity-linked classes act as a Controller for the Unity objects, whilst the other classes which are not directly linked to unity view the details from the objects and base calculations off of the data retrieved.

StyleCop[20] – A style-checking and useful refactoring tool

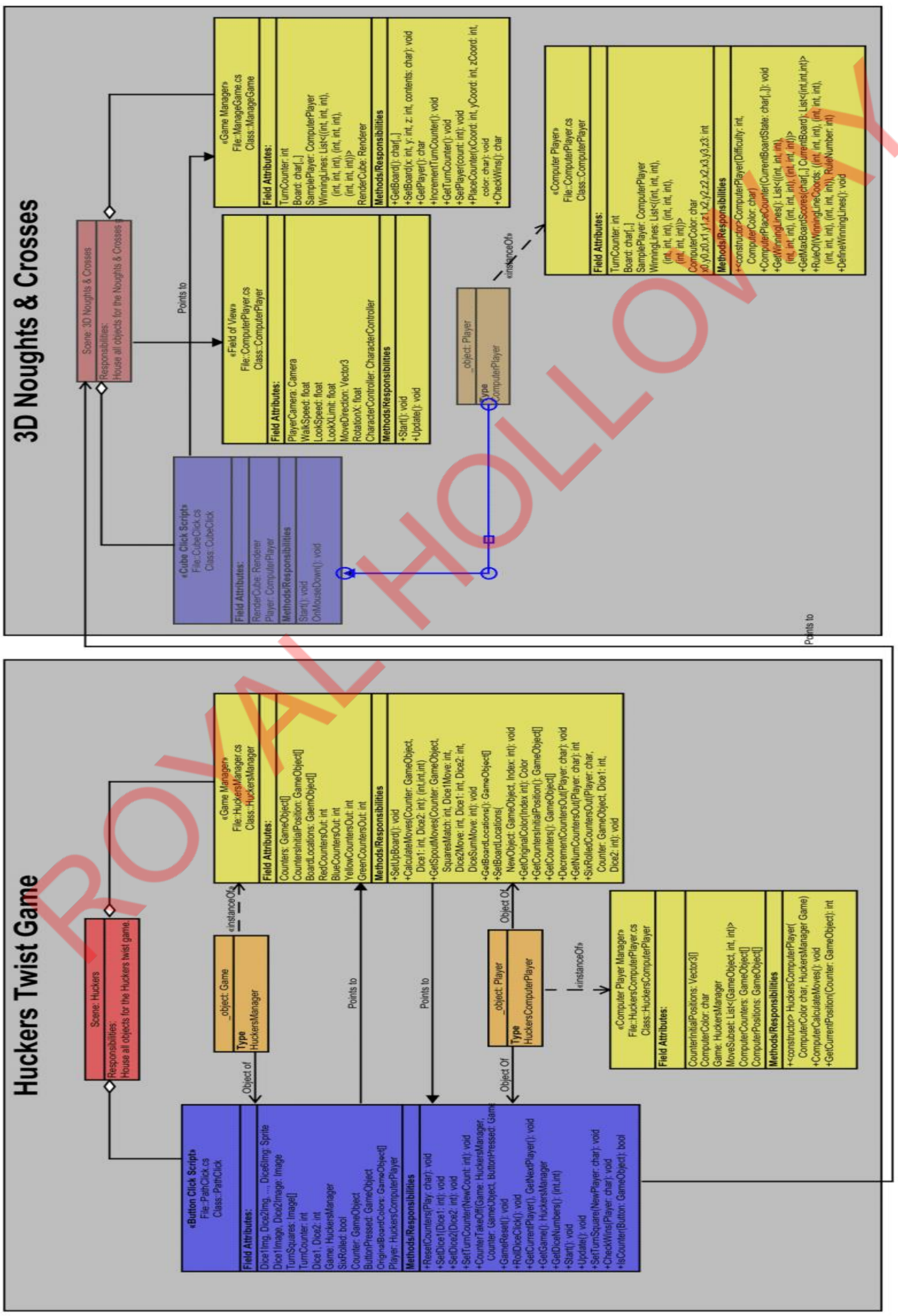
StyleCop is a commonly used style-checking tool created by Microsoft. Much like Checkstyle for Java, it analyses the source code and checks to make sure the correct indentation is implemented throughout. StyleCop can be particularly advantageous as it has multiple different tools to check that the code has been correctly formatted to meet particular conventions and clearly identifies issues to the user. Conventions can also be easily customised to ensure that code meets the standards as well as ensuring that the code is clear and readable, without loss of functionality.

StyleCop was originally created for Visual Studio 2008 and has had many updates. Unfortunately, development for the original StyleCop had development halted, however, a named successor for later versions of Visual Studio was StyleCopAnalyzers. Configurations to StyleCop have also been updated, enabling the extension to be used in later versions of Visual Studio (Such as Visual Studio 2019).

As well as StyleCop being a good tool for formatting code to meet coding standards and naming conventions, StyleCop can also produce documentation by utilising the feature with XML formatting.

UML diagram

UML diagrams are an excellent resource to developers. They allow the developer to visualise what is going on in the program, or just how the general structure may look. Therefore, by using a UML diagram, which shows the link between all the Scenes, Classes, Methods, etc. we can start to visualise the structure of the code. The UML model can become quite complex for larger scale development projects [9], so it is important to keep a track of every stage of the project. In order to create this UML diagram, a standalone version of the UMLet software was used.


Huckers Twist Game
Scene: Huckers
Responsibilities:
House all objects for the Huckers twist game.
«Button Click Script»
File::PathClick.cs
Class::PathClick
Field Attributes:
Dice1Img, Dice2Img, ..., Dice6Img: Sprite
Dice1Image, Dice2Image: Image
TurnSquares: Image[]
TurnCounter: int
Dice1, Dice2: int
Game: HuckersManager
SixRolled: bool
Counter: GameObject
ButtonPressed: GameObject
OriginalBoardColors: GameObject[]
Player: HuckersComputerPlayer
Methods/Responsibilities
+ResetCounters(Play: char): void
+SetDice1(Dice1: int): void
+SetDice2(Dice2: int): void
+SetTurnCounter(NewCount: int): void
+CounterTakeOff(Game: HuckersManager, Counter: GameObject, ButtonPressed: Game
+GameReset(): void
+RollDiceClick(): void
+GetCurrentPlayer(), GetNextPlayer(): void
+GetGame(): HuckersManager
+GetDiceNumbers(): (int,int)
+Start(): void
+Update(): void
+SetTurnSquare(NewPlayer: char): void
+CheckWins(Player: char): void
+IsCounter(Button: GameObject): bool
_object: Game
Type
HuckersManager
Object of
«instanceOf»
«Game Manager»
File::HuckersManager.cs
Class::HuckersManager
Field Attributes:
Counters: GameObject[]
CountersInitialPosition: GameObject[]
BoardLocations: GaemObject[]
RedCountersOut: int
BlueCountersOut: int
YellowCountersOut: int
GreenCountersOut: int
Methods/Responsibilities
+SetUpBoard(): void
+CalculateMoves(Counter: GameObject, Dice1: int, Dice2: int): (int,int,int)
+GetSpoutMoves(Counter: GameObject, SquaresMatch: int, Dice1Move: int, Dice2Move: int, Dice1: int, Dice2: int, DiceSumMove: int): void
+GetBoardLocations(): GameObject[]
+SetBoardLocations(NewObject: GameObject, Index: int): void
+GetOriginalColor(Index int): Color
+GetCountersInitialPosition(): GameObject[]
+GetCounters(): GameObject[]
+DecrementCountersOut(Player: char): void
+GetNumCountersOut(Player: char): int
+SixRolledCountersOut(Player: char, Counter: GameObject, Dice1: int, Dice2: int): void
Points to
Points to
_object: Player
Type
HuckersComputerPlayer
Object Of
Object Of
«instanceOf»
«Computer Player Manager»
File::HuckersComputerPlayer.cs
Class::HuckersComputerPlayer
Field Attributes:
CounterInitialPositions: Vector3[]
ComputerColor: char
Game: HuckersManager
MoveSubset: List<(GameObject, int, int)>
ComputerCounters: GameObject[]
ComputerPositions: GameObject[]
Methods/Responsibilities
+<constructor> HuckersComputerPlayer(ComputerColor char, HuckersManager Game)
+ComputerCalculateMoves(): void
+GetCurrentPosition(Counter: GameObject): int
3D Noughts & Crosses
Scene: 3D Noughts & Crosses
Responsibilities:
House all objects for the Noughts & Crosses
«Cube Click Script»
File::CubeClick.cs
Class::CubeClick
Field Attributes:
RenderCube: Renderer
Player: ComputerPlayer
Methods/Responsibilities
Start(): void
OnMouseDown(): void
Points to
«Field of View»
File::ComputerPlayer.cs
Class::ComputerPlayer
Field Attributes:
PlayerCamera: Camera
WalkSpeed: float
LookSpeed: float
LookXLimit: float
MoveDirection: Vector3
RotationX: float
CharacterController: CharacterController
Methods/Responsibilities
+Start(): void
+Update(): void
«Game Manager»
File::ManageGame.cs
Class::ManageGame
Field Attributes:
TurnCounter: int
Board: char[,,]
SamplePlayer: ComputerPlayer
WinningLines: List<((int, int, int), (int, int, int), (int, int, int), (int, int, int))>
RenderCube: Renderer
Methods/Responsibilities
+GetBoard(): char[,,]
+SetBoard(x: int, y: int, z: int, contents: char): void
+GetPlayer(): char
+IncrementTurnCounter(): void
+GetTurnCounter(): void
+SetPlayer(count: int): void
+PlaceCounter(xCoord: int, yCoord: int, zCoord: int, color: char): void
+CheckWins(): char
_object: Player
Type
ComputerPlayer
«instanceOf»
«Computer Player»
File::ComputerPlayer.cs
Class::ComputerPlayer
Field Attributes:
TurnCounter: int
Board: char[,,]
SamplePlayer: ComputerPlayer
WinningLines: List<((int, int, int), (int, int, int), (int, int, int), (int, int, int))>
ComputerColor: char
x0,y0,z0,x1,y1,z1,x2,y2,z2,x2,x3,y3,z3: int
Methods/Responsibilities
+<constructor>ComputerPlayer(Difficulty: int, ComputerColor: char)
+ComputerPlaceCounter(CurrentBoardState: char[,,]): void
+GetWinningLines(): List<((int, int, int), (int, int, int), (int, int, int), (int, int, int))>
+GetMaxBoardScores(char[,,] CurrentBoard): List<(int,int,int)>
+RuleOf(WinningLineCoords: ((int, int, int), (int, int, int), (int, int, int), (int, int, int)), RuleNumber: int)
+DefineWinningLines(): void
Points to

Technical Implementation

3D Tic-Tac-Toe

Throughout Term 1, multiple smaller programmes were written as proof-of-concept and test programs to see if the main strategies would work. This allowed time to experiment with the relevant tools as well as build a programme which would run successfully. Seen as 3D Noughts & Crosses was a new game to me, throughout Term 1, a basic 2-player 3D Noughts & Crosses was programmed in Java, as well as research being conducted, to show the main concept of the game. The goal of this proof-of-concept programme (as shown below) was to ensure that the rules of 3D Noughts & Crosses were understood and successfully implemented.

Java: The proof-of-concept program

Using Eclipse Integrated Development Environment (IDE), I was able to create a basic proof-of-concept program which allowed the implementation of the 3D Noughts & Crosses to be visualized by a series of four 4x4 grids (Each labelled with a different layer number). The aim of the program below was to develop this idea and ensure that all of the winning possible combinations were calculated.

Implementation of the board (as four 4x4 layers) printed at the beginning

```
public static void boardPrint(String[][][] boardSquares) { //Print out
the board in its current state. Once a counter is placed, this will
update based on the function placeCounter.
  System.out.flush();
  for (int i = 0; i < 4; i++) {
    System.out.println("Layer " + (i+1));
    System.out.println("|  " + boardSquares[0][0][i+1] + "  |  " +
boardSquares[0][1][i+1]+ "  |  "  + boardSquares[0][2][i+1]  + "  |  " +
boardSquares[0][3][i+1]  + "  |");
    System.out.println("_________________");
    System.out.println("|  " + boardSquares[1][0][i+1] + "  |  " +
boardSquares[1][1][i+1]+ "  |  "  + boardSquares[1][2][i+1]  + "  |  " +
boardSquares[1][3][i+1]  + "  |");
    System.out.println("_________________");
    System.out.println("|  " + boardSqua  res[2][0][i+1] + "  |  " +
boardSquares[2][1][i+1]+ "  |  "  + boardSquares[2][2][i+1]  + "  |  " +
boardSquares[2][3][i+1]  + "  |");
    System.out.println("_________________");
    System.out.println("|  " + boardSquares[3][0][i+1] + "  |  " +
boardSquares[3][1][i+1]+ "  |  "  + boardSquares[3][2][i+1]  + "  |  " +
boardSquares[3][3][i+1]  + "  |");
  }
}
```

This code demonstrates the printing of the board in its current state. Each position on the board is represented by an element in a 3D array. For example the back left corner on the bottom plane is named `boardSquares[0][0][0]` as it corresponds to the coordinate (0,0,0) in the 3-dimensional Cartesian plane. Each time a counter is placed, this is updated by clearing the console window, and printing out a new board with the updated counters, therefore, differing from the main program which will contain a UI with Scripting.

Checking to see if a space has already been taken

public static String[][][] placeCounter(String[][][] boardSquares, String whosTurn) {

```
  System.out.println("Please pick your space and write the numbers in
the form (X Y Z) excluding the brackets");
  Scanner sc = new Scanner(System.in);
  int xVal = sc.nextInt();
  int yVal = sc.nextInt();
  int zVal = sc.nextInt();
  if (boardSquares[xVal][yVal][zVal] != null) {
    System.out.println("Space is already taken");
    return boardSquares;
  } else {
    boardSquares[xVal][yVal][zVal] = whosTurn;
    return boardSquares;
  }
}
```

In Java, any variable that has not been initialised to a certain value will be automatically assigned a `null` value. As in the rules of 3D-Tic-Tac-Toe, a player can only enter a space that has not yet been taken by another counter. Therefore, placing a counter requires a check to be run first. Once the player has picked the space, this function will check whether an existing space is taken and will only place a counter if the space is not already taken.

