

Blackjack Player

Final Report

Submitted for the BSc in
Computer Science

October 2017

By

Geovan De Jesus Inacay

Word Count: 13760

Table of Contents

Contents

1 Introduction	5
2 Aim and Objectives	6
3 Background.....	7
3.1 History of Blackjack.....	7
3.2 Basic Strategy	8
3.3 Card counting.....	9
3.3.1 Counting fives.....	9
3.3.2 Complete point counting	10
3.3.3 Counting tens	10
3.4 Technology Comparisons#.....	11
3.4.1 Comparison of Game engines	11
3.4.1.1 Unity.....	11
3.4.1.2 Unreal Engine	11
3.4.2 Comparison of programming languages	12
3.4.2.1 C++	12
3.4.2.2 C#	12
3.4.2.3 Unreal Script	12
3.4.2.4 XML AND JSON.....	13
3.4.3 Comparison of Mobile phone operating systems	13
3.4.3.1 Android	13
3.4.3.2 Apple iOS.....	13
3.5 Comparison of algorithms	14
3.6 Conclusion.....	15
4 Design Overview	15
4.1 Unity Architecture	16
4.2 User Interface.....	17
4.2.1 Menu Scene.....	17
4.2.2 Game Scene	18
4.2.3 Simulation scene.....	20
4.3 Scene navigation	21
4.4 Class Design	22
4.5 Default Mode	23
4.6 Advice Mode.....	23
4.7 Computer Simulation	24
4.7.1 Strategy design	25

4.7.2 XML (Hi-Lo).....	26
5 Implementation	27
5.1 Representing a card (Card Prefab)	27
5.2 Representing decks (Card stack script)	29
5.2.1 Card rank	30
5.3 Card shuffle	31
5.4 Main Menu Implementation.....	32
5.5 Blackjack Simulation Implementation.....	33
5.5.1 Simulation logic.....	34
5.5.2 Card counter class	38
5.6 Blackjack Game Implementation	39
5.6.1 Blackjack game logic.....	40
5.7 UI Finalised	41
5.8 Testing overview.....	43
5.8.1 Test table	43
5.8.1 Testing the xml parsing	46
5.8.2 Simulation data	47
6 Evaluation	48
6.1 Project management report (Interim)	48
6.2 Project management report (Final progression)	48
Since the interim report progress, I have left the implementation of the blackjack split and focused on the simulation aspect as it is a critical feature of my application. When coding the simulation I experimented first parsing XML files and how they are used to stored data. I watched various YouTube tutorials online which boarded my understanding. Coding the simulation loop was extensive, and a lot of debugging was used to careful fix small bugs in the application constantly. I used visual studios watch list to observe different key variables that were dynamically critical to simulation a single hand correctly.	48
6.3 Project achievements	48
Further improvements and bugs	50
7 Conclusion	51
8 Appendix.....	52
Appendix A: Task List (Initial report).....	52
Appendix B: Initial task list progress review.....	53
Appendix C: Interim report task list.....	54
Appendix D: Initial Time Plan	54
Appendix E: Interim Report Time plan.....	56
Appendix F: Risk Analysis.....	57
Appendix I: Card sprites.....	58
Appendix J: UI: Game scene (Unity)	59
Appendix K: Card stack/card stack view/card model	60
Appendix L: Blackjack loop diagram.....	61

Appendix M: First deal (card stack representation)	62
Appendix N: Testing result (Screen shots)	63
Figure 4.1 – Setting up the development kits – page 15	68
Figure 4.2 – Unity Architecture – page 16.....	68
Figure 4.3 – Main menu (Hand drawn) – page 17.....	68
Figure 4.4 – Game scene UI (Hand drawn) – page 18.....	68
Figure 4.5 – UI concept in Unity for game scene – page 19.....	68
Figure 4.6 – Simulation scene UI (Hand drawn) – page 20.....	68
Figure 4.7 – Scene navigation diagram– page 21.....	68
Figure 4.8 – Card model UML– page 22.....	68
Figure 4.9 – Advice diagram– page 24	68
Figure 5.1 – Inserting sprite in card model class – page 27	68
Figure 5.2 – Animation curve – page 28	68
Figure 5.3 – Card flipping algorithm– page 28	68
Figure 5.4 – Pop and push methods – page 29	68
Figure 5.5 - illustration of card index to card rank – page 30.....	68
Figure 5.6 – Function that converts card index to hand value – page 30	68
Figure 5.7 – Fisher-yates shuffle implementation (Old method) – page 31	68
Figure 5.8 – Main Menu Hierarchy Architecture – page 32	68
Figure 5.9 – Blackjack Simulation Hierarchy Architecture – page 33	68
Figure 5.10 - Betting implementation – page 34	68
Figure 5.11 – Parsing the Hi-lo XML strategy – page 35.....	68
Figure 5.12 – Code for handling XML return advice – page 36	68
Figure 5.13 – While loop that simulates blackjack deals – page 37	68
Figure 5.14 – Implementation of converting card index to running count – page 38.....	68
Figure 5.15 - Blackjack Game Hierarchy Architecture – page 39.....	69
Figure 5.16 – Add card method in card stack view script – page 40	69
Figure 5.17 – Finalized UI (Main Menu) – page 41	69
Figure 5.18 – Finalized UI (Blackjack Game) – page 32	69
Figure 5.19 – Finalized UI (Blackjack Game) – page 42	69
References	71

1 Introduction

The main aim of the project is to develop a blackjack player that can consistently win blackjack games using various counting methods.

The project will explore various strategies that mathematically support a winning strategy to win blackjack and implement them thorough a simulation and a blackjack game.

Blackjack or otherwise known as twenty-one is a card game which is played between one or several players all against a dealer. It is important to understand that players do not play against each other. The game is played with one or more decks of 52 cards. Generally, the objective of the game is to beat the dealer in one of the following ways.

- The player gets a total of 21 with the first 2 cards dealt; this is called "blackjack" or "natural", without a dealer blackjack.
- Let the dealer draw additional cards until their hand exceeds 21.
- Reach a final score higher than the dealer without exceeding 21.

The game starts with each player being dealt 2 cards, face up or face down depending on the casino and table you sit on. Blackjack in the U.S the dealer is dealt also two, one faced up or exposed and one down or hidden. However, in remaining countries, the dealer is only dealt 1 face-up card.

The value of each card is their actual set value, this is true for 2 through 10, and additionally, face cards which are Jack, Queen, and Kind are all worth 10. The Ace card has two values of one and eleven. The definition of a player or dealers hand is the sum of the all the card values. While playing the game, players are allowed to ask more cards to improve their hand. When a player has the card Ace and doesn't hit a blackjack, the hand is automatically valued as "soft". This means that the hand will not bust by taking an additional card. This is useful as the ace value will be used as a one instead 11 to prevent the hand from exceeding 21. Else then the hand is "hard". (Henry Tamburin Ph.D.),

Once the two initial cards are dealt out clockwise from the dealer, this also includes the dealer, the dealer's next step is to ask each player in turn if they would want another card or to "hit". The player must then decide after totally their hand to accept another card in hopes to improve the hand. The "hit" call can be said my mouth by the player if he or she taps the table. If the player decided that to not receive any more cards, the player "stands", this pronounced by waving a hand over the cards. Furthermore, the player can hit as much as possible until the score of the hand reached 21 or exceeds 21. Whereas the score of a players hand exceeds 21, the bet of the player is lost to the dealer or house.

2 Aim and Objectives

The aim of this project is to develop software that can play and simulate the blackjack game with the basic rules. The application will have various modes, default, advice and simulation mode. The simulation mode will simulate a set number of hands that will be played by different strategies explored in the background. The simulation should bet according aiming to win as much as possible.

Objective 1 – Rules of the blackjack game implemented blackjack game and simulation

The program should let the user play the blackjack with the rules used in the casinos.

- Splitting, doubling down and insurance bets (Splitting will be only allowed once)
- 4 decks will be used and will be shuffled at different points of the game
- The dealer will have to stand on soft or hard 17.
- Maximum and minimum bets are defined
- When the player wins against the house (a “permanent bank”) bets are paid off.
- When the first two cards equal to 21, this is called blackjack or a natural, the player will be automatically paid 3:2 on their original bet.

Objective 2 – Develop 3 playing modes

Objective 2.1 – Default mode

This mode will be just like playing in an online casino, where you have a set amount to play blackjack with. Normal blackjack rules will be applied which is stated above in objective 1.

Objective 2.2 – Advice mode

Advice mode is where the application will advise the bet decision to play for example to stand on 17 against a dealer up card of 4.

Objective 2.2.1 – Basic strategy

The advice given should be taken from the basic strategy table, which is widely available. This table advises you on your tactics of play depending on yours and the dealers up card. See figure 1.1

Objective 2.3 – Computer simulation

When in this mode, the user will enter the number of hands which the computer player will play. In the result the information of the hands won and lost should be displayed, in addition to money lost or gained.

Objective 2.3.1 – Implement basic strategy

Card counting strategy should be implemented to the blackjack simulation that additionally increases bets accordingly. Hi-Lo or counting tens methods could be used.

Objective 2.3.2 – Implement hi-lo counting method

Hi-lo strategy should be implemented to the blackjack simulation that uses the Hi-lo index to make playing decisions.

Objective 2.3.3 – Devise a betting system

The simulation should make bets, with the knowledge of the true count. If the true count is high, the simulation player should increase bets accordingly to win more money.

Objective 3 – Graphical User Interface

A simple interface menu with implemented with options clearly visible to change what mode currently in. The cards for the dealer and player should be shown accordingly including in-game text for example when the player loses their bet.

Secondary Objectives

Not essential to be finished with the software as it won't affect the main functionality.

Objective 1 – Instructions for the game

If a new user would like to know how the applications works, they could do so by accessing a page where it tells you visually how the game operates and how to use its features clearly.

3 Background

3.1 History of Blackjack

Blackjack is one of the most popular casino card game in the 20th century, this is because of its simplicity and odds of winning. The game is played all over the world from the United States to Manila located in the Philippine islands. However the originality of blackjack is uncertain it began being played in French casinos around the year 1700, the version of the game called 'vingt-et-un' which translates to 20 and one is to be believed to be the precursor. (Edward O. Thorp, 1966). As to playing blackjack in the casino, many players have been using the other ways to play the game. For example, using online means, these websites offer the chance to play the game on different platforms.

Many forms of similar card games were played in early times, for example, the Italians played a game called 'Seven and a half' which goal was to reach a hand totaling 7.5. The values of each card were as follows. When a players hand exceed the 7.5 goal, the hand is busted, this is the same terminology used in blackjack games today (Henry Tamburin Ph.D.),

- Face cards were valued as a half point
- 8, 9 and 10's were valued at 1 point
- Lastly, the king of diamonds was a wild card.

As vingt-et-un gained popularity, the name of the game got corrupted and various names were used; England version became 'Van John', furthermore, in Australia, it was called 'pontoon'.

During the 40's and 50's, a few notable players developed counting systems to beat blackjack. Edward O. Thorp is referred to the father of card counting and with this method caused casino to change various rules. Card counting is where a player will give a set of cards a value of +1, -0 and 0. When the cards are dealt the player counts the cards in relation to their count value. This becomes the running count and this value is used in conjunction with how many cards are remaining in the deck and a true count is calculated.

This method of beating the game works because it looks at the relationship of the what cards have been dealt, if the deck or shoe is rich in high cards it favors the player on the other hand if the deck is rich in low cards, the dealer will have the advantage.

3.2 Basic Strategy

Basic strategy is a set of predefined rules that advises you the most optimal way of playing every hand dealt when then the only information available is your hand and the dealer's up card. The basic strategy is the mathematically correct approach to play every hand as it will result in maximizing the amount of money you will win simultaneously minimizing the amount you will lose resulting the casino to be at a disadvantage. Applying the basic strategy will give you about 0.2 – 0.5 % edge over the casino, therefore on average, for every £100 bet you will expect to lose £0.50.

Surrender strategy

The surrender rule is a misunderstood rule as it itself has a negative association within players as when you play blackjack you are supposed to win your hand every time and not surrender it. However playing this rule can have its advantages. It works by considering your two dealt cards and the dealer's up card if you think you have a low chance of winning, you can surrender which will in turn forfeit your hand along with half of your initial bet. There are two types of blackjack surrender; late surrender and early surrender.

Late surrender occurs after the dealer peeks at the hole card, though if the dealer has blackjack the option for surrender is no longer available thus losing your entire bet unless you also have a blackjack. In result if you use the late surrender appropriately, the house edge is reduced by 0.07 percent in multiple-deck games.

Early surrender is more widespread throughout Europe and Asian casinos rather than America, this is where dealers do not take a hole card until all of the players on the table have played their hand. Therefore players can have the option for an early surrender before the dealer checks if they have blackjack. When playing early surrender against a dealer's ace gains you have a 0.39 percent advantage and against a 10, a 0.24 percent. Strategy is seen below.

- Against a dealer ace, surrender hard 5 to 7 (including 3s), and 12 to 17 (including 6s, 7s, and 8s).
- Against a dealer ten, surrender hard 14 to 16, including 7s and 8s.
- Against a dealer 9, surrender hard 10-6 and 9-7 (but not 8s).

In conclusion you will benefit from playing the surrender strategy when your chance of winning is less than one out of four hands, meaning you expect losses worse than 50 percent. Mathematically if playing a hand that has a 25 percent chance of winning simultaneously a hand that has a 75 percent chance of losing, you will save money on the long run by playing the surrender rule rather than playing the hand.

Splitting strategy

The option for splitting is offered to the player when two cards are dealt are of the same value consequently being able to have two separate hands. For example if you bet £5 and you receive a pair of fours, you can play the hand as an 8 or split have two hands of 4. However you have to place a bet equal to your original bet and place it on the newly formed hand. The aspect that governs when to split using basic strategy is the number of decks of cards used in conjunction with the casino rules for example being able to re-split and doubling down.

Examples when to split;

DAS = Doubling after pair splitting

NDAS = Not doubling after pair splitting

- Double-deck game with DAS, you hold 4-4 and the dealer's up card is a 5. You should split.
- Double-deck game with NDAS, you hold 4-4 and the dealer's up card is a 5. You should not split.
- Six-deck game with DAS, you hold 9-9 and the dealer's up card is a 9. You should split.

Doubling down strategy

Doubling down is when you notify the dealer that you want to double the amount of the initial bet in return for only receiving one card. According to the strategy you are most like to double down on hard 9, 10, 11 and a soft 13 (Ace – 2) to 18 (Ace – 7).

Hit and stand strategy

This strategy is important as it is the most frequent playing decision. Hitting is the means of ask for another card whereas standing means you are contented with your hand and do not want more cards. You are able to stand and hit on any hand that totals 21 or less, however if in the process of taking a hit and your hand exceeds a total of 21, you are busted and automatically lose your bet. The basic strategy informs you when to hit or stand which is supported mathematically. For example if you are dealt a hard 13 against a dealers 3, if you stand you will win 35.2% of the hands and lose 64%, on average. However if you hit you will win 37.4% of the hands and lose 62.6 , on average, consequently if you were to bet £10 per hand, after 100 hands in which you hit on a 12, you will lose on average £25 which is less if you were to stand, losing £29 on average.

In conclusion the table below illustrates the basic strategy, it shows the combination of cards that you can be dealt on the left hand side along with the possible dealer up cards. You use this information to cross reference with the table to find the advised decision to make.

3.3 Card counting

3.3.1 Counting fives

This method involves counting how many fives remain in the deck. Edward O. Thorp *states that mathematically if one kind of card was removed from the deck, the greatest shift in the relative advantage of player and house is caused by removing four fives from the deck.* (Edward O. Thorp, 1966).

This five count strategy works similar to the basic strategy but giving a 3.6% advantage. Similarly to the basic strategy whereby soft standing numbers are the same and all basic doubling down situations also call for doubling down when fives are depleted from the deck. This is the same for splitting of pairs however a pair of Sixes are not split against a dealer's up card of value Seven. Furthermore, you can improve the method by keeping the number of

unseen cards, this way you can estimate whether or not the deck is five rich or five poor thus calculating your advantage. (Edward O. Thorp, 1966).

3.3.2 Complete point counting

With this strategy, we retain a count from the cards that have been dealt by the number they have been assigned to and then calculate our advantage from the cards remaining. This should be done in conjunction with the basic strategy.

1. The first step of counting cards is to assign a point value to each card rank. Card ranks from 2-6 are assigned as + 1, 7-9 are 0 and 10 to Ace are worth -1.
2. As the game starts you need to keep a track of the 'Running Count'. This should be zero before the deck/shoe is dealt. When a new card is revealed by the dealer, you should add or subtract according to the points which were assigned to the running count. For example, if the first 4 cards you see are 10, 9, 3, Ace, your running count will be equal to -1.
3. The next step is calculating the 'True count'. You derive this by dividing your running count by the number of decks remaining in the game. Determining how many decks are left can be difficult, therefore a rough estimate should be sufficient. For example, if you're running count is equal to 4, and there are roughly 3 decks remaining. The true count will be equal to $4/3 = 1.25$.
4. The purpose of the true count is to tell you your advantage against the house. When the true count is exactly 1, you and the casino have even odds. As the count increases the odds are more in your favor.
5. Lastly if the true count is high, the odds are in your favor and therefore you should bet big, otherwise, you should stay at the table betting minimum.

3.3.3 Counting tens

This strategy is on the same effectivity as of the complete point count system, by using this method a players advantage ranges from one to ten percent. This is different to counting of fives as for card by card, fives will have a greater effect than ten, this is because if to say only for ten value cards were added to the deck, it will give the player an advantage of 1.89 percent whereas the absent of fives will give an advantage of 3.58 percent. (Edward O. Thorp, 1966).

On the contrary, in a standard deck, there are 16 ten's in the deck and only four fives, consequently creating a larger deviation of tens appearing from the number of fives. Keeping a running count of the number of tens and unseen cards while playing in conjunction with the basic strategy is key for effective betting. (See page

3.4 Technology Comparisons#

3.4.1 Comparison of Game engines

As I am making a game, there are many ways of delivering the solution, for example, a simple console texted based blackjack game could be implemented from C# with the IDE visual studio. This part of the report explores all possible varied solutions.

3.4.1.1 Unity

Unity is used as a cross-platform game engine that makes three and two-dimensional games. You attached various components to 3D/2D objects which manipulate them in different ways. C#, Boo or unity script are all languages that are used as scripts. Unity has very fast development speed that also allows for facilitating quick prototyping and continuous release. When editing code in Unity by clicking on a code file within the project view, this opens a default cross-platform editor; Mono develop or visual studio. You can't use visual studio as a debugger as you are not bugging Unity.exe without UnityVS you are debugging a virtual environment inside of unity using a soft debugger that issues actions and commands (Tuliper, 2014).

In Unity games are developed in scenes, as when you package your game for a platform, the resultant collections are of one or more scenes. Within a scene, everything that you manipulate is a game object, it is like the System.Object in the .Net framework; as almost all types derive from it. In order to provide functionality to game objects, components are added. Components for different audio, camera, and physics related components are available; scripts are assigned to an object to give control. For the blackjack game, the engine can be used or deploy for a mobile build to Android, iOS and Windows phones with a 2D setting. In order to deliver an Android app, you need the Android/Java software development kit to be downloaded and linked with to unity, this enables you to access the libraries to create an Android software application.

