



Computer Science NEA

A-LEVEL H446/03

CHARLIE ROBERTS

2023-2025

Table of Contents

Analysis	4
Problem Identification.....	4
Introduction	4
Why should a computational approach be used?	4
Identifying Stakeholders.....	5
Interview One, Sam	5
Interview Two, Julia.....	7
Interview Three, George.....	9
Research of Existing Solutions.....	11
Game 1 – 8 Ball Pool	11
Game 2 – Chess.com	13
Game 3 – Tetris.....	15
References for the Research of Existing Solutions	16
Proposed Solution.....	17
Essential Features.....	17
Limitations.....	18
Hardware and Software requirements.....	18
Success Criteria and Requirements	19
Design.....	23
Structure of the Solution.....	23
Menu Structure	23
Game Structure	28
Further Explanation and Justification of the Structure of the Solution	30
Usability Features.....	31
Algorithms.....	33
Variables.....	38
Test Plan.....	42
Development of Coded Solution.....	48
Stage 1: Ball Physics and Collisions.....	49
Stage 1 Review.....	58
Stage 2: Pool table and Basic UI.....	60
Stage 2 Review.....	87

Stage 3: Cue Mechanics and Basic UI	89
Stage 3 Review.....	130
Stage 4: Game Rules and Logic	133
Stage 4 Review.....	155
Stage 5: AI Development	157
Stage 5 Review.....	162
Stage 6: Menu Creation.....	163
Stage 6 Review.....	200
Stage 7: Audio	202
Stage 7 Review.....	207
Stage 8: Final Additions.....	209
Stage 8 Review.....	215
Evaluation	217
Testing the final solution	217
Testing against the Test Plan	217
Stress Testing	240
Performance Testing.....	240
Inputs Testing	241
Menu Testing	242
Multiple User Testing.....	242
Has the success Criteria been met?.....	243
Evidence of implementation of Usability Features.....	251
What usability features have been implemented?	251
What usability features have not been implemented?	253
Stakeholder Feedback.....	254
Questions Asked.....	254
Stakeholder Feedback 1: Sam.....	254
Stakeholder Feedback 2: Julia.....	255
Stakeholder Feedback 3: George.....	255
Limitations of the Solution.....	257
What are the limitations of the solution?	257
How could these limitation be addressed?	257
Maintenance of the solution	258

What has been done well?	258
What could be done in the future?.....	258
Further Development	259
Spin shots.....	259
Tutorial.....	259
9-ball and Other Game modes.....	260
Final Code	262

Analysis

Problem Identification

Introduction

After visiting many game rooms, public houses, and arcades, it is very clear that Billiards, Pool, and Snooker are very popular games in these spaces. One of the most common is pool which is played on a large table covered with a distinctive green cloth. Players use a cue to hit a white ball towards other balls, aiming to knock them in a pot. One player aims to pot striped balls and the other, coloured balls. The first to pot theirs and then the eight ball wins. Given the huge popularity of the game, they are always being used in areas and consequently, many people are left disappointed when they cannot play with their peers. It's simple: the demand for pool tables is exceedingly high, but there are few resources and physical space to implement them to meet the demand. This can be especially difficult as not only are the pool tables expensive, but they also require a great deal of work maintaining the fabrics and requires lots of physical space which can prove problematic in denser cities.

A solution to this problem is manufacturing a virtual way of participating in the game. Currently there is no reliable Windows or Linux software that runs a pool game, which provides an opportunity. It would give an accessible medium for new players to learn the game and experienced players to further enjoy the game. This could also have social benefits as for example, an elderly person or someone with a disability may not have the physical requirements to play. This game gives them the means of enjoying the game. Therefore, the aim of the project is to construct a game that runs a virtual game of pool with a computer or another player.

Why should a computational approach be used?

The project is amenable to a computational approach given that the user will be interacting and viewing with a screen. A computer's CPU or GPU should compute the necessary calculations and programs for the billiard balls' collisions as doing these calculations by hand would be very time consuming. For the software, the user will control a cue stick and select the power at which the billiard ball is given. These necessary calculations will be done by the computer and then displayed on the screen. Both primary and secondary storage will be required for the processing and storage of data for the game. Additionally, an object-oriented programming approach is likely to be used with balls, as each ball shares similar properties (Inheritance). Computational methods like Decomposition, 'Divide and Conquer' and Abstraction will all be used in the solution. The problem will be continually broken down into smaller problems until each subproblem is solved. Then the solved subproblems will be merged and eventually made into a whole solution. While this is done, abstraction will be used to simplify the problem to make the divide and conquer approach easier to implement.

Identifying Stakeholders

The main stakeholders for the software are those who are not able to play pool regularly due to lack of availability, resources, or a given disability. However, due to the nature of the project being computer gaming, the demographic can be very wide: anyone from a child to an adult would find this game fun and enjoyable. Additionally, the project could be used as an educational resource which teaches people about the game, adding to possible demographics. Therefore, the stakeholders are all people who play computer games and those who are interested in pool, regardless of if they are experts or novices or their age.

Therefore, I will collect a group of people who both do and do not have experience with pub games. A similar approach will be done with their experience of computer games. The interviews will also include those who are enthusiastic about pool yet are not able to play due to reasons stated above. This will ensure I collect data which is both relevant and holistic to everyone's opinions and problems.

I have organised interviews with four different people, each representing a certain demographic explained above. These are the questions I have prepared:

1. What is your experience with the game pool, online and/or in person.
2. Do you like playing computer games in general?
3. What do you like and dislike most about playing computer games?
4. If given a pool game, how would you use the software, and would it meet your needs?
5. Would you prefer an arcade-style physics engine or a more realistic one?
6. Would you like to see power-ups or other special abilities included in the game?
7. What input device is best for you: touchscreen, controller or mouse and keyboard?
8. When playing, would you rather play against a computer or another player?
9. Do you have any accessibility requirements that would need to be implemented?
10. Do you have any other thoughts or ideas to add?

Questions one to three are designed to determine users' experience with the theme and their familiarity with computer gaming, whilst also finding what makes a computer game enjoyable to play.

Question four asks how they will use the software, hoping to provide information on the individual's wants and needs, so that they can be met.

Questions five to eight ask more specific questions on the game and hope to get a more detailed list of the requirements, e.g. must, should and could have.

Question nine is more catered to older players or people who have a disability, aiming to give an idea of how their needs are to be met.

Interview One, Sam

My first interview is with a teenager, Sam. Sam is 17 years old and enjoys skateboarding, cricket, and casual gaming. Sam represents someone who enjoys playing pool but has not played very often due to lack of available pool tables.

1. What is your experience with the game pool, online and/or in person.

"I've played pool a few times at a local arcade, mostly just for fun with friends. Often, the pool table is being used so my friends and I don't really get to play much. I've tried a few different

pool games on my phone and computer, and I wouldn't say I'm an expert, but I know the basic rules and enjoy playing it. My dad loves playing it though."

2. Do you like playing computer games in general?

"Yeah, I love to play video games on my computer. I play a range of video games, from sports to first-person shooters. Whenever I have an hour to spare, I will play on my laptop. Since I frequently play online with my friends, it helps me to stay connected with them. I also love to compete with them! Overall, gaming is a big social activity for me."

3. What do you like and dislike most about playing computer games?

"I really enjoy how immersive the games can be, helped by their amazing artwork. I find that playing video games with friends is a great social experience, and I enjoy the competitive and strategic aspects of it. On the other hand, I hate it when games frequently lag and have bugs. Also, pay-to-win games and excessive advertising annoy me as well."

4. If given a pool game, how would you use the software, and would it meet your needs?

"I would probably use it for entertainment since I quite like gaming. I especially like playing with friends so it would be great if I could play with them either in person or online. The software would allow me to meet my needs given that I find it difficult to find a free pool table at the arcade. It would give me a chance to learn and practice the game further."

5. Would you prefer an arcade-style physics engine or a more realistic one?

"I think I'd prefer an arcade-style physics engine because it would make the game more fun and less serious. Realistic physics are great, but sometimes they can be too complicated and difficult to predict. An arcade-style engine would make it easier to pull off cool trick shots and have fun without worrying too much about friction and other complex stuff."

6. Would you like to see power-ups or other special abilities included in the game?

"For me, I think this would be amazing. Power-ups could make the game more exciting and add a fun twist to the usual gameplay. Stuff like extra shots or increasing ball speed for example. However, I understand if someone would not like this as these are not in the actual game."

7. What input device is best for you: touchscreen, controller or mouse and keyboard?

"I would prefer using a controller, given I play most of my games on controller. For me, it feels more comfortable and intuitive compared to a mouse and keyboard, or even a touchscreen. Now that I think of it, I guess it would not matter as keyboard and mouse would work quite well for precise movements of the cue."

8. When playing, would you rather play against a computer or another player?

“I’d rather play against another player. It’s more challenging and fun, plus it’s great to compete with friends and see who’s better. Playing against a computer can be good for practice, but in my opinion, it can get boring after a while because the AI might not always provide a good challenge. With real players, I think you would get more unpredictable and exciting matches. Therefore, online multiplayer modes are necessary for me.”

9. Do you have any accessibility requirements that would need to be implemented?

“No, I don’t. But it’s always good for games to include options for those who need them.”

10. Do you have any other thoughts or ideas to add?

“Maybe having different game modes or challenges could keep things interesting. For example, timed challenges, trick shot challenges, or even mini tournaments could add a lot of variety. It would also be cool to customise the look of the pool table or the cues.”

Summary and Analysis of Interview with Sam

Sam enjoys playing pool but has little experience playing it. He really enjoys playing video games and loves sports games. It is important to him that he plays the game with his friends, therefore playing against another player is a must have. Sam dislikes it when games are not well optimised and littered with advertisements, so lots of time should be spent on the user’s experience when playing the game. Sam will use the product for entertainment and for improving his skill in the game. Sam would like an arcade-style physics engine, as well as power ups and special abilities. Sam does not mind what control method is used. He also believes that other game modes should be added as well.

Interview Two, Julia

The next interview is with a female adult, Julia, who is 31, enjoys baking, music, and a huge fan of story-telling games. Julia represents those who enjoy gaming but has little experience with pool or pub games in general.

1. What is your experience with the game pool, online and/or in person.

“I must admit that I don’t really know anything about pool. While I have occasionally played games in person at social events like bars or friends’ houses, I never really learnt the rules. Although I’ve never played online pool, if it’s interesting and well-designed I might give it a shot – Ha!”

2. Do you like playing computer games in general?

“Yes, I love playing video games, especially ones that have compelling stories and great graphics. My favourite games are those that take me completely into a game’s universe and immerse me in a great narrative. I’ve recently been enjoying games like ‘Life is Strange’ and ‘The Witcher 3’.”

3. What do you like and dislike most about playing computer games?

“For most games, I love how they can transport you to different worlds and let you experience stories in an interactive way. The creativity and artistry involved in game design is something I really appreciate. What I dislike is when games become repetitive or when there is too much emphasis on grinding and less on the narrative.”

4. If given a pool game, how would you use the software, and would it meet your needs?

“I would use the pool game as a relaxing activity, something to unwind with after a long day. Since I’m not very experienced with pool, the game should have a gentle learning curve and should offer a tutorial. Otherwise, the game would meet my needs.”

5. Would you prefer an arcade-style physics engine or a more realistic one?

“I think I would prefer an arcade-style physics engine. It seems like it would be more forgiving and fun for someone like me who isn’t very experienced with pool. A more realistic engine might be too challenging and could potentially take away from the enjoyment.”

6. Would you like to see power-ups or other special abilities included in the game?

“To be honest, I believe that including special abilities themselves could make the game too overwhelming and harder to understand. I reckon including these powers would make the game too complex, making the game less enjoyable.”

7. What input device is best for you: touchscreen, controller or mouse and keyboard?

“I’m most comfortable using a controller for games like this. It feels more intuitive and ergonomic compared to a mouse and keyboard. For mobile play, a touchscreen would be fine, but for a more immersive experience, I would prefer a controller.”

8. When playing, would you rather play against a computer or another player?

“I’d prefer to play against the computer at first to get a feel for the game. Once I’m more confident, I think playing against other players could be fun and add a competitive element to the game. Having both options available would be ideal.”

9. Do you have any accessibility requirements that would need to be implemented?

“Nah, I don’t. But for many games, customisable controls and difficulty levels would be helpful additions to the game.”

10. Do you have any other thoughts or ideas to add?

“Thinking about the games I play; the games often have fantastic music. I would love to be able to listen to a great soundtrack while I play the game. It adds character and life to the game.

Summary and Analysis of Interview with Julia

Julia has little experience playing pool, but like Sam, enjoys playing when she does. Julia is also a fan of video games and finds immersive story games entertaining. She does not like it when games get repetitive. Julia would use the software for amusement and relaxation as well as learning the game. She desires a tutorial with a gentle learning curve. In her opinion, the physics should not be realistic but more fun. Unlike Sam, she believes power ups and special abilities would ruin the game. Julia would rather use a controller for playing the game. There should be an option to play against both a computer and a real player. Music is especially important to Julia and therefore she desires a good soundtrack when playing the game.

Interview Three, George

My third interview is with an older member of the community, George. He is 71 years old and loves to play pool regularly in the local public house. George is very experienced with pool yet has no experience with computer games.

1. What is your experience with the game pool, online and/or in person.

“Since I was a teenager, I have spent a lot of my life playing pool. In the past, I participated in leagues and tournaments, and I still love to play pool in the local community centre. Although I’ve played pool for a long time in person, I’ve never actually tried playing it on a computer or online.”

2. Do you like playing computer games in general?

“I haven’t played many computer games. I spend my time playing traditional games like pool or cards. The few computer games I have tried were either too complicated or didn’t hold my interest for long. However, I’m open to the idea if it’s something that captures my interest and isn’t too difficult to get the hang of. In fact, pool might be perfect for a computer game.”

3. What do you like and dislike most about playing computer games?

“From the limited experience I have, I like that computer games can offer new ways to enjoy traditional games, like playing pool without needing a physical table. However, I dislike when the controls are overly complex, or the game isn’t intuitive. I would prefer something straightforward that doesn’t require a steep learning curve.”

4. If given a pool game, how would you use the software, and would it meet your needs?

“I would use the pool game to enjoy a quick game when I can’t get to the pool hall. It could be a wonderful way to practice my skills and keep my mind sharp. If the game is realistic and replicates the feel of playing actual pool, I think it would meet my needs. Simple controls and clear instructions would be essential.”

5. Would you prefer an arcade-style physics engine or a more realistic one?

“Given my background, I would prefer a more realistic physics engine. The more accurately the game can simulate the actual behaviour of the balls and the table, the better. This means simulating ball spin! It’s important for me that the game feels as close to the real thing as possible.”

6. Would you like to see power-ups or other special abilities included in the game?

“Not really. I think power-ups and special abilities might take away from the authenticity of the game. I’m more interested in a straightforward pool game that sticks to the traditional rules and mechanics.”

7. What input device is best for you: touchscreen, controller or mouse and keyboard?

“I think a touchscreen might be easiest for me to use since it’s more intuitive and direct. However, I wouldn’t mind using a mouse if it provides better control for aiming and shooting. Controllers might be a bit too finicky for me.”

8. When playing, would you rather play against a computer or another player?

“I’d prefer playing against a computer initially, especially if it can simulate different skill levels to provide a good challenge. But I also like the idea of playing against other real players, especially if there’s an option to play against friends or family members online. Both would be great.”

9. Do you have any accessibility requirements that would need to be implemented?

“My vision is not as good as it used to be, and I now wear glasses. Having adjustable text and vibrant colours would be much appreciated if I am to play the game. Given that I am not remarkably familiar with computer games, the game should be extremely easy to use.”

10. Do you have any other thoughts or ideas to add?

“It would be great if the game included a variety of pool games, like 8-ball, 9-ball, and straight pool. Also, a practice mode where I can set up specific shots would be beneficial for practice.”

Summary and Analysis of Interview with George

George is very experienced with pool, but less so with computer games. It is important for him that the controls are easy to grasp, and the game is straightforward. George would use the software to practice his skills and asks that the game is as realistic as possible (meaning realistic physics with spin simulation). George is completely opposed to the idea of power ups and special abilities. He also thinks that touchscreen or mouse and keyboard are the best input devices for the game. He thinks that a player should be able to play against both a computer and real player. Given George’s eyes, the text should be adjustable and have bright colours that help him to navigate the UI and play the game. He also suggested a practice mode to be implemented.

Research of Existing Solutions

Game 1 – 8 Ball Pool

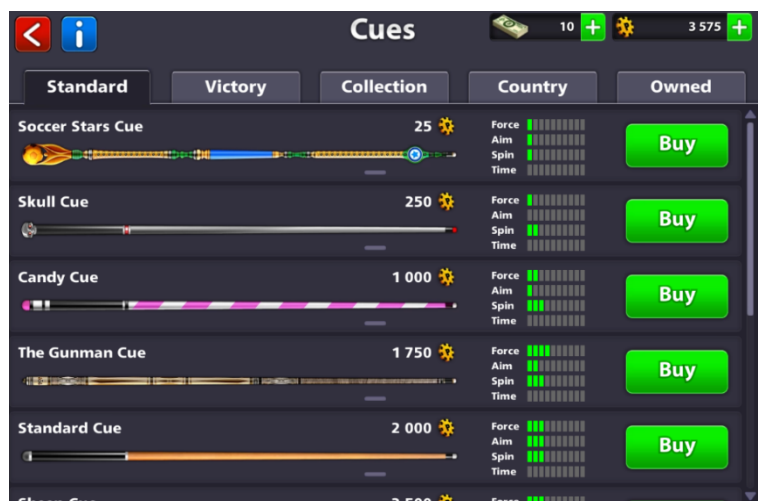
8 Ball pool is an immensely popular online pool game, some regarding it as one of the best in the genre of pub games. Made for IOS and Android by the Swiss Games Publisher 'Miniclip' in 2010. (Fandom, Inc, 2016). It offers many features and different game modes, appealing to a very wide demographic. These game modes include 8-ball, 9-ball, minigames and a practice arena. There is also a mode to play with friends and create tournaments. Although made for IOS and android, it can be played on any computer's browser that has flash enabled. (Fandom, Inc, 2016). 8 Ball pool is the most popular game in the genre given that it received a Guinness World Record for being the 'most downloaded pool mobile game' with 796,721,888 downloads. (Guinness World Records, 2021).

The image below shows the home screen for the game. The user interface for the game is particularly good all-round. Each game mode is easy to access and there many labelled buttons allowing for easy access to each section and feature of the software.



Some of the key features that differentiates the game from others is the inclusion of their own in game currency, Pool coins. These can be earned by winning matches, and then entering higher stake matches to earn more coins. These coins then allow you to purchase different skins for the pool cues and to choose different fabrics and colours for the billiard table.

The pool cues that you can buy have different skins and colours of course, but they also have different abilities and statistics of their own. The picture to the right demonstrates this. Despite the huge popularity of the game, one of the issues pointed out by the players and reviewers is the element of pay-to-win. You can buy coins in the game in exchange for real-world currency. Therefore, you would gain an advantage in the game if you were to spend money in the game.



The game also suffers from frequent advertising of their own, shown here. When you open the game, you are met with lots of advertising, promoting you to spend money to get coins. Although you do not need to pay, the frequency of this advertising gets annoying, shown by lots of the reviews on the Apple App Store. When researching, this particular advert (on the right) was shown after every game.



Despite this, the actual gameplay is extremely good. The game has a clear user interface with a fantastic user experience. The table is central, with relevant information scattered around it. After playing for a while, I found the available ball indicator at the top especially helpful to determine what ball to hit next. In addition, the trajectory line is helpful to aim the shots in the right direction. The line also changes length depending on the power given. In the top right, there is a brilliant display for where the cue hits the ball, allowing for spin shots to be simulated.



8 ball pool also has a feature which encourages players to log on daily. Every day, you have one chance to attempt to hit the golden ball and aim to stop it in the inner circle seen with the concentric rings to the right. This is an exceptionally innovative idea as it rewards the player for playing the game. This is definitely a consideration for the software I produce.



List of features that I should implement.