Checking for a winning line (Sample of many functions of a similar nature)

```
for (int z = 0; z < 4; z++) { //Checks to be performed on each 2D flat
layer.
  for (int y = 0; y < 4; y++) {
    if (boardSquares[0][y][z] == boardSquares[1][y][z] &&
boardSquares[1][y][z] == boardSquares[2][y][z] &&
boardSquares[2][y][z] == boardSquares[3][y][z] &&
(boardSquares[0][y][z] == 'X' || boardSquares[0][y][z] == 'O')) { //
Check \/ direction of each layer
    return boardSquares[0][y][z]; //Return the character of the player
that won.
    } else if  (boardSquares[y][0][z] == boardSquares[y][1][z] &&
boardSquares[y][1][z] == boardSquares[y][2][z] &&
boardSquares[y][2][z] == boardSquares[y][3][z] &&
(boardSquares[y][0][z] == 'X' || boardSquares[y][0][z] == 'O'))
//Check -> direction for each layer.
        return boardSquares[y][0][z]; //Return the character of the
player that won.
  }
}
```

This section of code is a snippet of all the lines of code that detect a win in any direction. In the case above resembles the checks that need to be performed on each 2D flat layer. Iterating through these layers will detect a win every time a counter is placed down (by calling this method after the placement of each counter).

Unity – The experiments

As Unity was a fairly new programming development environment for me, research and experimentation with features was crucial in the development of this project. Therefore, by the help of a Unity tutorial [12], taken together with the Unity documentations [5], I was able to develop part of a working 3D-Tic-Tac-Toe game (the basic set up of the User Interface) as seen below.

In this UI, the following code is implemented in order to change the colour of the cube (Either Red or Blue) to determine a player's intention to place a counter on a certain Cube. This is use by using built-in Unity method `GetMouseDown`. and the method in the game manager which alternates between each player to determine who should take a turn at any given moment in the game.

Changing the colour of a Cube – Unity [12], [13]

```
if (Input.GetMouseButtonDown(0)) {
  renderCube.material.color = ManageGame.getPlayer() == 'R' ?
Color.red : ManageGame.getPlayer() == 'B' ? Color.blue : Color.green;
  ManageGame.setPlayer(1);
}
```

Changing the counter of the player

```
public static void setPlayer(int count)  {
  turnCounter += count;
  Debug.Log(turnCounter);
}
```

Retrieving the colour of the player whose turn it is

```
public static char getPlayer(){
  return turnCounter % 2 == 0 ? 'R' : turnCounter % 2 == 1 ? 'B' : 'E';
//Returns either Red, Blue or Error. Upon returning error the rest of the
functions should have error handling.
}
```

For this piece of code, the use of ternary operators [14] have made it very easy to concatenate if statements into one, easy-to-understand, line of code that performs the same function.

Experimentation with Field of View (Some code and inspiration taken from references [15] and [16]):

```
void Update() {
  Cursor.lockState = CursorLockMode.None;
  #region Handles Movment
  Vector3 up = transform.TransformDirection(Vector3.up);
  #endregion
  #region Up-Down Movement
  float upDownDirection = walkSpeed * Input.GetAxis("Vertical");
  if (Input.GetKey(KeyCode.Space) || Input.GetKey(KeyCode.W))
//Checking if the space bar is pressed {
    moveDirection = (up * walkSpeed);
  } else if (Input.GetKey(KeyCode.LeftShift) ||
Input.GetKey(KeyCode.S)){
    moveDirection = (up * -1 * walkSpeed);
  } else{
    moveDirection = Vector3.zero;
  }
  characterController.Move(moveDirection * Time.deltaTime);
  #endregion
        #region Handles Rotation and Movement
   if (Input.GetKey(KeyCode.R)) //If the right click has been
pressed.{
         rotationX += -Input.GetAxis("Mouse Y") * lookSpeed;
         rotationX = Mathf.Clamp(rotationX, -lookXLimit, lookXLimit);
         playerCamera.transform.localRotation =
Quaternion.Euler(rotationX, 0, 0);
   }
   #endregion
}
```

This implementation is a manipulation of the code that is in [15] and [16]. This will change the player's field of view based on the different inputs from the player. This particular piece of code (Which has been changed slightly) demonstrates that the player can move by using the shift key to move down and the space bar to move up.

The basic UML diagram displayed below has the main layout of implementation for the Huckers game. This is entirely subject to change and the full report will include many more details on the UML diagram. (This diagram below is only a start).

Chapter 3: **Technical Documentation**

Huckers with a Twist game

PathClick.cs

```
    public void Start()
    {
        TurnCounter = 0;
        SpriteImages[1] = Dice1Img;
        SpriteImages[2] = Dice2Img;
        SpriteImages[3] = Dice3Img;
        SpriteImages[4] = Dice4Img;
        SpriteImages[5] = Dice5Img;
        SpriteImages[6] = Dice6Img;
        Game.SetUpBoard();
        for (int TurnIndex = 1; TurnIndex < 4; TurnIndex++)
        { // Setting the Blue Square, Yellow Square and Green Square visibility to
false as red starts.
            if (TurnSquares[TurnIndex] != null)
            {
                TurnSquares[TurnIndex].enabled = false;
            }
        }
        OriginalBoardColors = Game.GetBoardLocations();
        for (int InitPosComp = 0; InitPosComp < 4; InitPosComp++)
        {
            Player.CounterInitialPositions[InitPosComp] =
Game.GetCountersInitialPosition()[InitPosComp + 13].transform.position;
        }
    }
```

This method is run as soon as the programme is opened. In this method, we have the Array of different Sprite Images for each Dice face being assigned to a specific position in the array. This array becomes useful when assigning specific images to the Dice image boxes. In addition to this, the code also calls the method `Game.SetUpBoard()` which initialises the game from an instance of the game manager class.
`Start()` is a built in Unity method that is required for Unity to register the `PathClick` class as a class that needs to be run on startup.

```
     public void Update()
    {
        try
        {
            Player.ComputerCalculateMoves();
            SetTurnSquare(GetCurrentPlayer());

            if (Input.GetMouseButtonDown(0))
            {
                ButtonPressed =
GameObject.Find(EventSystem.current.currentSelectedGameObject.name);
                if (ButtonPressed.name == "GameResetBtn")
                {
                    GameReset();
                }
                if (ButtonPressed.name == "ClearCounterBtn")
                {
                    Counter = null;
                    ButtonPressed = null;
                    for (int BoardLocationIndex = 0; BoardLocationIndex < 53;
BoardLocationIndex++)
                    {

Game.GetBoardLocations()[BoardLocationIndex].GetComponent<Image>().color =
OriginalBoardColors[BoardLocationIndex].GetComponent<Image>().color;
                    }
                }
                if (ButtonPressed.name == "ManualTurnEnd")
                {
                    Dice1 = 0;
                    Dice2 = 0;
                }
                if (ButtonPressed.name == "ForceWinBtn")
                { //Forcing a win in the game for the purposes of marking.
                    ForceWin();
                    CheckWins('R');
                }
                if (ButtonPressed.name == "RollDice")
                {
                    RollDiceClick();
                    if (!SixRolled && Game.GetNumCountersOut(GetCurrentPlayer()) ==
0)
                    {
                        SetTurnSquare(GetNextPlayer());
                        Counter = null;
                    }
                }
                else if (IsCounter(ButtonPressed))
                {
                    if (Counter != null)
                    {
                        CounterTakeOff(Game, Counter, ButtonPressed);
                        Counter = null;
                        ButtonPressed = null;
                    }
                    else if (Dice1 != 0 || Dice2 != 0)
                    { // Making sure that the Dice have been rolled before the turn
is taken.
                        Counter = ButtonPressed;
                    }

                    if (Counter != null && Counter.name.ToCharArray()[0] ==
GetCurrentPlayer())
                    {
                        (Dice1Move, Dice2Move, DiceSumMove) =
Game.CalculateMoves(Counter, Dice1, Dice2);
                    }
                    else
                    {
                        Counter = null;
                    }
                }
                else if (ButtonPressed.name != "RollDice" && ButtonPressed.name !=
"RedTurnSquare" && (ButtonPressed.name != "BlueTurnSquare" || ButtonPressed.name !=
"YellowTurnSquare" || ButtonPressed.name != "GreenTurnSquare"))
                {
                    if (Counter != null && ButtonPressed.GetComponent<Image>().color
== Color.magenta)
                    { //Moving a counter to the new space if it is a space the
counter can move to.
                        Counter.transform.position =
ButtonPressed.transform.position;
                        GameObject[] NewlyColoredBoard = Game.GetBoardLocations();
                        for (int SetColorIndex = 0; SetColorIndex <
Game.GetBoardLocations().Length - 1; SetColorIndex++)
                        { // Setting space colours back to white when counter has
moved.
                            if
(NewlyColoredBoard[SetColorIndex].GetComponent<Image>().color == Color.magenta)
                            { // Resetting all possible moves.

NewlyColoredBoard[SetColorIndex].GetComponent<Image>().color =
Game.GetOriginalColor(SetColorIndex);

Game.SetBoardLocations(NewlyColoredBoard[SetColorIndex], SetColorIndex);
                            }
                        }
                        for (int SetColorSpoutSquares = 1; SetColorSpoutSquares < 7;
SetColorSpoutSquares++)
                        { // Setting the spout squares back to their original
colour.
                            GameObject.Find("RW" +
SetColorSpoutSquares).GetComponent<Image>().color = Color.red;
                            GameObject.Find("BW" +
SetColorSpoutSquares).GetComponent<Image>().color = Color.blue;
                            GameObject.Find("YW" +
SetColorSpoutSquares).GetComponent<Image>().color = Color.yellow;
                            GameObject.Find("GW" +
SetColorSpoutSquares).GetComponent<Image>().color = Color.green;
                        }
                        if (Counter.transform.position.y ==
Game.GetBoardLocations()[Dice1Move].transform.position.y &&
(Counter.transform.position.x ==
Game.GetBoardLocations()[Dice1Move].transform.position.x ||
Counter.transform.position.x ==
Game.GetBoardLocations()[Dice1Move].transform.position.x - 10))
                        { // Setting the Dice1 = 0
                            if (Dice1Move != 0)
                            {
                                Dice1 = 0;
                            }
                        }
                        else if (Counter.transform.position.y ==
Game.GetBoardLocations()[Dice2Move].transform.position.y &&
(Counter.transform.position.x ==
Game.GetBoardLocations()[Dice2Move].transform.position.x ||
Counter.transform.position.x ==
Game.GetBoardLocations()[Dice2Move].transform.position.x - 10))
                        { // Setting the Dice 2 = 0
                            Dice2 = 0;
                        }
                        else if (Counter.transform.position.y ==
Game.GetBoardLocations()[DiceSumMove].transform.position.y &&
(Counter.transform.position.x ==
Game.GetBoardLocations()[DiceSumMove].transform.position.x ||
Counter.transform.position.x ==
Game.GetBoardLocations()[DiceSumMove].transform.position.x - 10))
                        { // Setting Dice 1 & Dice 2 = 0
                            Dice1 = Dice2 = 0;
                        }
                    }

                    if (Dice1 == 0 && Dice2 == 0)
                    {
                        SetTurnSquare(GetNextPlayer());
                    }
                    CheckWins(Counter.name.ToCharArray()[0]);
```

```
                        Counter = null; // Setting the counter value back to null to
ensure that the counter just moved is not moved again.
                    }
                }

            }
            catch (NullReferenceException)
            {
                //Do nothing with the exception, player clicked in wrong location on the
board.
            }
        }
```

This method is run on every update that Unity runs. Once the left click button is pressed on any particular space on the board, this method will call the Computer Player method to make a move, if the Current Player is the computer's colour.
For every human player (Red, Blue and Yellow): It will run the rules of the game, including the game manager's `CalculateMoves()` method. Once the counter is clicked, this method manages the move to any possible space. Then calls the method that takes off the counters, and causes the 'Knock Out'.

```
    public static void RollDiceClick()
    {
        System.Random Rnd = new System.Random();
        Dice1 = Rnd.Next(1, 7);
        Dice2 = Rnd.Next(1, 7);
        GameObject.Find("Dice1Image").GetComponent<Image>().sprite =
SpriteImages[Dice1];
        GameObject.Find("Dice2Image").GetComponent<Image>().sprite =
SpriteImages[Dice2];
        if (GameObject.Find("Dice1Image").GetComponent<Image>().sprite ==
SpriteImages[6] || GameObject.Find("Dice2Image").GetComponent<Image>().sprite ==
SpriteImages[6])
        {
            SixRolled = true;
        }
        if (Dice1 == 6 || Dice2 == 6)
        {
            SixRolled = true;
        }
}
```

Generating two separate random numbers (between 1 and 6 inclusive) using the Next method from an instance `Rnd` of the Random class, the two Dice numbers are given and the pictures are assigned a different face of a dice based on the number that was rolled. It does this by making use of the Image array that was initialised in the `Start()` method.

```
    public void SetTurnSquare(char NewPlayer)
    {
       if (TurnSquares[0] != null && TurnSquares[1] != null && TurnSquares[2] !=
null && TurnSquares[3] != null)
        {
            switch (NewPlayer)
            {
                case 'R':
                    {
                        TurnSquares[0].enabled = true;
                        TurnSquares[1].enabled = false;
                        TurnSquares[2].enabled = false;
                        TurnSquares[3].enabled = false;
                        break;
                    }

                case 'B':
                    {
                        TurnSquares[0].enabled = false;
                        TurnSquares[1].enabled = true;
                        TurnSquares[2].enabled = false;
                        TurnSquares[3].enabled = false;
                        break;
                    }

                case 'Y':
                    {
                        TurnSquares[0].enabled = false;
                        TurnSquares[1].enabled = false;
                        TurnSquares[2].enabled = true;
                        TurnSquares[3].enabled = false;
                        break;
                    }

                case 'G':
                    {
                        TurnSquares[0].enabled = false;
                        TurnSquares[1].enabled = false;
                        TurnSquares[2].enabled = false;
                        TurnSquares[3].enabled = true;
                        break;
                    }
            }
        }
    }
```

This method is called upon moving to the next player (When a player's turn has finished). Each player has a purple border outlining their counter. Using a switch-case comparative statement, this method enables and disables based on the array of possible turn squares provided. This method changes the image visibility as opposed to changing the image itself.

```
    private char GetNextPlayer()
    {
        switch (TurnCounter)
        {
            case -1:
                {
                    TurnCounter = 0;
                    return 'R';
                }

            case 0:
                {
                    TurnCounter = 1;
                    return 'B';
                }

            case 1:
                {
                    TurnCounter = 2;
                    return 'Y';
                }

            case 2:
                {
                    TurnCounter = 3;
                    return 'G';
                }

            case 3:
                {
                    TurnCounter = 0;
                    return 'R';
                }
        }

        return 'E';
    }

    private char GetCurrentPlayer()
    {
        switch (TurnCounter)
        {
            case 0:
                {
                    return 'R';
                }

            case 1:
                {
                  return 'B';
                }

            case 2:
                {
                    return 'Y';
                }

            case 3:
                {
                    return 'G';
                }
        }

        return 'E';
    }
```