When scripting in unity, co-routines are made available and are useful for implementing different solutions. As you call a function, it will run to completion before returning, effectively meaning that any action taking place in a function must happen with a single frame update. Therefore a function call can't be used to contain a procedural animation or a sequence of events over time. To solve this problem, co-routines are used to have the ability to pause execution on a frame by frame basis and return control to unity but then continue where it left off on the following frame. To declare co-routine, a return type of I Enumerator with a yield statement included somewhere in the body. The yield return line is where the execution will pause and be resumed on to the following frame. In regards to my projects., co-routine could be implemented when cards are being dealt to the dealer and player from the shoe; just like in a casino setting, there is a brief pause when cards are being dealt.

3.4.1.2 Unreal Engine

This engine was created by Epic games lead by Tim Sweeney who built an editor that would allow 3D games to be developed. However, the most recent version is Unreal engine 4 which uses C++ to program in instead of its own scripting language (Unreal script used in Unity 3). In addition to Unreal 4, it encapsulated a new way of blueprint visual scripting which is node-based. This allows you technically to never need to write code, therefore, allows for quicker prototyping of levels and giving unexperienced programmers an advantage (Steiner, 2003). In terms of graphics, Unreal engine offer a better choice, as it has features to support complex particle simulations to advanced dynamic lighting. Therefore if you were making a graphically

intensive game made for next-generation consoles, the Unreal engine would be best suited but however Unreal 4 has the capabilities to create 2D/3D visual styles for PC, Mac, iOS, Xbox One and PlayStation 4. Subsequently, Unreal Engine 4 allows for development for Android application which is beneficial for my blackjack game.

3.4.2 Comparison of programming languages

As with every software application, a programming language has to be chosen, the table below will outline the three languages which are considered to be used in coding the blackjack player and two languages to represent the player strategies.

Language	Development Speed	Developer's Competence	Most Recent Used	Object-Oriented Ability	Performance
C++	Moderate/Low	Moderate	2017	Yes	High
C#	High	High	2017	Yes	Medium
Unreal Script	Moderate/Low	Moderate	n/a	Yes	High
XML	Moderate/Low	Low	2017	Yes	High
JSON	Moderate/Low	Low	2018	Yes	High

3.4.2.1 C++

The first language for consideration is C++, the language was developed from C. C++ is a high-level language which offers the choice of multi-device and platform development. It is also an object-oriented language which incorporates encapsulation, classes, and inheritance. C++ is mostly chosen when developing games or software is because it is a powerful, efficient language. The most useful tool of C++ is the use of pointers which allow the allocation of memory dynamically. The C++ compiler is very well optimized for resource management and also facilitates multiplayer with networking. However there are some disadvantages, for example, the development speed can be severely hindered by the languages type system and the lack of garbage collection. Another aspect that can affect development speed is the developer's lack of knowledge of the language.

3.4.2.2 C#

The second language for consideration is c#. This language was also developed from C to compete against Java. It is an object-oriented language which can be used for both mobile and desktop applications. In addition, the open-sourced language can accommodate various different platforms for example android where Xamarin or unity3D the game could be used. C# is also more type safe and protected meaning you get compiler errors and warnings that in turn create errors which C++ will allow. In regards to my experience with C#, choosing this language would be best suited as I have been programming for more than 2 years with it. The resources both in the university library and online sources will aid in the speed of delivery of the solution.

3.4.2.3 Unreal Script

The third language is what Unreal 3 engine uses as a scripting language. Its main key point when it developed, that it had games as a major focus there it contained many built-in features for examples states and timers. This made gameplay execution much smoother. The language is object-oriented and is derived from Java and C++ syntax. Learning this language would be useful if I was using the unreal engine 3 which offers high performance in making 3D games. With the Unreal script you don't have to worry about manual memory and is weakly type there

you can pass arguments to functions without always knowing their exact kind of argument. It shouldn't be confused with unreal engine 4 using C++.

3.4.2.4 XML AND JSON

To implement the two strategies for the computer simulation mode, extensible markup language (XML) is applied as it defines a set of rules that represent information in a format that is both human and machine readable. XML is extendable therefore tags can be created to represent all the possible combination of hands as attributes along with the dealers up card.

On the other hand JSON can also to implement this strategies, as JavaScript object notation (JSON) is a syntax for storing and exchanging data. JSON is displayed as text which can be convert any JavaScript object into JSON. JSON is lightweight and is 'self-describing' and easy to understand. Below shows individual examples of XML and JSON representing the same data.

JSON

```
{
  "company": Volkswagen,
  "name": "Vento",
  "price": 800000
}
```

XML

```
<car>
  <company>Volkswagen</company>
  <name>Vento</name>
  <price>800000</price>
</car>
```

3.4.3 Comparison of Mobile phone operating systems

A mobile operating system (OS) is software that allows smartphones, tablet PCs, and other devices to run applications and programs (Rouse, 2011). They give all the basic functionality of your smartphone from being able to make calls, access the internet and send/receive text messages. Google's Android, Apple IOS, and Microsoft's window phone are all examples of different types of operating systems. However Android and IOS have together a 97.1% of the global market stake for smartphones (IDC, 2015).

3.4.3.1 Android

The Android operating system is developed by Google and it's powered by Linux Kernel. A kernel is the lowest level of replaceable software that converses with hardware, its purpose is to provide an interface for all applications (Garrison, 2010). Android is an open source operating system which means that developers are given accessed to hardware and create new applications freely with minimal licensing restrictions. For that reason, Android has been adopted by phone manufacturers, for example, Samsung, HTC, LG, Motorola and Sony Ericsson. It stands as the dominant smartphone platform in line for its wide spectrum of users all having access to the Android market which is home to millions of apps that are mostly free (Uswitch, 2016). Nevertheless Android has it is a drawback as many phones are different therefore they vary in size, specifications, and versions.

3.4.3.2 Apple iOS

iOS is Apple's owned operating system designed to run on their iPhone, iPad, and iPod. iOS is based on when a user touches the screen, this allows you to be able to open programs with the control of a tap of a finger. The latest version of the operating is iOS 11 which currently operates 65% of all Apple devices.

One drawback if I were to use this operating system is that is it not allowed to be used in third-party systems meaning that only products made by Apple would be my target device. As the

poses a development ease from the less 'hardware' fragmentation. On the other hand, there are millions of applications readily available on the App store which is Apple's version of the play store owned by Google. In addition, Apple has more restrictions when publishing on their app store as you require consent, therefore, it adds a new layer of complexity in order for your app to be released.

3.5 Comparison of algorithms

When playing blackjack, shuffling the deck is a key part and many algorithms can be used to achieve this, however there can be implications where shuffling is not totally random. Fisher-yates is an algorithm that generates a random permutation of a finite sequence or shuffles a sequence of numbers. The algorithm works by effectively putting all the elements into a hat; it continuously determine the next element by randomly drawing an element from the hat until no elements remain.

On the other hand, when implementing Fish-yates shuffle, the generation of the random numbers can affect the biasness of the completed shuffle. Fisher-yates when implemented in code, will use a pseudorandom generator; as the sequence of numbers which is outputted is determined by its internal state at the start of the sequence. Fisher-yates algorithm has two versions, old and a modern approach, the big-O notation for the old method is $O(n^2)$ in the worst case whereas for the modern method, it is $O(n)$. Time complexity however only will have a greater affect with dealing with inordinately large data sets. The original and modern are defined below.

Original method

1. Given: A collection of items which we want to randomly sort
2. First: Randomly select one of the "unshuffled" item
3. Then: Place that item at the beginning of a new, separate collection
4. Finally: Remove the selected item from the source collection
5. Until : All numbers have been sorted

Modern method

1. Given: A collection of items which we want to randomly sort
2. First: Randomly select one of the "unshuffled" items
3. Then: Swap the selected item with the last item in the collection that has now be selected
4. Continue until: There are no remaining "unshuffled" items

A pseudorandom number generator can also be known as a deterministic random bit generator, is an algorithm for generating a sequence of numbers, however the generation is not truly random as the complete sequence is determined by an initial value which is called the PENG's seeds. The choice of a good random seed is fundamental in fields of computer security and online gambling systems. As if the seed is exposed, the system will be compromised. However if the random key is deliberately shared, it becomes a secret key. A secret key is used by two or more systems therefore they can create a matching sequence. This sequence of non-repeating numbers can be used to synchronize remote systems for example satellites and receivers. The random seeds are often generated from the state of the computer systems for example time.

The problem with PRNG's are that the sequence can be determined, in comparison, a true random number generators (TRNG) can be used. TRNG's determined their randomness from

physical phenomena's that generate low-level statistically random "noise" signals such as thermal noise therefore offer a better solution for where applications need unpredictability in areas such as data encryption and gambling. A website called 'Random.org' offers a way to generate a true random numbers from atmospheric noise, with their HTTP web API.

3.6 Conclusion

The project will be developed using C#, this is because I am more experienced with the programming language and it will enable to program confidently knowing that they are many resources available. With the addition to choosing Android as an operating system for my blackjack application, was done because it based on Androids dominance in the smartphone industry. When creating apps designed for Apple's iOS, specific tools are needed for development thus resulting in an overhead cost for acquisition of a Mac computer to run Xcode which is used for developers to create apps. Along with my platform and programming language, I have chosen to use the unity engine to develop my project, this engine incorporates C# as a scripting language. It is fairly easy to use and gives all the features that I need to in order to develop my blackjack player effectively and within a time frame.

4 Design Overview

As a phone application was chosen for the blackjack player, we need to start off by creating a 2D project in Unity. However, from my research and past programming two plugins need to be downloaded in order to run and test on an Android device. These are the Android and Java development kits (SDK/JDK). To set the file path for the development kits, we can do this in Unity by the edit tab and clicking preferences followed by the external tools, from this window you can locate where you saved the development kits My experience with Unity is at the beginner's level, therefore, I first familiarized myself by watching game tutorial via the Unity website. This enhanced my knowledge of scripting and how objects, for example, user interfaces are manipulated.

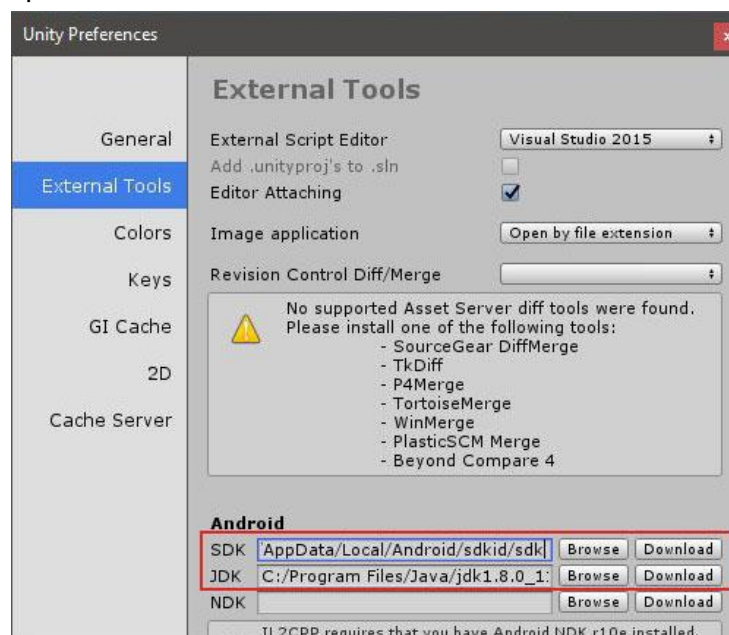


Figure 4.1 – Setting up the development kits

The blackjack game firstly consists of cards which play a vital role in the game. Both player and dealer use the cards to play the game and therefore need to be implemented in Unity as

a 2D sprite. Sprites are 2D bitmaps that are drawn directly to a render target without using the pipeline for transformations, lighting or effects (MSDN, 2018).

To get the graphics for the card sprites I used an online source that I spliced in order represent each of the card ranks in an array of sprites, see Appendix I, However, scripts need to be created in order to control the spliced images to design them to manipulate as a card in a deck to play the blackjack game. Further, in the report, I will elaborate my design choice in order to complete my objectives.

4.1 Unity Architecture

Unity 2D/3D is a game engine that comes with an integrated development and editor, scene builder, asset workflow, networking and scripting. In addition the unity community is vast where many forums exist to support the learning of unity. There are five main views used in the unity editor to develop a game; the project view, scene view, game view, hierarchy view and inspector view all of which are explained in more detail below.

The scene view is where all the game objects are placed and the scenes for the game are built. This view allows the programmer to move through the 2D/3D worlds where the game is built. The game view is what the user will see when they launch the game. Located at the top of this the window are several buttons and drop menus where perspective and screen sizes can be modified. The hierarchy view allows the programmer to create objects which can be grouped and manipulated to make the game. Any object that has an arrow next to it has the option to be further expanded to show more sub objects; the arrow indicates a group of objects. Next is the project view, this is where all the scrips and scenes are made available, the view is similar to file explorer on windows as it allows for the creation of files and folders to help organize the projects assets. Lastly the inspector view is where the all the properties of the objects are stored and manipulated from. For each game object, a transform property exist; this holds properties that hold elements such as scale, rotation and position. Depending on the object type, components can be attached which will affect they physical properties for example when they collide with another object. Textures and sound components can also be attached.

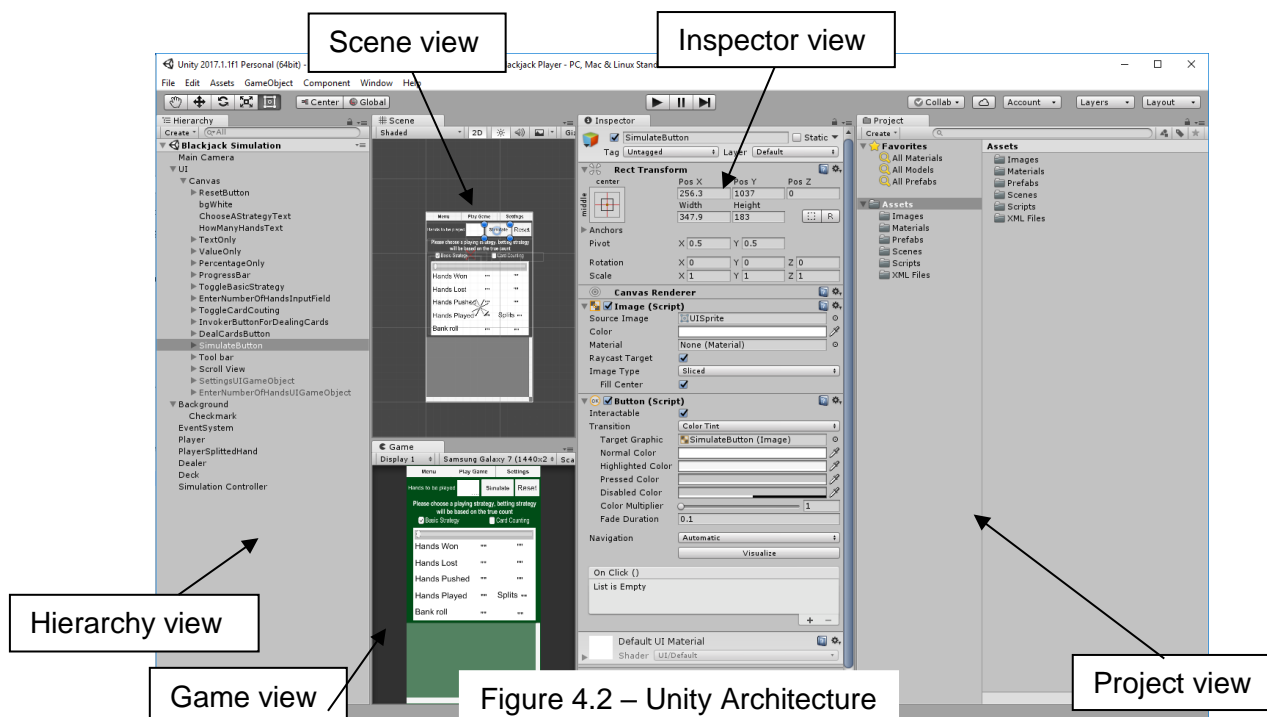


Figure 4.2 – Unity Architecture

4.2 User Interface

4.2.1 Menu Scene

When designing my interfaces I wanted to it to be simple and easy to use at the same time deliver functionality. For the menu, it should be eye-catching with the logo and title of the application standing out. It will have only three buttons; play, simulation and exit buttons with a checked box to give the option for the user if they would like the sound to be on or off. The play button would navigate to the main game scene where the user will play the blackjack game.

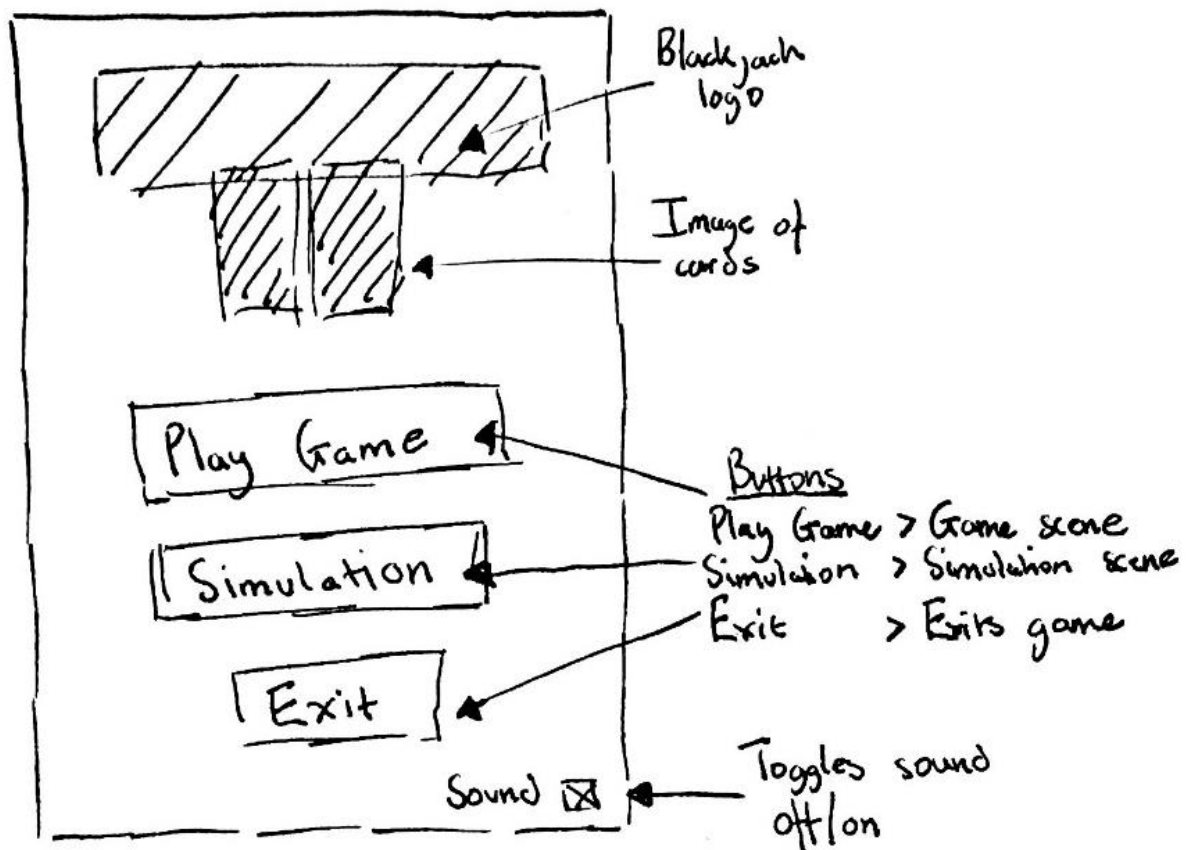


Figure 4.3 – Main menu (Hand drawn)

4.2.2 Game Scene

The game scene delivers the main objectives of the application. The design follows a simple theme similar to a casino blackjack setting which incorporates both player and dealer's cards. Figure 4.2 illustrates a hand drawn design concept for the game scene on the other hand, figure 4.3 is an initial user interface concept made in unity for the game scene. This version of the interface is not final and will need to be improved in later versions, as it doesn't include the option to change game mode thus showing the advice string below the player's card.