8 ball pool has a very clean and easy to use menu screen which should be a very important feature for the software that my stakeholders use. The physics of the game are a mix of arcade and realistic with the addition of ball spin. This should be met in the software. What many people do not like about the game is the frequency of the adverts. As a base, adverts will not be included in the software produced. Another reason this game thrives is due to the various minigames and practice shots you can set up. This increased the enjoyment for me and can be seen to be immensely popular among the players, shown by the reviews. This feature may be difficult to implement given the time constraints but is possible. The game should not have a pay-to-win aspect and players should be rewarded with their skill in the game instead. The game also includes good sound design, e.g. sound of balls colliding and being potted.

Game 2 – Chess.com

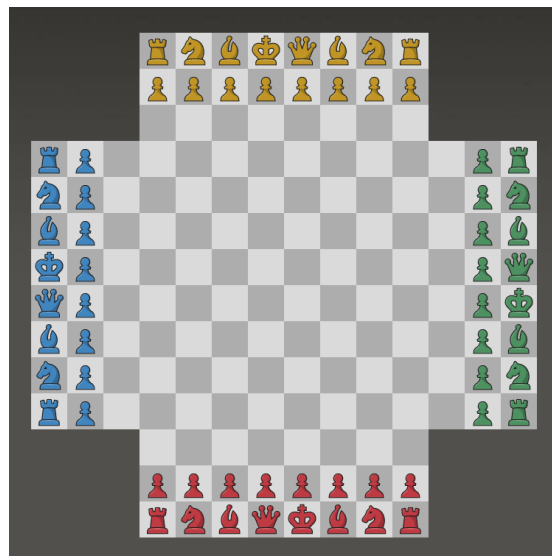
Chess.com is one of the most popular and largest online chess platforms in the world and as of April 2023, it has eleven million daily active players (New York Times, 2023). It is widely known for its fantastic features and game modes in chess as well as its very abundant, vibrant, and welcoming community. Below shows a picture of the main menu, with its many features to the left. Chess.com offers the opportunity to play chess as well as learning various techniques or movements through their lessons and videos. Its educational platforms also give you an opportunity to gain experience by playing skill games or connecting with others on their social platforms.



The UI of the game is unique, but very sleek and clean. This makes the game extremely easy to navigate. In addition to this, the HUD and UI are extremely customisable. There are over 50 different chess piece styles to choose from and even more different colours and patterns for the boards. This really allows players to have an impact on their game and helps the player enjoy the game further.

Like what I am trying to produce, Chess.com offers you to play against other players with its global community or its many bots as one is shown above. This attracts both those who want to play with others and those who want to practice against bots for an offline experience. You may also participate in different tournaments, fixing you with players of similar skill levels to compete for real-life winnings. For each game you play in chess.com, the system records your every move, allowing you replay your matches, with the application giving you recommendations on what to practice and the best moves in certain situations. Chess.com focuses greatly on education and enjoyability of the game.

Not only does Chess.com offer the ability to play chess, but it also offers more ways of playing the game through very niche game modes. An example would be four-player variants as shown to the left. In this mode, players take turns to progress up the board with the aim of checkmating all the other players. This could be done by targeting one player and hoping the other players follow. Or targeting all at once, cherry-picking the opportunities of reaching check. There are many tactics to this mode and therefore many can have lots of fun playing it. Additionally, the fact that it takes a well-known game, chess, and adds a slight twist makes the game remarkably interesting and I certainly found this very enjoyable to play.



Other interesting modes like Bullet exist where the rules are much like the normal chess, but the time for you to pick your move is considerably decreased, usually being about a minute in total. Other players have added to the chess variants on the Chess.com page and added other fun game modes like first to become check-mated wins and also ones that require playing on very oversized boards of 16x16.

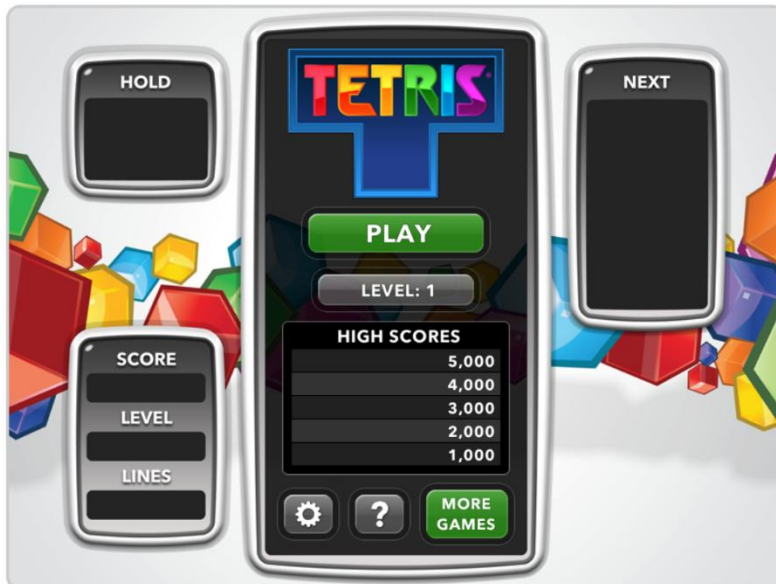
Chess.com has however had criticism with its technical implementation with many users reporting lots of lag and connectivity issues with the online play. People have also had issues with the subscription model, with many of the features mentioned above are only accessible to paid subscribers. Additionally, there have been many cheating concerns with the lack of anticheat in the games.

List of features that I should implement.

Although Chess.com has no thematic similarities to the game I am constructing, there are some vital elements that can be taken from Chess.com. The sleek UI and HUD elements are superb and should be replicated in my game. Chess.com's variety of game modes makes the application extremely enjoyable and packed with hours of fun, without getting bored. Chess.com is also a particularly good educational resource, with lots of tutorials, videos, and puzzles that the users can do to get better at the game. However, there are things that the users do not like including the fact that many of the features require a paid subscription.

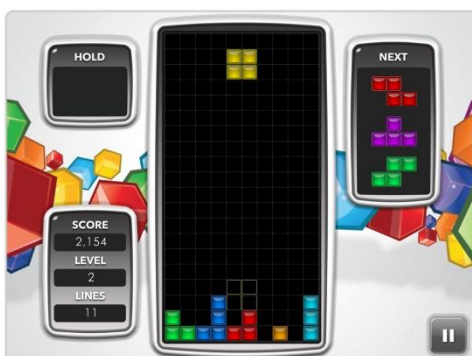
Game 3 – Tetris

Tetris is a classic video game that has survived as being one of the best arcade games of all time. The original game, designed by Russian designer Alexey Pajitnov in 1985, consists of the player attempting to manipulate falling blocks before they hit the ground, aiming to perfectly tessellate the region. When a complete row of blocks is filled, the row of blocks disappears, increasing the players points all while the speed of the blocks intensifies (Britannica, 2024). The most recent version of the game lies on the website, tetris.com.

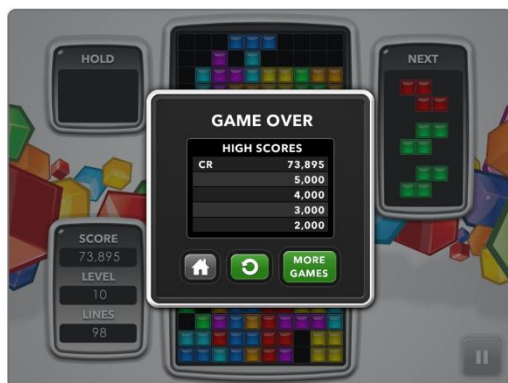


This updated version of Tetris has an interesting user interface. There are multiple bevelled boxes with lots of the boxes with no information, making the user think there are bugs to the game or slightly confusing the player. The main menu combines colours very well, there are vibrant and popping colours which increases the aesthetic of the game. With this, the game makes it very clear where you must click to start the game, with the use of a bright green, which is something I will seek to add to the final solution.

Despite being a basic game, the game offers high customisation. The player can easily change key bindings for the functions in the game. There are also toggles for animations which allows the game to be played on a range of different devices with varying computing powers. An improvement I would suggest would be to locate the volume control for the music to the main menu screen. When you open the game, the music is very loud and arguably obnoxious, therefore I would appreciate a relocation of the sound mixer.



Like any Tetris game, it is very addictive given its simple rules and easily graspable mechanics. Unlike previous versions, this newer version allows you to see the next three blocks that will fall, allowing the user to plan ahead where they shall place those blocks. In addition, the game gives a helpful indication of where the next block will land on the bottom of the screen shown in the image, with the addition of a 'ghost' block. These are vital quality of life features which should be included in the proposed solution.



List of features that I should implement.

Although the game is very simple in nature and mechanics, there are key points that can be taken from the game. My proposed solution should have lots of customisability with volume controls, ability to change key binds and adjustable graphics for different systems. The solution should also include features that are not required for the game to work yet improve the quality of life when playing the game, as it makes a large difference to people playing the game.

References for the Research of Existing Solutions

Britannica. (2024, August 12). *Tetris (Video Game)*. Retrieved from Britannica: <https://www.britannica.com/topic/Tetris>

Fandom, Inc. (2016, September 16). *8 Ball Pool Wiki*. (Dragonleaf5678, Producer) Retrieved June 30, 2024, from Fandom.com: https://8ballpool.fandom.com/wiki/8_Ball_Pool

Guinness World Records. (2021, July 4). *Most downloaded pool mobile game*. Retrieved June 30, 2024, from Guinness World Records: <https://www.guinnessworldrecords.com/world-records/662271-most-downloaded-8-ball-pool-mobile-game>

Richtel, M. (2023, April 24). *New York Times*. Retrieved July 1, 2024, from The Stealth Campaign That's Getting Your Kids Hooked on Chess: <https://www.nytimes.com/2023/04/24/science/chess-games-adolescents.html>

Proposed Solution

Essential Features

After having done research on existing solutions and consulting stakeholders, I now have key features that the solution should have. These features will determine whether the solution meets the needs of the stakeholders and the game. When designing the game, it is essential that the game is enjoyable and realistic.

The solution must have a main menu where the user can navigate to all parts of the game.

Upon opening the game, the user should be directed to a home page. This means the game offers a central hub where players can access various parts of the game. It also allows for a more intuitive way of adding game modes, should this be added in the future. The main menu will also include a settings page, which will allow players to adjust graphics, sound, and controls. The main menu also gives the ability of saving and loading progress and a leaderboard, should it be added at a later date.

The solution must have a game mode where the user can play a game of pool (8-ball) in 2D

This is a core feature of the solution; the game must be able to simulate a game of pool and satisfy the consumers. In addition, the game should be fun and enjoyable therefore encouraging players to continue playing. Other game modes like snooker or 9-ball are also very popular and should be considered to be added in a future update. The gameplay should also seek to be educational, meaning users should find the game easy to play and understand.

The solution must have realistic physics.

After conducting interviews about this question, it was clear that the stakeholders are interested in a physics engine that is both realistic and fun, pursuing both an authentic and enjoyable experience which players can take from the game to the real world. Given this, players should be able to use this software to practice their skills in an environment that closely resembles that of real life. Additionally, realistic physics provide added immersion to the game, helping players to have more fun when playing the game.

The solution must display relevant detail to the user when playing the game (suitable HUD).

For a greater level of immersion, the game should be able to display relevant detail to the user when playing the game. Examples of this information include a next player turn, which tells the user which player can go. Another is whether the player can pot coloured balls or striped balls, much like the interface of 8 ball pool, as researched before. There could also be a limit for how long a shot can take, which will be displayed here. An option to turn on real time information like ball velocities could be an option which would be helpful in debugging.

The solution must have sound effects and an appropriate background music/ambience.

As suggested by Julia in the interviews, music is a key part of any game. A soundtrack should be created that ties in nicely with the ambience of the game, increasing the level of immersion in the game. An upbeat soundtrack could increase the moral of the user, meaning the user is more likely to feel happier when playing the game. Sound effects should also be considered to increase the quality of the game. This could be done by recording the sound of a cue hitting a ball, balls colliding and balls being pot. Additional sound effects should be used when a foul is committed and when a player wins a game.

Limitations

When building the solution, I will inevitably uncover limitations that were unknown at the start. However, there are known limitations I have right now. The time restrictions may be difficult to combat as there are lots of features that I would personally like to add to the final solution. But after some research, it can take a team of people months or in some cases years to construct a game of a high quality like proposed here. Given that the implementation stage is considered to take the longest due to unforeseen bugs and maintenance, there may be physically not enough time to include could or should haves. The essential features will therefore be prioritised.

Another limitation is funding. Currently, there is no financial package or incentive to complete the project. This is being completed for A-Level coursework. If given more investment, more time, and money, the solution would be better. For example, extra software engineers could be hired or even artists that design textures for balls and the table. More funding would increase the quality of the game and ensure all the features would be added to the final solution.

Hardware and Software requirements

The software will be made and to be played on PC. Therefore, users will need a mouse, keyboard, or touchpad. The computer specifications should also meet the follow hardware requirements:

OS: Any Windows/Linux OS

Processor: Core i3 processor or Ryzen 3 processor

Memory: 1 GB RAM

Network: Internet connection not required.

Storage: 200 MB available space

GPU: Not required.

Software Dependencies: Python Interpreter, Pygame Library Installed Systemwide.

Success Criteria and Requirements

After considering the essential features and limitations along with the previous research, I have created a list of requirements that the solution should meet. All criteria will be cross-referenced and checked after the final solution is constructed.

Must have:

No.	Requirement	Justification
1	All elements of the game Pool are drawn to the screen.	The game should display all the relevant elements of 8 ball pool.
2	The player should be able to hit the balls with a cue.	This is needed to pot balls.
3	The game should be played from a singular camera angle from above.	This is the best camera for playing the game.
4	The game should have an intuitive user interface, where the user can navigate to all menus.	The user should be able to seamlessly navigate to play/settings/exit game.
5	The balls should be deleted after they are potted.	When the balls are potted, there is no further need to simulate them.
6	The system will implement visual aim assistance for the cue.	This helps users to aim the ball towards other balls.
7	The visual aim assistance will turn red/white depending on the prediction of what ball it will hit.	If the system thinks that the user is going to hit a ball which is not of their type, the system will tell the user that they are hitting the wrong ball.
8	When playing 8-ball, there should be 15 balls, each with different colours and markings. The game should follow the rules of 8 ball pool	The game should ensure each ball is unique and easy to recognise. Also follow the rules of the game
9	The player should be able to play the game against another player.	Two players should be able to play against each other.
10	After every game, the ball positions should be reset so a new game can be played after another.	There should be a starting function that starts a new game, and places all the balls in the correct positions.

11	When balls collide, the system should determine what direction and speed the balls have.	Appropriate calculations are required to determine the velocities of the balls after the collisions.
12	When balls collide with the walls, they are reflected off accurately.	When colliding with walls, the balls' velocities should be calculated accurately.
13	The game should display the winner after every game.	The game would follow the standard rules of the game and repetitively check if a user meets the win criteria.
14	The game should have a power meter to show how powerful each hit with the cue is.	This ensures that the player knows how powerful a shot is and can adjust the power to get the right type of shot.
15	The game should display which player is playing and who has the next turn.	There should be an indicator to show whose go it is.
16	The game should display which balls should be potted by each player.	The game should have a graphic that shows all the available balls to pot for each player.
17	The size of the game should be no more than 200MB	The file size should be small in general.
18	The game will have a music track to be played in the background.	Music helps players to enjoy the game, and a calming one would also help the players focus.
19	A relevant sound will be played if the player pots a ball	Repetitive sounds for certain game events are helpful to tell player if they committed a foul or won the game.
20	A relevant sound will be played when balls collide.	A ball-to-ball collision sound will be played when a collision is calculated. Enhances user experience.
21	When balls collide, the system should play a sound with a suitable volume, depending on the speed of collision.	If a ball collides with another ball at a high speed, then a loud sound should be played.
22	The game should play a click sound every time a menu is clicked.	This ensures user engagement and understanding of where the user is navigating to.
23	The game should run at 60fps or more for good performance.	The game should be able to run well when playing the game.

24	The game is visually aesthetic/appealing.	The game should be enjoyable to play, and the elements are well designed.
----	---	---

Should have:

No.	Requirement	Justification
25	The player should be able to play against an AI.	As well as another real player, the game should have a suitable algorithm that finds the best shot for a given state.
26	The game should include a practice mode	A practice mode would be helpful for players to increase their skills and attempt challenging shots.
27	The player should be able to choose between different colours and patterns of cues.	The player should be able to customise their experience and decide what colours and patterns to use.
28	The player should be able to choose between different colours and patterns of pool tables.	The player should be able to customise their experience and decide what colours and patterns to use.

Could have:

No.	Requirement	Justification
29	The game could be able to simulate 9-ball pool	The game could be able to incorporate another game mode.
30	The balls could rotate when moving (i.e. not static) as if in 3D.	When a ball is moving in real life, you see the ball rotating as it moves, this 3D effect should be added to the 2D game.
31	The game could be able to handle spin shots and simulate spinning balls.	When the cue hits the ball on the edge, it induces a spin on the ball, which should be simulated in the game.
32	The solution could have an adjustable font size.	This allows people who cannot see very well to understand and play the game.
33	The game could have a customisable UI (resizing and colouring)	This adds a more individualised touch to the game, as you determine where extra details and statistics are placed on the screen.

34	The game could have a leaderboard or tracker that tracks the player's win rates etc.	When a user plays, they either win or lose, this can be turned into a percentage and the best ones will be displayed on a leaderboard.
35	The game could include a replay feature.	If a user does a particularly challenging shot, they should be able to record it will in game tools or able to replay the shot.
36	The solution could simulate an animation for chalking the cue.	When playing pool in real life, you often chalk the cue. This feature would add immersion to the game.
37	The game could have player profiles, which save progress and scores.	This feature could allow players to stop playing and exit the game, without the fear of losing their stage in the game.

Won't have

No.	Requirement	Justification
38	The game will not allow you to play other players over the internet.	This feature will not be implemented as it would require lots of work developing servers for the game.
39	The game will not have modification support	Given this feature would require a large community and constant development, this feature is not feasible.
40	The game will not have a feedback system where the players would report bugs.	There are likely to be few bugs and also development of this system would not be essential for the game to function.
41	The game will not have a chat feature that lets you chat with players in game.	Given there are no online players, there is no use for this feature
42	The game will not have a spectator feature.	Given there are no online players, there is no use for this feature
43	The game will not function with other input controls	There will be no other user inputs for example a user will not be able to play the game with a gamepad.

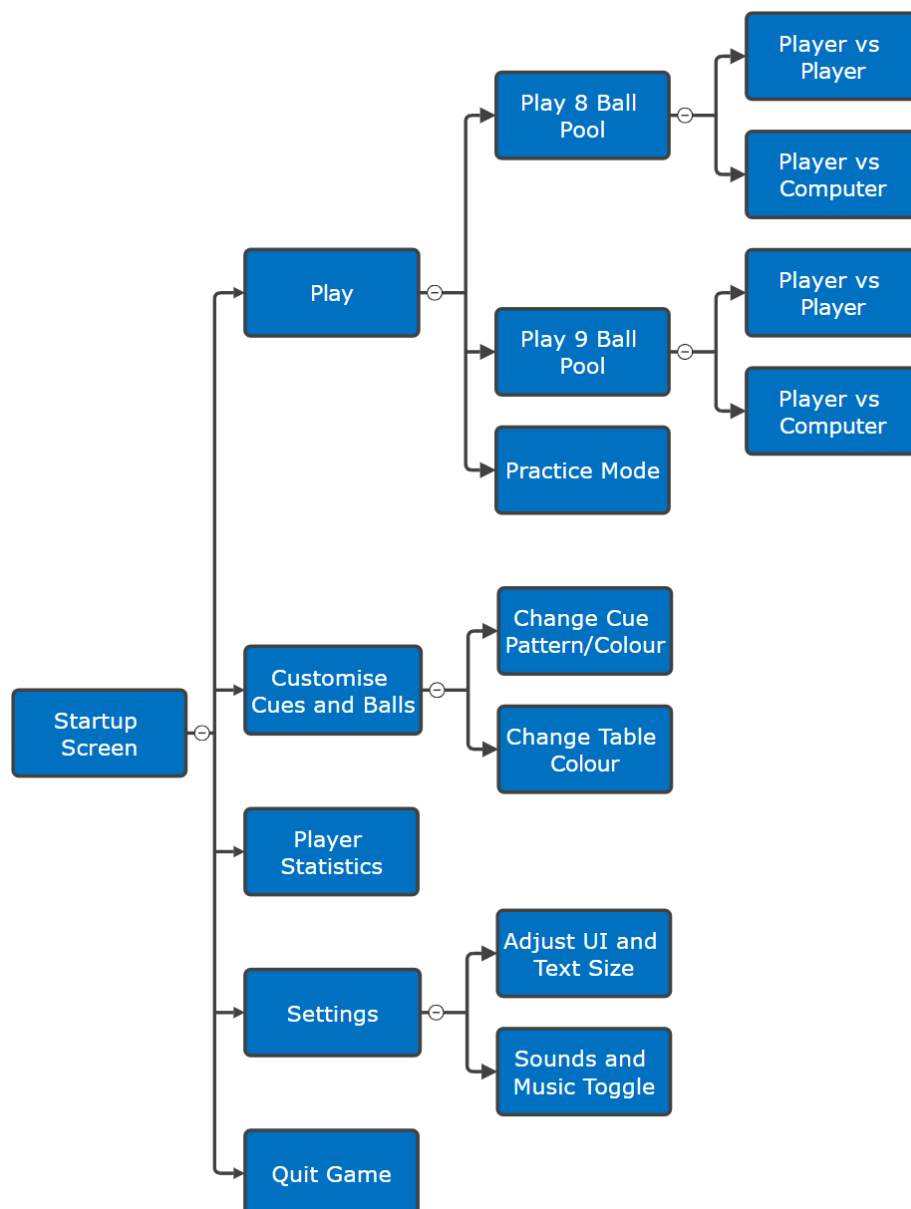
Design

Structure of the Solution

The solution can be broken down into two main sections: The menu structures and the game itself. For the game structure, both 8 ball pool and 9 ball pool have considerable similarities like the implementation of ball physics, ball collisions, potting balls, hitting the white ball with the cue. This means that the UI design algorithms for the game itself will be relatively similar, meaning that code can be reused.