Using the current private fields in the class to determine the next player and the current player respectively. These methods both make use of the switch-case comparative statements in order to ensure that there is as little code repetition as possible, whilst ensuring that the method performs for its intended purpose.

```
    public static void SetDice1(int D1)
    {
        Dice1 = D1;
    }
    public static void SetDice2(int D2)
    {
        Dice2 = D2;
    }
```

The relevant setting methods for the private fields `Dice1` and `Dice2`. The fields that save each of the dice numbers after the `RollDiceClick()` method has been run.

```
    public static void CounterTakeOff(HuckersManager Game, GameObject Counter,
GameObject ButtonPressed)
    {
        for (int CountersIndex = 1; CountersIndex < 17; CountersIndex++)
        { // Code block to take a counter off the board once landed on.
            bool KnockOff = ((ButtonPressed.name.ToCharArray()[0] == 'R' ||
ButtonPressed.name.ToCharArray()[0] == 'Y') && (Counter.name.ToCharArray()[0] == 'G'
|| Counter.name.ToCharArray()[0] == 'B'))
                || ((ButtonPressed.name.ToCharArray()[0] == 'G' ||
ButtonPressed.name.ToCharArray()[0] == 'B') && (Counter.name.ToCharArray()[0] == 'R'
|| Counter.name.ToCharArray()[0] == 'Y'));
            if (ButtonPressed == Game.GetCounters()[CountersIndex] && Counter !=
ButtonPressed)
            { // Sending the counter back to the starting squares.
                if (KnockOff)
                { // Code to detect a knock off for partnered players landing on top
of one another.
                    Counter.transform.position = ButtonPressed.transform.position;
                    Game.DecrementCountersOut(Counter.name.ToCharArray()[0]);
                    ButtonPressed.transform.position =
GameObject.Find(ButtonPressed.name + "S").transform.position; // Returning a counter
to its starting square
                }
                else
                { // Code for knock off of same colour.
                    float PosX = ButtonPressed.transform.position.x;
                    float PosY = ButtonPressed.transform.position.y;
                    float PosZ = ButtonPressed.transform.position.z;

                    ButtonPressed.transform.position =
GameObject.Find(ButtonPressed.name + "S").transform.position;
                    Counter.transform.position = GameObject.Find(Counter.name +
"S").transform.position;
                    Game.DecrementCountersOut(Counter.name.ToCharArray()[0]);
                    Game.DecrementCountersOut(ButtonPressed.name.ToCharArray()[0]);
                }
            }
        }
        PathClick.GetNextPlayer();
    }
```

Implementing the rules based on whether a counter has been landed on. If the counter is of a different team, it will knock off the counter. If the counter is on the same team, then it will cause a ‘Knock Out’ and both the Counter already in that position and the Counter that landed on it will be returned to the starting positions.

```
    public void GameReset()
    {
        Game = new HuckersManager();
        Game.SetUpBoard();
        for (int InitialCounterReset = 1; InitialCounterReset < 17;
InitialCounterReset++)
        {
            Game.GetCounters()[InitialCounterReset].transform.position =
GameObject.Find(Game.GetCounters()[InitialCounterReset].name +
"S").transform.position;
        }
        Dice1 = 0;
        Dice2 = 0;
        TurnCounter = -1;
        SetTurnSquare(GetCurrentPlayer());
    }
```

This method resets the game by taking the current board and moving all of the counters back to the starting squares. The Dice numbers are then reset back to 0 (the default value for an integer variable). Once this has taken place, the turn counter is reset to red and everything is done.

```
    public void CheckWins(char Player)
    {
        Vector3 Counter1pos = GameObject.Find(Player + "1").transform.position;
        Vector3 Counter2pos = GameObject.Find(Player + "2").transform.position;
        Vector3 Counter3pos = GameObject.Find(Player + "3").transform.position;
        Vector3 Counter4pos = GameObject.Find(Player + "4").transform.position;
        if (Counter1pos == Counter2pos && Counter2pos == Counter3pos && Counter3pos
== Counter4pos)
        {
            //Code to load scene.
            SceneManager.LoadScene("3DNCs");
        }
    }
```

Method that checks for a win in the main game (When the counter has been, once this has happened it will load in the Scene for 3D Noughts & Crosses.

```
    public void ForceWin()
    {
        for (int ForceTheWinIndex = 1; ForceTheWinIndex < 5; ForceTheWinIndex++)
        { //Set all of the red counters to trigger CheckWins to move on. **FOR THE
PURPOSES OF THE MARKERS ONLY**
            GameObject.Find("R" + ForceTheWinIndex).transform.position =
GameObject.Find("RW6").transform.position;
        }
    }
```

This part of the game automatically pushes Red to win by transporting all of the red counters to the home squares. Once this has taken place, the regular `CheckWins()` method detects a win.

HuckersManager.cs

```
    public (int, int, int) CalculateMoves(GameObject Counter, int Dice1, int Dice2)
    {
        int SquaresMatch = 0; // Creation of variable that is assigned to the
Counter's current location.
        int Dice1Move = 0, Dice2Move = 0, DiceSumMove = 0;
        for (int i = 1; i < 17; i++)
        {
            if ((Dice1 == 6 || Dice2 == 6) && Counter.transform.position ==
CountersInitialPosition[i].transform.position)
```

```
            {
                SixRolledCounterOut(Counter.name.ToCharArray()[0], Counter, Dice1, 
Dice2);

            }
        }

        for (int SquaresIndex = 0; SquaresIndex < BoardLocations.Length - 1; 
SquaresIndex++)
        { // Locating the counter on the board.
            if (Counter.transform.position != null & (Counter.transform.position == 
BoardLocations[SquaresIndex].transform.position))
            {
                SquaresMatch = SquaresIndex;
            }
            else if (Counter.transform.position != null & 
(Counter.transform.position.x == (BoardLocations[SquaresIndex].transform.position.x 
- 10)))
            {
                SquaresMatch = SquaresIndex;
            }
        }
        // Calculating moves for counters that are not up the spout (Winning 
squares).
        if (Counter.transform.position == 
BoardLocations[SquaresMatch].transform.position || Counter.transform.position.x == 
BoardLocations[SquaresMatch].transform.position.x - 10)
            {
                if (SquaresMatch + Dice1 >= 52)
                { // Calculating the moves based on the dice 1 roll
                    Dice1Move = SquaresMatch + Dice1 - 52;
                }
                else if (SquaresMatch + Dice1 < 52)
                {
                    Dice1Move = SquaresMatch + Dice1;
                }

                BoardLocations[Dice1Move].GetComponent<Image>().color = 
Color.magenta;
                if (SquaresMatch + Dice2 >= 52)
                { // Calculating the moves based on the dice 2 roll
                    Dice2Move = SquaresMatch + Dice2 - 52;
                }
                else if (SquaresMatch + Dice2 < 52)
                {
                    Dice2Move = SquaresMatch + Dice2;
                }

                BoardLocations[Dice2Move].GetComponent<Image>().color = 
Color.magenta;

                if (SquaresMatch + Dice2 + Dice1 >= 52)
                { // Calculating the moves based on the Dice Sum roll
                    DiceSumMove = SquaresMatch + Dice1 + Dice2 - 52;
                }
                else if (SquaresMatch + Dice1 + Dice2 < 52)
                {
                    DiceSumMove = SquaresMatch + Dice1 + Dice2;
                }

                BoardLocations[DiceSumMove].GetComponent<Image>().color = 
Color.magenta;
            }
            GetSpoutMoves(Counter, SquaresMatch, Dice1Move, Dice2Move, Dice1, Dice2,
DiceSumMove);
        return (Dice1Move, Dice2Move, DiceSumMove); // Returning all possible move
square locations to the PathClick class.
    }
```

Method that calculates possible moves after a specific counter has been pressed. The method takes in the Dice throws and returns a tuple with three elements. While doing this, it also lights up the squares of possible moves in the Magenta colour, ready for the relevant methods in the `PathClick` class to move the counter once a space in magenta has been pressed.

```
    public void GetSpoutMoves(GameObject Counter, int SquaresMatch, int Dice1Move,
int Dice2Move, int Dice1, int Dice2, int DiceSumMove)
    {
        char CounterColor = Counter.name.ToCharArray()[0];
        switch (CounterColor)
        {
            case 'R':
                {
                    if (SquaresMatch + Dice1 > 50)
                    {
                        BoardLocations[Dice1Move].GetComponent<Image>().color =
GetOriginalColor(Dice1Move);
                        Dice1Move = SquaresMatch + Dice1 - 50;
                        if (GameObject.Find("RW" + Dice1Move) != null)
                        {
                            GameObject.Find("RW" +
Dice1Move).GetComponent<Image>().color = Color.magenta;
                        }
                    }

                    if (SquaresMatch + Dice2 > 50)
                    {
                        BoardLocations[Dice2Move].GetComponent<Image>().color =
GetOriginalColor(Dice2Move);
                        Dice2Move = SquaresMatch + Dice2 - 50;
                        if (GameObject.Find("RW" + Dice2Move) != null)
                        {
                            GameObject.Find("RW" +
Dice2Move).GetComponent<Image>().color = Color.magenta;
                        }
                    }

                    if (SquaresMatch + Dice1 + Dice2 > 50)
                    {
                        BoardLocations[DiceSumMove].GetComponent<Image>().color =
GetOriginalColor(DiceSumMove);
                        DiceSumMove = SquaresMatch + Dice1 + Dice2 - 50;
                        if (GameObject.Find("RW" + DiceSumMove) != null)
                        {
                            GameObject.Find("RW" +
DiceSumMove).GetComponent<Image>().color = Color.magenta;
                        }
                    }

                    break;
                }

            case 'B':
                {
                    if (SquaresMatch < 12 && SquaresMatch + Dice1 > 12)
                    {
                        BoardLocations[Dice1Move].GetComponent<Image>().color =
GetOriginalColor(Dice1Move);
                        Dice1Move = SquaresMatch + Dice1 - 11;
                        if (GameObject.Find("BW" + Dice1Move) != null)
                        {
                            GameObject.Find("BW" +
Dice1Move).GetComponent<Image>().color = Color.magenta;
                        }
                    }

                    if (SquaresMatch < 12 && SquaresMatch + Dice2 > 12)
                    {
                        BoardLocations[Dice2Move].GetComponent<Image>().color =
GetOriginalColor(Dice2Move);
                        Dice2Move = SquaresMatch + Dice2 - 11;
                        if (GameObject.Find("BW" + Dice2Move) != null)
                        {
                            GameObject.Find("BW" +
Dice2Move).GetComponent<Image>().color = Color.magenta;
                        }
                    }

                    if (SquaresMatch < 12 && SquaresMatch + Dice1 + Dice2 > 12)
                    {
                        BoardLocations[DiceSumMove].GetComponent<Image>().color =
GetOriginalColor(DiceSumMove);
                        DiceSumMove = SquaresMatch + Dice1 + Dice2 - 11;
                        if (GameObject.Find("BW" + DiceSumMove) != null)
                        {
                            GameObject.Find("BW" +
DiceSumMove).GetComponent<Image>().color = Color.magenta;
                        }
                    }

                    break;
                }

            case 'Y':
                {
                    if (SquaresMatch < 25 && SquaresMatch + Dice1 > 25)
                    {
                        BoardLocations[Dice1Move].GetComponent<Image>().color =
GetOriginalColor(Dice1Move);
                        Dice1Move = SquaresMatch + Dice1 - 24;
                        if (GameObject.Find("YW" + Dice1Move) != null)
                        {
                            GameObject.Find("YW" +
Dice1Move).GetComponent<Image>().color = Color.magenta;
                        }
                    }

                    if (SquaresMatch < 25 && SquaresMatch + Dice2 > 25)
                    {
                        BoardLocations[Dice2Move].GetComponent<Image>().color =
GetOriginalColor(Dice2Move);
                        Dice2Move = SquaresMatch + Dice2 - 24;
                        if (GameObject.Find("YW" + Dice2Move) != null)
                        {
                            GameObject.Find("YW" +
Dice2Move).GetComponent<Image>().color = Color.magenta;
                        }
                    }

                    if (SquaresMatch < 25 && SquaresMatch + Dice1 + Dice2 > 25)
                    {
```

```
                        BoardLocations[DiceSumMove].GetComponent<Image>().color = 
GetOriginalColor(DiceSumMove);
                        DiceSumMove = SquaresMatch + Dice1 + Dice2 - 24;
                        if (GameObject.Find("YW" + DiceSumMove) != null)
                        {
                            GameObject.Find("YW" + 
DiceSumMove).GetComponent<Image>().color = Color.magenta;
                        }
                    }

                    break;
                }

            case 'G':
                {
                    if (SquaresMatch < 38 && SquaresMatch + Dice1 > 38)
                    {
                        BoardLocations[Dice1Move].GetComponent<Image>().color = 
GetOriginalColor(Dice1Move);
                        Dice1Move = SquaresMatch + Dice1 - 37;
                        if (GameObject.Find("GW" + Dice1Move) != null)
                        {
                            GameObject.Find("GW" + 
Dice1Move).GetComponent<Image>().color = Color.magenta;
                        }
                    }

                    if (SquaresMatch < 38 && SquaresMatch + Dice2 > 38)
                    {
                        BoardLocations[Dice2Move].GetComponent<Image>().color = 
GetOriginalColor(Dice2Move);
                        Dice2Move = SquaresMatch + Dice2 - 37;
                        if (GameObject.Find("GW" + Dice2Move) != null)
                        {
                            GameObject.Find("GW" + 
Dice2Move).GetComponent<Image>().color = Color.magenta;
                        }
                    }

                    if (SquaresMatch < 38 && SquaresMatch + Dice1 + Dice2 > 38)
                    {
                        BoardLocations[DiceSumMove].GetComponent<Image>().color = 
GetOriginalColor(DiceSumMove);
                        DiceSumMove = SquaresMatch + Dice1 + Dice2 - 37;
                        if (GameObject.Find("GW" + DiceSumMove) != null)
                        {
                            GameObject.Find("GW" + 
DiceSumMove).GetComponent<Image>().color = Color.magenta;
                        }
                    }

                    break;
                }
        }
        for (int SpoutSquaresIndex = 1; SpoutSquaresIndex < 6; SpoutSquaresIndex++)
        {
            if (GameObject.Find(CounterColor + "W" + (SpoutSquaresIndex + Dice1)) != 
null & GameObject.Find(CounterColor + "W" + SpoutSquaresIndex).transform.position == 
Counter.transform.position)
            {
                Dice1Move = SpoutSquaresIndex + Dice1;
                GameObject.Find(CounterColor + "W" + 
Dice1Move).GetComponent<Image>().color = Color.magenta;
            }
            if (GameObject.Find(CounterColor + "W" + (SpoutSquaresIndex + Dice2)) !=
null & GameObject.Find(CounterColor + "W" + SpoutSquaresIndex).transform.position ==
Counter.transform.position)
            {
                Dice2Move = SpoutSquaresIndex + Dice2;
                GameObject.Find(CounterColor + "W" +
Dice2Move).GetComponent<Image>().color = Color.magenta;
            }
            if (GameObject.Find(CounterColor + "W" + (SpoutSquaresIndex + Dice1 +
Dice2)) != null & GameObject.Find(CounterColor + "W" +
SpoutSquaresIndex).transform.position == Counter.transform.position)
            {
                DiceSumMove = SpoutSquaresIndex + Dice1 + Dice2;
                GameObject.Find(CounterColor + "W" +
DiceSumMove).GetComponent<Image>().color = Color.magenta;
            }
        }

    }
```

This method detects the moves on the spout squares. The regular `CalculateMoves` method takes in the necessary parameters and calculates moves for the main squares on the board. However, the CounterTakeOff method prevents the counters from moving around the board several times by redirecting the counter up the home squares of their relevant colour.