The user can play the game by firstly placing a bet using the betting chips each consisting of its set value followed by clicking the deal button. When the cards are dealt on the left side of the cards, there will be a text game object that will represent the hand score for both dealer and player. The information presented will make the user play more efficiently as there is no need to do arithmetic to calculate the hand score mentally. However in later implementation the text field can be removed to enable users to have a more realistic experience. The betting chips are located near the bottom of the screen along with the game buttons to hit and stand, this is intended for ease of use as the thumb will have smooth access to the lower half of a phone screen.

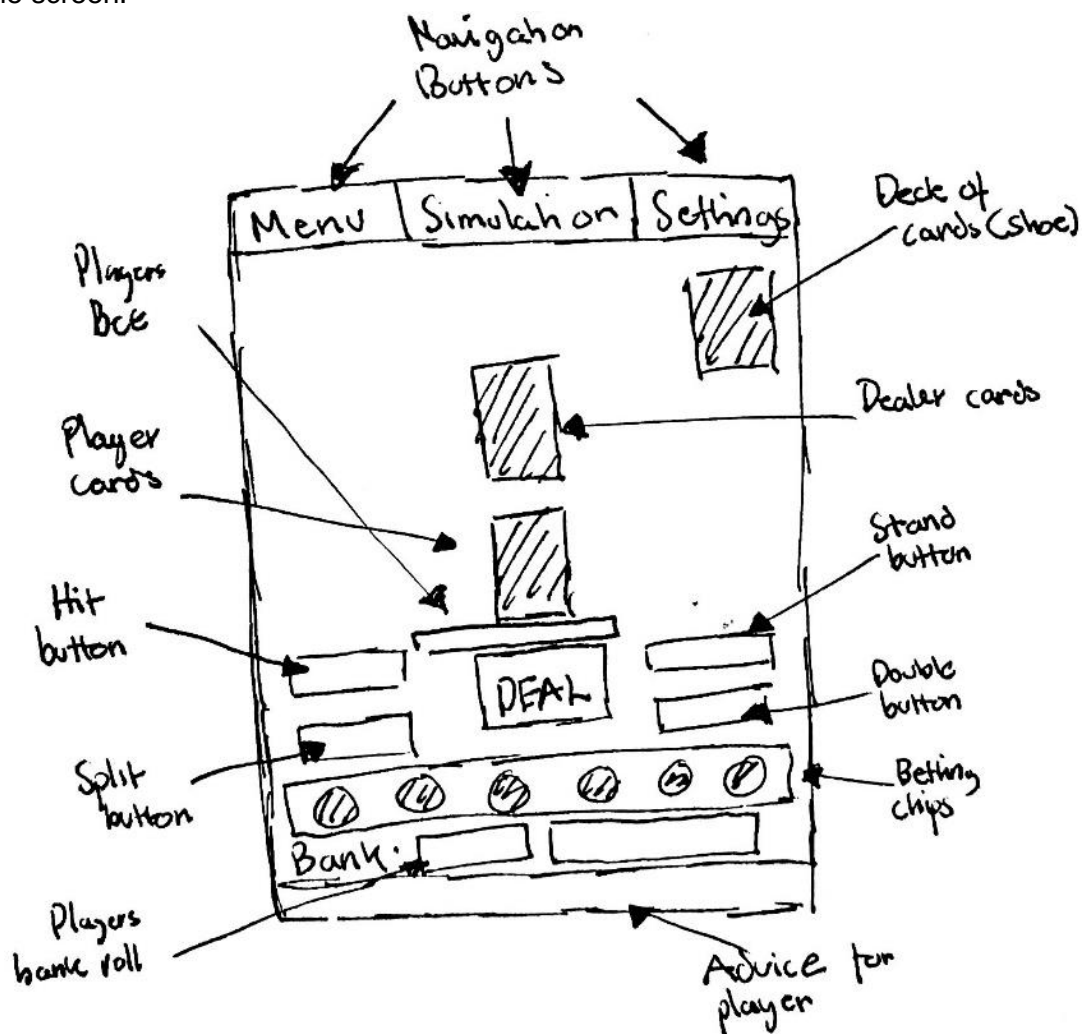


Figure 4.4 – Game scene UI (Hand drawn)



Figure 4.5 – UI concept in Unity for game scene

4.2.3 Simulation scene

The blackjack simulation scene will enable the user to enter a number of hands and simulate a defined blackjack game against a dealer. The user interface should be clear and concise as important data is needed to be shown. Input boxes need to be large enough to allow users to input the number of hands as well allowing the text to be easily seen.

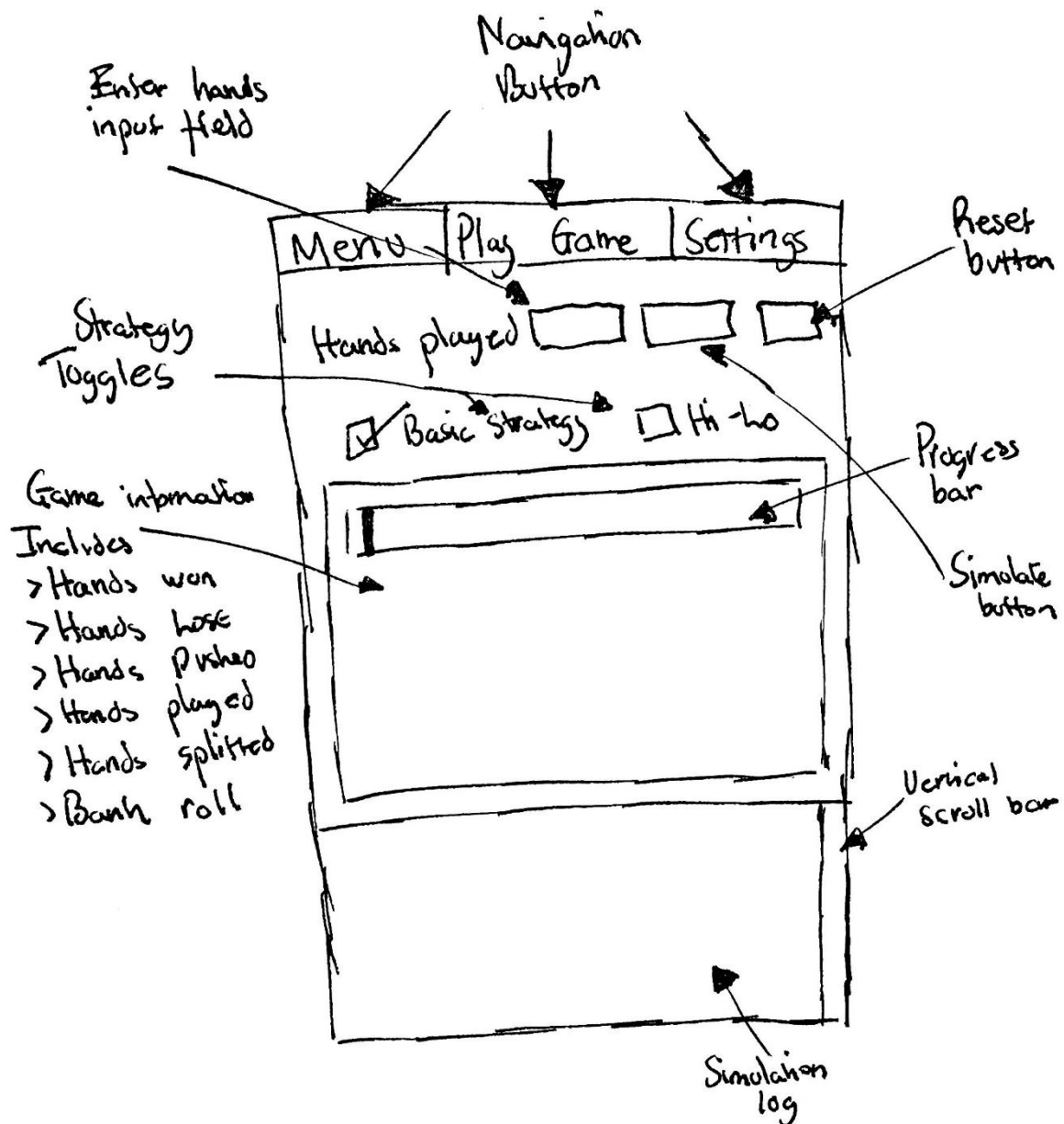


Figure 4.6 – Simulation scene UI (Hand drawn)

4.3 Scene navigation

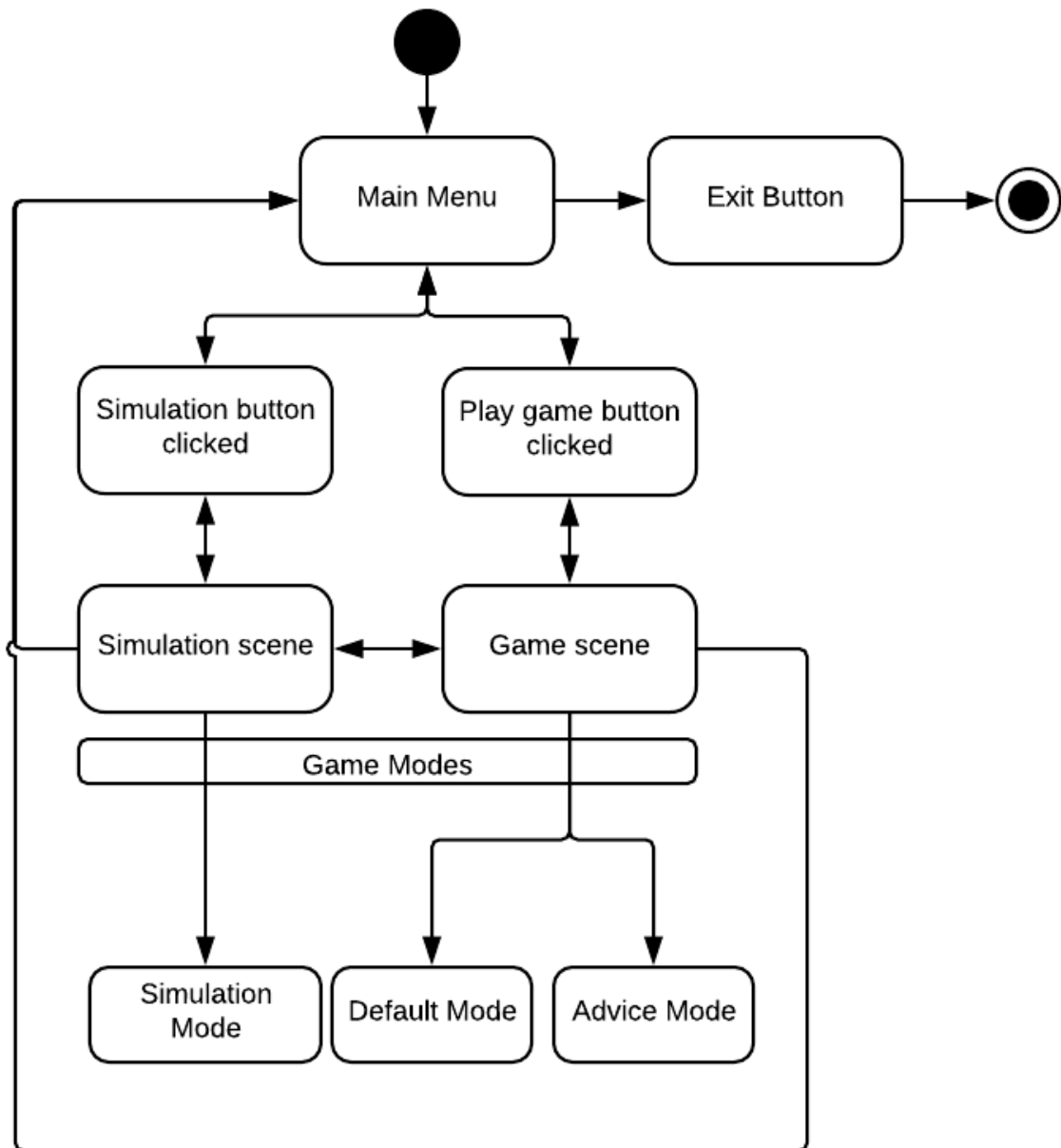


Figure 4.7 – Scene navigation diagram

The diagram above represents how the scene will interact with each other and what game modes are available to the player in that specific scene. When the application is loaded the first scene loaded in the application is the main menu, from this scene the user is able to play and simulate a blackjack game with also having the option to exit the application.

4.4 Class Design

4.4.1 Card and deck

For the game and simulation scenes, a blackjack game will need to be played and therefore cards and decks will need to be implemented. I will create three empty game objects that will hold each of the users, player, dealer and the deck which will act as the shoe for the game. The shoe is where the cards will be taken from to play the game. The application will be played with four decks, as a result 208 cards will be used that hold each of the four suits. Figure 4.6 below shows a Card Model class, which will represent a card. All the variables expect from the sprite renderer will be made public, so that it can be accessed in the unity architecture. This will enable the spliced card faces to be easily drags into the variable field.

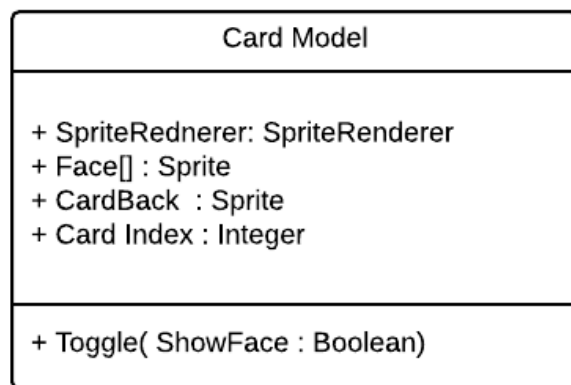


Figure 4.8– Card model UML

Sprites are used because they can represent game objects in 2 dimensional gameplay. The face array will hold all the 208 card images whilst the card back sprite variable will denote a specific card back. In addition card index will hold an integer that represents the position in the deck. However during implementation a prefab would have to make from an empty game object which will represent a singular card. A prefab in unity intuitively creates new instances of the same object in the scene. Appendix M Illustrates an extension of the class diagram show in

figure 4.7, which shows the relationship between a card stack and a card stack view. The card stack class will represent a deck whilst the card stack view class will handle how the image of the card will be displayed and dealt. A vital variable in the card stack class, will be the list of cards, which is of integer type. This variable refers to the card index in the card model.

In terms of the Card stack which will hold the stack of cards for the player, dealer and the shoe they will hold a list of integers. These integers represent a card index. Appendix M Shows a representation of a card deal where two cards are dealt to each of the player and dealer from the shoe. The shoe or deck card stack has the card indexes of 34, 12, 172, 109, 200, and 32. To proceed with the first deal, the card index from deck is popped and pushed to the players card stack, this is repeated for the dealer card stack and executed until the player and the dealer has two card indexes in their card stack list.

4.4.2 Game controller

This script will hold the game logic and will include all the game objects and will manipulate them to play blackjack in the game scene. Game objects for example the dealer will have attached to them a card stack script.

4.5 Default Mode

This mode will be the foundation for the game as it will execute the blackjack game as played in the casinos and online websites. The user will download the application and play the game from the menu screen by the clicking the play button. The mode will be set as default, but the user will have a chance to alter the mode while playing the game. The user will have a bank balance which will be used for betting, this data will be stored locally on their device, so that when they exit it will store their previous bank balance. In the game, if there is enough time, I will include a timer in which will increase the bank balance hourly, or a short promotional video can be watched fully in order to increase their bank balance.

The blackjack game loop will start by asking the user to place their bets by clicking the betting chips on the screen followed by the clicking of the deal button, appendix L, shows a full diagram of the blackjack game loop. The cards are dealt and the user will have the decision to hit, which will ask the dealer for a card or to stand which will let the dealer receive cards until a hand of 17 is reached or greater. The user will also have the option to double down which increases their bet by two and are enable to make an insurance bet when the dealers up card is an ace. These features will be implemented using unity's UI on a canvas which is the root component for rendering all user interface objects in unity.

4.6 Advice Mode

When the user wants to receive help with their playing strategy an option for advice can be enabled. This mode will prompt an advice string which will be either of the four options of hitting, standing, doubling up or splitting. The advice will be generated from using the basic strategy which will be hard coded in the system. Figure 4.7 illustrates the flow and decisions made for the logic of advice mode. In term of activating this mode of play in the game itself, a toggle user interface object can be used. However this mode can only be played in the game scene and not in the simulation.

4.7 Computer Simulation

This mode enables the user to simulate a blackjack game using two playing strategies, first being basic strategy and second being the Hi-lo counting method. Both of the strategies are similar as Hi-lo, uses the basic strategies playing method but the addition with a Hi-Lo index to decide from two options of play. For example if your hand gives a total value of 8 and the dealer up card is a 6, Hi-Lo strategy says to double down on the bet if the Hi-Lo index is only greater than 0.05. On the other hand if the index is lower the only option is to hit. Subsequently with the basic strategy no index is required therefore decision of the computer player is finite. Instead of hard coding both strategies, XML will be implemented. Figure 4.8 illustrates the logic of the simulation and advice mode as they both required to parse XML.

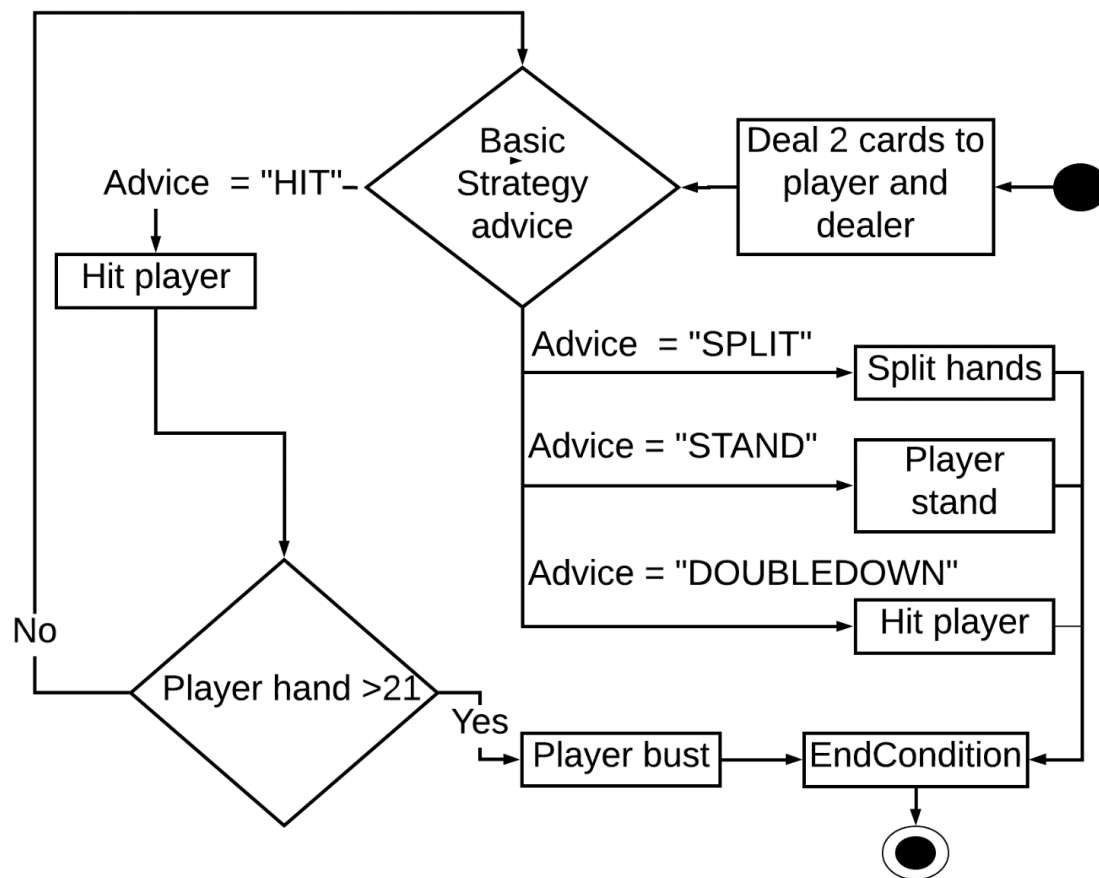


Figure 4.9 – Advice diagram

4.7.1 Strategy design

Seen below is extract from XML files that show basic and Hi-lo strategies for pairs and soft, hard hands.