Menu Structure

To start designing the Menus, I have engineered a structure diagram to help articulate the menu management in the solution. The diagram below shows all the menu screens to be added to the solution. After this, I have designed potential menu screens in designing software in subsequent pages.



Shown below will be the design of the main menu or startup screen. It allows the user to be able to play, see their stats, change the settings as well as quit the game. For every slide, the game name, "BILLIARD CLUB" is shown with an 8 ball underneath the text. For aesthetics, a blurred-out game of pool is being played in the background by the computer.



When the user presses the play button, the user will be directed to what be able to play 8 ball pool or 9 ball pool. The user will also have the option to practice in a sandbox-style environment. As with each menu screen, the 4th option will always be going back to the previous menu.



When the user clicks the Stats Screen, they will be directed to a screen showing the user's lifetime statistics. Here they will see information like how many balls they have potted, their win rate as well as the total play time in the game. This information will be stored on a text file for each user.



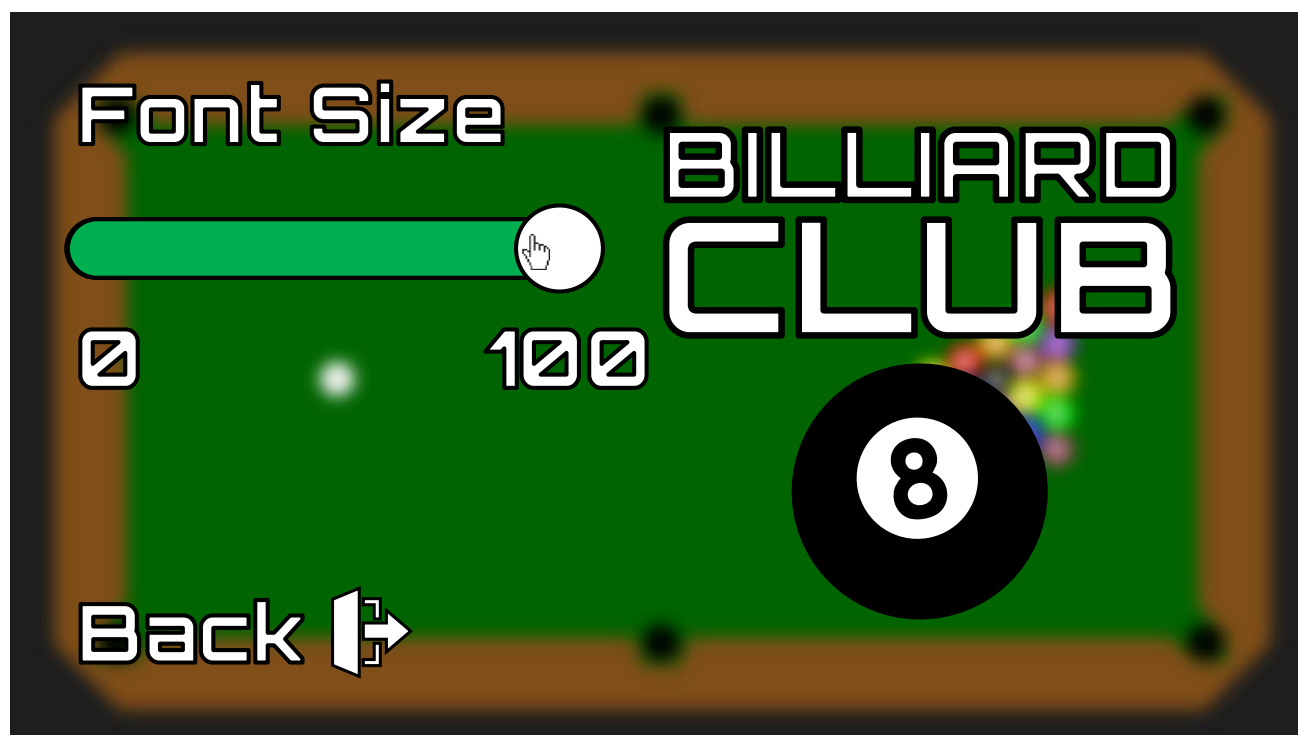
When the user clicks on settings, they will be transported to a subpage where they can change all the games' settings. This includes the table/cue options like colour, adjusting the font size, and to decrease the volume or turn music on or off.



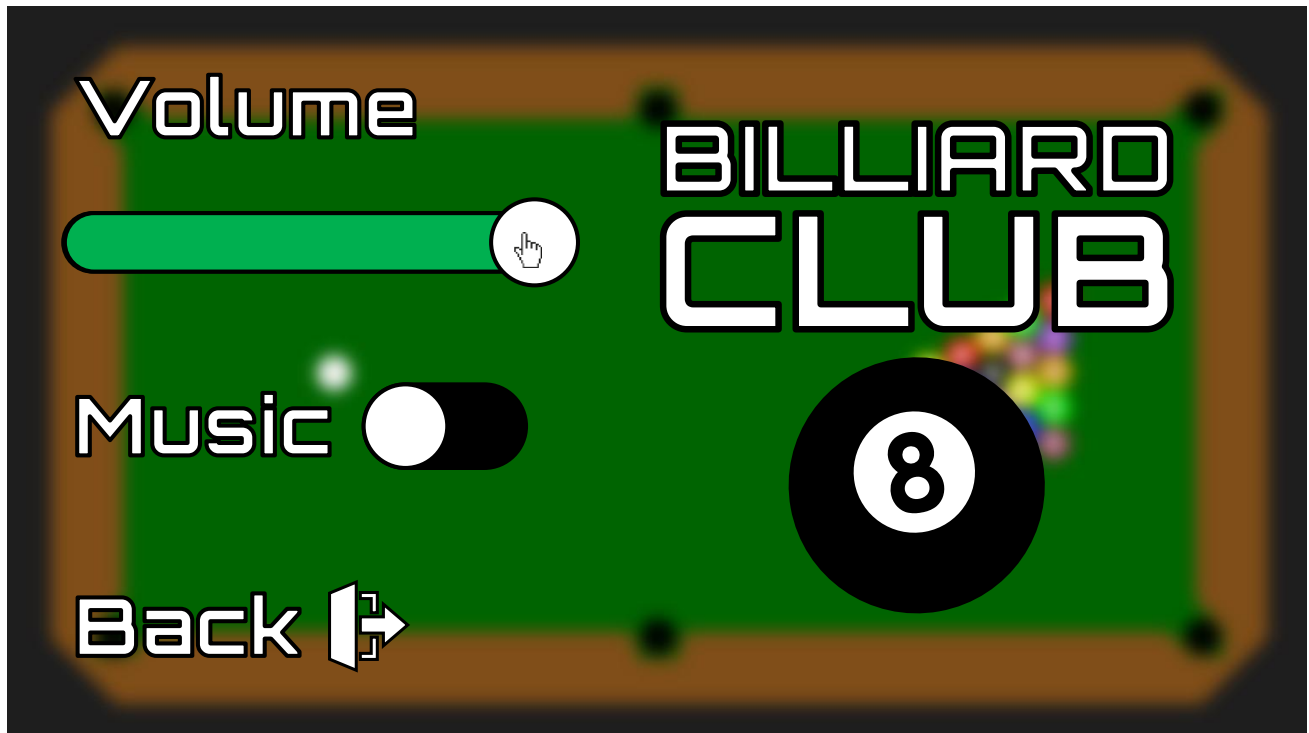
Below is the settings screen for altering table colours and the cue colours. The cue can be changed to any colour, while the table can only be changed to a few colours, which ensures that the game is still playable, avoiding colour clashes between the balls, background, cue, and text elements. The user will click on a colour whereas the cue colour can be changed by sliding the node.



Here is the 'Adjust UI' setting screen. Here the user will be able to change the size of all the text in the game. This will be done by the user sliding a node, resulting in a variable text size for all the elements. The Font size by default is set to the maximum to ensure no text clips into each other.



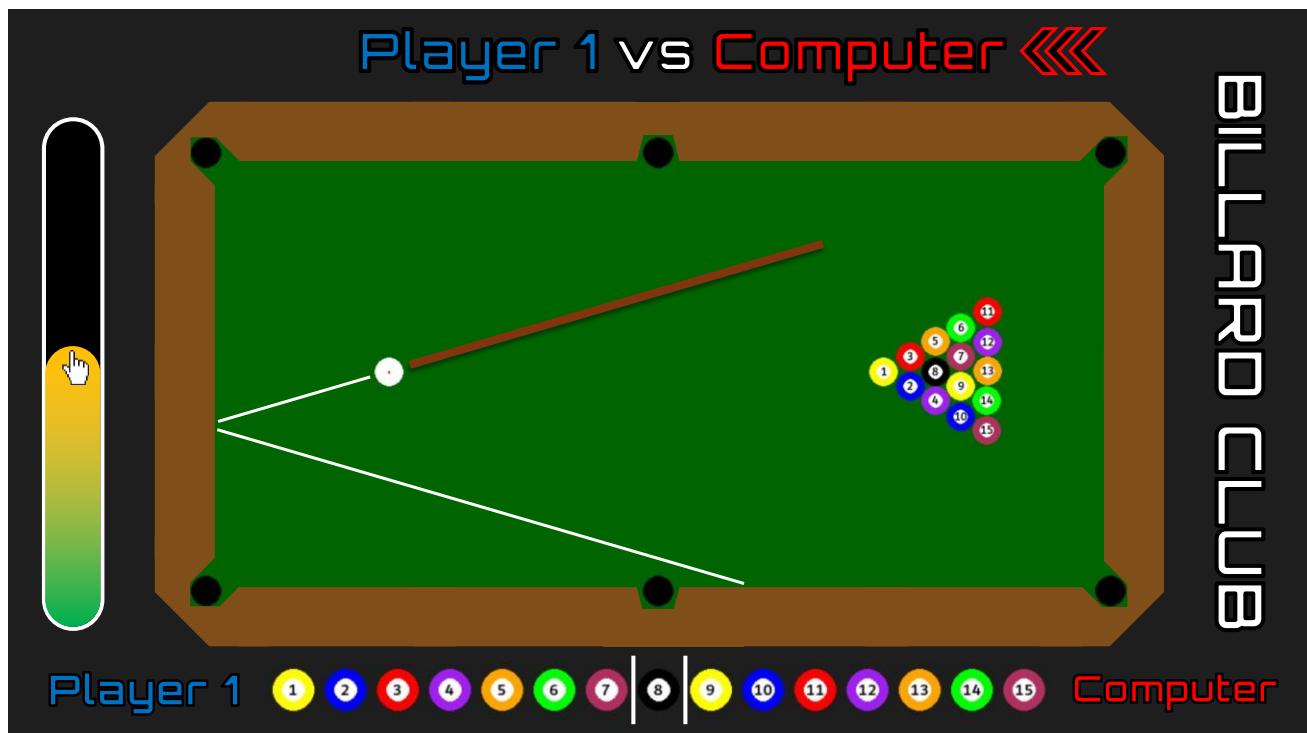
The sounds settings page is shown below. This allows the user to change the intensity of the volume in the game via a suitable slider. An appropriate music toggle has also been implemented. A sound effects button has not been implemented as they are integral for the playability of the solution.



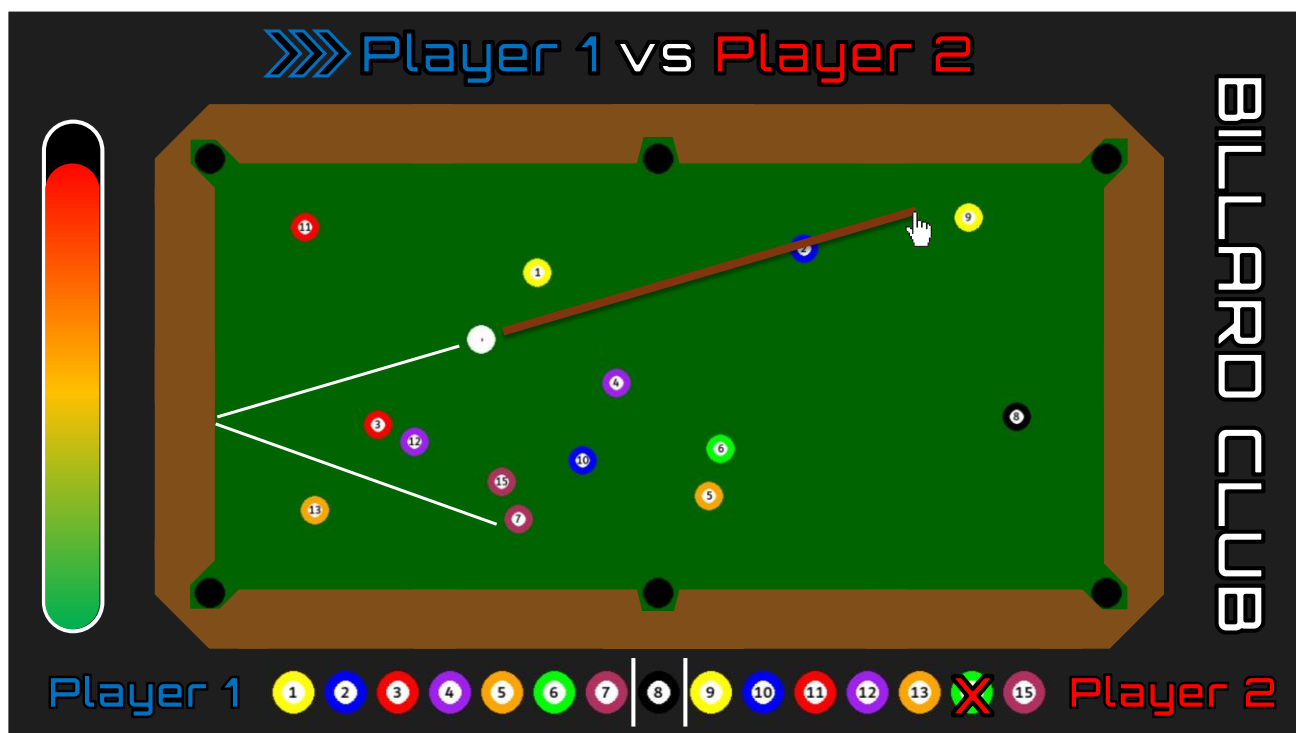
Game Structure

Before coding the solution, I have designed what the game screen should look like using design software. This ensures that I meet all the requirements and success criteria beforehand with the design, before dedicating time to coding the solution.

Below shows the start of a game of 8 ball pool. The player is playing against a computer, shown at the top of the screen. An arrow indicates whose turn it is to hit the white ball in the game. To the right, the game name is shown with no function. To the left of the screen is the power slider. Before the player performs a shot, the player can adjust the power of the shot. The slider has been made intentionally very large with bright colours to help convey the use and importance of the power of the shot in the game. In 8 ball pool, the bottom of the screen will display all the balls in play. As soon as a player pots one ball, the names either side of the balls will switch so that the left-side player is solids/dots, and the right-side player is stripes. This, again, helps to inform the user which balls they must pot to win the game. The 8 ball in the centre at the bottom has two white lines beside it, which again, reiterates the rules of 8 ball and shows that the 8 ball must be pot last to win the game.

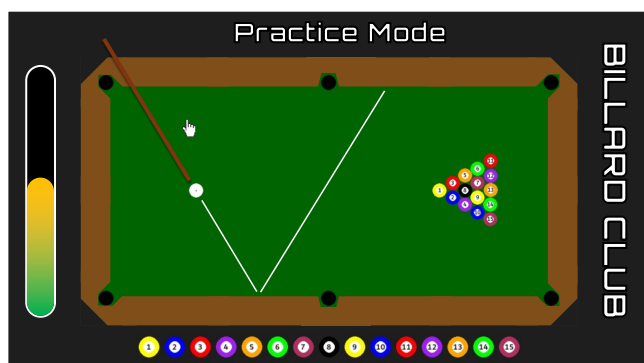
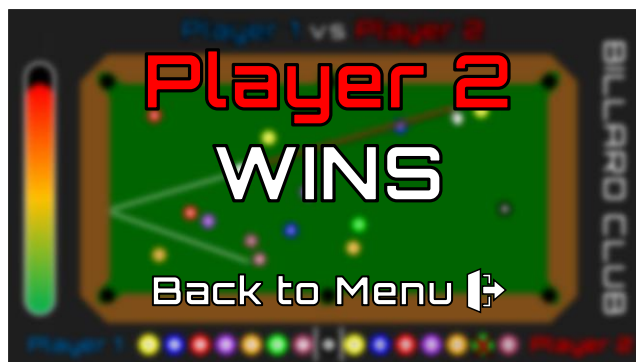


At the start of the game, the balls are set up as shown above. The cue hovers over the white ball and the coloured balls are arranged in a triangle on the right side of the table. When the user moves the cue, there will be a line that stems from the white ball, tracing the potential path of the ball. This helps the user perform more intermediate shots and trace the ball. The user will also be able to change the colour of the cue as well as the primary and accent colours of the billiard table. This gives the user an opportunity to improve the aesthetics of the game for their liking. The game will also include sounds and music. A soundtrack will play over each game to give ambiance. Sound effects will also be included like when a ball collides with the table edge or another ball. A win sound and foul sound will also be included. The player is notably playing against a computer. An algorithm will be in place to perform the necessary calculations to create a shot that mimics a player playing. In order to perform the shot, the play will have to drag the cue back and release the cue, this will hit the white ball at the power specified and at the direction of the cue.



The screen above shows a similar game of 8 ball pool, but the player is playing against another player locally. Here, they are one ball down. Player 2 at the moment is winning, shown by the X through ball number 14, conveying that the ball has been potted.

At the end of the game, a suitable win screen will be displayed. Shown below are designs for the event in which either player 1 or player 2 wins the game of pool.



To the left, there shows a potential design for a practice mode. Here the user is able to play a game of pool in a sandbox-style environment, with no rule or fouls. This allows the user to be able to practice shots and have fun. The text at the top of the screen has been replaced to "Practice Mode" and the player turn indicators have been removed from this mode.

Further Explanation and Justification of the Structure of the Solution

The solution has been split into two parts, the menu, and the game. This helps to break down the solution into manageable parts which will make coding the solution much easier.