The CounterTakeOff method takes in the counter, the current moves that are available, the current dice numbers and the index of the square that the counter is currently on. It then sets the relevant spout squares to magenta if the dice numbers are not too large. If the dice numbers are too large, it will not display any counter. (As exact numbers are required in order to get each counter home).

HuckersComputerPlayer.cs

```
    public HuckersComputerPlayer(char ComputerColor, HuckersManager Game)
    {
        this.ComputerColor = ComputerColor;
        this.Game = PathClick.GetGame();
    }
```

Constructor for the `HuckersComputerPlayer` class. Takes in parameters and initialises the private fields of colour of the computer player (Green) and the instance of the game manager class.

```
   public void ComputerCalculateMoves()
    {
        for (int MapIndex = 40; MapIndex < 53; MapIndex++)
        { //Filling in the 13 position in the new mapping.
            ComputerPositions[MapIndex - 39] = Game.GetBoardLocations()[MapIndex];
        }
        for (int MapIndex = 0; MapIndex < 38; MapIndex++)
        { //Calculate the rest of the mapping of board spaces.
            ComputerPositions[MapIndex + 13] = Game.GetBoardLocations()[MapIndex];
        }
        for (int MapIndex = 51; MapIndex < 59; MapIndex++)
        { //Adding the spout squares to the mapping
            ComputerPositions[MapIndex] = GameObject.Find("GW" + (MapIndex - 49));
        }
        for (int CounterCount = 0; CounterCount < 4; CounterCount++)
        {
            this.ComputerCounters[CounterCount] = GameObject.Find("G" +
(CounterCount + 1));
            this.CounterInitialPositions[CounterCount] = GameObject.Find("G" +
(CounterCount + 1) + "S").transform.position;
        }
        if (PathClick.GetCurrentPlayer() == 'G')
        {
            int CountersOut = 4;
            int CountersHome = 0;
            int Dice1 = 0, Dice2 = 0;
            PathClick.RollDiceClick();
            (Dice1, Dice2) = PathClick.GetDiceNumbers();
            Debug.Log((Dice1, Dice2));
            if (PathClick.GetCurrentPlayer() == 'G')
            {
                for (int CounterCount = 0; CounterCount < 4; CounterCount++)
                { //Loop to check if each counter is at its starting position (Not
out on the board yet).
                    if (ComputerCounters[CounterCount].transform.position ==
CounterInitialPositions[CounterCount])
                    {
                        CountersOut--;
                    }
                }
                Debug.Log(CountersOut);
                for (int CounterHomeCheck = 0; CounterHomeCheck < 4;
CounterHomeCheck++)
                {
                    if (ComputerCounters[CounterHomeCheck].transform.position ==
GameObject.Find("GW6").transform.position)
                    {
                        CountersHome++;
                    }
                }

                if (CountersOut - CountersHome == 0 && Dice1 != 6 && Dice2 != 6)
                { //If no counters are out and no sixes are rolled, immediately move
to the next player.
                    PathClick.GetNextPlayer();
                }
                else if (CountersOut - CountersHome == 0 && (Dice1 == 6 || Dice2 ==
6))
                {
                    if (Dice1 == 6)
                    {
                        Dice1 = 0;
                        ComputerCounters[CountersOut].transform.position =
ComputerPositions[Dice2].transform.position;
                    }
                    else if (Dice2 == 6)
                    {
                        Dice2 = 0;
                        ComputerCounters[CountersOut].transform.position =
ComputerPositions[Dice1].transform.position;
                    }

                }
                else if (CountersOut - CountersHome > 0)
                {
                    PathClick.GetNextPlayer();
                    if
(!(ComputerPositions[GetCurrentPosition(ComputerCounters[CountersOut - 1]) + Dice1 +
Dice2] != null))
                    {
                        ComputerCounters[CountersOut - 1].transform.position =
ComputerPositions[GetCurrentPosition(ComputerCounters[CountersOut - 1]) + Dice1 +
Dice2].transform.position;
                    } else if
(ComputerPositions[GetCurrentPosition(ComputerCounters[CountersOut - 1]) + Dice1] !=
null && ComputerPositions[GetCurrentPosition(ComputerCounters[CountersOut - 1]) +
Dice2] != null)
                    {
                        if (Dice1 > Dice2)
                        {
                            ComputerCounters[CountersOut - 1].transform.position =
ComputerPositions[GetCurrentPosition(ComputerCounters[CountersOut - 1]) +
Dice1].transform.position;
                        } else
                        {
                            ComputerCounters[CountersOut - 1].transform.position =
ComputerPositions[GetCurrentPosition(ComputerCounters[CountersOut - 1]) +
Dice2].transform.position;
                        }
                    }
                    if (ComputerCounters[CountersOut - 1].transform.position ==
ComputerPositions[54].transform.position && (Dice1 == 1 || Dice2 == 1)) {
                        ComputerCounters[CountersOut - 1].transform.position =
GameObject.Find("GW6").transform.position;
                    }
                }
            }
        }
    }
```

The method that moves the computer counter around the board and implements the strategy of the computer player. To avoid a computer player knock out, the simplest way to achieve this strategy is to work through each counter. Once one counter is fully home, the next counter comes out and repeats the process. This process repeats until all four counters are home.

```
    public int GetCurrentPosition(GameObject Counter)
    {
        for (int CurPosIndex = 1; CurPosIndex < 59; CurPosIndex++)
        {
            if (Counter.transform.position ==
ComputerPositions[CurPosIndex].transform.position)
            {
                Debug.Log(CurPosIndex);
                return CurPosIndex;
            }
        }
        return -1;
    }
}
```

Method that takes in the current Counter and returns the current point as an integer index.

3D Noughts & Crosses

```
    public static void SetBoard(int x, int y, int z, char Contents)
    {
        Board[x, y, z] = Contents;
    }
    public static char[,,] GetBoard()
    {
        return Board;
    }
```

These methods are the relevant Getting and Setting methods for the main Board itself. The Board is a 3-dimensional array that consists of index points `Board[x,y,z]` when setting the board space, the integers (x,y,z) are all required in order to set the contents at a specific location.

```
    public static char GetPlayer()
    {
        return TurnCounter % 2 == 0 ? 'R' : TurnCounter % 2 == 1 ? 'B' : 'E'; //
Returns either Red, Blue or Error. Upon returning error the rest of the functions
should have error handling.
    }
```

`GetPlayer()` returns the current player. For the two-player version of 3D Noughts & Crosses, the value of `TurnCounter` just increments each turn and therefore it is sufficient to check for either an odd number or an even number. However, the `TurnCounter` is adjusted to only use the numbers 1, 2 and 3 depending on whether it is Red's turn, Yellow's turn, Blue's turn.

```
   public static char CheckWins()
    {
        // Defining the list that stores the winning combinations.
        WinningLines = SamplePlayer.GetWinningLines();
        int x0, x1, x2, x3, y0, y1, y2, y3, z0, z1, z2, z3;
        for (int k = 0; k < WinningLines.Count; k++)
        {
            x0 = WinningLines[k].Item1.Item1;
            y0 = WinningLines[k].Item1.Item2;
            z0 = WinningLines[k].Item1.Item3;
            x1 = WinningLines[k].Item2.Item1;
            y1 = WinningLines[k].Item2.Item2;
            z1 = WinningLines[k].Item2.Item3;
            x2 = WinningLines[k].Item3.Item1;
            y2 = WinningLines[k].Item3.Item2;
            z2 = WinningLines[k].Item3.Item3;
            x3 = WinningLines[k].Item4.Item1;
            y3 = WinningLines[k].Item4.Item2;
            z3 = WinningLines[k].Item4.Item3;
            if (Board[x0, y0, z0] == Board[x1, y1, z1] && Board[x1, y1, z1] ==
Board[x2, y2, z2] && Board[x2, y2, z2] == Board[x3, y3, z3] && (Board[x0, y0, z0] ==
'B' || Board[x0, y0, z0] == 'R'))
            {
                return Board[x0, y0, z0];
            }
        }

        return 'G';
    }
```

This method takes in an array of all of the possible winning lines and checks them for a possible line of the same colour. It does this by checking each of the winning line combinations in turn. If there is a win, the programme will return the colour of the winner. If not, it will return the value 'G' and the game will continue. If it returns 'D' it means a draw has occurred.

ComputerPlayer.cs

```
    private void DefineWinningLines()
    {
        // Initalising the list that stores the winning combinations.
        this.WinningLines = new List<((int, int, int), (int, int, int), (int, int,
int), (int, int, int))>();

        // Defining the diagonal winning lines.
        this.WinningLines.Add(((0, 0, 0), (1, 1, 1), (2, 2, 2), (3, 3, 3)));
        this.WinningLines.Add(((3, 0, 0), (2, 1, 1), (1, 2, 2), (0, 3, 3)));
        this.WinningLines.Add(((0, 0, 3), (1, 1, 2), (2, 2, 1), (3, 3, 0)));
        this.WinningLines.Add(((0, 3, 0), (1, 2, 1), (2, 1, 2), (3, 0, 3)));

        // Defining the vertically diagonal, through-plane lines.
        for (int y = 0; y < 4; y++)
        {
            this.WinningLines.Add(((0, y, 0), (1, y, 1), (2, y, 2), (3, y, 3)));
        }

        for (int y = 0; y < 4; y++)
        {
            this.WinningLines.Add(((3, y, 0), (2, y, 1), (1, y, 2), (0, y, 3)));
        }

        // Defining the horizontal diagonal lines.
        for (int x = 0; x < 4; x++)
        {
            this.WinningLines.Add(((x, 0, 0), (x, 1, 1), (x, 2, 2), (x, 3, 3)));
        }

        for (int x = 0; x < 4; x++)
        {
            this.WinningLines.Add(((x, 3, 0), (x, 2, 1), (x, 1, 2), (x, 0, 3)));
        }

        // Define single plane wins
        for (int z = 0; z < 4; z++)
        {
            this.WinningLines.Add(((0, 0, z), (1, 1, z), (2, 2, z), (3, 3, z)));
            this.WinningLines.Add(((3, 0, z), (2, 1, z), (1, 2, z), (0, 3, z)));
            for (int x = 0; x < 4; x++)
            {
                // Defining the same-plane straight winning lines.
                this.WinningLines.Add(((0, x, z), (1, x, z), (2, x, z), (3, x, z)));
                this.WinningLines.Add(((x, 0, z), (x, 1, z), (x, 2, z), (x, 3, z)));

                // Defining the vertical winning lines (Cross-plane).
                this.WinningLines.Add(((z, 0, x), (z, 1, x), (z, 2, x), (z, 3, x)));
                this.WinningLines.Add(((z, x, 0), (z, x, 1), (z, x, 2), (z, x, 3)));
            }
        }
    }
```

This method defines the winning lines for use in all methods that require the winning lines.

```
    public void ComputerPlaceCounter(char[,,] CurrentBoardState)
    {
        ManageGame.IncrementTurnCounter();
        List<(int, int, int)> BoardScores =
this.GetMaxBoardScores(CurrentBoardState);
        System.Random rndGen = new System.Random();
        int RandomElementIndex = rndGen.Next(BoardScores.Count);
        int x = BoardScores[RandomElementIndex].Item1;
        int y = BoardScores[RandomElementIndex].Item2;
        int z = BoardScores[RandomElementIndex].Item3;
        GameObject CubePicked = GameObject.Find("Cube_" + x + string.Empty + y +
string.Empty + z);
        Renderer RenderCube = CubePicked.GetComponent<Renderer>();
        if (this.ComputerColor == 'R')
        {
            RenderCube.material.color = Color.red;
            ManageGame.SetBoard(x, y, z, 'R');
        }

        if (this.ComputerColor == 'B')
        {
            RenderCube.material.color = Color.blue;
            ManageGame.SetBoard(x, y, z, 'B');
        }
    }
```