XML (Basic Strategy)

The XML document consist of a root which contains a basic strategy child elements that will have two attributes that represent the player hands and the dealers up card. The basic strategy element contains a decision element which holds the information for what the computer simulation should execute in terms on their hand.

Hard hand of 16

```
<basicstrategy yourhand = "16" dealerhand = "2 3 4 5 6">
    <decision>STAND</decision>
</basicstrategy>
<basicstrategy yourhand = "16" dealerhand = "7 8 9 10 A">
    <decision>HIT</decision>
</basicstrategy>
```

Soft hand of 18

```
<basicstrategy yourhand = "A7" dealerhand = "2 7 8 ">
    <decision>STAND</decision>
</basicstrategy>
<basicstrategy yourhand = "A7" dealerhand = "3 4 5 6">
    <decision>DOUBLEDOWN</decision>
</basicstrategy>
<basicstrategy yourhand = "A7" dealerhand = "9 10 A">
    <decision>HIT</decision>
</basicstrategy>
```

A pair of 5's with total of a hand of 10

```
<basicstrategy yourhand = "55" dealerhand = "2 3 4 5 6 7 8 9 ">
    <decision>DOUBLEDOWN</decision>
</basicstrategy>
<basicstrategy yourhand = "55" dealerhand = "10 A">
    <decision>HIT</decision>
</basicstrategy>
```

4.7.2 XML (Hi-Lo)

The XML document consist of a root that contains card information child element that will have one attribute of the player's hands. Inside the element contains a dealer up card element that holds a value of the dealer up card as an attribute. Subsequently the dealer up card element will have a child element of the advice for the simulation and the Hi-Lo index. If the Hi-Lo index doesn't exist, the information taken from advice if index greater element should always be executed by the simulation If possible.

Hard hand of 8

```
<cardInfomation yourhand = "8">
  <dealerupcard value ="2">
    <adviceIfIndexIsGreater>HIT</adviceIfIndexIsGreater>
  </dealerupcard>
  <dealerupcard value ="3">
    <adviceIfIndexIsGreater>DOUBLEDOWN</adviceIfIndexIsGreater>
    <hiLoIndex>0.22</hiLoIndex>
  </dealerupcard>
  <dealerupcard value ="4">
    <adviceIfIndexIsGreater>DOUBLEDOWN</adviceIfIndexIsGreater>
    <hiLoIndex>0.11</hiLoIndex>
  </dealerupcard>
  <dealerupcard value ="5">
    <adviceIfIndexIsGreater>DOUBLEDOWN</adviceIfIndexIsGreater>
    <hiLoIndex>0.05</hiLoIndex>
  </dealerupcard>
  <dealerupcard value ="6">
    <adviceIfIndexIsGreater>DOUBLEDOWN</adviceIfIndexIsGreater>
    <hiLoIndex>0.05</hiLoIndex>
  </dealerupcard>
  <dealerupcard value ="7">
    <adviceIfIndexIsGreater>DOUBLEDOWN</adviceIfIndexIsGreater>
    <hiLoIndex>0.22</hiLoIndex>
  </dealerupcard>
  <dealerupcard value ="8">
    <adviceIfIndexIsGreater>HIT</adviceIfIndexIsGreater>
  </dealerupcard>
  <dealerupcard value ="9">
    <adviceIfIndexIsGreater>HIT</adviceIfIndexIsGreater>
  </dealerupcard>
  <dealerupcard value ="10">
    <adviceIfIndexIsGreater>HIT</adviceIfIndexIsGreater>
  </dealerupcard>
  <dealerupcard value ="A">
    <adviceIfIndexIsGreater>HIT</adviceIfIndexIsGreater>
  </dealerupcard>
</cardInfomation>
```

5 Implementation

5.1 Representing a card (Card Prefab)

In order to represent each of the 52 cards in a deck, first task was to splice an image taken from <http://www.thehouseofcards.com/img/misc/Deck-72x100x16.gif> and assign them to a sprite array. As the variable array is a public member, inside unity we can drag and drop the spliced images into the array. The card index will be the number of total cards in the deck meaning is there is 4 decks there will be 208 cards. On the other hand the card stack view class will only be used for the game scene as in the simulation the cards will not be seen, this is to improve on performance. After splicing the image containing card backs and card ranks into individual sprites, next stage was to set a size for the deck, which is 208 and select the corresponding card sprite for the face sprite array in the card model class for the card prefab; this is shown in figure 5.1.

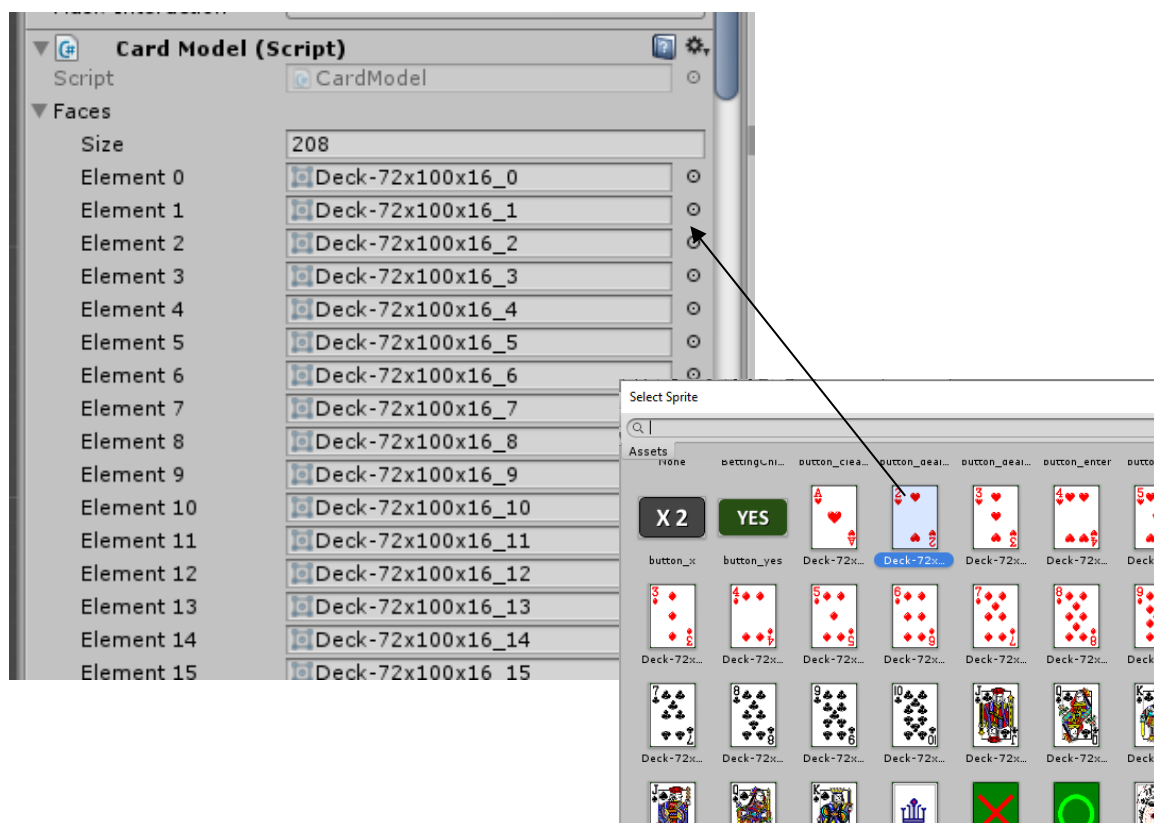


Figure 5.1 – Inserting
sprite in card model class

The card prefab has 3 components, first being a sprite renderer which allows for the image of the back of the card to be displayed, this component will be accessed through the card model to be used again to display a different type of sprite. The other two component is the card model and card flipper scripts. The card model stated in the design, will represent one card in the 3 decks. In addition the card flipper script will handle how the card prefab will graphically flip using a flip method which will have two parameters that include the start and end image that are of sprite data type. The flip method uses two co-routines which stop and starts the flip. The flip uses an animation curve which scales on the X-axis to flip the scale of the car shown in figure 5.2. The scale curve start from 1 which is the full size of the card followed by 0 which renders to card on its edge, then back to 1.

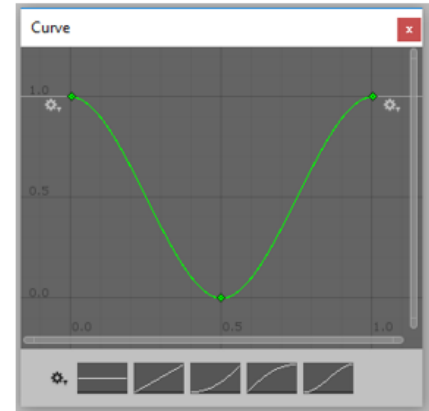


Figure 5.2 – Animation curve

The algorithm shown in figure 5.3 implements the card flip animation. The time variable will be incremented by `Time.deltaTime` which comes from the system clock, which is all inside a while loop for which time is less or equal to 1 second. Inside this while loop is where the scale is evaluated given the animation curve and point in particular time. Next part of the code scales the object that the image belongs, because if we scale the parent object, everything attached to that object or a child is also scaled. The local scale variable is set to the public animation scale predefined, where the X component is modified then reapply that local variable back to `transform.localScale`. Consequently when the time is greater than half a second, the sprite changes to the end image.

```
IEnumerator Flip(Sprite startImage, Sprite endImage)
{
    spriteRenderer.sprite = startImage;
    float time = 0f;

    while(time <= 1f)
    {
        //evaluate scale given
        float scale = scaleCurve.Evaluate(time);

        //increment counter
        time = time + Time.deltaTime / duration;

        Vector3 localScale = transform.localScale;
        localScale.x = scale;
        transform.localScale = localScale;

        if (time >= 0.5f)
        {
            spriteRenderer.sprite = endImage; // anytime after 0.5 seconds the user will see the front of the card
        }
        yield return new WaitForFixedUpdate();
    }
}
```

Figure 5.3 – Card flipping algorithm

5.2 Representing decks (Card stack script)

There is 3 'decks' that will need to be implemented, the player, dealer and deck or shoe which the cards will be dealt from. An empty game object will be created for each 'deck' that will have a Card stack script attached to it. To hold the cards in the card stack a list of integers is created which operate like a queue data structure where pop and push methods will be used to removed and add cards to the list in a first in first out manner. At the start of the deal after the player places their bets, four cards are dealt, meaning the values from the shoes card stack needs to be removed using the Pop() method. This method returns an integer value which represents a card index that is pushed to the player and dealer list data type. Figure 5.4 shows the implementation of the methods.

```
public int Pop()
{
    int temp = cards[0];
    cards.RemoveAt(0);

    //fires a message that card is removed to any subscribers
    if (CardRemoved != null)
    {
        CardRemoved(this, new CardEventArgs(temp));
    }
    return temp; // returns the first cards index from the stack
}

20 references
public void Push(int card)
{
    cards.Add(card);

    if (CardAdded != null)
    {
        CardAdded(this, new CardEventArgs(card));
    }
}
```

Figure 5.4 – Pop and push methods

This script has various method which enable card information to be retrieved for example, ReturnCardValueFromCardPositionInStack() method takes in what position in the list of integers as a parameter and returns the position of the value as a hand value. It converts the position in stack by using the ConvertCardIndexToHandValue() method which is show in figure 5.6.

5.2.1 Card rank

As the card indexes will range from a 0 to 207, the rank of the cards will have to be defined as it can only represent values from each suit from 1-11. Knowing the card indexes ranges from 0-207 and that each suit will have a total of 13 cards which hold ranks to 1-11. In order to separate the indexes to appropriate card ranks, modulo division is used. Modulo division is expressed as a percentage sign that performs the computation that returns the remainder of the division. For example for a card index of 27, to retrieve the card rank the calculation of $27 \% 13$ is performed which will return the remainder which is 12, this value is checked with various if condition to determine where that index refers in the card ranks. The result would return a 2. Below shows the calculations and references to return a card rank.

$$\begin{aligned} \text{Card index} &= 103 \\ \text{Card rank} &= \text{card index} \% 13 = 12 \\ 12 &= \text{King} \end{aligned}$$

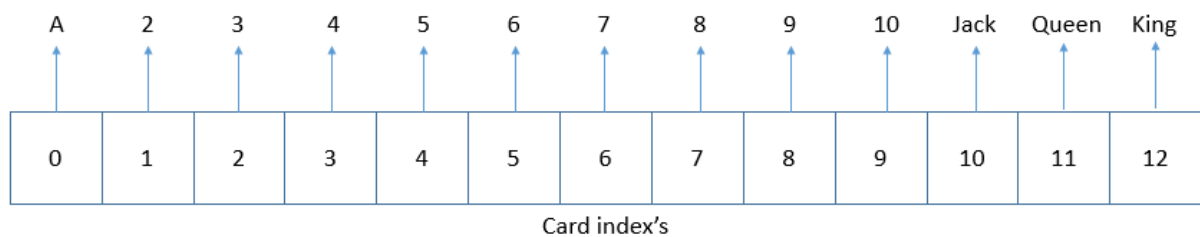


Figure 5.5 - illustration of card index to card rank

In terms of implementation, the function below which resides in the card stack script tackles the conversion.

```
public int ConvertCardIndexToHandValue(int cardIndex)
{
    // divides all images in 13's which separates the suits and assigns values (gets values from 0 -12)
    int cardRank = cardIndex % 13;

    //normal card
    if (cardRank >= 1 && cardRank < 10)
    {
        cardRank += 1;
    }
    // Face cards
    else if (cardRank > 9 && cardRank < 13)
    {
        cardRank = 10;
    }
    else
    {
        cardRank = 11; //Aces only return 11 as we need to check for blackjack
    }
    return cardRank;
}
```

Figure 5.6 – Function that converts card index to hand value

5.3 Card shuffle

For the card shuffling, I have chosen to implement a Fisher-yates algorithm which randomizes a list of integers which will form the card indexes which the card will refer to. The reason I have chosen this algorithm is that it is relatively simple to implement, and is fairly efficient running at linear complexity. The algorithm has been mathematically proven to be unbiased for as every permutation is equiprobable. The algorithm requires to shuffle 208 integers that represents each of the cards in a deck. The create deck method is called when a shuffled deck is needed for the shoe, it first clears the integer list then populates it with integers from 0 to 208.

The steps for the algorithm can be defined below,

```
public void CreateDeck()
{
    cards.Clear();

    for (int i = 0; i < 208; i++)
    {
        //Put the cards in the list from index 0 to 208
        cards.Add(i);
    }
    int n = cards.Count;
    while (n > 1)
    {
        n--;
        int k = Random.Range(0, n + 1);
        int temp = cards[k];
        cards[k] = cards[n];
        cards[n] = temp;
    }
}
```

Figure 5.7 – Fisher-yates shuffle implementation (Old method)

5.4 Main Menu Implementation

The main menu scene consist of various game objects, which functionality is to navigate to the main features of the application. I have created an empty game object named Main controller which has a script component attached to it. The script is called tool bar menu script and is used in the application in all scenes to enable the user to navigate around the application. Main menu has three buttons, play game, simulation and exit which navigate to their respective scenes. When the exit button is clicked the application is closed. Figure 5.8 illustrates the hierarchy architecture for the main menu scene.

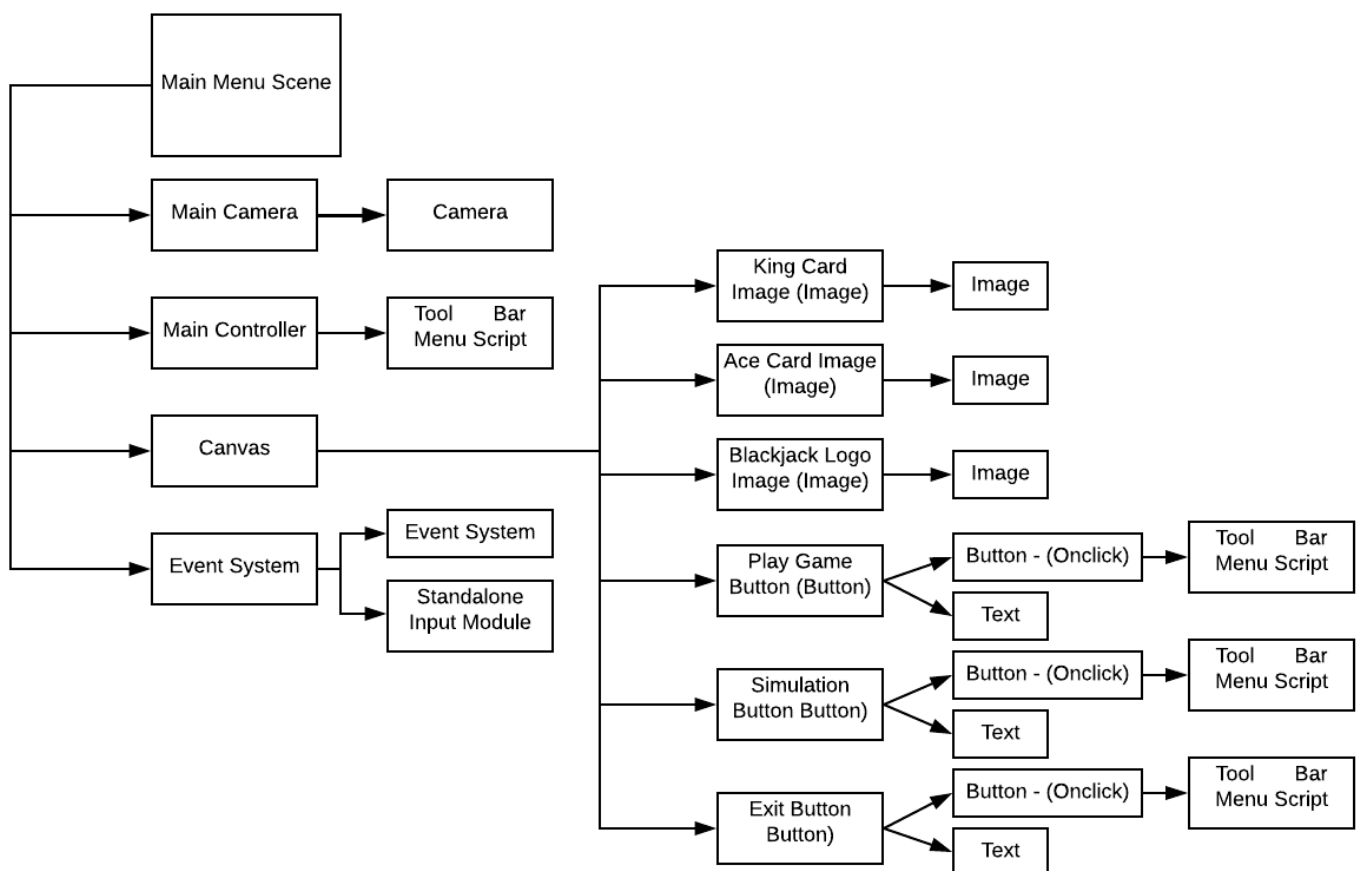


Figure 5.8 – Main Menu Hierarchy Architecture

5.5 Blackjack Simulation Implementation

The blackjack simulation scene will achieve the objective for the simulation playing mode, it comprises of various game objects that build the interface and hold the decks for player for example. The player, dealer and deck game object has a card stack script attached to it which enables cards to be drawn and kept to simulate the game. To handle if the player splits a hand, another game object was created called player splitted hand that also has a card stack script attached. The logic for the simulation to be executed resides in the simulation controller where the script, blackjack simulation is attached as a component, this script will be further explained below. Figure 5.8 illustrates the hierarchy architecture for the blackjack simulation scene.