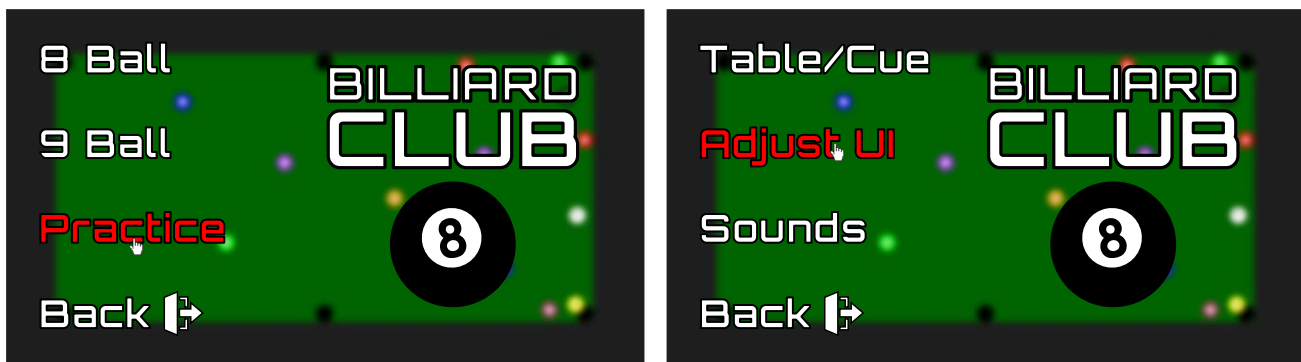
The menu structure has been broken down effectively into every section which will make development very simple as well as increasing the maintainability of the code. A modular approach to coding the menu structure will be used and allow lots of the code to be reused (such as the logo, text positions and background game). This modular approach will also allow for any further development or new features to be added in the future with ease, and each new feature will be unlikely to affect other areas of code, further increasing the maintainability of the code. Additionally, if there is a bug that must be fixed or changed, this way of implementation will ensure no further bugs are made. With the menu structure, various usability and quality of life features have been implemented, explained later in the usability features section.

The game structure has been decomposed into smaller sections or tasks to implement. A design for the game has been created, which include many features. This again suggests that a modular approach is required to code the solution up. For instance, a separate subroutine will manage each of the user interface elements (power slider, ball displayer, player turn indicator) so that they do not interfere with each other unwantedly. This modular approach, again, ensures that any feature added in the future can be done so easily and also that each bug encountered can be dealt with effectively with minimal impact to other elements in the coded solution.

A divide and conquer approach will be used with the whole solution. The game has been broken into the menu and game structures, each with their own child structures which will be combined to form a final solution.

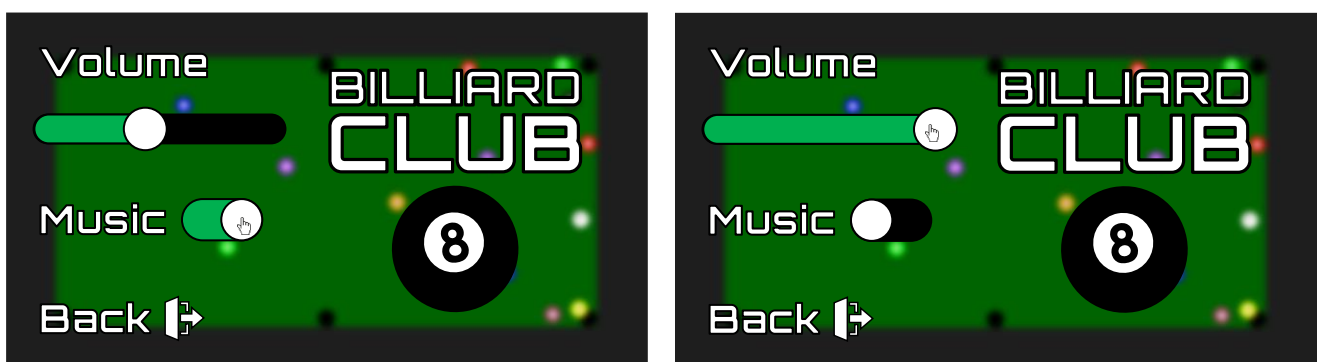
Usability Features

The usability features I will implement ensure that the program is accessible to all users as well as the menu being very easy to use and navigate. The UI and text elements will be resizable meaning the text will be large and easily legible giving the user the ability to cater their experience for their own needs if users have visual impairments etc. This has been done by using an appropriate bold and striking font with a large font size. The main window should also incorporate colour indicators to clearly indicate the where the mouse is or what the user is clicking on the screen. For example, on a menu screen, when the user's mouse hovers over a button, the button will dynamically respond, telling the user the button is able to be pressed. This has also been implemented in the designing of the menu screen, shown below.

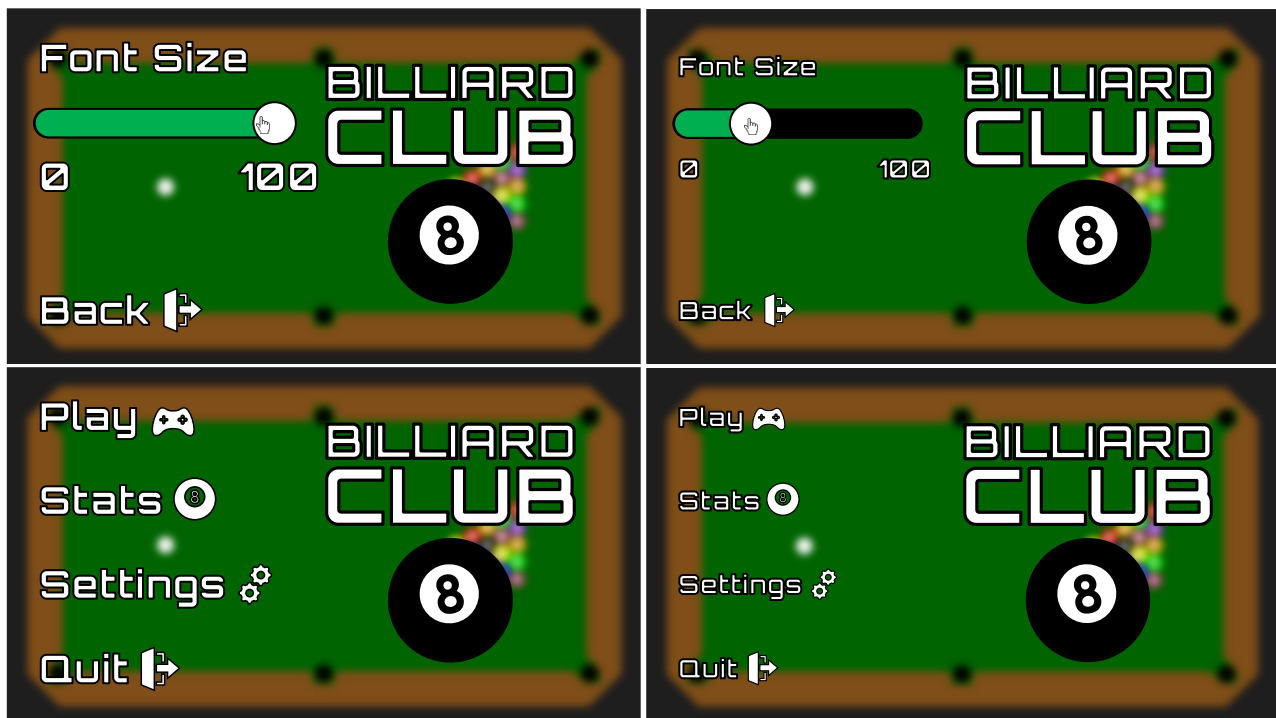


When the user hovers over a button on the menu screen, the button dynamically changes to notify the user that the button can be pressed. The design of the menu screen has also been engineered to ensure that the program can be navigated very easily. In each menu screen there are a maximum of four buttons to press, including a back button. This ensures the user does not become confused when navigating the menu and provides a familiar location for the user when they want to back out of the menu.

The game will also utilise sounds, which will need a way of toggling the music and changing the volume. This is needed to cater to those who have hearing problems. When researching the best medium of implementing a way of changing the volume, I found that most video games utilise a slider instead of inputting a percentage as it avoids the use of keyboard, making a slider feel quicker and more intuitive. The music should play a huge role in the game, immersing the player in the game, but I also acknowledge that some will find it annoying, this also means that not only should there be a volume slider but also a music toggle to be able to switch the music off. These features are shown below in the preliminary design of the sounds page.



Furthermore, to ensure the user can read the text and buttons, the UI could be scalable. All the font on the site would be set to the maximum from the start to account for this. However, if the user wishes to reduce the text on the site, the font size can be changed with a slider as shown below. This slider will act as a master font size switch, which will change the size of all the text in the solution.



Algorithms

Here, I explain the majority of the algorithms I intend to use in the development of the game. There is a description of the algorithm as well as pseudocode to illustrate how the problem will be solved.

1. Ball Update: Each ball will have attributes of position and velocity. Each game tick, every ball will add the velocity vector onto the position vector to get the new position vector.

```
1. for each ball
2.     ball.pos = ball.pos + ball.velocity
3. next ball
```

2. Friction: Each game tick, the velocity of the ball will be reduced, this reduction will be determined by a friction coefficient.

```
1. for each ball
2.     ball.velocity = ball.velocity * frictionCoefficient
3. next ball
```

3. Ball-to-Ball Collisions Detection: This algorithm will determine if two balls are colliding by comparing the distance between their centres to the sum of their radii.

```
1. for each pair of balls (ball1, ball2):
2.     if distance(ball1.pos, ball2.pos) < (ball1.radius + ball2.radius) then
3.         collisionDetected = true
4.     end if
5. next pair
```

4. Ball-to-Ball Collision Resolution: This will calculate the impulse from the collision. The algorithm will then change the velocities of each of the balls so that an accurate collision is calculated.

```
1. if collisionDetected then
2.     impulse = computeImpulse(ball1, ball2)
3.     ball1.velocity = ball1.velocity + impulse / ball1.mass
4.     ball2.velocity = ball2.velocity - impulse / ball2.mass
5. end if
```

5. Fixing sticking balls: When collisions are calculated, the balls may be clipping each other. This algorithm will incrementally move balls apart until they no longer clip each other.

```
1. while ball1 and ball2 overlap
2.     ball1.pos += normalVector
3.     ball2.pos -= normalVector
4. end while
```

6. Wall Collision Detection: This will check if the ball is colliding with the walls of the pool table.

```

1. for each ball
2.     if ball.pos.x - ball.radius < table.left or ball.pos.x + ball.radius > table.right
   then
3.         wallCollision = true
4.     end if
5.     if ball.pos.y - ball.radius < table.top or ball.pos.y + ball.radius > table.bottom
   then
6.         wallCollision = true
7.     end if
8. next ball

```

7. Wall Collision Resolution: This will reverse the corresponding velocity component (horizontal or vertical) when a ball collides with a wall.

```

1. if horizontal wall collision then
2.     ball.velocity.x = -ball.velocity.x
3. end if
4. if vertical wall collision then
5.     ball.velocity.y = -ball.velocity.y
6. end if

```

8. Hole Collision Detection: This algorithm will determine whether a ball is colliding with a hole and therefore should be deleted. Does this by computing the distance and then comparing the distance with the radius of the hole.

```

1. for each ball
2.     for each hole
3.         if distance(ball.pos, hole.center) < hole.radius then
4.             ball.isPotted = true
5.         end if
6.     next hole
7. next ball

```

9. Delete Ball When Potted: This will Remove a ball when it is detected in a hole.

```

1. for each ball
2.     if ball.isPotted then
3.         ballList.remove(ball)
4.     end if
5. next ball

```

10. Resetting the White Ball: When the white ball is potted, the program will put the white ball in its original starting position.

```

1. if whiteBall.isPotted then
2.     whiteBall.pos = startingWhiteBallPosition

```

```
3.     whiteBall.velocity = (0, 0)
4. end if
```

11. Cue Vector Calculation: This will determine the vector direction for the cue shot by subtracting the mouse position from the white ball's position and normalising the vector.

```
1. cueVector = normalise(whiteBall.pos - mousePosition)
```

12. Virtual Ball Projection: To assist the user when using the cue, the program will implement a virtual ball that will estimate the position of the ball after it is hit. Advancing a virtual white ball along the cue vector to predict collision points with rails or other balls.

```
1. virtualPos = whiteBall.pos
2. while not near collision and withinBounds(virtualPos)
3.     virtualPos = virtualPos + cueVector * stepSize
4. end while
```

13. Virtual Ball Reflected Projection: When the virtual ball collides with a rail, the program should reflect the virtual ball's velocity vector so that the ball forecasts an accurate rebound from the rail.

```
1. if ball collides with rail then
2.     reflectedVector = reflect(cueVector, railNormal)
3.     reflectedVector = normalise(reflectedVector)
4. end if
```

14. Drawing Cue Line and Projections: If there is a virtual ball projection that has been reflected off a wall, then this code will draw the relevant lines to provide a visual to the user.

```
1. drawLine(screen, whiteBall.pos, virtualBall.pos)
2. if reflectedVector exists then
3.     drawLine(screen, virtualBall.pos, virtualBall.pos + reflectedVector * length)
4. end if
```

15. Vertical Slider Update: This will check if the user is interacting with the power slider and if it is, the program will adjust the power slider's value based the mouse input.

```
1. if mousePressed within verticalSlider then
2.     slider.value = (mouseY - slider.top) / slider.height
3. end if
```

16. Horizontal Slider Update: This will check if the user is interacting with the power slider and if it is, the program will adjust the power slider's value based the mouse input.

```
1. if mousePressed within horizontalSlider then
2.     slider.value = (mouseX - slider.left) / slider.width
```

```
3. end if
```

17. Dynamic Sounds: This algorithm will be used for when two balls collide. The difference in magnitude of the velocities will be calculated and then the volume of the sound will be changed depending on this value.

```
1. impactSpeed = magnitude(ball1.velocity - ball2.velocity)
2. volume = scale(impactSpeed, minSpeed, maxSpeed)
3. playCollisionSound(volume)
```

18. Game Queue dequeuing: The main game loop will have a game queue which will need to be dequeued when the player has stopped playing. The program will also execute each function.

```
1. if all balls stopped then
2.     while gameEventQueue not empty
3.         execute(gameEventQueue.dequeue())
4.     end while
5. end if
```

19. Turn Switching Logic: Deciding whether to change turns based on which ball was potted and ball type assignments.

```
1. if pottedBall not in currentPlayerBallType then
2.     switchTurn()
3. end if
```

20. Ball type assignment: Once a player has potted a ball, this will set the player ball types (striped or dotted).

```
1. if firstNonWhiteBallPotted then
2.     player1BallType = determineType(pottedBall)
3.     player2BallType = oppositeOf(player1BallType)
4. end if
```

21. Determine winner algorithm: This will calculate who wins the game, based on the eight ball being potted after all of a player's balls are potted or not.

```
1. if eightBall.isPotted and currentPlayer has no remaining balls then
2.     winner = currentPlayer
3. end if
```

22. Setup ball function: This will be positioning balls at the start of the game using a set of predefined positions and patterns. These positions will be calculated by arranging the balls in a triangle formation. This will also be called when the game is reset.

```
1. function setupBalls()
2.     positions = computeTrianglePositions()
3.     for i in range(totalBalls)
4.         ball[i].pos = positions[i]
```

```
5.     next i
6. end function
```

23. Collision checks for walls: This will continuously check and resolve collisions with rails until the ball is free from overlap.

```
1. while ball collides with wall
2.     resolveWallCollision(ball)
3. end while
```

24. Table and Rail Drawing: This algorithm will be run continuously throughout the program as it will compute and render table boundaries, rails, and all the holes that will allow the user to play the game.

```
1. drawRectangle(screen, table.outerBounds, colour)
2. for rail in rails
3.     drawRail(screen, rail.position, rail.dimensions)
4. next rail
5. for hole in holes
6.     drawCircle(screen, hole.center, hole.radius, colour)
7. next hole
```

25. UI Handling: For every UI element, the program will detect if the mouse interacts with them (e.g. buttons, sliders, text boxes). Once interacted with, the program will callback the function to run on click/press.

```
1. for each uiElement
2.     if mouse over uiElement and clicked then
3.         uiElement.trigger_callback()
4.     end if
5.     if uiElement is a slider then
6.         uiElement.update_value(mousePosition)
7.     end if
8. next element
```

Variables

Here I have explained some of the potential variables and classes that will be used in the solution. This will help with coding the solution.

Key Variables in the Program:

Name	Data Type	Description
FPS	int	The target frames per second for the program to run.
resolution	list	The height and width of the program – dimensions of the display.
borderName	str	The game window title.
friction	float	This will be a value between 0 and 1 which will be multiplied by the velocity vector of the ball to slow the ball down.
restitutionCoeff	float	A value between 0 and 1 which will be used to slow reduce the Kinetic energy of the ball after every collision.
primaryTableColour	str	This will be a hex code that will store the primary colour of the table.
secondaryTableColour	str	This will be a hex code that will store the secondary colour of the table.
outerTableColour	str	This will be a hex code that will store the tertiary colour of the table.
tableDimensions	list	This list will store the dimensions of the pool table playing area.
centre	list	This will be calculated to be the centre point of the screen from the resolution list. This will make layout calculations much easier in the program.
holeRadius	int	A constant which will set the radii of the holes in the table.
ballRadius	int	A constant which will set the radii of the balls on the table.
tableCorners	list	Four corner coordinates of the table calculated from centre and dimensions.
holePos	dict	This dictionary maps each hole in the table to their respective coordinates on the table.
tableRails	list	This will be a 2D list that will store the values of each of the coordinates of the table rails.
ballInfo	tuple	This will also be a 2D list that will contain information for each ball: its colour, number, and whether it is striped or not.
ballStartPos	list	This dictionary will map the ball identifiers to their initial positions on the table.
computerPlaying	bool	This Boolean value will indicate if a computer opponent is active. This will activate computer mode.
player1Turn	bool	Indicates whether it is player 1's turn.

player1BallType	str	Will be either "striped" or "dotted" that player 1 must pot.
player2BallType	str	Will be either "striped" or "dotted" that player 2 must pot.
winner	str	Holds the winning player's name once the game is decided.
player1Name	str	The current player1's name.
player2Name	str	The current player2's name.
player1Colour	str	The current player1's colour that they have chosen
player2Colour	str	The current player2's colour that they have chosen

Key Classes:

Ball Class:

Attribute / Method	Data Type / Return Type	Description
Attributes:		
colour	str	The colour of this ball.
radius	float	Radius of the ball to draw.
pos	list	A 2D vector holding the ball's current position.
velocity	list	A 2D vector representing the ball's movement per frame.
mass	float	Mass value used in collision calculations.
number	int	The ball's number that should be displayed
isStriped	Boolean	Indicates if the ball is striped or not.
Methods:		
update()	None	This method will move the ball by applying its velocity vector every tick. It will also apply friction, and checks for wall collisions every game tick.
checkResolveCollision(other)	Boolean	This will determine whether this ball is colliding with another ball. If so, adjusts positions and velocities based on collision physics. This will also play a sound if there is a collision
checkResolveWallCollision()	Boolean	This checks collisions with the rails of the pool table and will adjust the position so that balls do not overlap with boundaries.
checkHoleCollision()	Boolean	This will check whether the ball has been potted.

draw(screen)	None	Draws the ball to the screen. This will also include the number and striped graphic.
delete()	None	Will remove the ball from the game.

Slider Class:

Attribute / Method	Data Type / Return Type	Description
Attributes:		
outerColour	str	Stores the outer colour for the slider's border.
sliderColour	str	Stores the current colour of the slider picker.
position	list	A 2D position vector that stores the position of the slider on the screen.
width, height	int	Dimensions of the slider.
value	float	A value between 0 and 1 representing the slider's current value.
Methods:		
draw(screen)	None	Renders the whole slider to the screen.
update(event)	None	Will update the slider's value based on if the mouse interacts with it.
returnValue()	float	Returns the slider's current value after adjustment.

Button Class:

Attribute / Method	Data Type / Return Type	Description
Attributes:		
text	str	The text displayed on the button.
font	str	What font should be used to render the button text.
callback	Callable function	The function that will be called when the button is interacted with
hovering	bool	Will be set to true when the mouse cursor is hovering over the button.
boxColour	str	The colour to render for the button background.
boxColourClicked	str	The colour to render for the button when the button has been clicked.
textColour	str	The colour of the text when the mouse is not hovering over the button.
textColourHovering	str	The colour of the text when the mouse is hovering over the button.

rect	list	This will define the clickable area of the button.
Methods:		
draw()	None	The method renders every element of the button: text and box. Changes based on current state hovering/clicked/neither.
checkHover(mousePos)	bool	This will return true if the mouse is within the button's rectangle.