Taking in the board in its current state, this method uses many different sources (other methods and randomised scores) to calculate the move of the computerised player. If a winning line is impossible, it will assign a score of 0 to those winning lines. However, the rules can overwrite this and place the counter. The randomised point where the player makes their move is determined in this method.

```
    private List<(int, int, int)> GetMaxBoardScores(char[,,] CurrentBoard)
    {
        this.DefineWinningLines();
        List<(int, int, int)> BoardMaxScores = new List<(int, int, int)>();
        this.Board = ManageGame.GetBoard();
        this.BoardStates = new int[4, 4, 4];
        int CurrentMaxScore = 0;
        for (int x = 0; x < 4; x++)
        {
            for (int y = 0; y < 4; y++)
            {
                for (int z = 0; z < 4; z++)
                {
                    if (this.Board[x, y, z] == 'R' || this.Board[x, y, z] == 'B')
                    {
                        this.BoardStates[x, y, z] = 0;
                    }
                    else if (this.DifficultyLevel == 0)
                    {
                        this.BoardStates[x, y, z] = 1; // Assigning a score of 1 to
any space that has not already been taken by a counter.
                    }

                    if (this.DifficultyLevel == 1)
                    {
                        for (int a = 0; a < this.WinningLines.Count; a++)
                        {
                            this.RuleOf(this.WinningLines[a], 1);
                        }
                    }

                    if (this.DifficultyLevel == 2 || this.DifficultyLevel == 1)
                    { // Only run these methods for the Medium or Hard Player.
                        for (int i = 0; i < this.WinningLines.Count; i++)
                        {
                            this.RuleOf(this.WinningLines[i], 2);
                        }

                        for (int k = 0; k < this.WinningLines.Count; k++)
                        {
```

```
                                    this.RuleOf(this.WinningLines[k], 3);
                                }
                            }

                            CurrentMaxScore = CurrentMaxScore < this.BoardStates[x, y, z] ?
this.BoardStates[x, y, z] : CurrentMaxScore; // Checking each space to see if the
current score needs to be updated to a higher score.
                        }
                    }
                }

                for (int x = 0; x < 4; x++)
                {
                    for (int y = 0; y < 4; y++)
                    {
                        for (int z = 0; z < 4; z++)
                        {
                            if (this.BoardStates[x, y, z] == CurrentMaxScore)
                            {
                                BoardMaxScores.Add((x, y, z));
                            }
                        }
                    }
                }

                return BoardMaxScores;
            }
```

This method feeds into the `ComputerPlaceCounter`. It takes the board in its current state and generates a subset of positions with the maximum scores based on the rules of the computerised player.

```
    private void RuleOf(((int, int, int), (int, int, int), (int, int, int), (int,
int, int)) WinningLineCoords, int RuleNumber)
    {
        char[,,] Board = ManageGame.GetBoard();
        (int, int, int, char) Return = (0, 0, 0, 'G');
        int RedLineCounter = 0;
        int BlueLineCounter = 0;
        this.x0 = WinningLineCoords.Item1.Item1;
        this.y0 = WinningLineCoords.Item1.Item2;
        this.z0 = WinningLineCoords.Item1.Item3;
        this.x1 = WinningLineCoords.Item2.Item1;
        this.y1 = WinningLineCoords.Item2.Item2;
        this.z1 = WinningLineCoords.Item2.Item3;
        this.x2 = WinningLineCoords.Item3.Item1;
        this.y2 = WinningLineCoords.Item3.Item2;
        this.z2 = WinningLineCoords.Item3.Item3;
        this.x3 = WinningLineCoords.Item4.Item1;
        this.y3 = WinningLineCoords.Item4.Item2;
        this.z3 = WinningLineCoords.Item4.Item3;
        if (Board[this.x0, this.y0, this.z0] == 'R')
        { // Checking if there is a red or blue counter in the first space of the
winning line.
            RedLineCounter++;
        }
        else if (Board[this.x0, this.y0, this.z0] == 'B')
        {
            BlueLineCounter++;
        }
        else
        {
            Return = (this.x0, this.y0, this.z0, 'G'); // Returning the coordinates
of the blank point and the G empty space character.
        }

        if (Board[this.x1, this.y1, this.z1] == 'R')
        { // Checking if there is a red or blue counter in the second space of the
winning line.
            RedLineCounter++;
        }
        else if (Board[this.x1, this.y1, this.z1] == 'B')
        {
            BlueLineCounter++;
        }
        else
        {
            Return = (this.x1, this.y1, this.z1, 'G');
        }

        if (Board[this.x2, this.y2, this.z2] == 'R')
        { // Checking if there is a red or blue counter in the third space of the
winning line.
            RedLineCounter++;
        }
        else if (Board[this.x2, this.y2, this.z2] == 'B')
        {
            BlueLineCounter++;
        }
        else
        {
            Return = (this.x2, this.y2, this.z2, 'G');
        }

        if (this.Board[this.x3, this.y3, this.z3] == 'R')
        { // Checking if there is a red or blue counter in the fourth space of the
winning line.
            RedLineCounter++;
        }
        else if (this.Board[this.x3, this.y3, this.z3] == 'B')
        {
            BlueLineCounter++;
        }
        else
        {
            Return = (this.x3, this.y3, this.z3, 'G');
        }

        if (RedLineCounter == 3)
        {
            Return.Item4 = 'R';
        }
        else if (BlueLineCounter == 3)
        {
            Return.Item4 = 'B';
        }
        else
        {
            Return = (this.x3, this.y3, this.z3, 'G');
        }

        if (BlueLineCounter == RuleNumber)
        {
            if (this.Board[Return.Item1, Return.Item2, Return.Item3] == 'R' ||
this.Board[Return.Item1, Return.Item2, Return.Item3] == 'B')
            {
                this.BoardStates[Return.Item1, Return.Item2, Return.Item3] = 0;
            }
            else
            {
                this.BoardStates[Return.Item1, Return.Item2, Return.Item3] =
RuleNumber;
            }
        }
        else if (RedLineCounter == RuleNumber)
        {
            Debug.Log((Return.Item1, Return.Item2, Return.Item3));
            if (this.Board[Return.Item1, Return.Item2, Return.Item3] == 'R' ||
this.Board[Return.Item1, Return.Item2, Return.Item3] == 'B')
            {
                this.BoardStates[Return.Item1, Return.Item2, Return.Item3] = 0;
            }
            else
            {
                this.BoardStates[Return.Item1, Return.Item2, Return.Item3] =
RuleNumber;
            }
        }
    }
```

The `RuleOf` method applies the relevant rules to achieving a computerised player. It takes in four points as parameters which represent a winning line. Part of the refactor was ensuring that there was no repeated code. The `RuleOf` method therefore takes in an extra parameter as `RuleNumber` and uses the parameter to run the method on the Rule of 3 followed by the Rule of 2 and then the Rule of 1, depending on the difficulty of the computer player.

Three Player Computer – Optimisations explained

In order to create a Computerised Player for the 3D Noughts & Crosses game in three-player, an additional yellow colour had to be introduced and comparisons needed to be made in order to ensure that the yellow counters were taken into account as well.

In addition to this, the TurnCounter had to be optimized to ensure that the Game was checking for multiples of three as opposed to just odd and even numbers (as in the two-player game). The algorithms remained the same and performed well under the relaxed constraints of the TurnCounter. No major overhaul was required for the 'Rule of 3-2-1' algorithm, but the yellow colour had to be included in order to detect Yellow wins.

FieldOfViewControl.cs

```
//------------------------------------------------------------------------
// <author>David Playfoot</author>
// <copyright company="Royal Holloway" file="FieldOfViewControls.cs">
// All rights reserved
// </copyright>
//------------------------------------------------------------------------
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// Class that controls the up and down movement in the Noughts and Crosses game.
/// </summary>
[RequireComponent(typeof(CharacterController))]
public class FieldOfViewControls : MonoBehaviour
{
    /// <summary>Camera for player</summary>
    public Camera PlayerCamera;

    /// <summary>Standard speed of the movement</summary>
    public float WalkSpeed = 6f;

    /// <summary>Standard gravity constant gravity</summary>
    public float Gravity = 10f;

    /// <summary>Speed for the field of view - looking up and down</summary>
    public float LookSpeed = 2f;

    /// <summary>Standard limit constant for Look X and Y Limit</summary>
    public float LookXLimit = 45f;

    /// <summary>Direction of movement vector</summary>
    Vector3 MoveDirection = Vector3.zero;

    /// <summary>Rotation element</summary>
    float RotationX = 0;

    /// <summary>Controller for the character during the movement</summary>
    CharacterController CharacterController;

    /// <summary>
    /// Method that is run when starting up the program
    /// </summary>
    void Start()
    {
        this.CharacterController = this.GetComponent<CharacterController>();
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = true;
    }

    /// <summary>
    /// Method that is run when movement is required.
    /// </summary>
    void Update()
    {
        Cursor.lockState = CursorLockMode.None;
        Vector3 up = transform.TransformDirection(Vector3.up);
        if (Input.GetKey(KeyCode.Space) || Input.GetKey(KeyCode.W))
        { // Checking if the space bar is pressed
            this.MoveDirection = up * this.WalkSpeed;
        }
        else if (Input.GetKey(KeyCode.LeftShift) || Input.GetKey(KeyCode.S))
        {
            this.MoveDirection = up * -1 * this.WalkSpeed;
        }
        else
        {
            this.MoveDirection = Vector3.zero;
        }

        this.CharacterController.Move(this.MoveDirection * Time.deltaTime);
        if (Input.GetKey(KeyCode.R))
        { // If the right click has been pressed.
            this.RotationX += -Input.GetAxis("Mouse Y") * this.LookSpeed;
            this.RotationX = Mathf.Clamp(this.RotationX, -this.LookXLimit,
this.LookXLimit);
            this.PlayerCamera.transform.localRotation =
Quaternion.Euler(this.RotationX, 0, 0);
        }
    }
}
```

The field of view is a very important feature of the User Interface. Without the Field of View, the players would not be able to visualise every plane in the 3-Dimensional game. This code has been taken directly from a YouTube tutorial for games editing by a YouTuber called “All Game Things Dev”. https://www.youtube.com/watch?v=qQLvcS9FxnY

The code takes constant values into account, such as speed of movement, gravity in 3D space and uses the keyboard keys to adjust the UI up and down.

There are multiple additional different methods that have not been explained on here. This is because the method looks almost identical to a method that has been mentioned (For example, the relevant getting and setting methods). The most complex parts of the program have been mentioned in the above examples. Source code has been provided in an additional section of the folder that contains the final deliverables.

Chapter 4: **Strategy to achieving a computerised player**

3D Noughts & Crosses – How the game works

The implementation of 3D Noughts & Crosses works by making use of the x-y-z 3D coordinate plane. It relies on a singular coordinate (x,y,z) representing a space on the board. The entire board is represented as a set of points (x,y,z) on a zero-based coordinate system such that each point is represented as a series of integer coordinates. An example of this is as follows:

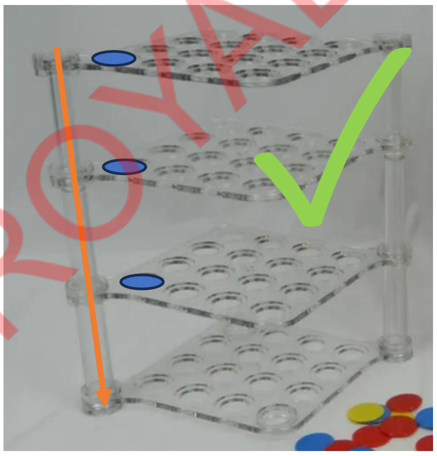
(0,0,0) represents the point that appears as at the back of the board in the bottom left-hand corner. This series of points progresses through to (3,3,3) which is in the top right-hand corner.

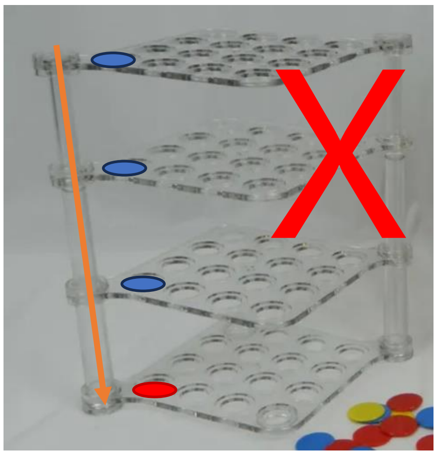
The 3D Noughts & Crosses programme that is written is heavily reliant on the finite number of winning lines [17]. There are 76 winning lines in 3D Noughts & Crosses, meaning that on a modern-day computer, it is safe to assume that checking these will be done in fairly quick time and, because of this, checking each of the possible winning lines for a win each turn is a feasible way of running the programme.

3D Noughts & Crosses – The 'Rules of 3-2-1'

The first rule, and arguably the most crucial rule for finding the wins or blocking the opposing player's win is the rule of 3. The rule of 3 will make use of a sequence of steps to detect 3 in a row of any colour, the fourth space must be an empty space. Once the sequence of steps has been executed, the computer player will take it's turn and strategically place its counter in a position that has been detected as empty by the "Rule of 3", therefore completing that row, column or diagonal.

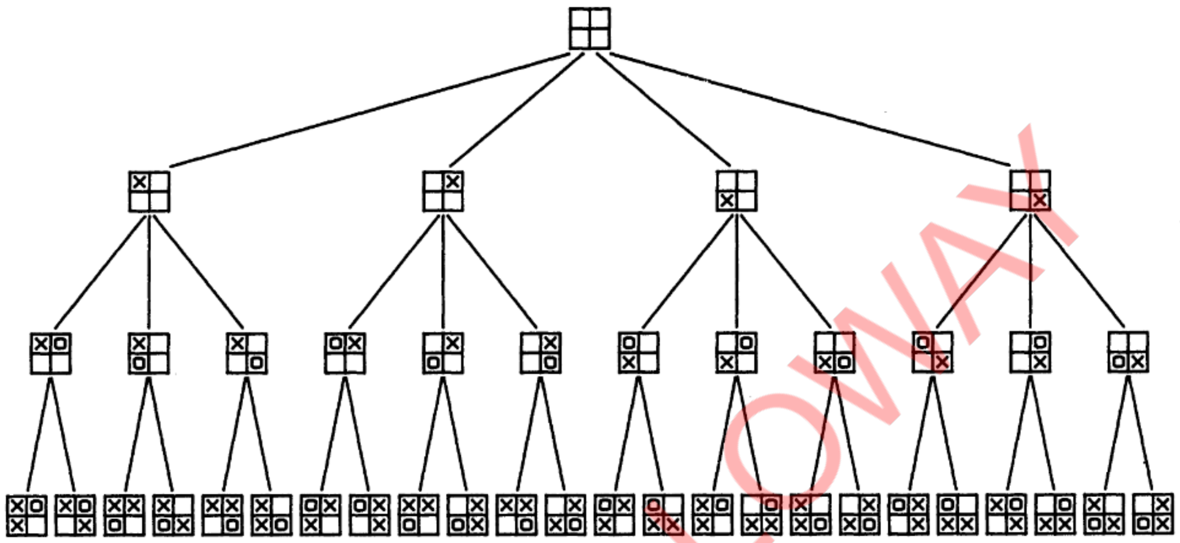
Example scenarios


[10] An example of where the rule of 3 will be used


[10] An example of where the rule of 3 will not be used

The Rule of 2 and Rule of 1 works in exactly the same way as the above rule of 3. However, there is a direct corollary of these two rules.