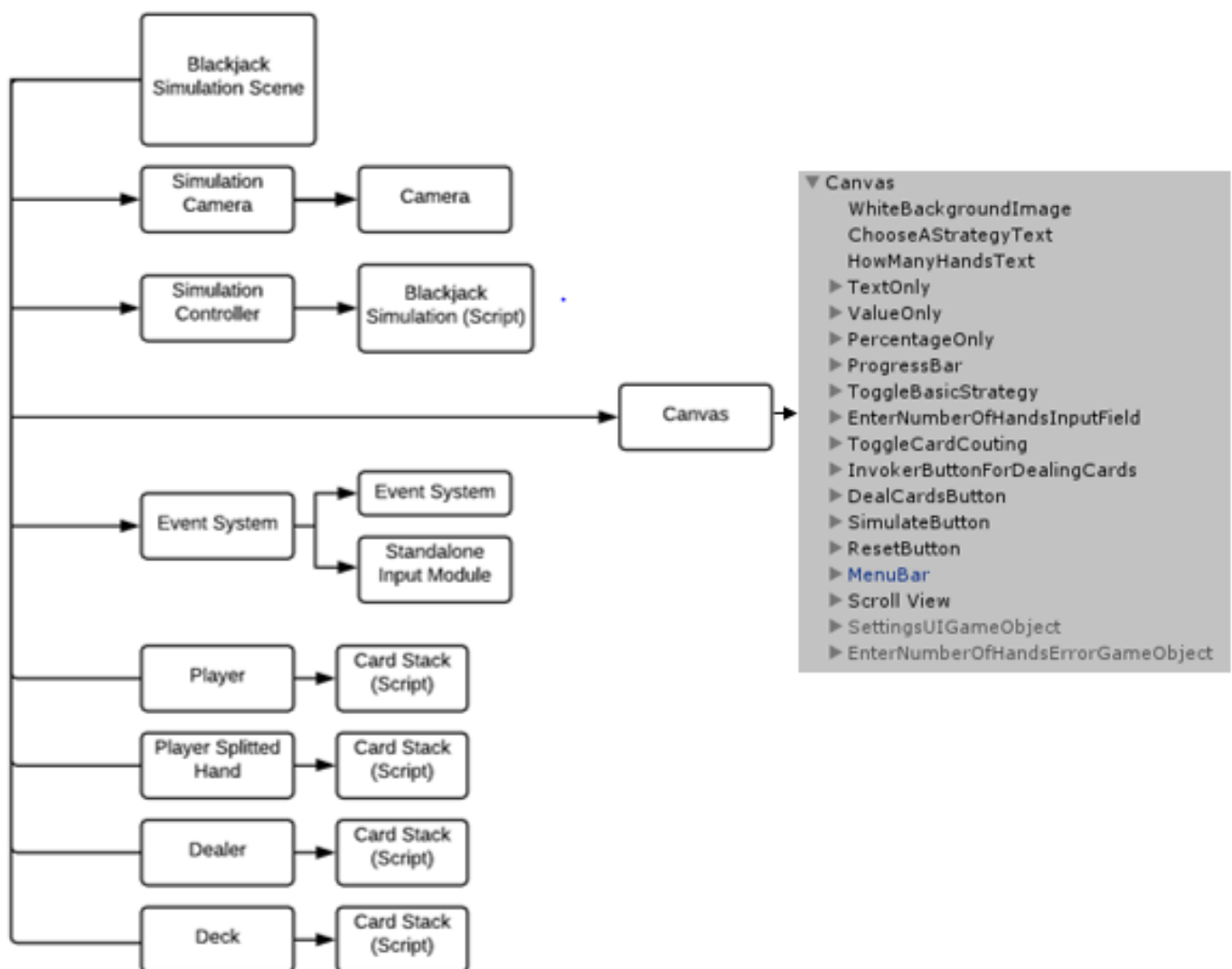


Figure 5.9 – Blackjack Simulation Hierarchy Architecture

5.5.1 Simulation logic

The logic for the simulation of a blackjack game is all handled in one script, called blackjack simulation. The script is arranged in regions to help organise the code in different functionalities. When the simulate button is clicked it goes through an invoker method which includes a while loop that stop invoking a deal when the number of hands to played reaches it limit. Bets are made in this while loop for each deal. How the bets are made is determined by the true count, the table below illustrates how the true count affects the betting for the simulation and the code that implements it. For example if the true count is 3.5, and your basic betting unit is £10, you would multiply your betting unit by 4, which will result in a £40 bet. Furthermore if the true count exceeds 5, it is estimated you have more of advantage against the house and should bet top of the bet spread which is 12 units. The true count is calculated with a card counting class which will be explained later in this section.

True count	How much to bet
+1 or lower	1 x betting unit
+2	2 x betting unit
+3	4 x betting unit
+4	Double the true count and use that for how many units to bet
+5 or higher	12 x betting unit

```
if (cardCounter.CalculateTrueCount() <2)
{
    PlayerBettingUnit = 1;
    PlayerBet = PlayerBettingUnit * PlayerBet;
}
else if (cardCounter.CalculateTrueCount() >= 2 && cardCounter.CalculateTrueCount() <3)
{
    PlayerBettingUnit = 2;
    PlayerBet = PlayerBettingUnit * PlayerBet;
}
else if (cardCounter.CalculateTrueCount() >= 3 && cardCounter.CalculateTrueCount() <4)
{
    PlayerBettingUnit = 4;
    PlayerBet = PlayerBettingUnit * PlayerBet;
}
else if (cardCounter.CalculateTrueCount() >= 4 && cardCounter.CalculateTrueCount() <5)
{
    int DoubledTrueCount = Convert.ToInt32(cardCounter.CalculateTrueCount() * 2);
    PlayerBet = DoubledTrueCount * PlayerBet;
}
else if (cardCounter.CalculateTrueCount() >= 5)
{
    PlayerBettingUnit = 12;
    PlayerBet = PlayerBettingUnit * PlayerBet;
}
```

Figure 5.10 - Betting implementation

The deal cards method executes a single simulation of a deal with the dealer and player, it checks which strategy the user has chosen and pick the correct xml document which to retrieve the strategy from. I have created two methods that retrieve the xml strategies, it takes two parameters first being a card stack data type for the players hand and a Boolean value if the simulation player has decide to split. The first task was to change the formatting of the both the deal and player hands so that it be used to check against the xml, for example if the hand containing an ace card, will be change to 'A' followed by the next card values. A node list is created where for each loops are used to sort through the strategies. However the xml strategy for the hi-lo is different from the basic strategy as it use the true count to make its decision, figure 5 handles the parsing if a hi –lo index exist and stores the variable accordingly.

```
foreach (XmlNode node in myNodeList)
{
    string test = node.Attributes[0].InnerText;
    if ((playerHandXMLFormatted == test))
    {
        foreach (XmlNode child in node.ChildNodes)
        {
            if (child.Attributes[0].InnerText == dealerUpCardString)
            {
                if (is2ndHand == false)
                {
                    adviceString = child.ChildNodes[0].InnerText;
                }
                else
                {
                    adviceStringForSplittedHand = child.ChildNodes[0].InnerText;
                }

                //checks if the Hi-Lo Index is null
                try
                {
                    string indexTemp = child.ChildNodes[1].InnerText;
                    if (indexTemp.Contains(" "))
                    {
                        // if this is true, the condition for the advice will have two paramters
                        // for example if a player has 'A6' versus a dealers 2, the player will only double down when the
                        // index is between 0.01 and 0.10
                        string[] indexParameters = indexTemp.Split(' ');
                        HiLoIndexFloat[0] = decimal.Parse(indexParameters[0]);
                        HiLoIndexFloat[1] = decimal.Parse(indexParameters[1]);
                    }
                    else
                    {
                        HiLoIndexFloat[0] = decimal.Parse(child.ChildNodes[1].InnerText);
                    }
                }
                catch
                {
                    HiLoIndexActive = false;
                }
            }
        }
    }
}
```

Figure 5.11 – Parsing the Hi-lo XML strategy

When the advice string is populated by the xml methods, an 'if statement' handles and executes the correct code the advice string, figure 5.12 shows the implementation.

```
if (adviceString == "SPLIT")...  
  
else if (adviceString == "STAND" && playerSplit == false)...  
  
else if (adviceString == "DOUBLEDOWN" && playerSplit == false)...  
  
else if (adviceString == "HIT" && playerSplit == false)...
```

Figure 5.12 – Code for handling XML return advice

The advice string can be only either one of the four, 'SPLIT', 'STAND', 'DOUBLEDOWN' and 'HIT'. First advice string I implemented was the when the simulation ask for the play to hit, or ask for another card. If the strategy chosen was hi-lo, on a return of a 'HIT' the players hi-lo index is checked against the hi-lo index from the xml strategy if it has that property exist as some hands for example if you had a hand value of 20 against a dealers up card of 6, a hi-lo index does not exist as you always 'STAND'. This results in, if the index is larger, player will stand on the contrary if the index is less or equal to the simulation player will hit. The hit player method I created is recursive as multiple hits can occur when playing blackjack until a hand is bust or exceeds 21. The hit method consist of a card taken from the deck or shoe and then being popped to the player's hand. The specified xml parsing method are called again to retrieve the advice string followed by examining if the hand is bust. If the hand is bust or the following advice return after a hit is a stand, the next method execute is called DealersTurnAndEndConditions () which deals card to the dealer and handles all the winning conditions and pay outs.

When the return advice is 'DOUBLEDOWN' the simulation player will receive only one card and the bet is doubled. If the hi-lo strategy is chosen, the players hi-lo index is checked against the parsed xml hi-lo index. If the player hi-lo index is greater, the bet is doubled, however if it is lower a normal hit is called. On the contrary if basic strategy is chosen the bet is always doubled. For both strategies on a 'DOUBLEDOWN' if the players hand exceeds 21, they bust and DealersTurnAndEndConditions () is called.

When the return advice is 'SPLIT', one of the cards from the player hand is transferred to a new card stack variable called playersplithand. Each card stack is pushed a card from the deck resulting in both hands having two cards. The specified xml parsing methods are called to retrieve the advice for both hands. The advice for both hands are execute with their respective method, for example a stand will not ask for a card and therefore DealersTurnAndEndConditions () is called, for doubling down, it can only occur on the first hit and therefore a normal hit player method is executed. The hit method takes in a parameter for a card stack type and therefore can be used for the split hand. In some cases when the simulation player has two aces and splits, the blackjack rules state that only one card can be dealt out for each hand and no blackjacks can occur.

When the return advice is 'STAND' the simulation player does not hit to receive another card but stands on their hand followed by DealersTurnAndEndConditions () method being called.

DealersTurnAndEndConditions () method gets called when the simulation player busts or decided to stand on their hand. The purpose of this procedure is to handles who wins the blackjack game recently simulated. It first check if the player has used a split as further checks are made. If a split has not occurred the code checks if the dealer or player has blackjack from their first two cards. In the blackjack rules if the dealer has blackjack and the player does not, the dealer win and the player loses their bet. However if the player has blackjack and the dealers up card reveals for an ace, the player can ask for an insurance bet which plays 2:1; or when both player blackjack, a push occurs and the money is neither lost nor paid. After checking for no blackjacks the dealer plays their hand which consist of asking for a card from the deck until the hand value is 17 or better, in other words, the dealer must draw on a 16's but stand on all 17's; this is achieved by a while loop. In this implementation, the dealer will not draw if the simulation player has busted. Next it checks who has won but comparing each hand value, if a split occurs, that hand is also compared and the simulation player is paid accordingly if it has won against the dealer.

After, the player and dealer hand information are stored in local variables and displayed in an easy format to read in a list box structure using a text field prefab that is dynamically coded. The text colour changes for each different end conditions, red for a dealer win, green for a player win, yellow for a player blackjack and black for splits and pushes. I have chosen to implement text colours as it make the ending conditions more readable and eye catching.

Following in order to simulate another blackjack deal, the various variables used must be reset; each card stack class has a public method called reset which clears the integer list in the class that previously held the card indexes. The data for number of wins, loses, pushes splits and players bank roll and their percentage equivalent are displayed in text views after all the number of hands have be simulated.

In support of simulating blackjack deals, the cards from the deck or shoe will eventually be depleted and the deck will need to be populated and shuffled again. Inside the while loop is an 'if statement' that checks if the cards left in the deck is less than 40, and if this condition is true, the deck is repopulated and shuffled. I have chosen to shuffle when the shoe is less than 40 cards as it rules out the exception where the dealer and player do not have enough cards to play which will affect functionality. Figure 5.13 illustrates the implementation of the while loops that simulates the number of hand of what the user inputs in the text field after pressing the simulate button.

```
do
{
    MakeBet();

    ///Calls the method from the button on click listner that executes the cards to be dealt
    DealCardsButton.onClick.Invoke();
    HandsPlayed += 1;

    "Debug logs and Deck resets"

    //populates Progress Bar
    ProgBar.value += (float)(progressBarIncrement + timeSlicedForProgressBarIncrement);
} while (HandsPlayed != NumberOfHandsToBePlayedInt);
```

Figure 5.13 – While loop that simulates blackjack deals

5.5.2 Card counter class

The card counter class enables a systematic implement to manage the true counts and hi-lo indexes. The true count is calculate by the running count divided by how many decks are remaining, this is used for deciding how much the simulation player should bet. On the other hand, the Hi-lo index is used for deciding what action the player should perform. Hi-lo index is calculated by the running count divided the number of remaining cards in the deck. Both are of decimal type as it allows for precision and not prone to round-off errors. The card counter class is used in the simulation logic when a card is dealt to the dealer and player, as it needs to update the running count for the true count and hi-lo index to be valid to be used.

Figure 5.14 illustrates the implementation to calculate the running count, it take in a integer parameter which is the card index and converts it into a card value from a public method in the card stack called `ConvertCardIndexToHandValue()`. The card value is then evaluated to determine to what value the card represents and then added to the running count variable inside the `AddToRunningCount()` method.

```
private int FindtheCountValueOftheCard(int cardIndex)
{
    cardValue = player.ConvertCardIndexToHandValue(cardIndex);
    if (cardValue == 2 || cardValue == 3 || cardValue == 4 || cardValue == 5 || cardValue == 6)
    {
        return 1;
    }
    else if (cardValue == 10 || cardValue == 11 || cardValue == 1)
    {
        return -1;
    }
    else
    {
        return 0;
    }
}

public void AddToRunningCount(int CardValue)
{
    runningCount = runningCount + FindtheCountValueOftheCard(CardValue);
    CardsLeftInDeck -= 1;
    cardsUsed += -1;
}
```

Figure 5.14 – Implementation of converting card index to running count

5.6 Blackjack Game Implementation

The blackjack game part of the application has various game object and scripts working in conjunction with each other. The main function of this scene is to play a blackjack game as close to the real experience in a casino. The game controller script is the main script that manages how the game is played, in addition to the chip betting controller game object that consist of two scripts which handle betting for standard and insurance bets. The player is not allowed to split as this was not implemented due to time constraints therefore only three card stacks are only used; the player, dealer and the deck. The canvas consist of unity user interface objects for example buttons, images and text views to allow to users to interact and play the game. Figure 5.15 illustrates the hierarchy architecture for the blackjack game scene.

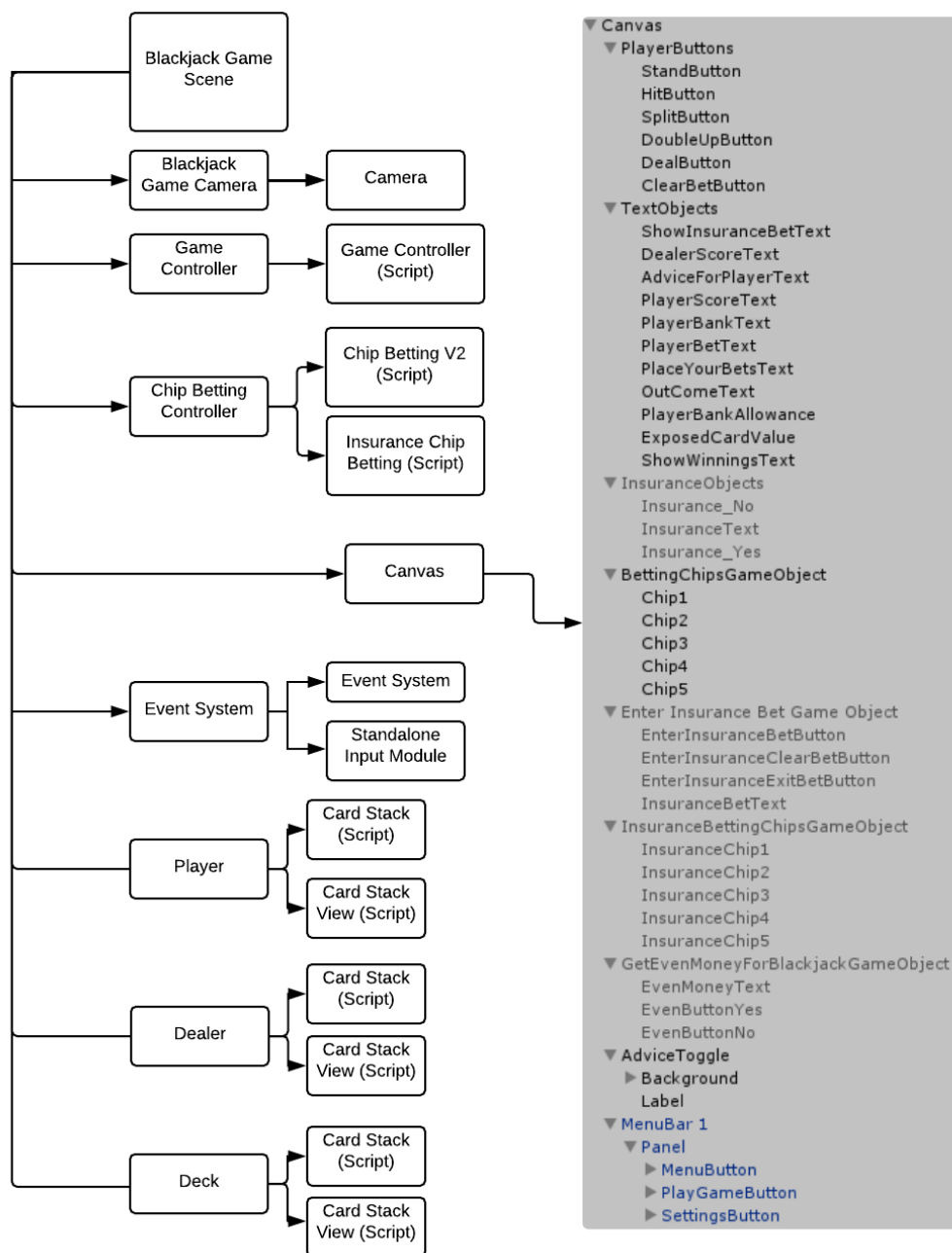


Figure 5.15 - Blackjack Game Hierarchy Architecture

5.6.1 Blackjack game logic

To create the logic and flow of this scene, appendix L, illustrates the flow of how a blackjack should be implemented. I have created various methods and co-routines to implement the blackjack game logic, for example the StartGame() method invokes a co-routine that deals card to both player and the dealer. In this scene the cards are displayed in the UI, therefore the dealer, player and deck must have a card stack view script which is attached to the game object.