Menu Class:

Attribute / Method	Data Type / Return Type	Description
Attributes:		
fontSize	int	The font size used for menu options.
pos	List	A 2D position vector that stores the starting position for the menu.
options	List	A list of text options for each menu item.
functions	List	All the callback functions corresponding to each menu option.
buttons	List of instances of Button	Collection of Button instances created for each menu option. (using Button class outlined above)
Methods:		
createButtons()	None	Instantiates each Button for each text option.
draw()	None	Will draw each button and then draw the menu.
update()	None	Iterates through each button and updates their hover states based on the current mouse position.
handleEvent(event)	None	This method will forward mouse events to each button, so they can determine if they are being clicked.

Test Plan

The idea for this section is to define an overall plan for testing the final product without needing to define specific test data in advance. This is because I am going to split development up into stages. This modular approach will allow me to test each section and module of the application individually which means that all components of the game function correctly when integrated together at the end of development. The test plan outlined is shown in the table below.

Throughout development, for each function and method created, the inputs and outputs will be validated using print statements and IDE functionalities such as variable watching and breakpoints. These will be tested to ensure they function as expected. White box testing and black box testing will also be used to assist debugging.

#	Item Tested	Purpose of Test	Method of Test	Expected Result	Link to Success Criteria
1	Game Start	Ensures game starts correctly	Open the executable file for the game	Game starts	1
2	Game Exit	Ensures the game starts	Press exit game in the menu	Game exits	4
3	Balls are given a velocity	Ensures the balls move on the table	Play the game, see if balls move.	Balls should move on table if playing	2
4	Friction applied to balls	Ensures balls eventually stop moving	Play game and see if balls eventually stop	Balls should eventually stop moving	2
5	Ball-to-Ball collisions	Ensures ball-to-ball collisions are recognised and resolved	Play game and hit one ball into another	Balls should accurately collide and reflect off each other	11
6	Ball-to-Wall Collisions	Ensures ball-to-wall collisions are recognised and resolved	Play game and hit one ball into a wall/rail	Balls should accurately reflect off walls and rails	12
7	Balls should remain within table	Ensures all the balls are interactable and game functions	Play game and after check balls after one round	All balls that are not potted, should still be on the table.	1
8	Balls should handle multiple collisions	Ensures game calculates physics correctly	Break in the game, watch all the collisions occurring	Ensure the break is accurate.	11
9	Potting White Ball	Ensures white ball is not deleted when potted	Pot the white ball	White ball should reappear after potted in the next round	5

10	Pot any ball	Ensures any ball (except white) is deleted.	Pot any ball except the white ball	Ball should be deleted	5
11	Setting player type	Ensures a player is given a ball type on potting a ball	Pot a ball when no player types have been decided	Player should have that type of ball	15
12	Potting 8 ball	Ensures the game gives a win to someone	Pot the 8 ball	Player should either win/lose depending on if they potted all other balls	5
13	Foul Detection	Ensures player turn switches if player fouls	Commit a foul (e.g. pot another player's ball)	Should switch turn	8
14	Ball Pocket Detection	Ensures balls are detected to be potted	Pot a ball	Ball should be deleted	5
15	Game End	When all balls have been potted, the game should end	Pot all the balls in the game	Game should give the win to the correct player.	8
16	Win screen display on win	Ensures a winner is determined at the end of each game	Pot the 8 ball	Game should display the correct winner in an appropriate message	13
17	Game initialisation	Ensures at the start of each game, all the balls are put in the correct place	Start a new game after another	Game should reset and a new game should start	10
18	Main Menu Start button	Ensures that when "start" is pressed, the game mode selection launches	Press "start" in the main menu	Directs user to game mode selection	4
19	Main menu Options button	Ensures that when options are pressed, the options screen launches	Press "options" in the main menu	Launches options screen	4
20	Game Mode Selection	Ensures that each game mode is launched when pressed	Launch each game mode.	The correct game mode is launched	4
21	Options Screen	Ensures each menu in the options screen is functional	Press each button in the options screen	Each button should link to the correct screen.	4
22	Cue is drawn when user's go.	Ensures no cue is drawn when simulating	Run game and use cue.	After confirmation of shot, the cue should disappear	2
23	Virtual Ball UI	When aiming, the virtual ball should be shown to the user	On player's turn verify the ball projection is drawn	Ball projection is drawn to screen	6

24	Virtual Ball Colour Indicator	When aiming to another ball, the colour should change depending on the user's ball type	Aim the cue to a ball of the player's type and another not of their type.	Should indicate which ball not to hit	7
25	Virtual ball projection on another ball	Ensures the user knows where the collision will project the ball to be hit.	Hover the cue over the ball	The projection should appear and be accurate	6
26	The cue colour should match with the player's colour	This ensures each player knows whose turn it is	Observe the cue colour	The cue colour is the same as whoever's turn it is.	24
27	The power slider's colour should match with the player's colour	This ensures each player knows whose turn it is	Observe the power slider's colour	The cue colour is the same as whoever's turn it is.	24
28	Smooth power slider	Ensures the slider is easy and smooth to use	Use the power slider and decide if it is smooth	Slider should be responsive and smooth to use	24
29	Fluid ball motion	Ensures ball movement is smooth and accurate	Observe balls when moving	Balls should have accurate and fluid motion	24
30	All table elements should be drawn to the screen	Ensures all functionality and data is displayed to the user	Observe table	All elements/balls have been drawn	1
31	Ball numbers should be rendered on each ball	Ensures that the user can tell which ball to aim for.	Observe the balls	All balls except for the white ball should have a number	1
32	Balls should be either striped or dotted	Ensures balls can be differentiated from each other for the user	Observe the balls	All balls should have a graphic which displays them as dotted or striped	1
33	Player turn indicator	Ensures the user knows which player's turn it is	Observe the indicator	The indicator should be clearly shown to be whoever's turn it is	1
34	FPS is consistent and at least 60	Ensures the game is smooth to play and has good performance	Observe the FPS display	FPS should fluctuate above 60	23

35	Pause Screen Activation	Ensures the user can pause the game at any point in the game	Press the ESC key while playing a game	Should direct the user to a pause screen	3
36	Pause Screen Exit	Ensures the user can replay the game after pause	Press the ESC key while paused	Should return the user back to the game	3
37	No game interaction while paused	Ensures the user can interfere with the game while paused	Press keys and attempt to break the game while paused	The game will only return when the ESC key is pressed	3
38	The game is blurred in the background while paused	Ensures the pause screen is aesthetic.	Observe the blur in the pause screen	Game is blurred in background	24
39	The game should be blurred in all menus	Ensures all menu screens are aesthetic.	Observe the blur in menus screen	Game is blurred in background	24
40	Each button in the menu should be clickable	Ensures each button has functionality	Click various buttons in menus	They should all have functionality	4
41	Each button should change colour when hovering	Ensures user knows that box is a button	Hover over button	Should change colour	4
42	Each button should change colour when clicked	Ensures user knows that button has been clicked	Click button	Should change colour	4
43	Each textbox should change colour when clicked	Ensures the user knows when to start typing	Click a text box	Changes colour	4
44	Each textbox should allow the user to type	Ensures text can be input into a text box	Click and start typing in a text box	Characters pressed show up in textbox	4
45	When delete is pressed in a textbox, the letter is deleted	Ensures that characters can be removed in the text box if the user mistypes	Press delete in the text box	Last character is deleted	4
46	Back button in menus	Ensures the user can go to the previous menu	Press back in menus	Game should revert to previous menu	4

47	When the user types a new name, it updates to the game	Ensures the user can change their name	Change name, observe it in a new game	New name is updated to game, observable in game	4
48	Colour picker	Ensures the user can pick a colour for their indicator	Use the colour picker to pick any colour	That colour should be updated, is observable in a game	4
49	Colour Slider is smooth and easy to use	Ensures the user can operate the slider well and is easy to use	Use the slider and determine if easy to use	Slider is easy to use	4
50	Master Volume Slider	Ensures user can change the volume for all elements in the game	Use slider, check before and after	Slider should correspond to how loud every sound is	4
51	Music Volume Slider	Ensures user can change the volume for only the music	Use slider, check before and after	Slider should correspond to how loud the background music is	4
52	Ball sound Volume Slider	Ensures user can change the volume every ball sound effect	Use slider, check before and after	Slider should correspond to how loud ball collisions and potting is	4
53	Sound Played on cue hitting white ball	Ensures a response to the user after they have hit the ball	Hit the white ball with a cue	Sound Is heard	20
54	Sound played on ball collisions	Ensures user knows when a ball collision occurs	Project one ball into another	Sound is heard	20
55	Sound volume should depend on speed of collision	Ensures the user is met with a dynamic volume for more immersion	Project one ball into another at a slow speed, then at a higher speed then compare	Sound at higher speed is louder than at a lower speed	21
56	Background music is heard everywhere on game start	Ensures music plays	Open game and check if music playing	Background music playing	18
57	Background music should replay after it ends.	Ensures the music plays indefinitely	Leave the game for 20 mins, and check if music still playing	Music continues to play	18
58	Computer calculates	Ensures the user can play against a computer	Select Player vs AI and start playing	When it is the computer's turn, they will calculate a shot	25

	next shot on turn				
59	The computer should calculate the shot instantly	Ensures the user is not waiting around for a long time	Give the computer the turn, time how long they take	Should be near instant	25
60	The user's cue should not be drawn when the computer is playing	Ensures there is no redundant information in the UI when playing	Give the computer the turn	The cue should not be drawn here	1
61	When balls are potted, the UI updates to show the user which have been potted	Ensures the user knows which ball are potted and which need to be potted	Pot a ball	Ball is shown to be potted in UI	4
62	A sound should be played when a ball is potted	Ensures the user knows when someone pots a ball	Pot a ball	Sound is heard	19

Development of Coded Solution

I have split up the development of the game into key stages to ensure that I take an iterative approach to coding the solution. At the end of each stage, I will review what has been added, how the code has been tested, and what success criteria have been met and which may have not.

Stage 1: Ball Physics and Collisions

Stage 2: Pool table Implementation

Stage 3: Cue Mechanics and Basic UI

Stage 4: Game Rules and Logic

Stage 5: AI Development

Stage 6: Menu Creation

Stage 7: Audio and Sounds

Stage 8: Final Testing

Stage 1: Ball Physics and Collisions

The solution will be constructed with Python, utilising the Pygame library extensively. Given the physics of the game is integral for a successful game, I will start developing a 2D environment where I can simulate elastic collisions. This will be in the form of a whole new and separate python file that will just allow me to develop the physics of the game first, with the ability of coping the code later to the final solution.

Here I have created a new python file that opens a window, with a running game loop.

```
import pygame

pygame.init()

# Variables
FPS = 60
resolution = (1000, 600) # Screen dimensions
borderName = "ballPhysics.py" # Displays at the top of the program

# Pygame Functions
pygame.display.set_caption(borderName)
screen = pygame.display.set_mode(resolution) # Sets up the display
clock = pygame.time.Clock()

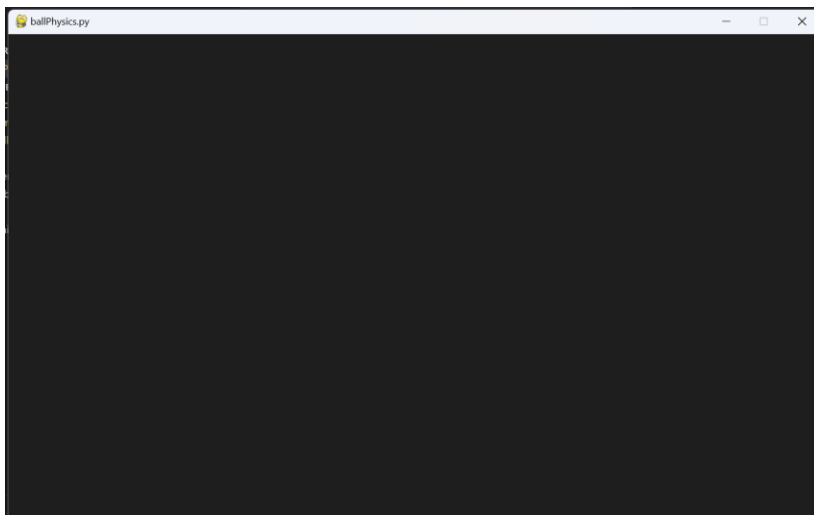
# Game Loop
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT: # Checks if user closes program
            pygame.quit()
            exit()

    screen.fill((30, 30, 30)) # Fills the screen with a grey colour

    pygame.display.flip() # Updates the display
    clock.tick(FPS) # Caps the frame rate
```

The code above shows variables and functions sorted and shows the game loop where future functionality will be added.

The code is shown working on the next image:



Next, I am going to use OOP to create a class named “ball” that will allow me to create a ball on the screen, given its position, radius etc.

```
class Ball(pygame.sprite.Sprite): # Defines a class for a ball
    def __init__(self, colour, radius, pos, direction, speed): # Constructor Method
        super().__init__()
        self.colour = colour
        self.radius = radius
        self.pos = VEC(pos) # Position Vector
        self.direction = VEC(direction).normalize() # Normalised direction Vector
        self.speed = speed # Initial speed of the ball

    def update(self): # Updates the position of the ball
        self.pos += self.direction * self.speed

    def draw(self, screen): # Draws the ball to the screen
        pygame.draw.circle(screen, self.colour, self.pos, self.radius)

ballGroup = pygame.sprite.Group() # Groups all balls for easier maintenance

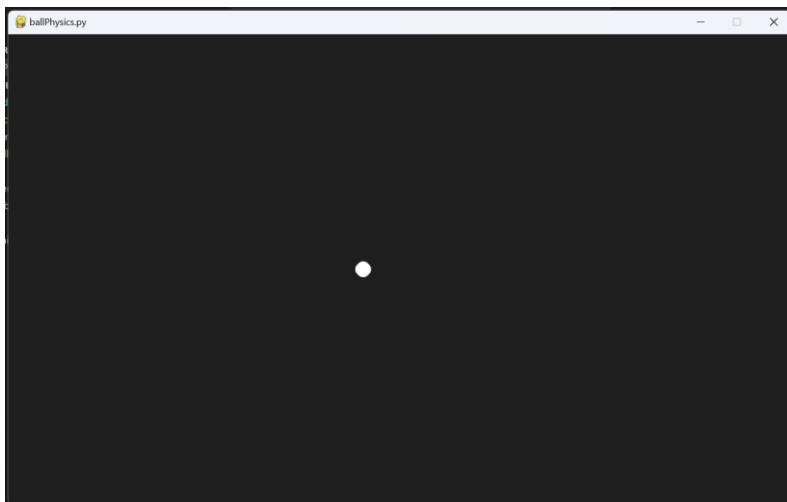
ball1 = Ball("white", 10, (300,300), (1,0), 3) # Instantiates the Class
ballGroup.add(ball1) # Adds the ball to the group
```

The function VEC() is shortened from “pygame.math.Vector2” for easier readability. This shortcut will be used extensively and therefore will save lots of time later in development.

In the game loop, each ball is updated and drawn:

```
ballGroup.update()
for ball in ballGroup.sprites():
    ball.draw(screen)
```

Running code produces this:



The ball is moving to the right, meaning the code is working.

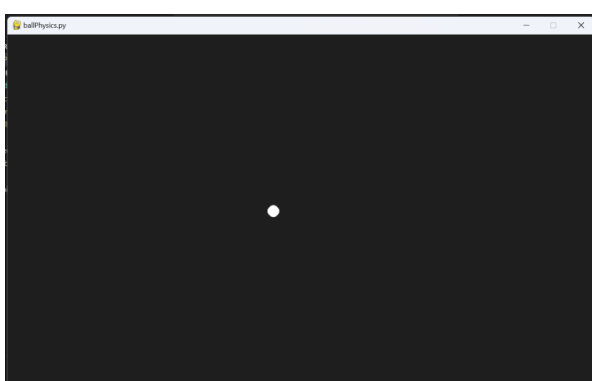
Now I will work on adding collisions with the walls:

In the ball class, I have changed the update() method in the ball class to add collisions.

```
def update(self): # Updates the position of the ball
    self.pos += self.direction * self.speed

    # Checking collisions with the walls:

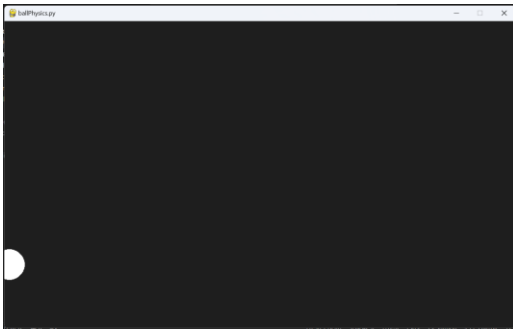
    # Left Wall
    if self.pos[0] - self.radius < 0:
        self.direction[0] *= -1 # Reverses X component
    # Right Wall
    if self.pos[0] + self.radius > resolution[0] :
        self.direction[0] *= -1 # Reverses X component
    # Top Wall
    if self.pos[1] - self.radius < 0:
        self.direction[1] *= -1 # Reverses Y component
    # Bottom Wall
    if self.pos[1] + self.radius > resolution[1]:
        self.direction[1] *= -1 # Reverses Y component
```



The code has been tested and the ball bounces off the walls.

However, there is a problem when the ball starts outside the area.

The problem is that the ball gets stuck on the side and constantly reverses direction, forever being stuck in the wall.



This is fixed by putting ball on the edge of the wall if the ball is outside the arena, then simulating as normal:

```
def update(self): # Updates the position of the ball
    self.pos += self.direction * self.speed

    # Checking collisions with the walls:

    # Left Wall
    if self.pos[0] - self.radius < 0:
        self.pos[0] = self.radius # Put ball on edge of wall
        self.direction[0] *= -1 # Reverses X component
    # Right Wall
    if self.pos[0] + self.radius > resolution[0]:
        self.pos[0] = resolution[0] - self.radius # Put ball on edge of wall
        self.direction[0] *= -1 # Reverses X component
    # Top Wall
    if self.pos[1] - self.radius < 0:
        self.pos[1] = self.radius # Put ball on edge of wall
        self.direction[1] *= -1 # Reverses Y component
    # Bottom Wall
    if self.pos[1] + self.radius > resolution[1]:
        self.pos[1] = resolution[1] - self.radius # Put ball on edge of wall
        self.direction[1] *= -1 # Reverses Y component
```

Up until now, I have been using two vectors to store and calculate the speed and direction of the ball.

Shown here:

```
class Ball(pygame.sprite.Sprite): # Defines a class for a ball
    def __init__(self, colour, radius, pos, direction, speed): # Constructor Method
        super().__init__()
        self.colour = colour
        self.radius = radius
        self.pos = VEC(pos) # Position Vector
        self.direction = VEC(direction).normalize() # Normalised direction Vector
```

I am now going to change the code so that the ball will only store one attribute, velocity vector. This has been done to keep the ball collisions as simple as possible and avoid further unnecessary calculations on the system.

Given the velocity vector is just the direction vector multiplied by the scalar value of speed, the change is minor and can easily be done.

New class is shown here:

```
class Ball(pygame.sprite.Sprite): # Defines a class for a ball
    def __init__(self, colour, radius, pos, velocity, mass): # Constructor Method
        super().__init__()
        self.colour = colour
        self.radius = radius
        self.pos = VEC(pos) # Position Vector
        self.velocity = VEC(velocity[0], velocity[1]) # Velocity vector
        self.mass = mass
```

The updated method is also shown here:

(the only change being the first line in the method)

```
def update(self): # Updates the position of the ball
    self.pos += self.velocity

    # Checking collisions with the walls:

    if self.pos[0] - self.radius <= 0: # Left Wall
        self.pos[0] = self.radius # Put ball on edge of wall
        self.velocity[0] *= -1 # Reverses X component

    if self.pos[0] + self.radius >= resolution[0]: # Right Wall
        self.pos[0] = resolution[0] - self.radius # Put ball on edge of wall
        self.velocity[0] *= -1 # Reverses X component

    if self.pos[1] - self.radius <= 0: # Top Wall
        self.pos[1] = self.radius # Put ball on edge of wall
        self.velocity[1] *= -1 # Reverses Y component

    if self.pos[1] + self.radius >= resolution[1]: # Bottom Wall
        self.pos[1] = resolution[1] - self.radius # Put ball on edge of wall
        self.velocity[1] *= -1 # Reverses Y component
```

Next is computing the ball-to-ball collisions.