The following game tree [18] represents a brute-force search (much like the Rule of 3-2-1). The brute force method is a perfectly valid method within the modern-day computer world as it takes a lot less time to solve the issue of 3D Noughts & Crosses on the modern-day Computer with a fairly efficient algorithm. Due to the finite number of winning lines, the brute-force search is quite efficient, however, in a game where there is an arbitrarily large grid **n**x**n**x**n** where **n** is a particularly large number, this brute force algorithm can get complicated rather quickly.



[Figure 3]

This game tree can get much more complex with the 4x4x4 grid, as opposed to the strategic examples of certain sections of the main game that are shown in Figure 3. From the top to the bottom of the tree, in any direction, it denotes all of the possible moves from the empty squares to the squares with two crosses and one nought.

The one-line winning strategy

The Rule of 3-2-1 is heavily based around finding a one-line winning strategy for a 2-player game. In this circumstance, the Game Manager will force the player to make the first move and will iterate through each of the winning lines. It will then assign a score to the empty spaces depending on the importance of picking that move in order to force a win (For example, if a row of 3 is detected, the computer will need to go in that position in order to either block a win, or win itself). Once a score has been assigned, a subset of points that have been assigned the highest score will be generated, the Player will then pick a move from this subset of possible moves at random.

The Rule of 3-2-1 is one of the simple methods to implement a basic Computer Player for the 3D Noughts & Crosses.

In conclusion, the Rule of 3-2-1 will create a Computer Player that will analyze the winning lines to find a row of 3, followed by finding a row of 2 and then rows with a single counter to provide a subset of possible moves.

The two-line strategy - Does it work?

In 2D Noughts & Crosses, typically played on a 3x3 grid, there is the option to use a two-line strategy, which can ultimately guarantee a win against the opposing player or, at least, force a draw. This means that the expected Payoff for the player with the two-line strategy is always going to be at least 0, thus forcing the opposing player (The Second Player) to draw or completely lose (in the best case for the first Player).

3D Noughts & Crosses, in comparison, has a lot more winning lines, given the fact that the grid can take a line of four in any given direction. This makes it much more difficult to analyze the winning combinations by immediate inspection.

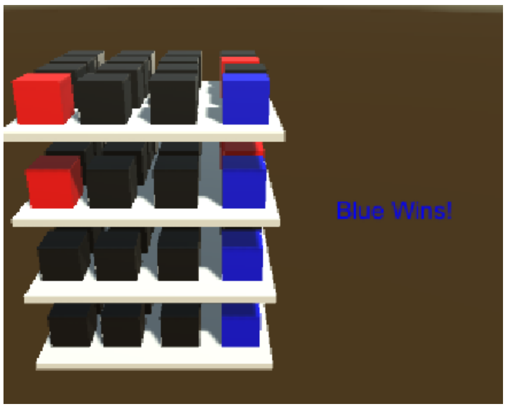
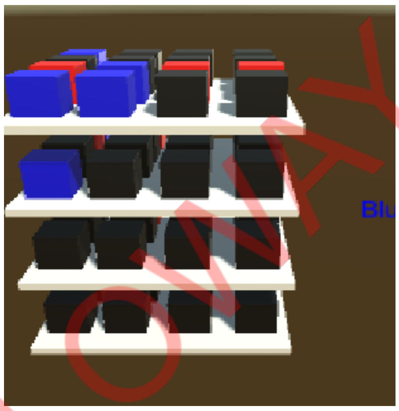
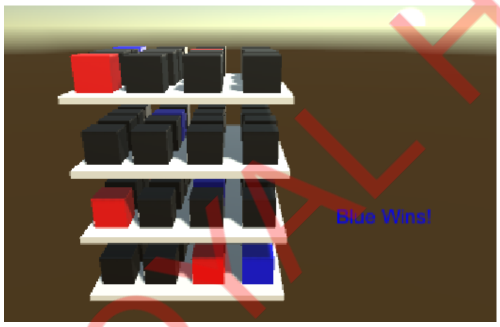
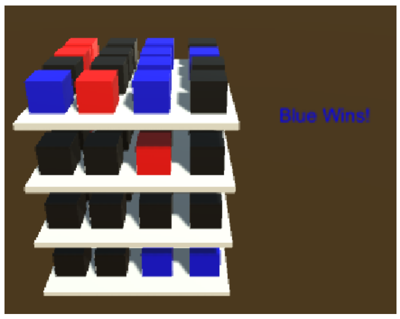
The Rule of 3-2-1 – Algorithmic Optimisation & The elimination of potentially repeated code.

During the course of the creation of the Rule Of 3-2-1, there were many different factors to consider with respect to the efficiency, the possible audience and the level of machine that this the code could work on. Due to the efficiency of modern-day computers, it was safe to assume that everyone will have a powerful enough machine to run an algorithm of at least O(n^2) in terms of Big-Oh notation, completing the algorithm of O(n^2) time complexity in quick time.

The Rule of 3-2-1 algorithm itself has a complexity of O(n) as the checks that need to be run only need to be run once for each winning line. The optimisation comes from combining the rule of 3-2-1 into one method that can be used for each of the Rules, as opposed to having a different method per class. This was a major refactor in the code as it removed a lot of repeated code that resulted from the test-driven development strategy.

Experiments for two player 3D Noughts & Crosses

As part of the experiment to see whether the strategy of the Computer Player was working, two computer players were pitted against each other. The examples of the two players are displayed below, with the specific scenario listed.

Scenario 1 – A randomised Computer Player (Red) Vs. A medium-level Computer Player (Blue)	
 In this example, blue has taken a very visible winning line strategy.	 In this example, blue has taken a less visible winning line, however, the winning line that is given is at the back
 A winning line was created on one of the diagonal lines (cross-plane).	 A winning line was created on a vertical line in the top 2D plane.

As shown in this image, the blue player wins this game in each of the given examples (A separate run of the board on each turn).

Taking all of these examples together, amongst other examples that have been conducted throughout program testing, against a random computer player, the strategic computer player which is implementing the Rule of 3 and the Rule of 2 will always react to the current state of the board and win against a random game play.

Three Player Strategy and adjusting for three-player game play

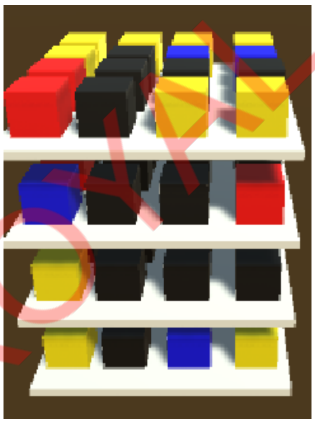
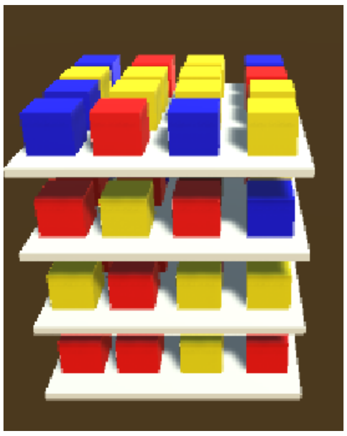
Three Players with the Rule of 3-2-1

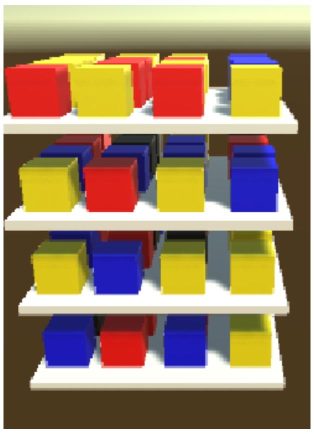
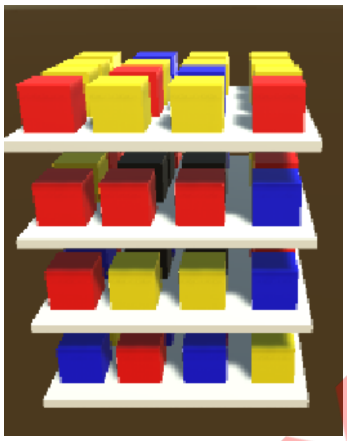
The Rule of 3-2-1 was purposely created in such a way that, regardless of the colour of the Computerised Player, a successful winning game would be played when playing against a Computer Player. By using the colours Red, Blue and Yellow, the same board was used with different optimisations which ensured that a Computerised Player could function properly and a successful working game could be played.

The three-player strategy works in a very similar way to that of the two Player. The players take it in turns to fill in the board and the Computerised Players the rule of 3-2-1 algorithm. The Rule of 3-2-1 is the same algorithm as that of the two-player computerised player.

The optimisations that are required in order to turn the two-player game into the 3D version of the same game were:

- Instead of checking just for a space taken by red or a space taken by blue. The programme was optimized to include checking for a space of Yellow as well.
- Instead of the `TurnCounter` variable checking for an odd or an even number to determine a turn, it now checks for whether the number is one, two or three (Therefore corresponding to either Red, Blue or Yellow respectively). Once Yellow has taken a turn, the counter is then set back to 1.
- The CheckWins() method has to check for Yellow cubes as well as blue and red.

Scenario 2 – A randomised Computer Player (Red) Vs. A Medium Computer Player (Blue) Vs. A Hard Computer Player (Yellow)	
	
In this example, yellow takes quite a prominent winning line.	In this example, Blue takes a very subtle winning line. Blue has taken the diagonal.

 In this example, blue takes a fairly obvious winning line strategy. On the third plane up, blue takes a horizontal line which is visible in this example.	 In this final example, blue takes a fairly obvious winning line on the bottom plane. Although it cannot be seen in this example very well, blue uses a singular plane diagonal to traverse the bottom plane.

Overall, in the three player 3D Noughts and Crosses, using the examples given in Scenario 2, the conclusion that I can take from this is that the Blue Player hard computer wins most of the time. This is due to the subset of possible positions being identical as soon as the Rule of 3 and the Rule of 2 are implemented.

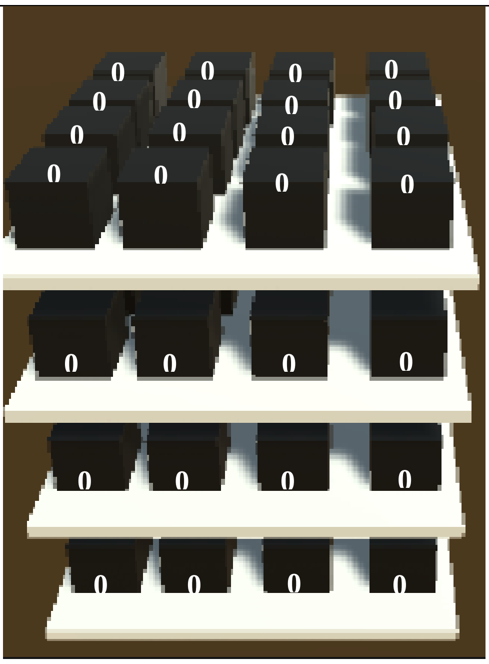
The Hard Computer Player – Too much strategy?

During the course of the writing of 3D Noughts & Crosses strategic computerised player, the most interesting part of the code to conclude was the idea that the computer can be creating a large subset of points to choose from based on the strategy it is using at any one point. This, therefore, means that the Rule of 1 (Taken from the Rule of 3-2-1) may produce a much larger subset, thus meaning that the computer player has multiple possible moves to choose from.

Taking this argument, we may be able to conclude that the notion of ‘Too much strategy’ has occurred. Throughout the course of the many experiments with the two-player game, during the process of eliminating the spaces with the least score, the computer generates a subset with those points of the highest score. This subset becomes increasingly larger, meaning that the computer has multiple spaces on the board to pick from. By using the “Rule of 1” specifically, this creates a large subset of strategic points that the computer can choose, but will also, consequently, effect the positioning of the successive moves from the computer player.

Analysis of Rule of 3-2-1 with labels on the User Interface

Step 1: Each of the blank spaces starts off with a score of 0.

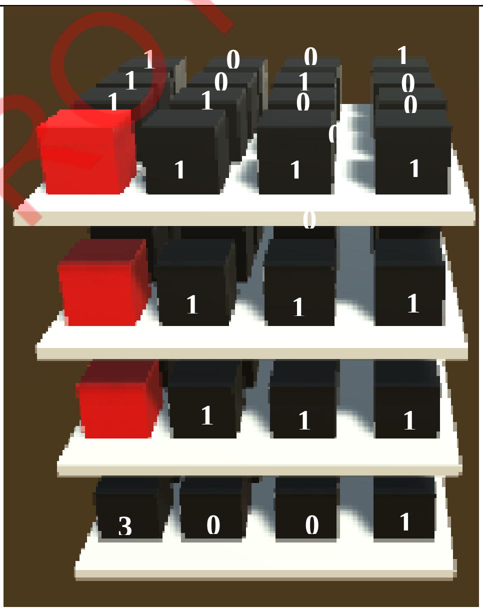


In this example, all of the squares start off with a starts off with an assigned score of 0. In order to do this, an O(n^3) algorithm was assigned to do this.

If the blank board occurs, the subset `MaxScore = {`*`List of all possible cubes on the board`*`}`. This means that the starting square is entirely random.

Code Snippet

```
for (int x = 0; x < 4; x++) {
    for (int y = 0; y < 4; y++) {
        for (int z = 0; z < 4; z++) {
            if (Board[x, y, z] == 'R' || Board[x, y, z] == 'B' || Board[x, y, z] == 'Y')
            {
                BoardStates[x, y, z] = 0;
            }
```



This next happens when red takes 3 moves and the rule of 3-2-1 algorithm is run (See Snippet 1). The code then creates a subset using the following code and picks a random location from that subset. In this case, the subset generated will be `MaxScore = {(3,0,0)}`. This contains one location based on the (x,y,z) coordinates for the board.

Snippet 1 exhibits all of the code required to run this step.

The Rule of 3-2-1 has been run three times for each space.

Huckers with a Twist - Computer Player Strategy

Due to the twisted version of the traditional Huckers game, the Knock Outs are a big risk to the play of the game, one wrong move and the game could change completely. In order to prevent ‘knock outs’, the strategy of the computer player is quite a simple one. Unlike the original game where it is encouraged to have two counters land on top of one another, in the Twist version, it is in fact quite the opposite, these details are outlined in the rules document listed in the Appendix.

The computer player will roll the dice and move one counter around the board at a time, once this counter is home, the computer player will get the next counter out and move it round the board. This process repeats for all counters.

The computer player also reacts to counters being taken off by detecting whether a counter has returned to its starting position by detecting that the counter is back at home and continuing with the game play by rolling a six to get that counter out onto the board again.