This script work with event delegates in the card stack script that trigger when a card is add and removed. A method called AddCard() takes in three parameters which function is to display and flip the card to the scene. It creates a copy of the card prefab and transforms its position along displaying the set card image from the card index variable as a parameter. Figure 5.16 implements this algorithm.

```
void AddCard(Vector3 position, int cardIndex, int positionalIndex)//positional index is the position of the hand
{
    //handles toggling the card face up or down
    if (fetchedCards.ContainsKey(cardIndex))
    {
        if (faceUp)
        {
            CardModel model = fetchedCards[cardIndex].Card.GetComponent<CardModel>();
            model.ToggleFace(fetchedCards[cardIndex].IsFaceUp);
        }
        return;
    }

    GameObject cardCopy = (GameObject)Instantiate(cardPrefab);
    cardCopy.transform.position = position;
    CardModel cardModel = cardCopy.GetComponent<CardModel>();

    //Flips the card
    flipper = cardCopy.GetComponent<CardFlipper>();
    flipper.FlipCard(cardModel.faces[cardIndex], cardModel.cardBack);

    cardModel.cardIndex = cardIndex;
    cardModel.ToggleFace(faceUp);

    SpriteRenderer spriteRenderer = cardCopy.GetComponent<SpriteRenderer>();
    if (reverseLayerOrder)
    {
        spriteRenderer.sortingOrder = 51 - positionalIndex;
    }
    else
    {
        spriteRenderer.sortingOrder = positionalIndex;
    }
}
```

Figure 5.16 – Add card method in card stack view script

The first deal method is a co-routine that gives the player and dealer two cards, the second card of the dealer must be faced down. However before the cards are dealt the player must place a bet, this is implemented by the chip betting controller, which houses the chip betting v2 script. This scripts takes in public button variables which are created in the UI to represent betting chips. The images are sourced from <https://nl.vecteezy.com/vector-kunst/99332-poker-fiches> and is cropped to be the image background for each betting chips, the value of chips are of 1, 5, 10, 25 and 50 dollars. After the player places their bets, the cards are dealt by a co-routine which suspends the execution of the code for 0.3 seconds each time when a new card is dealt.

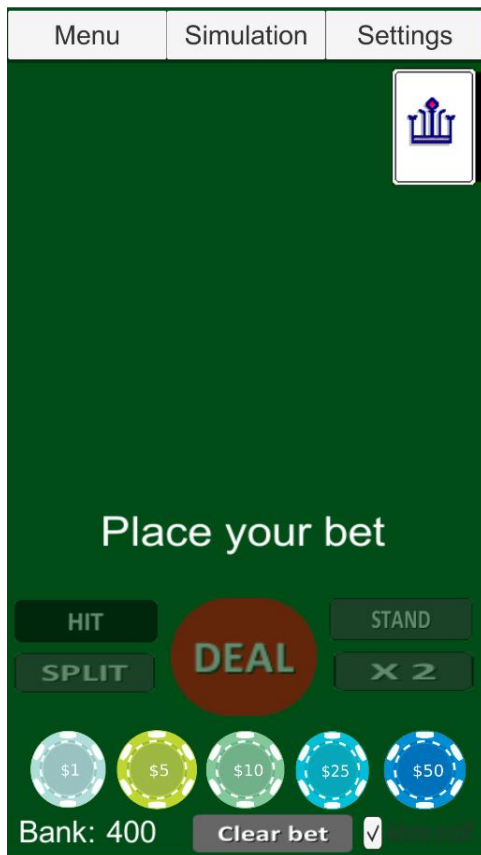
The player starts off with 400 dollar in the bank to place bet. Next, the advice toggle is set to true, the same method used in the blackjack simulation to parse to XML basic strategy is used against the result is display in the scene. The advice toggle can be set to true or false anytime the game is active. The player or user then has the 'hit', stand' and 'double down' buttons made available to make a decision of what to play. The split button is implemented in the UI but not coded, this will be discussed in the evaluation. Hitting, standing and doubling down all are implemented in the same style as in the blackjack simulation. To follow another co-routine is used to simulate the dealer drawing cards, standing on all 17 and above but hitting on 16's. In conclusion to determine who has won, two methods are used, EndConditions() and CalculateWinningsAndLosings(). End condition method determines who wins the game by the players and dealers hands or who has busted, while the calculate winnings and losings method, pays the player if they have won.

When the dealers up cards show for an ace, the player has the option to make an insurance bet. If the player want to place an insurance bet a new UI object is displayed in the screen along with new betting chips game object being used.

5.7 UI Finalised

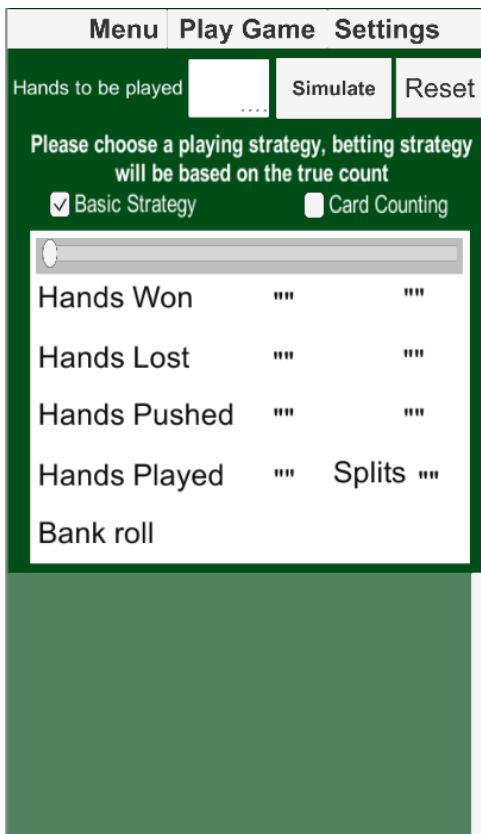


The UI for the main menu consist of a blackjack logo along with two cards representing a blackjack. The menu included 3 buttons which navigate to the blackjack game and blackjack simulation scene. The 3rd button exits the application.



The blackjack game UI consist of various buttons which are used to play the game, along with betting chips which are displayed towards the bottom of the screen. There is a tool bar at the top of the interface which navigates to different scenes.

Figure 5.18 – Finalized UI (Blackjack Game)



The blackjack simulation UI consist of the same menu tool bare which navigates to different scenes. Toggles are used to switch betting strategies. A progress bar is also implemented to signify growth in simulation achievement. The middle of the UI contains various text fields that display the simulation information clearly in a large font and which is spaced out.

Figure 5.19 – Finalized UI (Blackjack Game)

5.8 Testing overview

When debugging in Unity, there are various ways of testing the application, one option is to build the project and export it every time to an Android device, however, this can take too long. So to aid in the speed of development Unity introduced a way to test your projects at runtime on any android device via Unity remote. In terms of testing code to determine functionality, unity has a built in tool called Unity Test Runner that enables you to test code both in both edit and play mode.

Unity remote is located in Play store. You have first set your Android device in developer mode and enable USB debugging. Then plug your device in via USB and change project settings to show on all available Android devices. In conclusion when the play button is pressed, what is shown on the game screen is mirrored on the android device along with full its functionality.

In conclusion unity unit testing was not implemented but the use of the console window was helpful as output from algorithms can be viewed by using `Debug.Log()` which prints out values on to the console window.

To test my application I have created a testing table below which encompasses various test to showcase working and non-working aspects of the project.

5.8.1 Test table

The test screen shots can be found in appendix N.

Test No.	Description	Expected Outcome	Actual Outcome
Main menu scene			
1.1	Scene navigation test; main menu, blackjack simulation and blackjack simulation scenes can all navigate to each other	Selected scene is loaded	As expected
1.2	In the main menu scene, when the exit button is clicked, application should be closed	Application closes	As expected
Blackjack simulation scene			
2.1	Basic strategy: Player hands is 20 and dealer up card is a 7, therefore player should stand. Dealer draws to 20 and player wins 10 units	Simulation player stands and wins, bank roll = 410	As expected
2.2	Basic strategy: Player hands is 15 and dealer up card is a 5, therefore player should stand. Dealer draws to 20 and player loses 10 units	Simulation player stands and loses, bank roll = 390	As expected
2.3	Basic strategy: Player hands is 17 and dealer up card is a 10, therefore player should stand. Dealer draws to a blackjack and player loses 10 units	Simulation player stands and loses, bank roll = 390	As expected
2.4	Basic strategy: Player receive a blackjack and dealer does not deal	Player win, receive pay for 3:2 which is equal to 15 units	As expected

2.5	Basic strategy: Player receives a hand value of 8, advice says to hit, however dealer has blackjack and player loses.	Player loses as dealer has blackjack	As expected
2.6	Basic strategy: Player receives a hand value of 11, advice says to double down. Dealer draw for an 11 and hits; hand value of 17 so dealer stands.	Player doubled bets and win against the dealer as 21 is a better hand 17	As expected
2.7	Hi-Lo strategy: Player receives blackjack while the dealer draw for a 15.	Player wins with and get paid 3:2 which is 15 units. The dealer does not deal as the player has already won	As expected
2.8	Hi-lo strategy: Player stand on a hard 18 while the dealer draw for a 20.	Dealer wins and the player loses 10 units	As expected
2.9	Hi-lo strategy: Player receives a hard 16, and hi-lo strategy says to stand against a dealers 6. The deal deals has the hand value is a hard 16.	The dealer bust and player wins, bank roll is increased by 10	As expected
2.10	Hi-lo strategy: Player receives an 11 and hi-lo says to double down. Dealer draws hard 13 and bust when hits.	The dealer bust and player wins, bank roll is increased by 20	As expected
2.11	Hi-lo strategy: Player receives 2 aces against a dealers 8. Player draws to a bust of 23. Dealers hand = 13	Dealer win as player busts	Aa expected
2.12	Hi-lo strategy: Player receives 2, 10's and should split against a dealers up card of 9. Player split, first hand = 20, so stands. Second hand equals to 19, also stands. Dealers draws and stands to a hard 19.	Hand 1 wins, Hand 2 pushes. Bank roll should equal 410	The text views displaying how many hands, won, loses and pushes are incorrect. Bank roll update is wrong
Blackjack game scene			
3.1	When the betting chips are clicked the players bet increases accordingly and is displayed in the UI as a text field.	50 is clicked once followed by a 10	As expected
3.2	Card are dealt to both player and dealer and are displayed in the scene. The first card of the dealer will be faced down.	Cards are displayed, in addition to their respective hand values as a text field.	As expected
3.3	When the user wants to hit, they should be dealt a card from the shoe	A new card is drawn from the shoe and added to the player's hand.	As expected
3.4	When the user wants to stand, the user does not ask for another card and therefore the dealer will draw cards.	The dealer will hit until 17 or higher is reached but less than 21	As expected

3.5	When the user wants to double down, original bet is doubled and one card is only dealt to the player, if the player does not bust	The original bet is doubled and	As expected
3.6	Advice mode is turned on by the user	Advice is displayed with a text view	As expected
3.7	The player gets dealt a hard 13 and hit against a dealers up card of 9.	The player receives a 10 and bust therefore the dealer wins.	As expected
3.8	The player gets dealt a hard 6 and hits against a dealers up card of 8	The player receives a 10 which equals to hand value, while the dealer bust on a hand of 22. The players and therefore a winning text is shown with the amount of money won in total, including the original bet.	As expected
3.9	The player stands on a hard 18 against a dealers up card of 8.	Both player and dealer have a hard 18 and the players hand is pushed. The money is returned.	As expected
3.10	Player stands on 15 and the dealer get blackjack	Player lost, 50 units lost	As expected
3.11	Dealers up card is an ace, the insurance game object should appear, asking if the player wants to make an insurance bet. Player clicks yes and make an insurance bet of 50. The dealer does have black and the player hands is pushed	The money for the original bet is returned, however it is unclear if the insurance bet was successful	Unclear if insurance bet is successful
3.12	Player receive a blackjack and the dealer does not	The players original bet was 50 therefore 75 is won. Text field displays	As expected

5.8.1 Testing the xml parsing

One important feature that was critical for the simulation to work was parsing the xml strategies. I was not overly confidence with xml so I had to try out different testing methods in order to get the results I needed. I created methods to test if my parsing was working correctly in the blackjack simulation script. The methods function was to write to the text file the all the information for each hand strategy's output. So for example for basic strategy, if the player, has a hand value of a soft 16 and the dealer has a up card value of 5, the advice would be to double down. Figure 5.20 shows the code for the testing and the results from parsing the XML.

```
void ParseXmlAndWriteToFile(string playerHand)
{
    string[] array = { "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "A" };

    String data = xmlFileBasicStrategy.text;
    XmlDocument xmlDoc = new XmlDocument();
    xmlDoc.Load(new StringReader(data));

    string xmlPathPattern = "//root/basicstrategy";
    XmlNodeList myNodeList = xmlDoc.SelectNodes(xmlPathPattern);

    foreach (string upcard in array)
    {
        foreach (XmlNode node in myNodeList)
        {
            List<string> StrLineElements = node.Attributes[1].InnerText.Split(' ').ToList();

            if ((playerHand == node.Attributes[0].InnerText) && StrLineElements.Contains(upcard))
            {
                foreach (XmlNode child in node.ChildNodes)
                {
                    adviceString = child.InnerText;

                    sr.WriteLine("Player hand=" + playerHand + " Upcard =" + upcard + "||| " + adviceString);
                }
            }
        }
    }

    sr.WriteLine("");
}
```

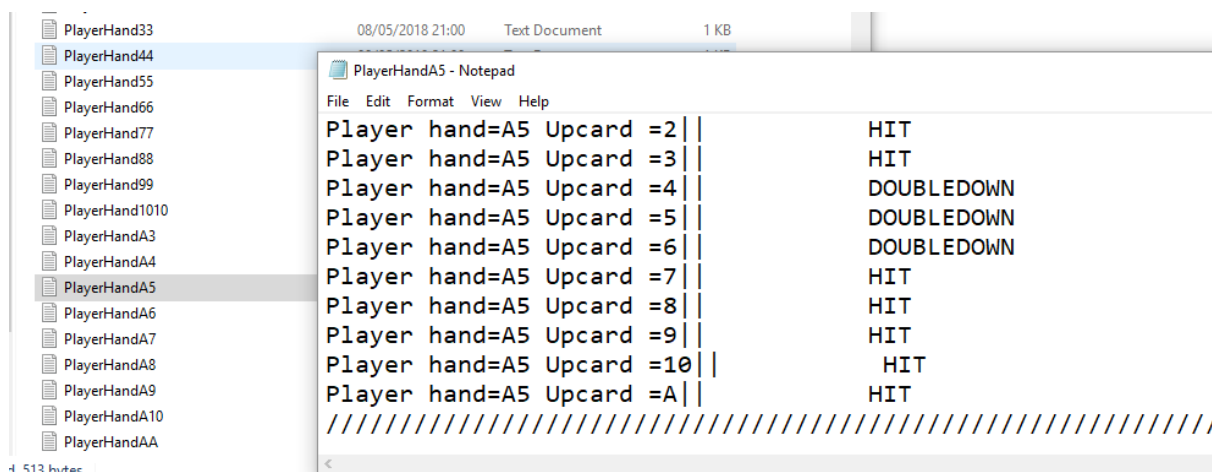


Figure 5.20 – Code and text file for XML parsing test

5.8.2 Simulation data

In order to test if the blackjack simulation made skillful bets in order to win as much possible, 10, 100 hand simulation were conducted for each of the strategies. The data retrieved was however inaccurate for the basic strategy as when a split occurred the number of wins, losses and pushes did not add up. Therefore in the code, a bug is present. The code, when a split occurs currently, adds an extra win, lost or push to the final total, the data is incorrect but does show useful information on a larger scale.

On the other hand, testing for the Hi-lo simulation is accurate as all the hand information equaled to the number of hands simulated which is 100. For both strategies they produced profits of more than double the original starting bank roll which is 400. In comparison a higher of wins occurred for the hi-lo method and a lower lost average however more money was won overall by basic strategy. In addition the reason why basic strategy may have had a higher profit average is from the errors in the code where splits occur. Figure 5.21 and figure 5.22 shows the results from the simulation test.

Figure 5.21- Basic strategy simulation test

Test	Won	Lost	Pushed	Splits	Total played	Bank roll	Profit
1	41	55	6	2	102	1175	775
2	35	60	8	1	103	135	-265
3	39	53	8	0	100	920	520
4	47	52	1	0	100	870	470
5	47	49	8	3	104	1345	945
6	43	52	7	1	102	720	320
7	40	56	7	1	103	490	90
8	54	40	8	1	102	1660	1260
9	42	60	5	4	107	300	-100
10	40	50	12	1	102	820	420
Average	42.8	52.7	7	1.4	102.5	843.5	443.5

Figure 5.22 - Hi-lo simulation test

Test	Won	Lost	Pushed	Splits	Total played	Bank roll	Profit
1	53	42	5	0	100	595	195
2	38	55	7	0	100	470	70
3	38	54	8	0	100	495	95
4	47	45	8	0	100	1245	845
5	46	48	6	0	100	950	550
6	42	51	7	0	100	1090	690
7	49	43	8	0	100	1695	1295
8	41	53	5	0	100	565	165
9	53	46	9	0	100	103	-297
10	40	50	10	0	100	1115	715
Average	44.7	48.7	7.3	0	100	832.3	432.3

6 Evaluation

6.1 Project management report (Interim)

Since the initial report progress on my project has been steady. The first objective I wanted to achieve on a large scale was to complete the fundamentals of the blackjack game. With help from online resources, I managed to create card class which holds all of the card types as a sprite. The images I used for the suits and the back images were sourced online. Within a few weeks, I managed to create code to shuffle only one deck as my deck object only consisted of 52 cards, I did this to make debugging test easier. A deck is represented by a card stack class, which is an array of card model prefabs. A prefab is a reusable game object stored in Project View. The card stack class is also reused for the player and dealer to signify their hands. The card stack view class is a function that collects the card model prefab and physically displays it in the game scene. It includes a flipper which uses physics components to graphically show a card flip.