Firstly, I instantiate lots of balls, shown here:

```
balls = [
    Ball("blue", radius=ballRadius, pos=(300,800), velocity=3*VEC(-1,5).normalize(), mass=5), # Instantiates the Class
    Ball("green", radius=ballRadius, pos=(100,300), velocity=3*VEC(6,5).normalize(), mass=5),
    Ball("yellow", radius=ballRadius, pos=(100,400), velocity=3*VEC(-1,5).normalize(), mass=5),
]
```

And add all the balls to a group:

```
for ball in balls:
    ballGroup.add(ball) # Adds the ball to the group
```

Next, the computer should loop through each ball in the group, which ensures that each pair of balls are only checked once.

Game Loop:

```
for ball1 in ballGroup:
    for ball2 in ballGroup:
        if ball1 != ball2:
            ball1.checkResolveCollision(ball2)
```

Ball Class:

```
def checkResolveCollision(self, other): # Check if balls are colliding, then resolves collision
    # Check if balls colliding
    posDifference = self.pos - other.pos # find difference in positions
    distance = math.hypot(posDifference[0], posDifference[1]) # distance between points

    if distance < self.radius + other.radius: # If colliding
        self.pos -= self.velocity # undoes last move, to prevent clipping

        # Calculate Velocities after collision
        n = VEC(posDifference).normalize() # creates a normalised vector as a origin for calculations
        relVel = self.velocity - other.velocity # calculates the relative velocity of the balls
        impulse = 2 * (n * relVel) / (self.mass + other.mass) # calculates the impulse each ball will get

        self.velocity -= impulse * other.mass * n # applies impulse to each ball
        other.velocity += impulse * self.mass * n
```

This algorithm aims to input all the information about two balls: ball positions, ball velocities, ball masses and calculate the velocities after the collisions.

The code then performs the following:

Checks if two objects are colliding (based on their distance and radii).

Resolves any overlap between the objects to prevent clipping.

Calculates the new velocities after the collision using conservation of momentum and impulse physics.

Updates the velocities of both objects accordingly.

Upon testing this algorithm, I encountered an error.

This algorithm seems to apply collision calculations twice, cancelling out the first collision.

Therefore, meaning the algorithm compares a pair of balls twice with this code:

```
for ball1 in ballGroup:
    for ball2 in ballGroup:
        if ball1 != ball2:
            ball1.checkResolveCollision(ball2)
```

e.g.: The algorithm would compare the following:

A vs B

B vs A (duplicate) (cancels out first collision)

C vs A

A vs C (duplicate) (cancels out first collision)

This is updated to:

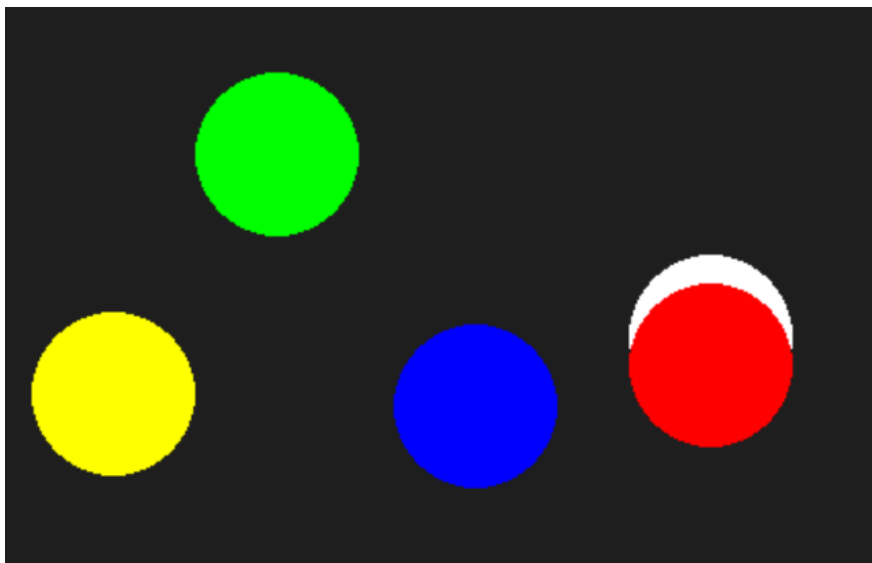
```
# Checks collision between every pair of balls
for i in range(len(ballGroup)):
    for j in range(i + 1, len(ballGroup)):
        ballGroup.sprites()[i].checkResolveCollision(ballGroup.sprites()[j])
```

Final view:

```
balls = [
    Ball("white", radius=40, pos=(500,490), velocity=3*VEC(-1,5).normalize(), mass=5), # Instantiates the Class
    Ball("red", radius=40, pos=(500,500), velocity=3*VEC(-1,5).normalize(), mass=5),
    Ball("blue", radius=40, pos=(300,800), velocity=3*VEC(-1,5).normalize(), mass=5),
    Ball("green", radius=40, pos=(100,300), velocity=3*VEC(-1,5).normalize(), mass=5),
    Ball("yellow", radius=40, pos=(100,400), velocity=3*VEC(-1,5).normalize(), mass=5),
]
```



More errors found: If you place a ball on top of another ball, this results in abnormal behaviour shown here:



The ball is constantly colliding and produces irregular results.

This can be solved by pushing out each ball to avoid clipping balls.

Done here in the checkResolve() method.

```
def checkResolveCollision(self, other): # Check if balls are colliding, then resolves collision
    # Check if balls colliding
    posDifference = self.pos - other.pos # find difference in positions
    distance = math.hypot(posDifference[0],posDifference[1]) # distance between points

    if distance < self.radius + other.radius: # If colliding
        self.pos -= self.velocity # undoes last move, to prevent clipping

        # Calculate Velocities after collision
        n = VEC(posDifference).normalize() # creates a normalised vector as a origin for calculations
        relVel = self.velocity - other.velocity # calculates the relative velocity of the balls
        impulse = 2 * (n * relVel) / (self.mass + other.mass) # calculates the impulse each ball will get

        self.velocity -= impulse * other.mass * n # applies impulse to each ball
        other.velocity += impulse * self.mass * n

        # Push out the colliding balls
        while distance < (self.radius + other.radius):
            self.pos += n # move balls away from each other
            other.pos -= n
            posDifference = self.pos - other.pos # find difference in positions
            distance = math.hypot(posDifference[0],posDifference[1]) # distance between points
```

The last while loop here repeatedly separates the two balls from each other, by adjusting their positions until they are not colliding.

Another error occurs if the distance between the balls is zero. This means that the normalised vector can never be 1. This results in a division of zero error:

ValueError: Can't normalize Vector of length Zero

This is fixed by adding a statement to ensure that the function quits if two balls are in the same position:

```
if distance == 0: # prevents normalisation error
    return
```

Further Optimisation:

Computers are very slow at square-rooting values. This happens when the computer calculates the hypotenuse or the distance between two points. This can be changed by squaring the values, meaning the computer does not have to calculate the exact Euclidean distance. Shown here:


```
def checkResolveCollision(self, other): # Check if balls are colliding, then resolves collision
    # Check if balls colliding
    posDifference = self.pos - other.pos # find difference in positions
    distanceSquared = self.pos.distance_squared_to(other.pos) # distance squared between points

    if distanceSquared == 0: # prevents normalisation error
        return

    if distanceSquared < (self.radius + other.radius)**2: # If colliding
        self.pos -= self.velocity # undoes last move, to prevent clipping

        # Calculate Velocities after collision
        n = VEC(posDifference).normalize() # creates a normalised vector as a origin for calculations
        relVel = self.velocity - other.velocity # calculates the relative velocity of the balls
        impulse = 2 * (n * relVel) / (self.mass + other.mass) # calculates the impulse each ball will get

        self.velocity -= impulse * other.mass * n # applies impulse to each ball
        other.velocity += impulse * self.mass * n

        # Push out the colliding balls
        while distanceSquared < (self.radius + other.radius)**2:
            self.pos += n # move balls away from each other
            other.pos -= n
            distanceSquared = self.pos.distance_squared_to(other.pos) # distance squared between points
```

Here I am using distanceSquared instead of exact distance. This saves resources on the computer and will improve performance for when I have lots of balls in the ball group.

In order to smooth out the collisions, the pushing out of the balls should be changed from a while loop to an exact distance. Previous code:

```
# Push out the colliding balls
while distanceSquared < (self.radius + other.radius)**2:
    self.pos += n # move balls away from each other
    other.pos -= n
    distanceSquared = self.pos.distance_squared_to(other.pos) # distance squared between points
```

This just repeats until they do not collide, but a better way is to find an exact distance for the balls to be displaced.

Upon looking for a solution, an exact value would require an exact distance to be calculated (requiring a square root), which would slow the program further.

Stage 1 Review

What has been done?

- A Ball class has been created to represent each ball that will be on the table.
- The ball class includes functions that simulate movement in a 2D environment.
- Collisions with balls and walls has been coded and now functions correctly.
- Collisions with balls and other balls has been coded and now functions correctly.
- Improved performance by implementing small changes that reduce computation for calculations with the balls.
- Successfully created an environment that simulates 2D elastic collisions between balls with mass.

How is code being tested?

- For each new element/function (e.g., ball movement, wall collisions, ball-to-ball collisions), I add a small bit of functionality and then observe the result, documenting and debugging it accordingly. This incremental approach ensures I follow the computational method of decomposition which ensures for swift development of the solution.
- Bugs are identified and resolved by modifying code of the program, rerunning the simulation, and documenting errors.
- Comments are used extensively throughout the solution. This helps debugging enormously as I sift through hundreds of lines of code.

Errors Fixed:

Error	Fix
If the ball starts outside the game area, it continuously reverses direction and therefore gets stuck.	I adjusted the ball's position to be exactly on the edge of the arena before calculating vectors after collisions.
The algorithm checks collisions twice per pair (e.g., A vs B and B vs A), which cancels out the effect of the first collision.	I modified the loop so that each pair of balls is only checked once.
If a ball starts on top of another ball, collisions are continuously calculated, which causes erratic motion from the balls.	I implemented a method that pushes balls apart until they no longer overlap, to prevent collisions being calculated when clipping.
If two balls occupy the exact same coordinates, normalising their distance vector results in a dividing by zero error.	I added an exception to exit the function if two balls have the same position.

Success criteria that has been met:

11	When balls collide, the system should determine what direction and speed the balls have.
12	When balls collide with the walls, they are reflected off accurately.

Summary of Stage 1

This stage has been successful given it has established foundational mechanics for the physics simulation and will allow for further, more complex interactions such as the collisions between irregularly oriented walls on a pool table.

Stage 2: Pool table and Basic UI

Given the ball physics and collisions have been coded up, I can start with creating the foundations of the game itself. I have created a new python file and declared key variables here:

```
import pygame

pygame.init()

# Variables
FPS = 60
resolution = (1280, 720) # Screen dimensions
borderName = "Billiard Club" # Displays at the top of the program

# Pygame Functions
pygame.display.set_caption(borderName)
screen = pygame.display.set_mode(resolution, pygame.FULLSCREEN) # Sets up the display
clock = pygame.time.Clock()

# Game Loop
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT: # Checks if user closes program
            pygame.quit()
            exit()

    screen.fill((30, 30, 30)) # Fills the screen with a grey colour

    pygame.display.flip() # Updates the display
    clock.tick(FPS) # Caps the frame rate
```

I have adjusted the screen so that that it fits the whole screen, 1280x720 and so that the game plays in Fullscreen mode shown in pygame functions comment

First thing to do is to draw the table:

I want to centre the table on the screen, so I have created a tuple for where the centre of the screen is. I have also defined where each corner of the table is. This is shown below:

```
# Table Variables and Setup
tableDimensions = (1024, 512)

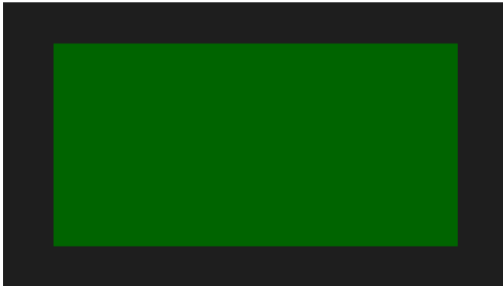
centre = (resolution[0]/2, resolution[1]/2)

tableCorners = (
    (centre[0] - tableDimensions[0] / 2, centre[1] - tableDimensions[1] / 2), # Top Left
    (centre[0] + tableDimensions[0] / 2, centre[1] - tableDimensions[1] / 2), # Top Right
    (centre[0] - tableDimensions[0] / 2, centre[1] + tableDimensions[1] / 2), # Bottom Left
    (centre[0] + tableDimensions[0] / 2, centre[1] + tableDimensions[1] / 2), # Bottom Right
)
```

I then drew this to the screen with the following code in the game loop:

```
# Draw Table
pygame.draw.rect(
    screen,
    "darkgreen", # Colour
    pygame.Rect(
        tableCorners[0][0], # Top Left x
        tableCorners[0][1], # Top Left y
        tableDimensions[0], # Width
        tableDimensions[1] # Height
    )
)
```

The screen now displays this:



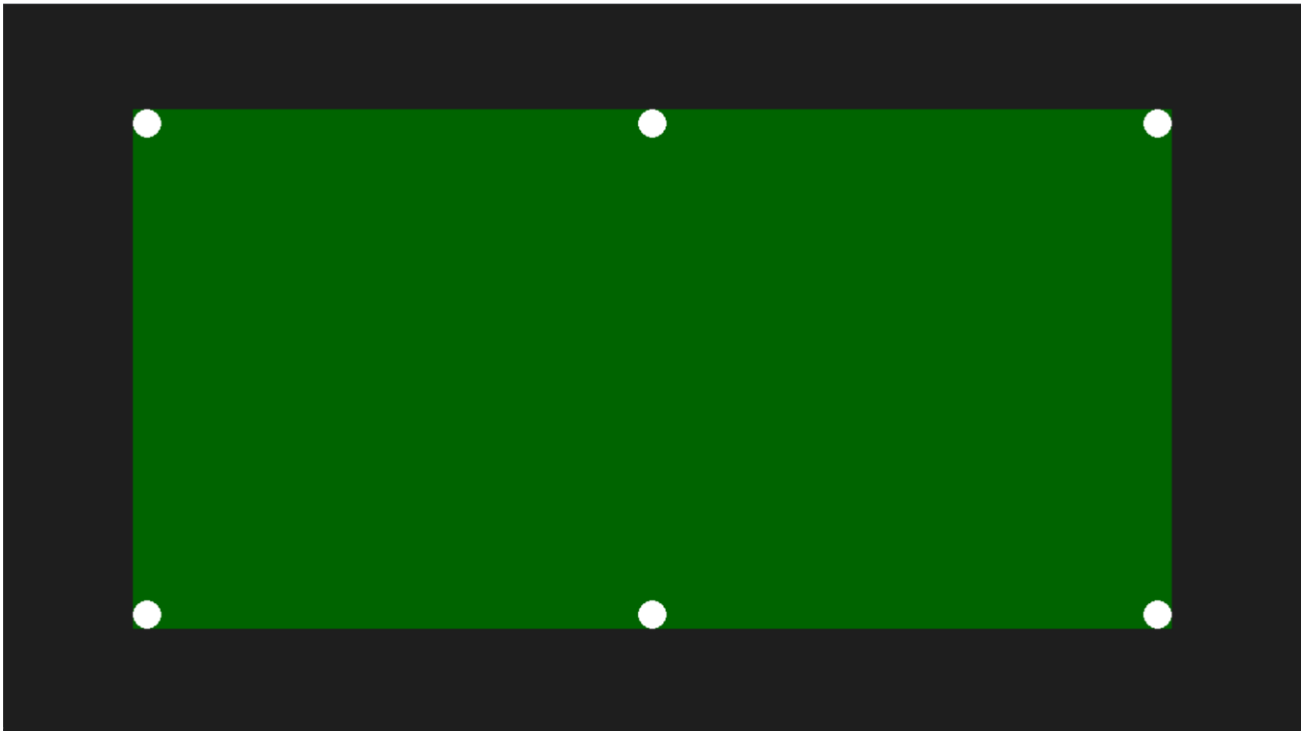
Next, I defined the positions of the holes:

```
holePos = {
    "TL": (tableCorners[0][0]+holeRadius, tableCorners[0][1]+holeRadius), # Top Left
    "TR": (tableCorners[1][0]-holeRadius, tableCorners[0][1]+holeRadius), # Top Right
    "BL": (tableCorners[0][0]+holeRadius, tableCorners[2][1]-holeRadius), # Bottom Left
    "BR": (tableCorners[1][0]-holeRadius, tableCorners[2][1]-holeRadius), # Bottom Right
    "TM": (centre[0], tableCorners[0][1]+holeRadius), # Top Middle
    "BM": (centre[0], tableCorners[2][1]-holeRadius), # Bottom Middle
}
```

And drew the holes to the screen with this code:

```
# Draw Holes
for i in holePos:
    pygame.draw.circle(screen, "white", (holePos[i][0],holePos[i][1]), holeRadius)
```

Screen: (dots are white to highlight them)



Next, I will be adding an outer ring and further aesthetics to the table. I will start by defining a few variables for the colours:

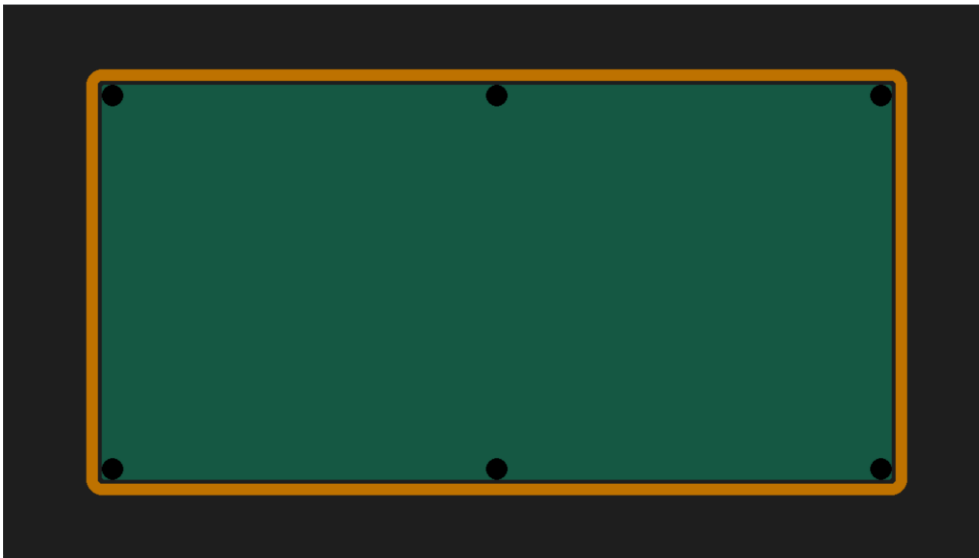
```
# Table Variables and Setup
primaryTableColour = "#1B6875"
accentTableColour = "#36454F"
outerTableColour = "#36454F"

tableDimensions = (1024,512)
centre = (resolution[0]/2, resolution[1]/2)
holeRadius = 14
```

I will add an outer ring to the table and update the table to have rounded edges

```
# Outer Ring
pygame.draw.rect(
    screen,
    outerTableColour,
    pygame.Rect(
        tableCorners[0][0]-20, tableCorners[0][1]-20,
        tableDimensions[0]+40, tableDimensions[1]+40
    ),
    15, # Width
    20 # Border Radius
)
```

```
# Draw Table
pygame.draw.rect(
    screen,
    primaryTableColour, # Colour
    pygame.Rect(
        tableCorners[0][0], # Top Left x
        tableCorners[0][1], # Top Left y
        tableDimensions[0], # Width
        tableDimensions[1] # Height
    ),
    border_radius=3 # Border Radius
)
```



Next, I will add the more complex edges to the table like the image shown here:

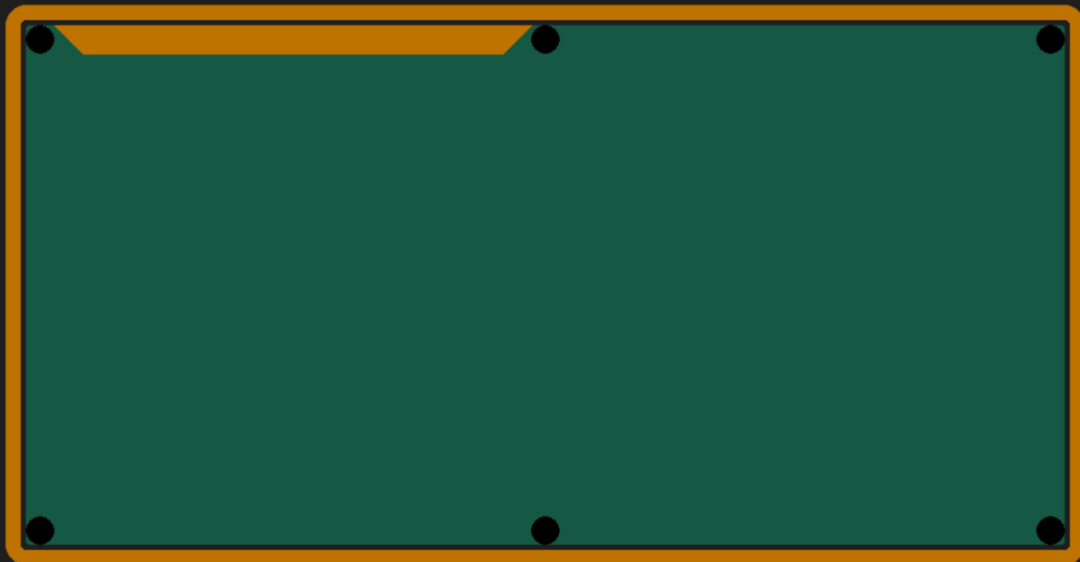


These are called the cushions or rails. This is the code for the top left cushion/rail:

```

# Top Left
pygame.draw.polygon(
    screen,
    outerTableColour,
    (
        (tableCorners[0][0] + 4*holeRadius, tableCorners[0][1]+2*holeRadius),
        (tableCorners[0][0] + 2*holeRadius, tableCorners[0][1]),
        (centre[0] - 1*holeRadius, tableCorners[0][1]),
        (centre[0] - 3*holeRadius, tableCorners[0][1]+2*holeRadius),
    )
)

```



```

# Top Left Side
pygame.draw.polygon(
    screen,
    outerTableColour,
    (
        (tableCorners[0][0] + 4*holeRadius, tableCorners[0][1]+2*holeRadius),
        (tableCorners[0][0] + 2*holeRadius, tableCorners[0][1]),
        (centre[0] - 1*holeRadius, tableCorners[0][1]),
        (centre[0] - 3*holeRadius, tableCorners[0][1]+2*holeRadius),
    )
)

# Top Right Side
pygame.draw.polygon(
    screen,
    outerTableColour,
    (
        (tableCorners[1][0] - 4*holeRadius, tableCorners[0][1]+2*holeRadius),
        (tableCorners[1][0] - 2*holeRadius, tableCorners[0][1]),
        (centre[0] + 1*holeRadius, tableCorners[0][1]),
        (centre[0] + 3*holeRadius, tableCorners[0][1]+2*holeRadius),
    )
)

```



```

# Bottom Left Side
pygame.draw.polygon(
    screen,
    outerTableColour,
    (
        (tableCorners[0][0] + 4*holeRadius, tableCorners[2][1]-2*holeRadius),
        (tableCorners[0][0] + 2*holeRadius, tableCorners[2][1]),
        (centre[0] - 1*holeRadius, tableCorners[2][1]),
        (centre[0] - 3*holeRadius, tableCorners[2][1]-2*holeRadius),
    )
)

# Bottom Right Side
pygame.draw.polygon(
    screen,
    outerTableColour,
    (
        (tableCorners[1][0] - 4*holeRadius, tableCorners[2][1]-2*holeRadius),
        (tableCorners[1][0] - 2*holeRadius, tableCorners[2][1]),
        (centre[0] + 1*holeRadius, tableCorners[2][1]),
        (centre[0] + 3*holeRadius, tableCorners[2][1]-2*holeRadius),
    )
)

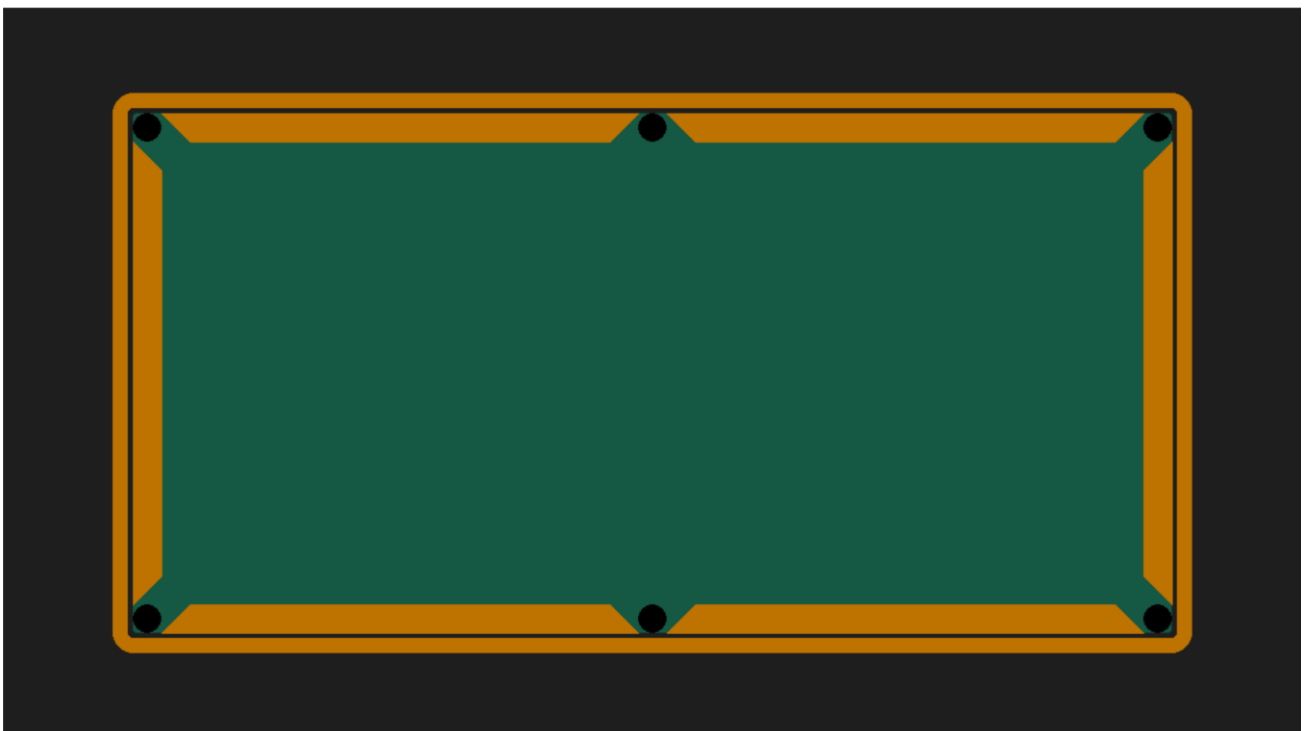
```

```

# Left Middle Side
pygame.draw.polygon(
    screen,
    outerTableColour,
    (
        (tableCorners[0][0]+2*holeRadius, tableCorners[1][1] + 4*holeRadius),
        (tableCorners[0][0], tableCorners[1][1] + 2*holeRadius),
        (tableCorners[0][0], tableCorners[3][1] - 2*holeRadius),
        (tableCorners[0][0]+2*holeRadius, tableCorners[3][1] - 4*holeRadius),
    )
)

# Right Middle Side
pygame.draw.polygon(
    screen,
    outerTableColour,
    (
        (tableCorners[1][0]-2*holeRadius, tableCorners[1][1] + 4*holeRadius),
        (tableCorners[1][0], tableCorners[1][1] + 2*holeRadius),
        (tableCorners[1][0], tableCorners[3][1] - 2*holeRadius),
        (tableCorners[1][0]-2*holeRadius, tableCorners[3][1] - 4*holeRadius),
    )
)

```

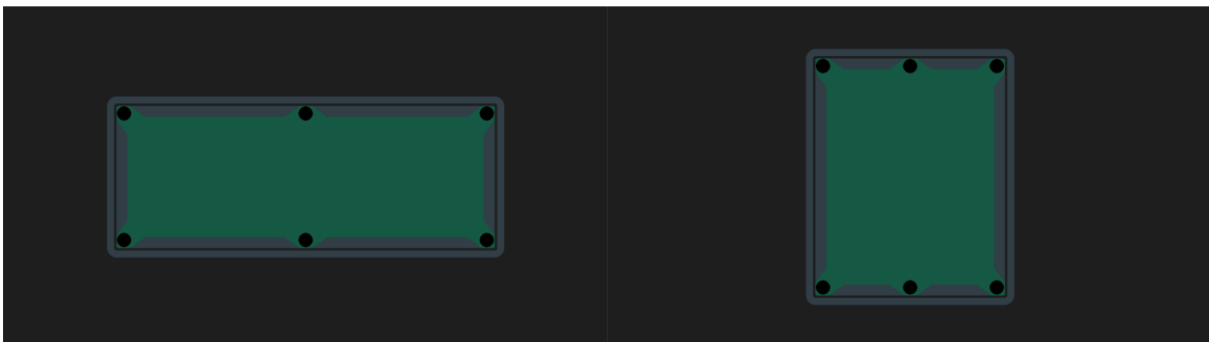


Updating Colours and other adjustments: (sides were shortened slightly)



Overall, the table is looking good now. I need to scale the table down now slightly to allow for other elements of the UI.

The way I have coded the solution will allow for this as shown below with multiple sizes of the table:



Here is the final table:



Now the table is done, I must add the balls to the table and the relevant user interface elements.

Here I have copied the ball class from Stage 1 to the main program, containing the pool table.

```
# Table Variables and Setup
primaryTableColour = "#155843"
secondaryTableColour = "#313e46"
outerTableColour = "#313e46"

tableDimensions = (900,450)
centre = (resolution[0]/2, resolution[1]/2)
holeRadius = 15
ballRadius = 10

# Ball Class
class Ball(pygame.sprite.Sprite): # Defines a class for a ball...

ballGroup = pygame.sprite.Group() # Groups all balls for easier maintenance
```

(I have used the collapse function on my IDE to shorten the Ball Class)

Since there are 7 coloured balls, 7 striped balls, one 8 ball and a white ball, they need to be instantiated from the class.

I will need to define how large each ball should be, and this is done via this variable:

```
ballRadius = 10
```

Each ball is going to be placed relative to the absolute centre of the table, using this tuple as the relative coordinates:

```
centre = (resolution[0]/2, resolution[1]/2)
```

```
# Ball Class
class Ball(pygame.sprite.Sprite): # Defines a class for a ball
    def __init__(self, colour, radius, pos, velocity, mass): # Constructor Method...

    def update(self): # Updates the position of the ball...

    def checkResolveCollision(self, other): # Check if balls are colliding, then resolves collision...

    def draw(self, screen): # Draws the ball to the screen...

ballGroup = pygame.sprite.Group() # Groups all balls for easier maintenance

# Define where each ball should be relative to the start position
ballStartPos = {
    "white0": (centre[0] - 20*ballRadius, centre[1]),
    "yellow1": (centre[0] + 15*ballRadius, centre[1]),
    "blue2": (centre[0] + 15*ballRadius - root3 * ballRadius, centre[1] + ballRadius),
    "red3": (centre[0] + 15*ballRadius - root3 * ballRadius, centre[1] - ballRadius),
    "purple4": (centre[0] + 15*ballRadius + 2 * root3 * ballRadius, centre[1] - 2 * ballRadius),
    "orange5": (centre[0] + 15*ballRadius + 2 * root3 * ballRadius, centre[1] + 2 * ballRadius),
    "black8": (centre[0] + 15*ballRadius + 2 * root3 * ballRadius, centre[1]),
}
```

Under the class, I have defined where each ball should be relative to the central position based on the radius of the balls. This is stored in a dictionary called “ballStartPos”

Under this, I have added code to instantiate each ball and place it in the position stated in the dictionary. I have not included every ball yet, as I will first test that a few are working first, then add more afterwards.

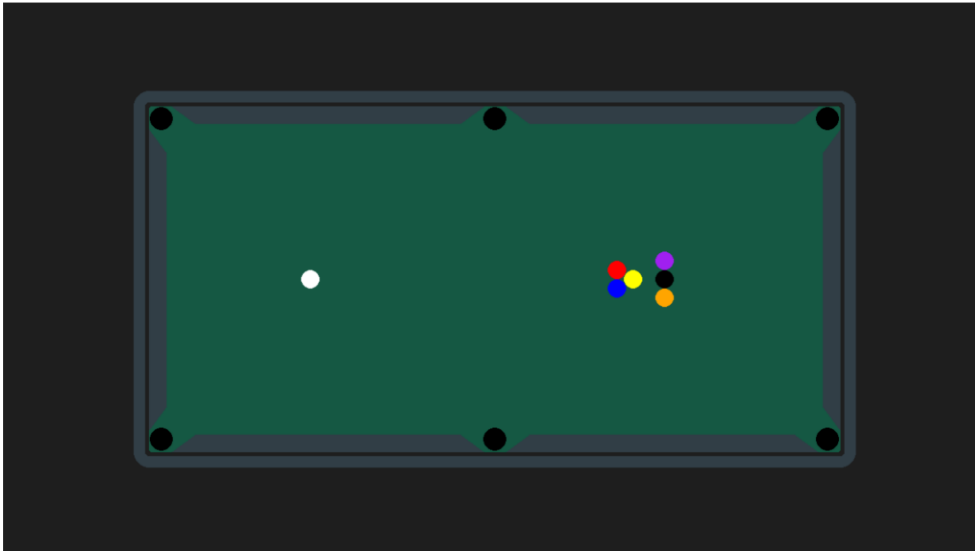
```
# Instantiate each ball
balls = [
    Ball("white", ballRadius, ballStartPos["white0"], (0, 0), 1), # Whiteball
    Ball("yellow", ballRadius, ballStartPos["yellow1"], (0, 0), 1), # Yellow 1
    Ball("blue", ballRadius, ballStartPos["blue2"], (0, 0), 1), # Blue 2
    Ball("red", ballRadius, ballStartPos["red3"], (0, 0), 1), # Red 3
    Ball("purple", ballRadius, ballStartPos["purple4"], (0, 0), 1), # Purple 4
    Ball("orange", ballRadius, ballStartPos["orange5"], (0, 0), 1), # Orange 5
    Ball("black", ballRadius, ballStartPos["black8"], (0, 0), 1), # Black 8
]

for i in balls:
    ballGroup.add(i)
```

I have also added all the balls in the array to a group so that they can be updated and drawn together with a few lines of code shown here in the game loop:

```
ballGroup.update() # update each ball
for ball in ballGroup.sprites():
    ball.draw(screen) # draws each ball
```

However, when I run the code:

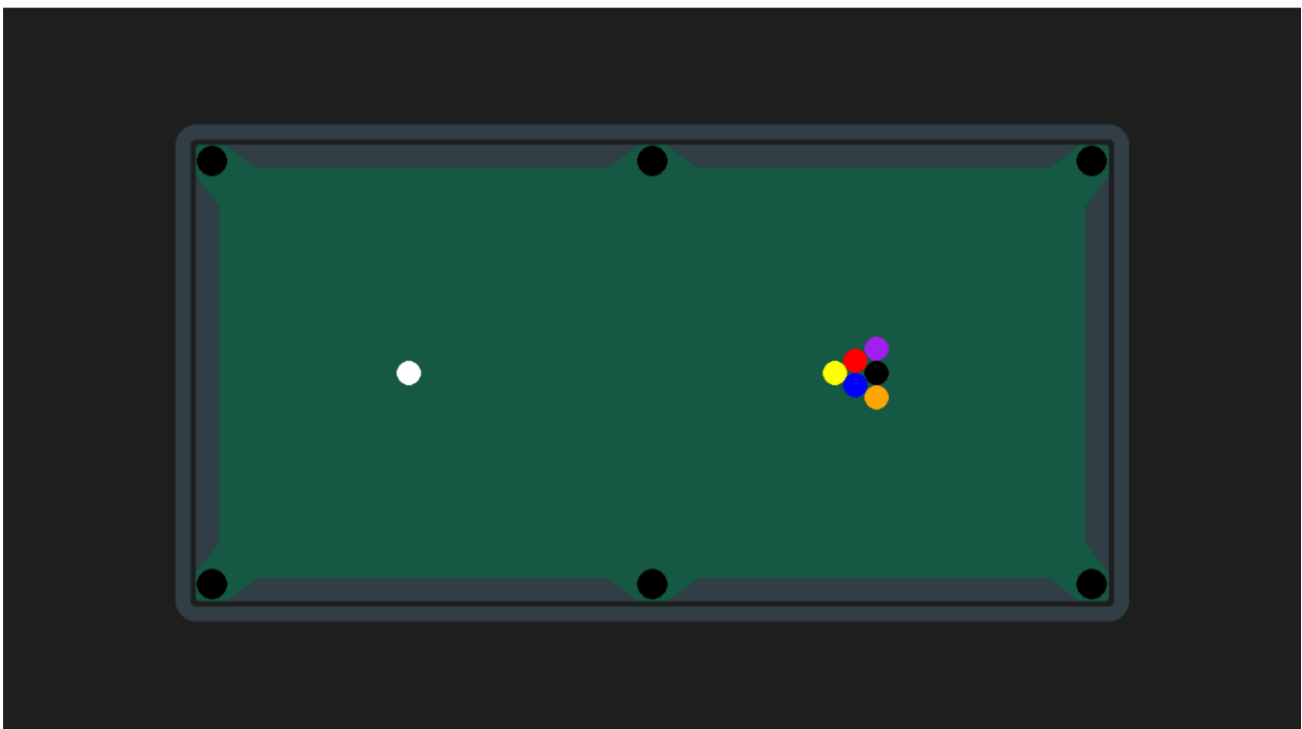


The balls are not in the correct position.

To fix this I have changed the position of the balls to these values:

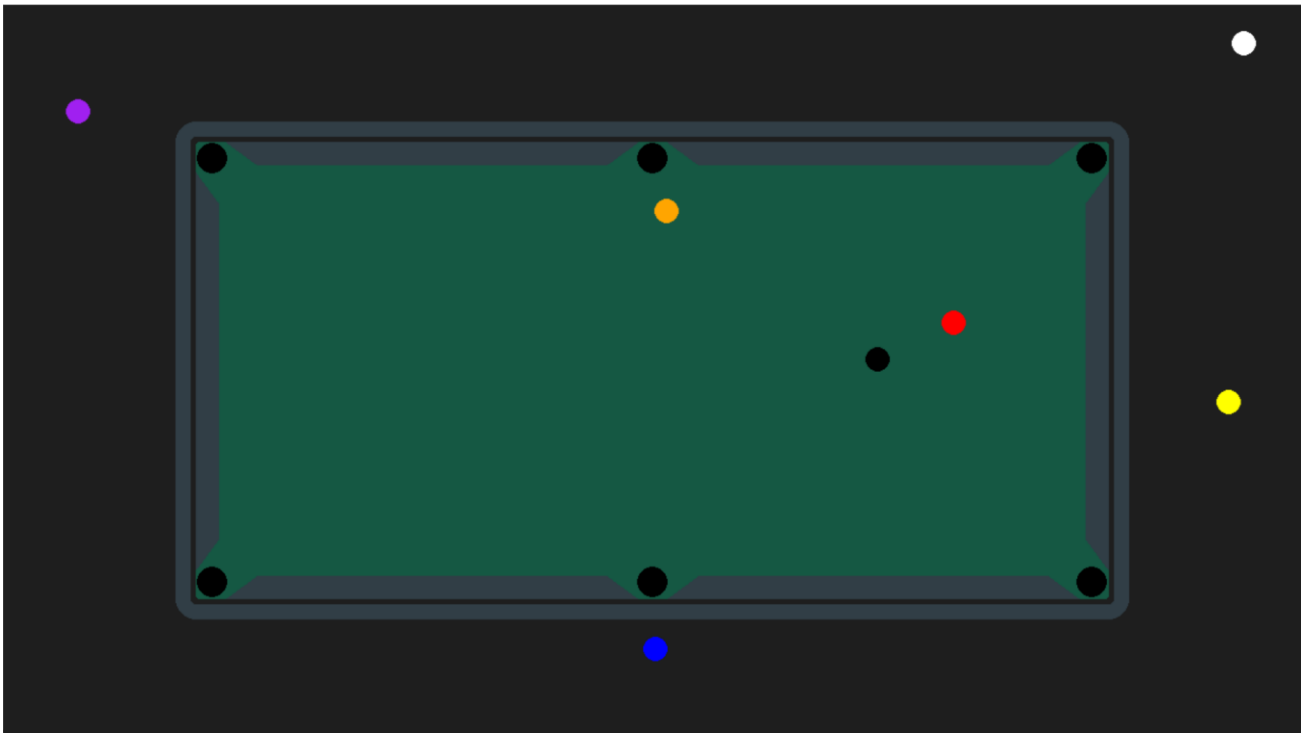
```
ballStartPos = {
  "white0": (centre[0] - 20*ballRadius, centre[1]),
  "yellow1": (centre[0] + 15*ballRadius, centre[1]),
  "blue2": (centre[0] + 15*ballRadius + root3 * ballRadius, centre[1] + ballRadius),
  "red3": (centre[0] + 15*ballRadius + root3 * ballRadius, centre[1] - ballRadius),
  "purple4": (centre[0] + 15*ballRadius + 2 * root3 * ballRadius, centre[1] - 2 * ballRadius),
  "orange5": (centre[0] + 15*ballRadius + 2 * root3 * ballRadius, centre[1] + 2 * ballRadius),
  "black8": (centre[0] + 15*ballRadius + 2 * root3 * ballRadius, centre[1]),
}
```

Now the balls are in the correct position:



When checking collisions, I realised that I have not included collisions for the walls of the billiard table, therefore the balls will clip through the table and move around on the background.