Chapter 5: **Risks & Mitigations**

In every project, including this one, there are always chances for things to go wrong, some of which are unavoidable. These 'Risks' are based around what could happen if things do not quite go to plan. In addition to this, with every risk, comes a potential mitigation to prevent the fallout from the risk from being unmanageable.

Hardware and Software Failures

Whilst undertaking any long-term project with a plan that lasts for at least two months, there is always a chance that any piece of Hardware could fail, or any piece of Software (or indeed project files) could become corrupted, all of which could result in a catastrophic loss of project work. To avoid this, the project should be regularly backed up to a different device and everything should be kept up to date.

A Version Control System (VCS) should be used to ensure that every change made to the code should be regularly updated and stored on an additional machine.

No Software works one-hundred percent of the time, and as such, every software should always be kept up to date. The risk of not keeping software up-to-date could be the reason behind bugs and code smells appearing in the program, even later on down the line. Keeping softwares up-to-date at all stages of development can prevent this from happening.

Bad Code-Report Balance

The report writing in any project has the same importance as the code that is being written. Unbalanced report-code ratios can result in too much code that hasn't got its own detailed explanation. On the contrary, have too much report writing can result in over-explaining and leaving a lack of time to plan and write the code for the project. In order to mitigate this, a well-rounded project plan should be created and implemented that contains a good balance of research, code writing and report writing.

Should a bad code-report balance occur in the first term, use the second term to catch up on the programming and put the research to one side, it will have been completed by this point.

Lack of compatibility

The "Lack of compatibility" risk can occur for multiple different reasons. This mitigations for this risk are similar to that of the "Software Failures" risk. All software should be maintained and kept up-to-date with the latest versions, or at least the version that corresponds to your program code. However, in addition to out-of-date client software, the audience for this game will have to be taken into account. Huckers has ties to the Royal Navy, dating as far back as WW2[1] and may have clients that have out-of-date machines. Choosing the correct (not necessarily the latest) version of the software being used is going to be instrumental in ensuring that this risk does not occur.

Plagiarism – Resulting in loss of credibility

With all research, references have to be provided to works that have been referenced in the research. Failure to do this can result in a complete loss of credibility for the project that has been produced. In order to mitigate this, ensure that there is a clear list of references provided and regular referencing to the bibliography has been made in the correct locations.

Loss of Time

There could be multiple reasons for the loss of time arising. There are many risks behind the "Loss of Time" umbrella. Some of these risks include: Insufficient project plan, Insufficient Final Report, Insufficient code writing resulting in lack of functionality. In order to mitigate this, a project plan with realistic timelines should be set, taking into account all of the major and minor milestones. In addition, a diary and notebook should be sought in order to keep a steady progress log. The use of a site such as Trello can also be useful to keep steady progress logs of individual tasks, as with any website, there is always the possibility of downtime and/or complete shutdown during the course of the project.

Imbalance of Luck and Strategy in the game

With games in general, they have to be appealing to the client as no one is going to realistically play a game that is deemed as boring. The risk resulting from an imbalance of luck and strategy is that the client will find the game boring and will stop playing. A possible mitigation for this is ensuring the game has enough room for critical thinking to be enhanced, whilst including the element of the dice, and pot-luck in that sense.

Chapter 6: **Personal Evaluation**

The following section is about comparing the initial objectives with the final objectives and seeing how well the programme functions in line with the main objectives. As mentioned above, all of the main objectives were met to a certain level.

Primary Objective 1: *"Create a Huckers board game, combining all of the rules of the standard Huckers game into one game".*

The primary objective of creating the Huckers board game worked successfully, algorithms for each rule were implemented and were optimized to work in the cases for all counters. A very easy-to-use User interface was provided and ensured that each of the Players were clear about when their turn was. In addition to this, blobs were removed and 'Knock Outs' were implemented to comply with the rules of the Twist game.

This section could have further algorithm optimisations by using the buttons on the board instead of the x-y-z locations on the board, therefore making it much easier to visualise how the code is working, whilst ensuring the same functionality. However, due to the development of the Computerised player and meeting other primary objectives, code that worked was higher on the priority list for this objective.

Primary Objective 2: *"Creating 3D Noughts & Crosses using a 4x4x4 grid"*

This objective was met very well. A friendly User-Interface was provided with a series of four planes, each containing a 4x4 grid, thus making up the 4x4x4 board. The Unity 3D Cube tools were used in order to generate each square. Instead of using 'X's and 'O's (Much like the regular Noughts & Crosses in 2D space), different colours of red, blue and yellow have been utilised to ensure that the UI remains clear and concise.

The working two-player game was developed in a relatively short space of time, which allowed a bit of extra time to work on the primary objectives, while still ensuring that the game maintained optimal functionality.

Overall, this Primary Objective was met very successfully. By taking the necessary steps in the secondary objectives (Broken down primary objectives into smaller manageable steps), this was very helpful.

Primary Objective 3: "Implementing a computerised player for both of these games".

The vast majority of time spent on this project was spent ensuring that the algorithms were fully functional when detecting moves for the 3D Noughts & Crosses. As 3D Noughts & Crosses is entirely based on the strategy of the players, it was very interesting to see how the computers played against each other in the experiments. By assigning a score to each of the spaces in the board upon the computer detecting each turn, the use of a list to act as a smaller subset was the best course of action to determine where the computer was placing the game, thus creating potentially pivotal points and multiple winning strategies within the game.

By generating the Rule of 3-2-1 as a simple solution, it was clear to see how the game operated and how effective, simple algorithms could easily be built on to ensure that there was a much more secure way of almost guaranteeing a win against that of the Human Player, as the Human Player is more than likely going to make at least one error.

As Huckers is similar to a game of Ludo, with a few added rules, the points-based system was a useful tool to implement a Computerised strategy. The computerised player looks through each counter and decides which counters to move. The computerised player using a point scoring system is made much easier to visualise given that there are a maximum of twelve possible moves (A maximum 3 moves for every counter based on the dice throws) for any given player.

Primary Objective 4: ***"Figure out a 'twist' to the Huckers game which allows for the two games to be combined into one larger game"***

Thinking of a link to join the two games together to give the individual unique selling point of the game was quite a difficult part. However, by joining the two games together, the Huckers game has an exciting twist that means that winning the game may not be as easy as in the original game. In order to do this, by making use of individual scenes in Unity and switching between scenes at the end became very useful.

In the standard game of Huckers, there is a particular rule which states that, once you have all of your counters on the corresponding centre square, it then allows you to throw for your team mate (Diagonally opposite player), upon throwing a six, you are then able to do this on successive turns.

In the exciting twist, instead of attempting to throw the dice for a six, the player must play a game of 3D Noughts & Crosses against a computer player in the "Hard mode". Therefore, incorporating the 3D Noughts & Crosses into the main game of Huckers and making it increasingly more difficult to win the game. In addition to this, the game also allows people to improve their strategic gameplay by playing a game that is entirely strategic, with luck having no effect on the outcome of the final game.

Overall, this objective was met by linking the 3D Noughts & Crosses into the game once the game of Huckers has almost ended and, hence, adding the strategic element to the game.

The General use of Unity in aiding the development of the project:

Throughout the course of Term 1, the time spent on the project was spent on writing proof-of-concept programs and researching the games and technologies required to build a successful working project. This became a very good use of time as it stood the project in a really good position to start the programming. Throughout the Christmas Break and the second term, the programming was able to become the main focus, this combined with innovative ways to solve the problem of the 3D Noughts & Crosses for the Computer.

By using the Unity UI, building the game Interface itself was made much easier due to the relevant tools that Unity provides. Two main examples of Unity tools that have aided the development of the game are:

- The ability to have multiple different scenes for one project has really helped in the development of the project as it means that each game can be separated out into its own individual scenes. Changing between scenes is also possible, therefore allowing for the easy scene change from the Huckers game to the 3D Noughts & Crosses game.

- The vast ranges of objects in the UI allowed for an easy generation of the grid for the Huckers game, and the creation of the four planes in 3D space that allows for the generation of the Huckers board. Throughout the project, it has always been the case that the clickable field for the objects has also been very accurate, meaning that buttons within close proximity do not have an overlapping field for which the button can be pressed.

Learning new coding conventions for C# was made much easier given the fact that I had prior knowledge of Java. As C# has multiple similarities to Java’s syntax (Such as method declarations and the use of curly brackets to define different classes, methods and routines), this made it a very useful programming language.

Chapter 7: **Professional Issues**

Computational Efficiency and inter-generational games

With all online games, any machine that is running them must meet certain resource requirements. Defining the expectations of any particular machine can be challenging and, particularly when it comes to games that provide a User Interface, a number of factors have to be considered. Three examples of these types of issues are the Graphics Processing Power of the machine, the efficiency and number of cores in the Central Processing Unit (CPU). The original significance of the Huckers with a Twist game was to provide a modern solution to a problem that has been occurring for quite some time, the lack of unique board games that do not have as much of a degree of popularity as the rest of the more common board games, such as original Ludo or Snakes & Ladders. As mentioned in the initial introduction, this board game can be used as an inter-generational part of a family and can therefore bring people closer together. Generally, a young person is thought to be more likely to have a more powerful computer than someone who may be much older, therefore, by providing a simple, easy-to-use user interface, it becomes much easier to navigate around the game.

Correlation between – Loneliness and Computer Games

Technology has now become a major factor in everyday life. Single-player online games can increase the amount of time one can spend on any one activity. After a study conducted into the effects of computer gaming and loneliness (Conducted by Zaliha Tras)[21], it was concluded that *"A positive linear relationship"* was observed between the loneliness of adolescents and amounts of time spent on computer games. After taking this into careful consideration, a version of the online game was not published for the reason that it would require people spending increased amounts of time playing online without the requirement of human interaction. In addition to this, there is only an option given to have a maximum of one computer players at any given moment for the Huckers game, meaning that two additional human players would be required.

Physical Health and Computer Gaming – The correlation

There are many health concerns that surround excessive use of the computer. A study conducted by Ofcom into the time spent on Computer games [22] concluded that *"for those aged 13-64"* the average time spent on computer games was *"Seven and a half hours"* per week with 85% of people stating that their children spend longer online. According to NHS UK, there is a strong correlation between a lack of exercise and conditions like high blood pressure. Due to this, the game of Huckers is intentionally short. After conducting a study into playing a series of Huckers games with family members, I have concluded that the average game of Huckers takes between 30 minutes and 45 minutes, indicating that a standalone game of Computerised Huckers, generally, will not exceed 1 hour.

Unity Development – Controversies surrounding updates

Throughout the second half of 2023, the company behind the Unity development environment faced extreme backlash when making drastic changes to the pricing of the Unity tool. This led, to a lot of developers feeling betrayed as the developers felt that the company maintained a lack of transparency with respect to the pricing of the runtime environment feature [24].

Upon the abrupt changes to the pricing plans being made, two weeks of chaos emerged for the company which led to the resignation of the Chief Executive at the time. This bad press could have an impact on the distribution of Unity games and payments that needed to be made in order to keep any Unity-made game running. This would have made it increasingly more difficult, however, on 22nd September 2023, the Unity Create president issued an apology[24] and Unity developers were informed that Unity were no longer forcing the developers that use the platform to pay for the Runtime, a big backtrack to the original drastic price plan changes.

Quality Control Measures

In order to ensure that the game provided an easy-to-use simple User interface that is required by the users of the software, some points had to be taken into account:

Bug Testing and Fixes:

In order to ensure that there were as little bugs as possible in the software system, there were weeks allocated in both the initial and revised plan for the project. These weeks were broken down into smaller, manageable tasks which allowed for fixes to take place for the features that were not working at the required level to meet the demands of the other features of the program.

Some of the time that was spent working on bug fixes just made sure that features could be adjusted so that one feature would not have an impact on the other features' functionality.

A Robust Testing strategy:

In order to ensure that there was a robust testing strategy in place, multiple testing strategies were used in order to ensure that the tests were well written and made sure that the methods were performing the task that they were programmed to do. Test Driven Development was used wherever possible in order to create methods based on fields that already existed. (For example, the `SetDice1` method would be created using Test-Driven development to pass the tests firstly using a constant value, then setting the field to the value of an input), there are some more examples above on Pages 14 and 15.

Wherever Test-Driven Development was not possible or was not deemed to be a sufficient use of time and resources, White Box Testing was used in order to test different values of the variables at different stages of the program runtime. The `Debug.Log()` method was very useful in ensuring that the variables were clearly defined and any bugs were quickly identified and fixed.

Compatibility

Although C# is not tied to the Windows Operating System, it requires the .NET framework, the framework that is provided by Microsoft for multiple different programming languages. This heavily limits Unity and the platforms it can run on.

Unity can only be installed on Windows or MacOS, meaning that the commonly used Linux operating systems, such as Ubuntu, cannot run the Unity interface. Unfortunately, due to the constraints that Unity provides on the free version, the project requires a Windows 64-bit operating system in order to run the program. However, research conducted into how many people own a Windows PC concluded that Microsoft dominate the market, with approximately 72% of the PC market being Windows-based operating systems [25]. In the future, I will aim to extend this to enable people using other Operating Systems to use it.

Intellectual Property and Copyright

As with any project, many works of other people may be used in order to generate a successful piece of work. As well as referencing the work, copyright must be considered and appropriate accreditations must be provided. In order to use some works, such as images, copyright statements must be read and adhered to in order to use the images provided online.

For example, the main Ludo board in the Huckers twist game required a Ludo board as a background. To ensure that the board that the board was appropriate for use, the copyright statement had to be read and stated that "You can copy, modify, distribute and perform the work, even for commercial purposes, all without asking permission" meaning that the board was suitable for use in the context of the project.

For all of the counters that are on the board and dice images: All of these images were obtained on a website called clker.com from a publisher called OCAL[27]. This website obtains their images from openclipart if the publisher is OCAL, the publisher of the image from Openclipart.org these were covered under the openclipart.org disclaimer which reads *"We try and make it clear that you can use all Clipart from Openclipart even for unlimited commercial use"*. [27]

Accessibility & Dependencies

Both compatibility and accessibility play a very important role in the development of the project. Easy to use up and down controls such as the space bar and the left shift key enable the user to see many of the spaces in the User Interface. Combined with the controls and a clear document on the "Rules and Directions for Use", the 3D Noughts & Crosses also makes use of colours instead of the traditional 'X's' and 'O's' of 3D Noughts & Crosses in order to create the board in the way that it should be created. By using colours, this made the three-player experimental scenario much easier to program and, therefore, use.