In terms of functionality to the game, the main brain is the game controller object which in itself has a script. This script controls most of the blackjack gameplay including currently how the advice mode works however in the future, I would separate this script into smaller scripts accordingly to my design brief. In terms of functionality the player can play a simple game of blackjack that has the features to hit and stand and that recognize if the dealer or the player has blackjack or not. There are some bugs towards betting but being able to double down is operational. Advice mode is currently working but only with the use of basic strategy and not card counting implementation. In regards to project risk, no major risk has occurred, I have been saving work regularly and also keeping various backups on various devices.

To conclude there are improvements that can be made as I started tasks while other tasks were not completed, therefore affecting the timeline of my task list. In the coming weeks, I hope to get on track and aim to achieve each objective chronologically. See appendix E.

6.2 Project management report (Final progression)

Since the interim report progress, I have left the implementation of the blackjack split and focused on the simulation aspect as it is a critical feature of my application. When coding the simulation I experimented first parsing XML files and how they are used to stored data. I watched various YouTube tutorials online which boarded my understanding. Coding the simulation loop was extensive, and a lot of debugging was used to careful fix small bugs in the application constantly. I used visual studios watch list to observe different key variables that were dynamically critical to simulation a single hand correctly.

6.3 Project achievements

In this section, the project objectives are evaluated.

Objective 1 – Rules of the blackjack game implemented for both blackjack game and simulation scenes

In terms of decks used, 4 decks or 208 cards from each of the 4 suit are created and shuffled using a fisher-yates shuffle.

For the blackjack simulation, most of the rules were implemented and working, hitting, standing and doubling is correctly functioning. On the contrary splitting technically works, in the aspects that two hands are created and played normally, and the results are correct.

However a bug in the code alters the number of win, loses and pushes, that affect the end number of total hands played. In regards to the blackjack game, hitting, standing and doubling functions correctly, however UI for the splitting feature was created however was never coded due to time constraints as more time was taken to finish the splitting functionality in the simulation.

The dealer in the simulation and blackjack game is implemented where standing on soft and hard 17 takes place. The only different in dealer features, is that in the simulation the dealer will not finish their hand or deal if the player bust; the dealer will always deal card even if the player busts. I have chosen to do this, to show different type ways that blackjack can be played. Both the simulation and game detect if the blackjack has occurred, evidence is shown in the test table.

Objective 2 – Develop 3 playing modes

Objective 2.1 – Default mode

This mode was partially completed, as stated above in objective 1, splitting was not implemented. On the other the card were show graphically on the scene and also had a flipping animation. The application recognized if the player or dealer won the game and bets were paid accordingly. Insurance was also implemented but partial functional as the text view showing if the insurance was successful was dysfunctional. UI objects were created for even money, however was not implemented. The game controller script implements the blackjack game logic. The setting button across the top is obsolete.

Objective 2.2 – Advice mode

When the application is loaded, advice mode is set to false, however if the advice toggle is set to true. The advice string will be displayed on the next time a card is dealt.

Objective 2.2.1 – Basic strategy

The advice objective is achieved as the basic strategy xml is parsed correctly, evidence is shown in the test table and showcase video. The information from the player hand and the dealer up card are used to retrieve a basic strategy decision for that hand. The advice is shown clearly on the scene below the player's card.

Objective 2.3 – Computer simulation

The computer simulation partially functions but how ever has some bugs in terms of displaying the correct number of wins, losses and pushes after a simulation. The correct information is displayed in the list box as the it states clear what hand value the player and dealer have, in addition to who has won and if a blackjack has occurred. Splits display the correct hand information as seen in the test 2.12 but do not register correctly in the wins, losses and pushes text views. Another aspect that was not designed but not functioning is the reset button, the button should clear all previous variables and allow for another simulation to concur.

Objective 2.3.1 – Implement basic strategy

Implementing this objective with the blackjack simulation is partially successful as parsing the xml was done correctly. When an advice is return from basic strategy, I have implemented code for each of the 4 possible outcomes. The hit method is recursive which aids in efficient.

Objective 2.3.2 – Implement hi-lo counting method

Parsing of the xml was done correctly and tested thoroughly, and with the help of the card counting class, the Hi-lo index can be calculated to determine what the decision the simulation player should execute.

Objective 2.3.3 – Devise a betting system

This objective partially works as when a user simulates a low number of hands, it is clear from test 2.1 and 2.4 that the simulation player is paid correctly. The betting system works using the card counting class which calculates the true count and bets higher if the true count is higher.

Objective 3 – Graphical User Interface

This object is achieved the 3 main scenes; main menu, blackjack game and blackjack simulation all have a corresponding graphical user interface. The background for all the scenes is green which relate to a real life blackjack table.

Secondary Objectives

Objective 1 – Instructions for the game

Instructions for the game was not implemented as the more time needed to be allocated for the main functionality for example the blackjack simulation to fully work.

Further improvements and bugs

Many aspects of the application can be improved, first, all the bugs can be fixed. This includes displaying the correct number of win, losses and pushes when a simulation is running. Secondly, the betting system is sometimes inaccurate and needs to be more consistence when dealing with the greater number of simulations. Thirdly, splitting needs to be fixed as there is some confusion what circumstances need to be set for a split to fully function.

On the contrary, if the application was fully working and the resulting data was accurate. A new feature could be added whereby a further simulation could be performed. For example in testing, I simulated 10, 100 simulations but this was done separately. A new feature could offer a solution whereby further simulations can occur and the resulting data displayed in graphs or transposed to an excel spreadsheet.

In regards to the blackjack game scene, music can be added to enchase user experience, along with animations that pop up when the user has won by blackjack. The settings button could also be implemented which allows different rules to be played. Deck sizes could be altered, alongside if the dealer is allowed to stand on all hard and soft 17's.

Time management could be improved, as time was held short because of time spent programming left the report weak in terms of the conclusion.

7 Conclusion

In conclusion, most of the objectives were met and most of the blackjack rules were implemented in a reasonably standard. The various bug affects the compliance of the results but however give the general sense that the simulation wins progressively. The simulation was able to develop profits with both playing strategies by parsing XML and using the Hi-lo index. The blackjack game visually is pleasing and functions partially, able to play a simple blackjack game without the feature of splitting and no even option available.

Using unity to develop my application for android gave many advantages as it was fairly easy to use and enable me to use C# which I have experience with. The Blackjack player project was enjoyable to program and was an interesting topic that I have learned a lot from.

8 Appendix

Appendix A: Task List (Initial report)

#	Task Name	Description	Duration (days)
1	Initial report (Deliverable)	DELIVERABLE: Write the Initial report deliverable	7
2	Design GUI	Design the main page which will be used for the application, the page should be green just like most blackjack tables, the theme should be consistent throughout.	7
3	Blackjack strategies research	This task will help get familiar with the different ways to play blackjack with different counting strategies. For example basic strategy and the hi-low card counting method.	5
4	Code for creating and shuffling the deck	Implementation of code that can automatically shuffle the deck of 4.	14
5	Bets should be able to be made	The user will have a set amount on which they can bet. They can change this bet value.	3
6	Code for implementing deal for the player and the dealer	The code should implement the deal, where 2 cards are shown for each player. The cards should be shown graphically on the graphical user interface. The photos of the cards will be stored locally.	4
7	Implementation of hitting, standing and busting.	This code will give the opportunity for the player and dealer to gain a new card within their set limits. For example, when the dealer has the hand value of 17 - 21, the dealer should stop receiving more cards. Busting is when a player's hand is over the value of 21.	7
8	Code to implement blackjack for both player and dealer	The application should recognize when the player receives a blackjack and should pay their bets straight away, this is true unless the first card of the dealer is an ace. When this occurs the player can either stand or take even money. When the dealer receives a blackjack and the player doesn't, the player's bets are lost.	3
9	Implementation of splitting, doubling, and insurance	Splitting occurs when a player receives the same number value as their first 2 cards, this enables the player to play two hands. On a special occasion 2 aces appear, you are only allowed to receive 1 card further on each hand, no blackjack can occur as blackjack only happens on the player's first 2 cards.	7

10	Interim report (Deliverable)	DELIVERABLE: Write the interim report deliverable	14
11	Basic strategy advice	Implement the advice mode where advice is given using the basic strategy chart/table.	21
12	Implement card counting for blackjack player simulation	Implement the ability to input a number of hands wanting to be played, in which when clicking the deal button, the hands will be dealt and information about this history will be clearly shown. This is the main part of the application in where card counting will be in action.	21
13	Bonus objective : Create a login page / Database	Bonus Objective: Creates a user functionality that enables them to create an account in which personal and in-game information will be stored on a server.	7
14	Final Report (Deliverable)	DELIVERABLE: Write the final report deliverable	25

Appendix B: Initial task list progress review

#	Task Name	Progress comment
1	Initial report (Deliverable)	Completed
2	Design GUI	The design is completed and is implemented for the game scene, however, the menu is not yet implemented in Unity.
3	Blackjack strategies research	Research into card counting method has been achieved
4	Code for creating and shuffling the deck	Completed
5	Bets should be able to be made	Code is in place however so issues are slowing down progress
6	Code for implementing deal for the player and the dealer	Yes this code is in place via the game controller game object
7	Implementation of hitting, standing and busting.	Yes this code is in place via the game controller game object
8	Code to implement blackjack for both player and dealer	Yes this code is in place via the game controller game object and the card stack class
9	Implementation of splitting, doubling, and insurance	Splitting is not implemented as the way I have to design my project I may have to limit the number of splits allowed. Code for doubling is implemented but not for insurance bets
10	Interim report (Deliverable)	Soon to be submitted
11	Basic strategy advice	The foundations have been coded by the use of blackjack basic strategy but could be improved by the addition of card counting algorithms
12	Implement card counting for blackjack player simulation	I have written a basic script for the class however still needs a lot of work

13	Bonus objective : Create a login page / Database	Not yet started
14	Final Report (Deliverable)	In progress

Appendix C: Interim report task list

#	Task Name	Description	Duration (days)
1	Modify the card model class to incorporate 4 decks instead of 1	Decks currently works with 1 deck, import more cards to the card model	2
2	Enable for player to allow betting	There are currently some bugs towards this objective. It may be because the mono develops update loop is interfering somewhere else.	3
3	Code insurance bets and finish implementing double down feature	Insurance bets are made when the dealer up card is an ace	3
4	Code for managing bank balances	This include when the dealer pays the player. In addition to the timer which increases the user's bank balance after a certain time and an option to watch a promotional video which will also increase the balance.	2
5	Code for splitting of cards	Splitting the cards occurs when the player has two of the same card type, in my application, I will only allow splitting once due to complexity	5
6	Basic strategy advice (with card counting)	The string will be outputted via a text game object. Card counting algorithms could be implemented.	5
8	Implement card counting for blackjack player simulation	The complete full counting system will be implemented here. The simulation will open up a window which will display all the relevant information regarding the computer's decisions and results.	10
9	Implement UI for the main menu	This will be what the user will see first.	2
10	Implement UI and information for the instruction scene	This task is on the secondary objectives and will not hard the progress of the main objectives	3

Appendix D: Initial Time Plan



This represents a time buffer for other projects/tasks.

		University Calendar Weeks																																												
#	Task Name	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36															
1	Initial report (Deliverable)	D					Buffer time to complete other projects/task																Buffer time to complete other projects/task																							
2	Design GUI																																													
3	Blackjack strategies research																																													
4	Code for creating and shuffling the deck																																													
5	Bets should be able to be made																																													
6	Code for implementing deal for the player and the dealer																																													
7	Implementation of hitting, standing and busting.																																													
8	Code to implement blackjack for both player and dealer																																													
9	Implementation of splitting, doubling, and insurance																																													
10	Interim report (Deliverable)																																													
11	Basic strategy advice																																													
12	Implement card counting for blackjack player simulation																																													
13	Bonus objective : Create a login page / Database																																													

Appendix F: Risk Analysis

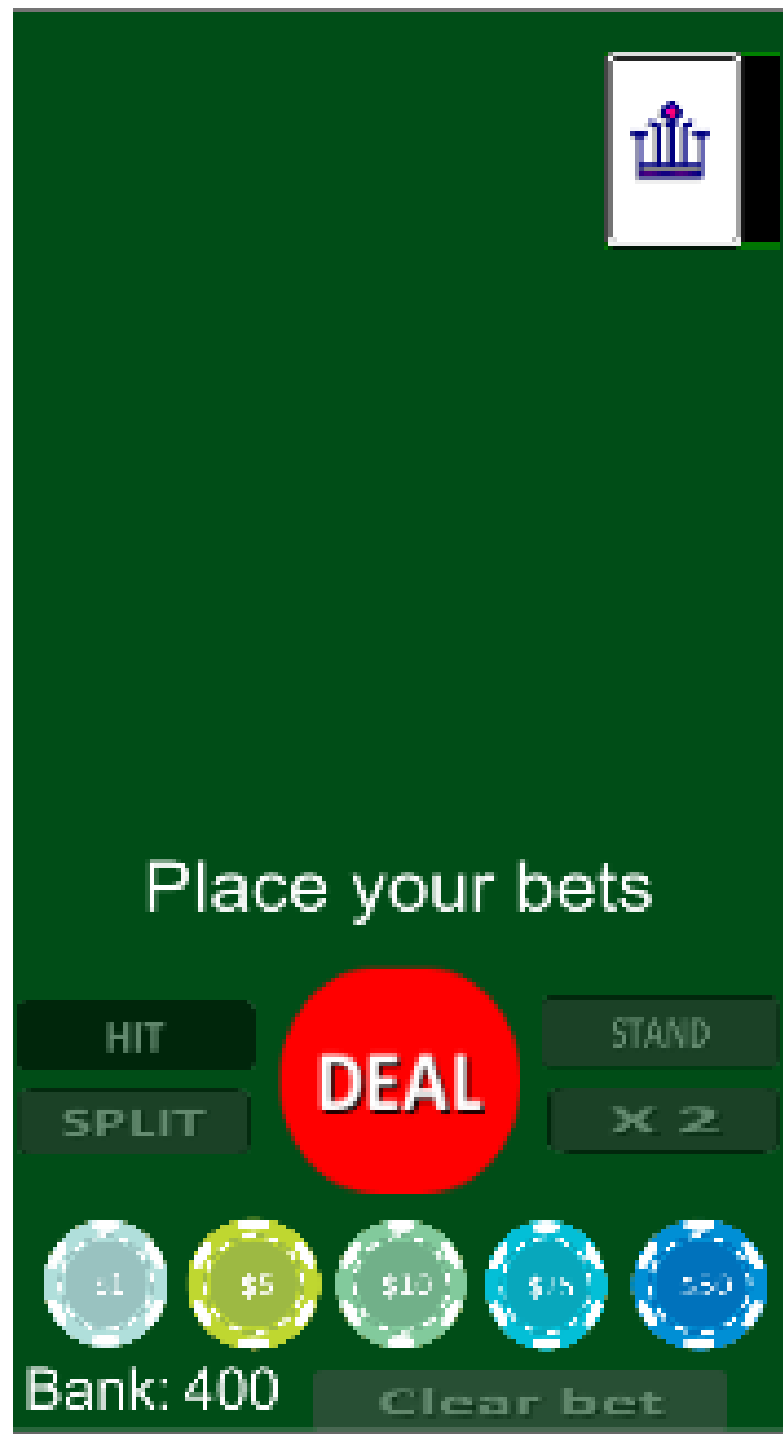
Risk	Severity (L/M/H)	Likelihood (L/M/H)	Significance (Sev. x Like.)	How to Avoid	How to Recover
Data loss	H	M	HM	Each time a major milestone in both report writing and implantation of code, multiple versions should be stored. In addition, backups should be taken	Reinstate from backups
Loss of backups	H	L	HL	Store versions of data on different technologies	Use alternate data storage, for example, memory sticks.
Bonus objectives	L	L	LL	To avoid, focus on delivering the main features of the application and use time efficiently	Remove the bonus objectives.
Lack of programming knowledge / Platform	M	M	MM	Study the platform knowledge in depth using videos tutorials and books	Different solutions will be needed to be considered. Is there a simpler feature available? Get help from AST?
Multiple deadlines	H	H	HH	Manage time effectively in order to balance time for working on different modules.	Concentrate on delivering the essentials to each project.

Appendix I: Card sprites

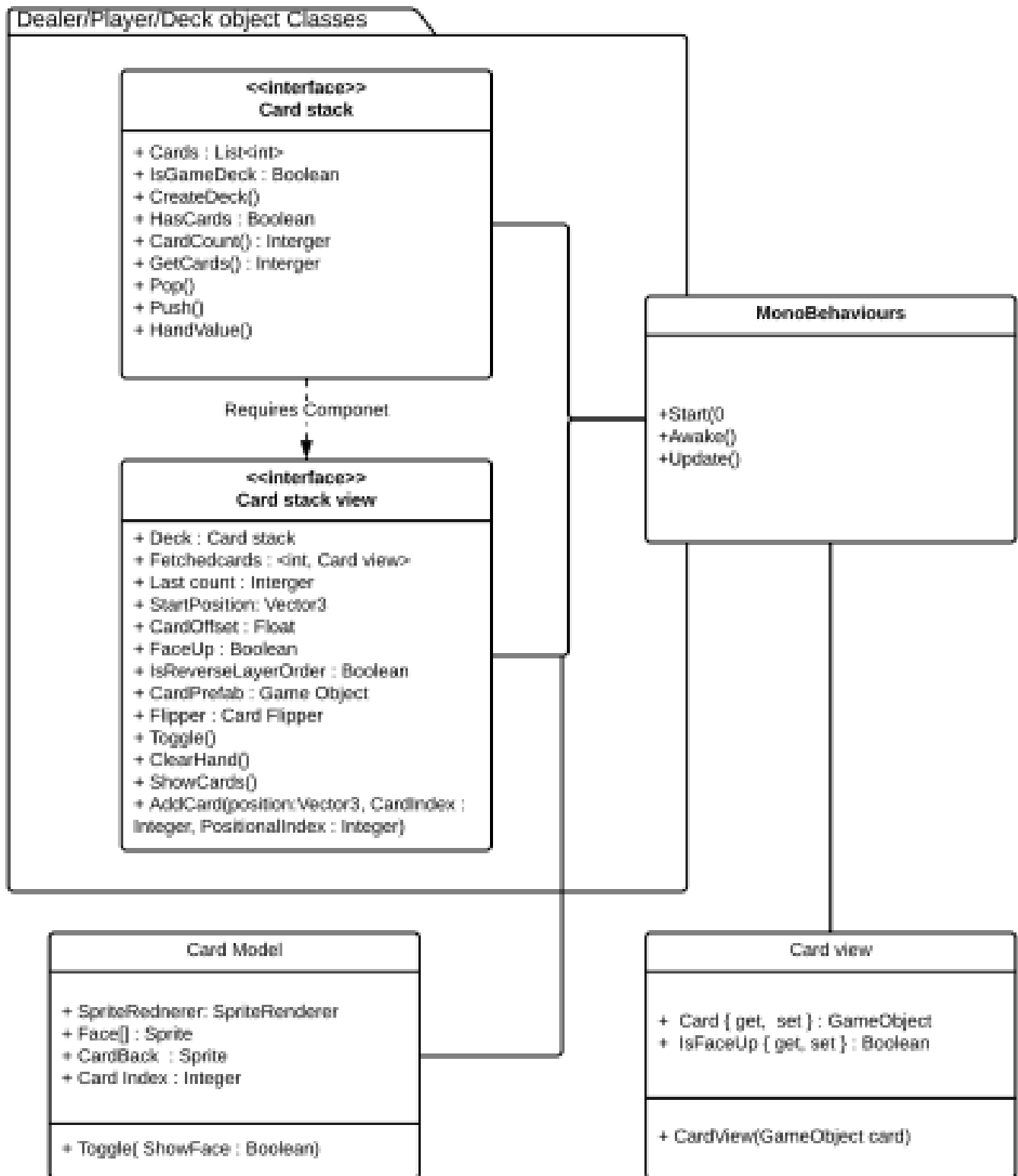


Image sourced from - <http://www.thehouseofcards.com/img/misc/Deck-72x100x16.gif> - Original image name = Deck-72x100x16.gi