Dependencies of the project or prerequisites are essential for the deployment of the project. Clearly defining dependencies in the User manual means that the user knows what kind of requirements are needed prior to purchasing any game. Normally, for games that run on a specific platform, such as Steam, requirements are set by the game manager (in the case of Steam, it would be Steam themselves) in order for all of the games to be accessible to all people.

New coding conventions for a fairly new programming language

With learning a new programming language comes the different conventions and the different ways to use the specific programming language. One of the main hurdles to overcome in this project was ensuring that there was a good balance of code and write up, however, in order to do this, the programming language of C# had to be widely understood. After learning the Java programming language, C# was fairly similar and had some quite nice similarities between the two, however, the fundamental differences were also quite clear. E.g. Using PascalCase (as mentioned above) Vs. using CamelCase (same as PascalCase but the letter is smaller at the beginning).

Unity tools

Learning the Unity tools took up a significant amount of time, slightly more time than was initially expected which meant that the project plan was pushed back slightly. This was an unavoidable step as it was to ensure that all the tools needed were ready to use. As Unity was a completely new technology, it took a long time to learn where everything was and to set up the Unity user interface in a way that was effective to the development of the project.

Managing Exceptions in Unity

One of the main issues that was faced throughout the development of the project was the fact that Unity and Visual Studio (for the C# Scripts) could become unsynchronised when it came to handling errors in the program and exceptions. This meant that, if an error was fixed in the code, Unity would sometimes miss the save and the project would either have to be restarted (worst case) or another edit and resave would have to be performed to trigger Unity's script updates to occur. Sometimes, if an error was fixed from a different script, the original error could be quite difficult to pinpoint and fix if this occurred, as well as knowing that an error had been fixed but Unity had not detected that. All of this led to more time being taken up by fixing errors than was possibly needed.

Use of Test-Driven Development (TDD) – The testing strategies

The nature of TDD is that you have to write tests for a piece of code that is not yet written to ensure that you get the desired functionality that you need. Unfortunately, trying to link C# version of Unit testing (NUnit tests) to the individual scripts in C# became a bigger task than originally anticipated and therefore quite a large task which was possibly not as imperative as producing results from the experiments that needed to be performed, especially when trying to produce a piece of code that was quite simple. Therefore, a manual TDD strategy was adopted as I went along. This strategy, I found, was just as effective due to the use of the `Debug.Log()` command to display results from a test, along with writing temporary variables for method calling to a method that needed to be created. This, combined with manual inspection of the variables, all contributed to a Test-Driven development strategy.

A combination of Test-Driven development and white box testing was used in order to ensure that there was a rigorous testing strategy throughout the development of the project.

Unity and GitLab (The Version Control System)

There were a number of clashes with Git and Unity libraries throughout the project development. As library files were updating over a long period of time, it was difficult to commit library files, then push them without a new update occurring in the local changes to the project. To overcome this, the directory path of folder that contained all of the libraries was placed in the `.gitignore`. However, throughout the project, the odd different Library file within that directory would still cause issues when committing even though it was in the `.gitignore` folder. To overcome this, manual changes had to be written and a few manual merges had to take place between the remote repository and the local repository for a specific branch.

As a direct result of this, issues between the remote repository branches came to light. This was due to the many commits made to the branches in quick succession without the merge taking place. This, therefore, meant that there was an issue with the heads of branches. To solve this, manual merges crossing branches and head resets had to take place. However, this could sometimes cause issues with the `.csproj` assembly files.

Unity Assembly Files

Sometimes, the C# assembly files could become detached from the main Library files and project as a whole. This resulted in errors, such as different libraries not being recognised or different dependencies appearing as if they were obsolete or no longer existed. This meant that certain scripts that had always worked would suddenly stop working as they did not recognise one of the Library files that were being used. This specifically happened to the `UnityEngine.UI` dependency and therefore UnityEngine.UI was not being picked up as a package that was contained within the Library UnityEngine namespace. In order to repair this, deleting the .csproj file and allowing it to automatically regenerate a new file fixed this error as it meant that the new assembly was being built around the correct Libraries and dependencies that C# required in order to run the individual scripts.

Appendix

Project Diary (Timeline of events so far)

Term 1

Fresher's Week (Week commencing the 18th September):

- Met with supervisor meeting meet with supervisor.
- Started work on the 'Abstract' & 'Project Plan' sections of the project plan document.

Week 1 (Week commencing 25th September):

- Drafted up the risks and mitigations section of the project plan.

Week 2 (Week commencing 2st October):

- Finalised the details and submitted the project plan document.

Week 3 (Week commencing 9th October):

- Researched into 3D Noughts & Crosses implementations and the history of Huckers.

Week 4 (Week commencing 16th October):

- Started creating an implementation of a 'proof-of-concept' Java program in Eclipse.

Week 5 (Week commencing 23st October):

- Started experimenting with Scripts, tools and researching into which Unity items would best aid the development of the game as a whole.
- Microsoft Teams meeting with personal adviser.

Week 6 (Week commencing 30th October):

- Carried on experimenting with the different tools in Unity. Created the first objects for the board.

Week 7 (Week commencing 6th November):

- Concluded experiments in Unity and started making strategies for the implementation of a computerised player.
- Concluded the strategies that would work the best.

Week 8 (Week commencing 13th November):

- Working on the field of view experimentation.

Week 9 (Week commencing 20th November):

- Completing the field of view experimentation.

- Began work on the Unity 3D Noughts & Crosses board.

Week 10 (Week commencing 27th November):

- Continued with work on the interim report.

Week 11 (Week commencing 4th December):

- Completed the interim deliverables and submitted them.

Term 2

Week 12 (Week commencing 8th January):

- Check and act on feedback for interim deliverables.
- Continue work on 3D Noughts & Crosses two player game.

Week 13 (Week commencing 15th January):

- Complete work on 3D Noughts & Crosses Unity game (for two-players).
- Start work on the Huckers UI, importing the board and counters.

Week 14 (Week commencing 22nd January):

- Finish UI of Huckers and implement basic game of Ludo.
- Start working on the Rule of 3-2-1, using the test programs to implement a working algorithm for a random computer player and then build up strategy.

Week 15 (Week commencing 29th January):

- Start adding the rules of Huckers to the basic game of Ludo.
- Build up levels of difficulty in the random computer player for 3D Noughts & Crosses.

Week 16 & 17 (Week commencing 5th February and week commencing 12th February):

- Continue the implementation of the rules in order to obtain a working version of Huckers.

Week 18 (Week commencing 19th February):

- Write Up some experiments on 3D Noughts & Crosses computer players. Make a conclusion about the findings of the experiments. (For example, making computer players play against each other in order to verify the strategic algorithms winning strategy).

Week 19 (Week commencing 26th February):

- Write Up some experiments on the working Huckers game. Fixing any bugs.

Week 20 (Week commencing 4th March):

- Fix any bugs with the Huckers game, ensuring that the game play is running well. (For example, the multiple sixes deducting multiple turns from the turn counter, ensuring that the game runs with as few errors as possible).

Week 21 (Week commencing 11th March):

- Begin work on the final report (Start setting up a contents page to make sure that there is a set criterion of what to include).

- Meeting with adviser to ensure that the project is on the right track and progress level is maintained.

Week 22 (Week commencing 18th March):

- Continue work on final report. Report to adviser on progress updates.

Week 23 (Week commencing 25th March):

- Continue to work on final report. Report to adviser on progress updates.

- Submit a first draft of final report to adviser, gain feedback.

Week 24 (Week commencing 1st April):

- Record video of code running.

- Write Up a user manual and adjust the README.txt file in GitLab.

- (Friday 5th April) submit the final deliverables.

Markers' Use Button

The Huckers Twist game can take quite a long time to get round the board. Each player takes it in turns to move once or twice depending on sixes being thrown and when all four counters have to be home before moving on to the 3D Noughts & Crosses, this can take some time. To prevent unnecessary time being spent on getting all of the counters round on the board, a Markers' button was created. The markers' button takes all four counters belonging to the red player and moves them to the home square, thus triggering the `CheckWins()` method to detect the red player has won the Huckers Twist game.

User manual

Prerequisites and Dependencies

Due to the use of Unity on a Windows machine for the development of the project, a computer running either a 32-Bit or a 64-Bit Windows Operating System is required to run the game.

Instructions for Use

Step 1: Enter the Final Submission folder.

Step 2: Enter the Deployable Code (n-Bit) folder where n is either 32 or 64 depending on the type of Operating System. If you are unsure about this, it is generally safe to stick to 32 (Therefore pressing on the Deployable Code (32-Bit) folder).

Step 3: Click on 'HuckersWithATwist.exe' and the game will load.

In order to close the game, simply click Win and then right click the icon on the task bar to close the game.

Rules Of Huckers with a Twist

1. In order to get your counters out and around the board, you must roll a six on at least one of the dice to start off with.

2. The aim of the game is to use the dice numbers to get all of your counters out and around the board, finishing in the middle squares. At any time, you can get an additional counter out. The dice numbers can be used on the same counter, or on two different counters of your colour.

3. Once your counters are around the board and on the colored squares that lead up to the middle, you are considered to be "Up the Spout", a counter that is in the middle-most square is considered to be "Home". In order to go from "Up the Spout" to "Home", you must roll the exact numbers required.

4. Your partner is the player that is directly opposite you, they are an opponent that you cannot knock off. If you land on any of your partner's counters (or indeed any of your own counters), you will return ALL counters on that square back to the beginning. This is called a "Knock Out". *(Creating a big difference from a traditional game of Naval 'Uckers where counters are required to create a blob to block the opponent from moving further).*

5. At any time, you can choose to end your turn by pressing the "Manually End Turn" button. The game will simply move on to the next player. Manually ending your turn could mean that your opponents may get round the board quicker than you.

6. Upon your counters all returning to the home square, you will be asked to play a game of 3D Noughts & Crosses against a strategic computer player. If you win this game against the computer, you will win the entire game. However, if you lose this game, the Computer will steal your win and declare itself the winner.

7. The Green Player of this game is a computerised player. It has a simple secret strategy behind it. In the event that the Green Player should win, everyone immediately loses this game and the game is reset.

Huckers Twist - Controls

The board is immediately set to the red player beginning play. The dice can be rolled by clicking the "Roll Dice" button followed by a Red Counter. Once the Counter has been pressed, the square will light up Magenta in colour. Once moved out the counter can be clicked again to move the remaining squares or a 'Manual End Turn' can take place by clicking the 'Manual End Turn' button.

The Green Counters are a computer player, it will automatically roll the dice and see if a six is rolled. It will then implement its own simple but secret strategy to getting its own counters around the board. **Beware:** If the computer player wins the Huckers game, it will automatically win the whole game.

3D Noughts & Crosses – Controls

Left Click	Clicks a specific square on the board
Left Shift	Moves the field of view down
Space Bar	Moves the field of view up

Bibliography

[1] Uckers – Paragraph's 1 and 2. History of 'Uckers' - https://www.mastersofgames.com/cat/board/uckers-board-game.htm

[2] Uckers – History https://en.wikipedia.org/wiki/Uckers#:~:text=Uckers%20is%20a%20board%20game,well%2C%20including%20the%20Commonwealth%20Forces.

[3] "Qubic" Subsection, "Games and Analysis" 4x4x4 subsection and "Computer Implementations" - https://en.wikipedia.org/wiki/3D_tic-tac-toe#:~:text=%22Qubic%22%20is%20the%20brand%20name,3D%20Tic%20Tac%20Toe%20Game%22.

[4] Subsection IV – 3-dimensional Tic-Tac-Toe, https://www.ripon.edu/wp-content/uploads/2014/11/Summation-2010.pdf

[5] What is Scripting in Unity - https://unity.com/how-to/learning-c-sharp-unity-beginners#:~:text=The%20language%20that's%20used%20in,(pronounced%20C%2Dsharp).

[5a] History of Unity - https://en.wikipedia.org/wiki/Unity_(game_engine)

[6] Naming Conventions, Basic Best Practices and Coding best practices (Unity) - https://avangarde-software.com/unity-coding-guidelines-basic-best-practices/

[7] Access Level Modifiers table W3Schools - https://www.w3schools.com/cs/cs_access_modifiers.php

[8] GitLab documentation (2 Factor Authentication section) - https://docs.gitlab.com/

[9] Use of a UML diagram (Introduction) - https://ceur-ws.org/Vol-1078/paper1.pdf

[10] Picture of 3D Qubic board - https://www.ebay.co.uk/itm/234355208287

[11] Applications of educational games to enhance student learning - https://www.frontiersin.org/articles/10.3389/feduc.2021.623793/full

[12] Input.GetMouseDown scripting API – Unity - https://docs.unity3d.com/ScriptReference/Input.GetMouseButtonDown.html

[13] Material.Color scripting API – Unity - https://docs.unity3d.com/ScriptReference/Material-color.html

[14] The Ternary Operator – Explanation - https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/conditional-operator

[15] Experimentation with Field of View (YouTube) - https://www.youtube.com/watch?v=qQLvcS9FxnY

[16] Experimentation with Field of View (Pastebin) - https://pastebin.com/RXZ1dXgw

[17] 3D Noughts & Crosses https://en.wikipedia.org/wiki/3D_tic-tac-toe (4x4x4 two-players).

[18] Qubic 4x4x4 Tic-Tac-Toe – Oren Patashnik - https://www.jstor.org/stable/2689613?seq=5

[19] Information on C# Switch Statements - https://www.w3schools.com/cs/cs_switch.php

[20] StyleCop – A style checking software for .NET framework. - https://en.wikipedia.org/wiki/StyleCop

[21] Results of the experiment on online gaming and loneliness - https://eric.ed.gov/?id=EJ1222950

[22] Times spent on online games per week - https://www.uswitch.com/broadband/studies/online-gaming-statistics/#:~:text=Furthermore%2C%20online%20gaming%20research%20by,by%20those%20between%2055%2D64.

[23] Links between physical health and prolonged time periods of inactivity- https://www.nhs.uk/live-well/exercise/why-sitting-too-much-is-bad-for-us/

[24] Unity C# pricing controversy - https://www.polygon.com/23885373/unity-technologies-install-fee-pricing-change

[25] Market Research into Windows - https://www.statista.com/statistics/218089/global-market-share-of-windows-7/

[26] Ludo Board - https://en.wikipedia.org/wiki/File:Ludo_board.png

[27] Disclaimers on Clker.com and Openclipart.org – https://www.clker.com/faq.html#q7 & Licensing Section - https://openclipart.org/share