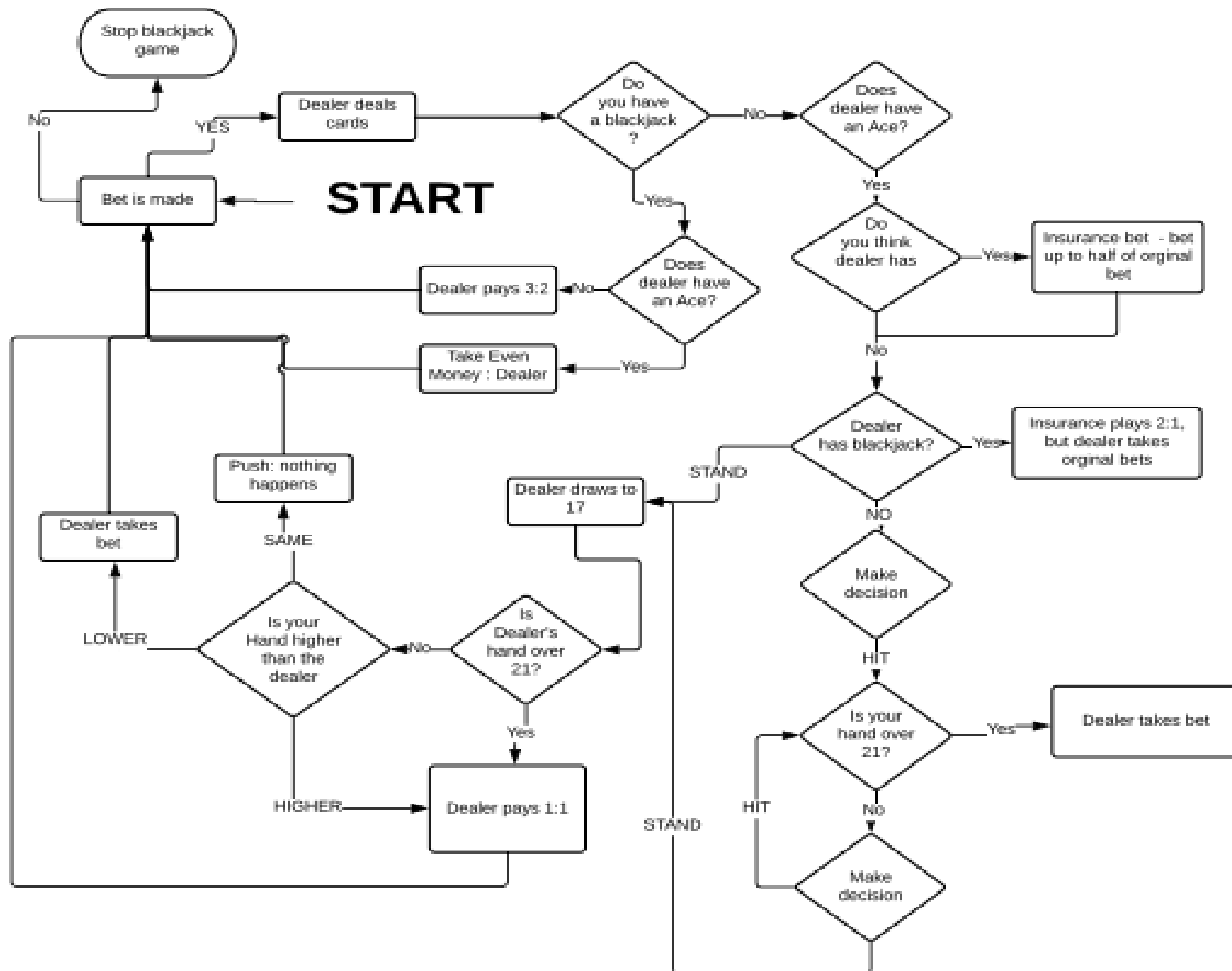
Appendix J: UI: Game scene (Unity)



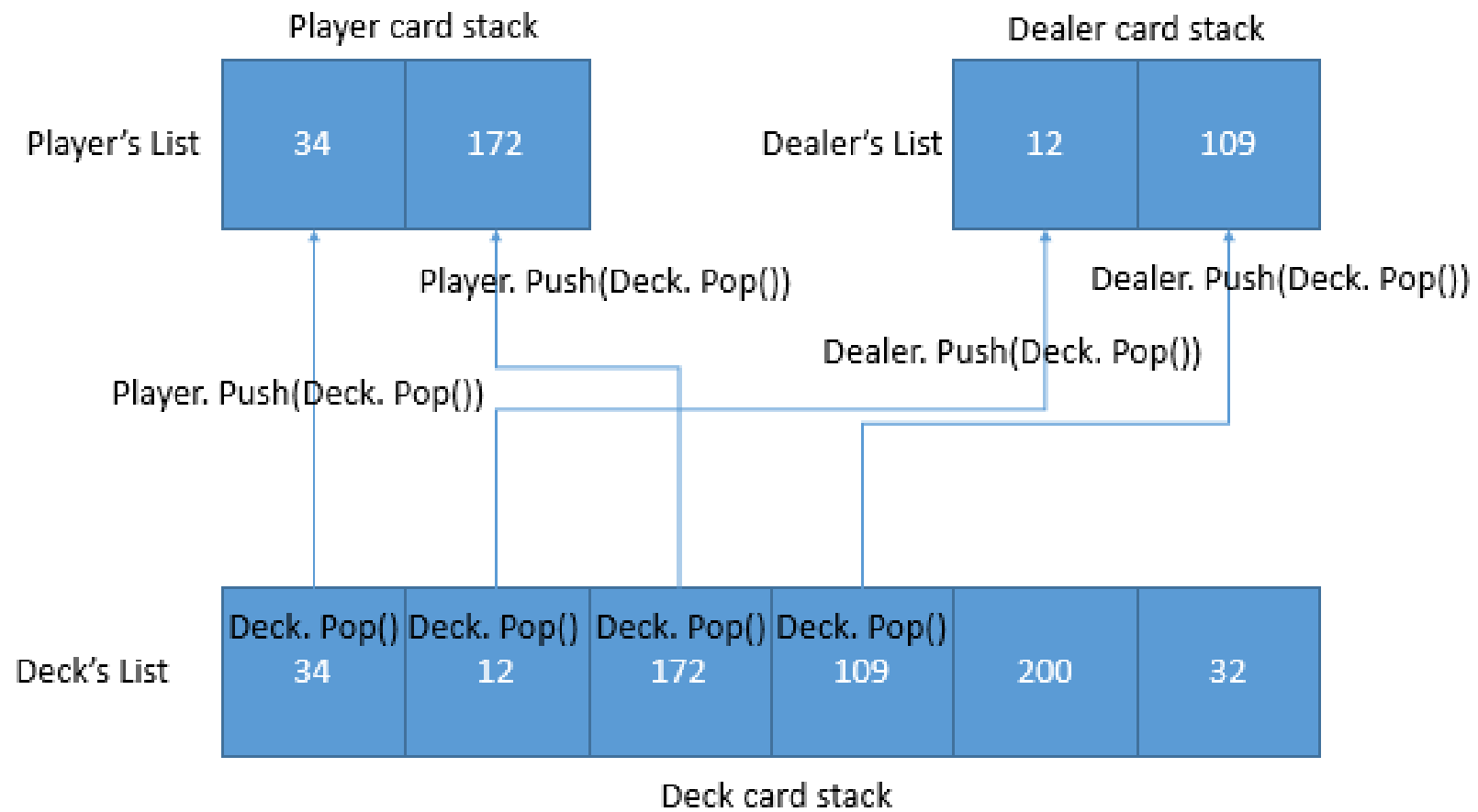
Appendix K: Card stack/card stack view/card model



Appendix L: Blackjack loop diagram

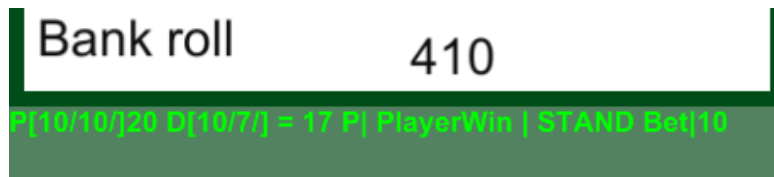


Appendix M: First deal (card stack representation)



Appendix N: Testing result (Screen shots)

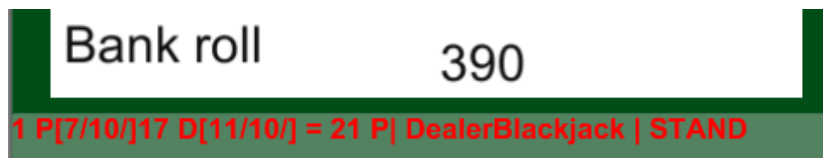
Test 2.1



Test 2.2



Test 2.3



Test 2.4



Test 2.5



Test 2.7

Bank roll

415

1 P[11/10/]21 D[5/10/] = 15 P| PlayerBlackjack | STAND Bet|10

Test 2.8

Bank roll

390

1 P[9/9/]18 D[10/10/] = 20 P| DealerWin | STAND Bet|10

Test 2.9

Bank roll

410

1 P[10/6/]16 D[10/6/9/] = 25 P| PlayerWin | STAND Bet|10

Test 2.10

Bank roll

420

1 P[3/8/126 D[3/10/2/10/] = 25 P| PlayerWin | DOUBLEDOWN Bet|20

Test 2.11

Bank roll

390

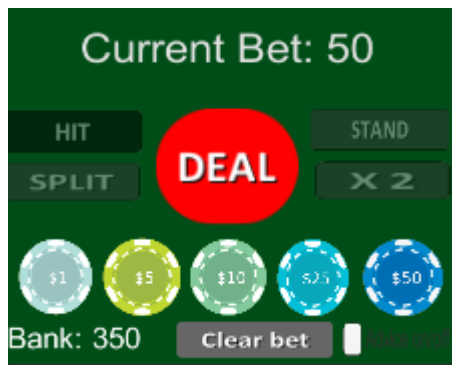
1 P[11/11/11/10/]23 D[5/8/] = 13 P| DealerWin | HIT Bet|10

Test 2.12

Hands Won	3	150%
Hands Lost	0	0%
Hands Pushed	3	150%
Hands Played	2	Splits 1
Bank roll	405	

1 P1[10/10/]=20| P2[10/9/] D[10/9/]=19 P1|PlayerWin P2|Push Bet|10

Test 3,1



Test 3.2



Test 3,3



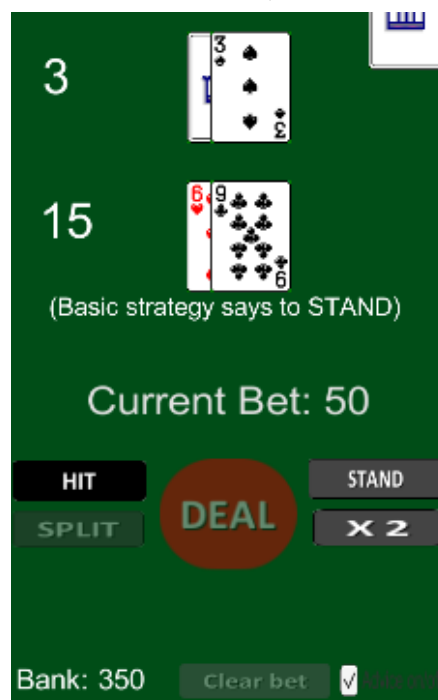
Test 3,4



Test 3,5



Test 3,6



Test 3,7

20



23



House wins
losings : 50

Test 3,8

22




16




Dealer bust/Better Hand
Winnings : 100

Test 3,9

18




18




Push
50 returned

Test 3,10

21

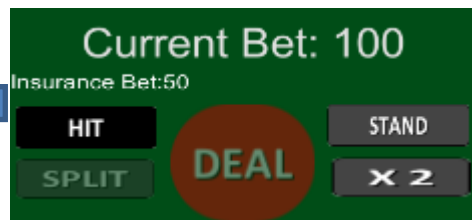
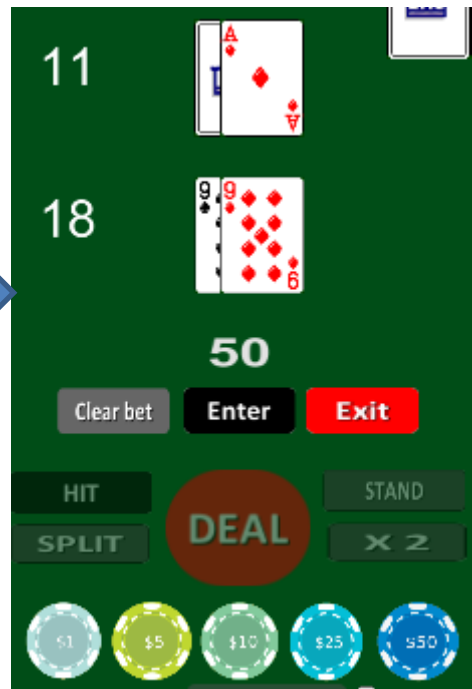


15



(Basic strategy says to HIT)
Lost : Dealer blackjack
losings : 50

Test 3.11



Test 3.12

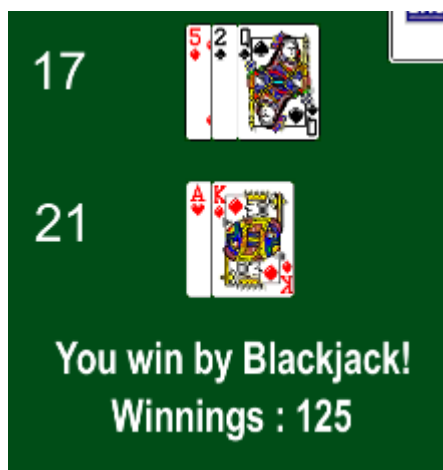


Figure 4.1 – Setting up the development kits – page 15

Figure 4.2 – Unity Architecture – page 16

Figure 4.3 – Main menu (Hand drawn) – page 17

Figure 4.4 – Game scene UI (Hand drawn) – page 18

Figure 4.5 – UI concept in Unity for game scene – page 19

Figure 4.6 – Simulation scene UI (Hand drawn) – page 20

Figure 4.7 – Scene navigation diagram– page 21

Figure 4.8 – Card model UML– page 22

Figure 4.9 – Advice diagram– page 24

Figure 5.1 – Inserting sprite in card model class – page 27

Figure 5.2 – Animation curve – page 28

Figure 5.3 – Card flipping algorithm– page 28

Figure 5.4 – Pop and push methods – page 29

Figure 5.5 - illustration of card index to card rank – page 30

Figure 5.6 – Function that converts card index to hand value – page 30

Figure 5.7 – Fisher-yates shuffle implementation (Old method) – page 31

Figure 5.8 – Main Menu Hierarchy Architecture – page 32

Figure 5.9 – Blackjack Simulation Hierarchy Architecture – page 33

Figure 5.10 - Betting implementation – page 34

Figure 5.11 – Parsing the Hi-lo XML strategy – page 35

Figure 5.12 – Code for handling XML return advice – page 36

Figure 5.13 – While loop that simulates blackjack deals – page 37

Figure 5.14 – Implementation of converting card index to running count – page 38

Figure 5.15 - Blackjack Game Hierarchy Architecture – page 39

Figure 5.16 – Add card method in card stack view script – page 40

Figure 5.17 – Finalized UI (Main Menu) – page 41

Figure 5.18 – Finalized UI (Blackjack Game) – page 32

Figure 5.19 – Finalized UI (Blackjack Game) – page 42

References

Thorp, Edward O. (1966) *Beat The Dealer*, Vintage Books, a Division of Random House, London Often, 2017, "The history of Blackjack". Source[online] Available from: <https://www.blackjackapprenticeship.com/resources/history-of-blackjack/> [Accessed: 10/10/2017].

Henry Tamburin Ph.D., "The ultimate blackjack strategy guide". Source [online]. Available from: <https://www.888casino.com/blog/blackjack-strategy-guide> [Accessed: 10/10/2017].

John Grochowski, 2006, "How to Play Blackjack". Source [online] Available from: <http://entertainment.howstuffworks.com/how-to-play-blackjack.htm> [Accessed: 12/10/2017].

Burton, B. (2017), "Twenty-One: How to Play Casino Blackjack. Source [online] Available from: <https://www.thoughtco.com/how-to-play-blackjack-537106> [Accessed: 12/10/2017].

Radeck, K. (2003). *C# and Java: Comparing Programming Languages*. [online] Msdn.microsoft.com. Available at: <https://msdn.microsoft.com/en-us/library/ms836794.aspx> [Accessed 8 Dec. 2017].

Tuliper, A. (2014). Unity - Developing Your First Game with Unity and C#. [online] Msdn.microsoft.com. Available at: <https://msdn.microsoft.com/en-gb/magazine/dn759441.aspx> [Accessed 8 Dec. 2017].

Pluralsight.com. (2014). Unreal Engine 4 vs. Unity: Which Game Engine Is Best for You?. [online] Available at: <https://www.pluralsight.com/blog/film-games/unreal-engine-4-vs-unity-game-engine-best> [Accessed 18 Jan. 2018].

Steiner, B. (2003). The Unreal Engine's Rise to Video Game Fame. [online] Popular Mechanics. Available at: <http://www.popularmechanics.com/culture/gaming/a9178/how-the-unreal-engine-became-a-real-gaming-powerhouse-15625586/> [Accessed 18 Jan. 2018].

Rouse, M. (2011). What is mobile operating system? - Definition from WhatIs.com. [online] SearchMobileComputing. Available at: <http://searchmobilecomputing.techtarget.com/definition/mobile-operating-system> [Accessed 18 Jan. 2018].

International Data Corporation, (2015). Smartphone OS Market Share, 2015 Q2 Available Online: <http://www.idc.com/prodserv/smartphone-os-market-share.jsp> [18 Jan. 2018]

Garrison, J. (2010). *What is the Linux Kernel and What Does It Do?*. [online] Howtogeek.com. Available at: <https://www.howtogeek.com/howto/31632/what-is-the-linux-kernel-and-what-does-it-do/> [Accessed 18 Jan. 2018].

USwitch. (2016). Mobile operating systems - what are they and which is best?. [online] Available at: <https://www.uswitch.com/mobiles/guides/mobile-operating-systems/> [Accessed 20 Jan. 2018].

Android (2018). Platform Versions Available Online: <https://developer.android.com/about/dashboards/index.html> [Accessed 20 Jan. 2018].

Developer.apple.com. (2018). App Store - Support - Apple Developer. [online] Available at: <https://developer.apple.com/support/app-store/> [Accessed 20 Jan. 2018].

Msdn.microsoft.com. (2018). What Is a Sprite?. [online] Available at: <https://msdn.microsoft.com/en-us/library/bb203919.aspx> [Accessed 21 Jan. 2018].