There is also no friction for the balls, therefore the balls move indefinitely. This will be fixed afterwards.



Firstly, I will restrict the balls to the table dimensions with the following code:

```
def update(self): # Updates the position of the ball
    self.pos += self.velocity

    # Checking collisions with the walls:

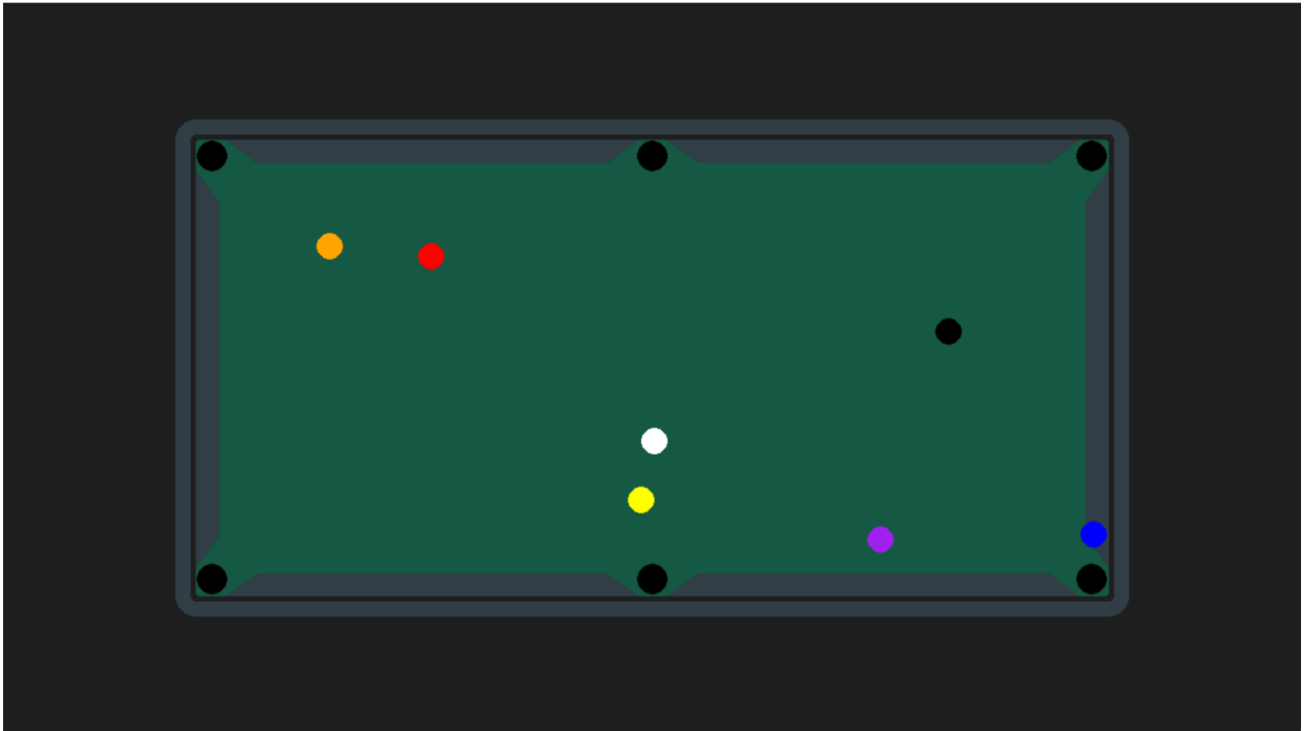
    if self.pos[0] - self.radius <= tableCorners[0][0]: # Left Wall
        self.pos[0] = self.radius + tableCorners[0][0] # Put ball on edge of wall
        self.velocity[0] *= -1 # Reverses X component

    if self.pos[0] + self.radius >= tableCorners[1][0]: # Right Wall
        self.pos[0] = tableCorners[1][0] - self.radius # Put ball on edge of wall
        self.velocity[0] *= -1 # Reverses X component

    if self.pos[1] - self.radius <= tableCorners[0][1]: # Top Wall
        self.pos[1] = self.radius + tableCorners[0][1] # Put ball on edge of wall
        self.velocity[1] *= -1 # Reverses Y component

    if self.pos[1] + self.radius >= tableCorners[2][1]: # Bottom Wall
        self.pos[1] = tableCorners[2][1] - self.radius # Put ball on edge of wall
        self.velocity[1] *= -1 # Reverses Y component
```

This restricts all the balls to the table dimensions but does not account for the polygons on each on the walls. (shown with the clipping of the blue ball).



The way that these polygons were created was by collecting four coordinates and the pygame library would create a surface. Given that this creates an irregular shape, the collision detection and simulation will be slightly more complicated.

Firstly, I will harvest all the coordinates in the polygon and store them in a dictionary shown here:

```

tableRails = {
    "TL": (
        (tableCorners[0][0] + 4*holeRadius, tableCorners[0][1]+1.5*holeRadius),
        (tableCorners[0][0] + 2*holeRadius, tableCorners[0][1]),
        (centre[0] - 1*holeRadius, tableCorners[0][1]),
        (centre[0] - 3*holeRadius, tableCorners[0][1]+1.5*holeRadius),
    ),

    "TR": (
        (tableCorners[1][0] - 4*holeRadius, tableCorners[0][1]+1.5*holeRadius),
        (tableCorners[1][0] - 2*holeRadius, tableCorners[0][1]),
        (centre[0] + 1*holeRadius, tableCorners[0][1]),
        (centre[0] + 3*holeRadius, tableCorners[0][1]+1.5*holeRadius),
    ),

    "BL": (
        (tableCorners[0][0] + 4*holeRadius, tableCorners[2][1]-1.5*holeRadius),
        (tableCorners[0][0] + 2*holeRadius, tableCorners[2][1]-1),
        (centre[0] - 1*holeRadius, tableCorners[2][1]-1),
        (centre[0] - 3*holeRadius, tableCorners[2][1]-1.5*holeRadius),
    ),

    "BR": (
        (tableCorners[1][0] - 4*holeRadius, tableCorners[2][1]-1.5*holeRadius),
        (tableCorners[1][0] - 2*holeRadius, tableCorners[2][1]-1),
        (centre[0] + 1*holeRadius, tableCorners[2][1]-1),
        (centre[0] + 3*holeRadius, tableCorners[2][1]-1.5*holeRadius),
    ),

    "ML": (
        (tableCorners[0][0]+1.5*holeRadius, tableCorners[1][1] + 4*holeRadius),
        (tableCorners[0][0], tableCorners[1][1] + 2*holeRadius),
        (tableCorners[0][0], tableCorners[3][1] - 2*holeRadius),
        (tableCorners[0][0]+1.5*holeRadius, tableCorners[3][1] - 4*holeRadius),
    ),

    "MR": (
        (tableCorners[1][0]-1.5*holeRadius, tableCorners[1][1] + 4*holeRadius),
        (tableCorners[1][0]-1, tableCorners[1][1] + 2*holeRadius),
        (tableCorners[1][0]-1, tableCorners[3][1] - 2*holeRadius),
        (tableCorners[1][0]-1.5*holeRadius, tableCorners[3][1] - 4*holeRadius),
    ),
}

# Draw Table Rails
pygame.draw.polygon(screen, secondaryTableColour, tableRails["TL"]) # Top Left Side
pygame.draw.polygon(screen, secondaryTableColour, tableRails["TR"]) # Top Right Side
pygame.draw.polygon(screen, secondaryTableColour, tableRails["BL"]) # Bottom Left Side
pygame.draw.polygon(screen, secondaryTableColour, tableRails["BR"]) # Bottom Right Side
pygame.draw.polygon(screen, secondaryTableColour, tableRails["ML"]) # Left Middle Side
pygame.draw.polygon(screen, secondaryTableColour, tableRails["MR"]) # Right Middle Side

# Draw Holes
for i in holePos:
    pygame.draw.circle(screen, "black", (holePos[i][0],holePos[i][1]), holeRadius)

```


This has been done so that I can now implement an algorithm that will connect each point, drawing a line segment. This will then be used to check if each ball collides with the railing, meaning a collision algorithm can be implemented.

Firstly, I will check which walls need to have collisions.

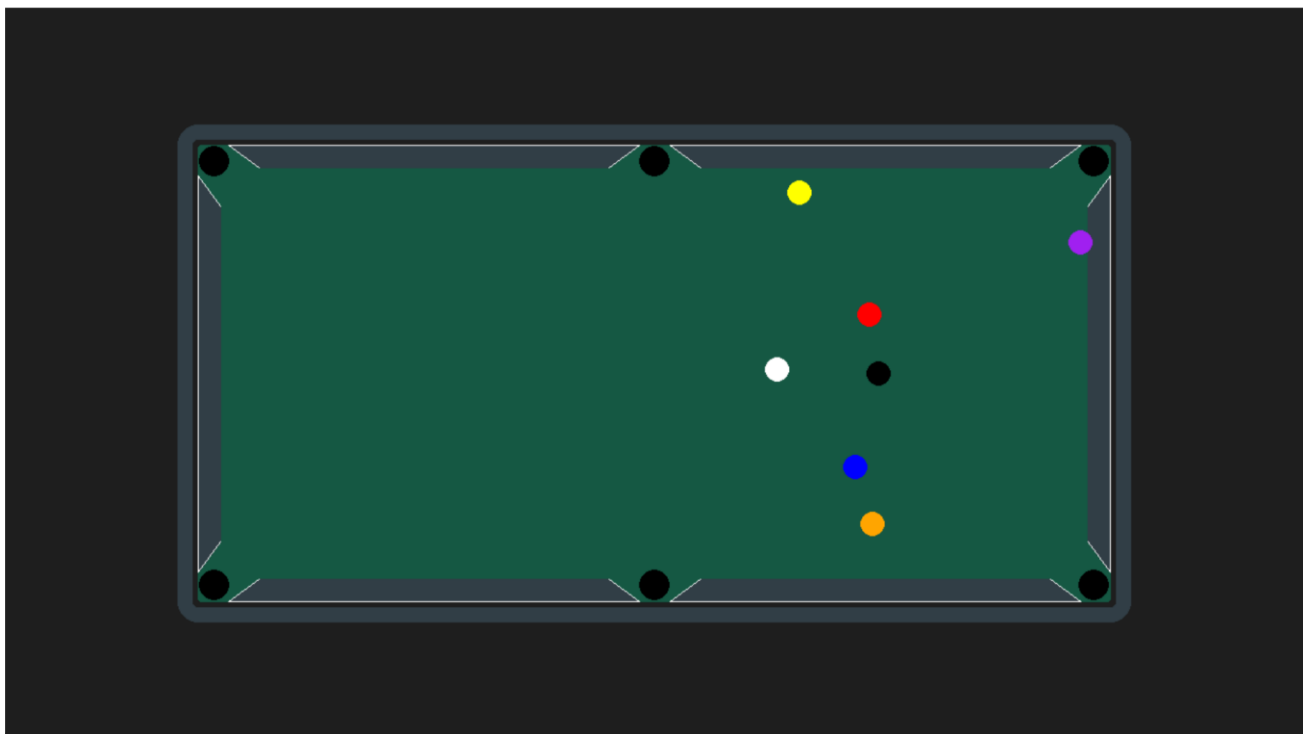
I will draw each line that needs a collision on the table for better visualisation.

Using this for loop:

```
# Draws a white line on each of the Rails (where collisions should be done)
for i in tableRails:
    for j in range(3):
        pygame.draw.line(screen, "white", tableRails[i][j],tableRails[i][j+1])
```

This will iterate through each of the points in the polygon and draw a line segment between all but one line that does not need collisions.

Here is the result I got:



The code works well, but the rail that faces the table must have a line and the line that faces away from the table should not. Therefore, I will reorder the coordinates in the dictionary.

With the same code in the game loop, I have changed the table rails shown below so that loop code is as simple as possible.

```

tableRails = {
  "TL": (
    (centre[0] - 1*holeRadius, tableCorners[0][1]),
    (centre[0] - 3*holeRadius, tableCorners[0][1]+1.5*holeRadius),
    (tableCorners[0][0] + 4*holeRadius, tableCorners[0][1]+1.5*holeRadius),
    (tableCorners[0][0] + 2*holeRadius, tableCorners[0][1]),
  ),

  "TR": (
    (centre[0] + 1*holeRadius, tableCorners[0][1]),
    (centre[0] + 3*holeRadius, tableCorners[0][1]+1.5*holeRadius),
    (tableCorners[1][0] - 4*holeRadius, tableCorners[0][1]+1.5*holeRadius),
    (tableCorners[1][0] - 2*holeRadius, tableCorners[0][1]),
  ),

  "BL": (
    (centre[0] - 1*holeRadius, tableCorners[2][1]-1),
    (centre[0] - 3*holeRadius, tableCorners[2][1]-1.5*holeRadius),
    (tableCorners[0][0] + 4*holeRadius, tableCorners[2][1]-1.5*holeRadius),
    (tableCorners[0][0] + 2*holeRadius, tableCorners[2][1]-1),
  ),

  "BR": (
    (centre[0] + 1*holeRadius, tableCorners[2][1]-1),
    (centre[0] + 3*holeRadius, tableCorners[2][1]-1.5*holeRadius),
    (tableCorners[1][0] - 4*holeRadius, tableCorners[2][1]-1.5*holeRadius),
    (tableCorners[1][0] - 2*holeRadius, tableCorners[2][1]-1),
  ),

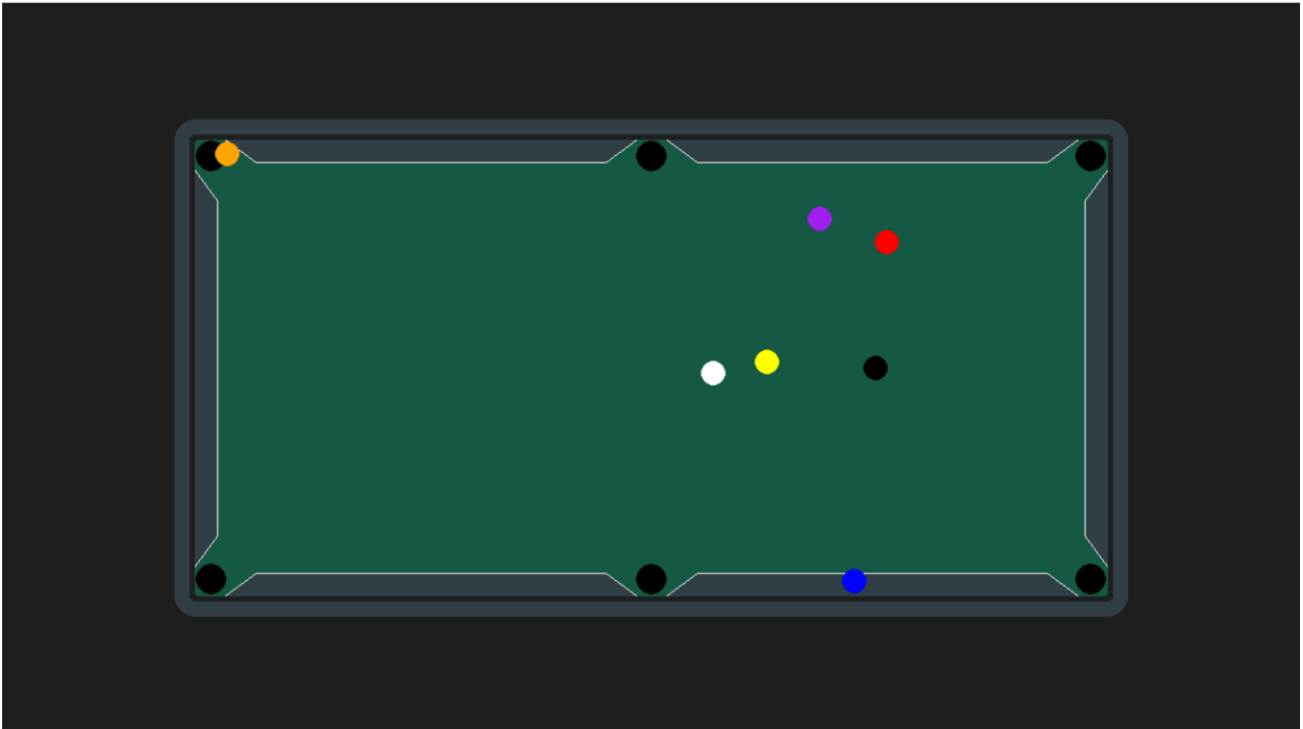
  "ML": (
    (tableCorners[0][0], tableCorners[3][1] - 2*holeRadius),
    (tableCorners[0][0]+1.5*holeRadius, tableCorners[3][1] - 4*holeRadius),
    (tableCorners[0][0]+1.5*holeRadius, tableCorners[1][1] + 4*holeRadius),
    (tableCorners[0][0], tableCorners[1][1] + 2*holeRadius),
  ),

  "MR": (
    (tableCorners[1][0]-1, tableCorners[3][1] - 2*holeRadius),
    (tableCorners[1][0]-1.5*holeRadius, tableCorners[3][1] - 4*holeRadius),
    (tableCorners[1][0]-1.5*holeRadius, tableCorners[1][1] + 4*holeRadius),
    (tableCorners[1][0]-1, tableCorners[1][1] + 2*holeRadius),
  ),

},
}

```

With this code, the following table is executed:



Since I now know which sides need collisions and the fact that they are accessible from the looping code, I can now start work detecting and simulating the collisions.

For each line segment, I will compute whether each ball is overlapping with the line, meaning a collision can be computed.

Here is the code to check if each circle is overlapping with each line on the pool table: Note that the actual collision simulation will be done after, but this is important to check if the balls collide with the edges correctly.

```
def checkResolveWallCollision(self):
    # Loops through each of the Rails (where collisions should be done)
    for i in tableRails:
        for j in range(3):
            lineStart = tableRails[i][j] # Coordinates of the Line Start and End
            lineEnd = tableRails[i][j+1] # e.g. [ [x1,y1] , [x2,y2] ]

            lineVec = VEC(lineEnd[0] - lineStart[0], lineEnd[1] - lineStart[1]) # Vector of the Line
            circleVec = VEC(self.pos[0] - lineStart[0], self.pos[1] - lineStart[1]) # Vector of the circle to the start of the line

            t = max(0, min(1, (circleVec[0] * lineVec[0] + circleVec[1] * lineVec[1]) / lineVec.magnitude_squared()))
            # This creates a parameter that will calculate the s.f of the normal projection that maps the circle onto the line segment

            linePoint = (lineStart[0] + t * lineVec[0], lineStart[1] + t * lineVec[1]) # Calculate closest point on the line to the circle
            circlePointVec = VEC(linePoint[0] - self.pos[0], linePoint[1] - self.pos[1]) # Vector of the circle to the closest point

            distance = circlePointVec.magnitude() # calculates distance between the circle and closest point

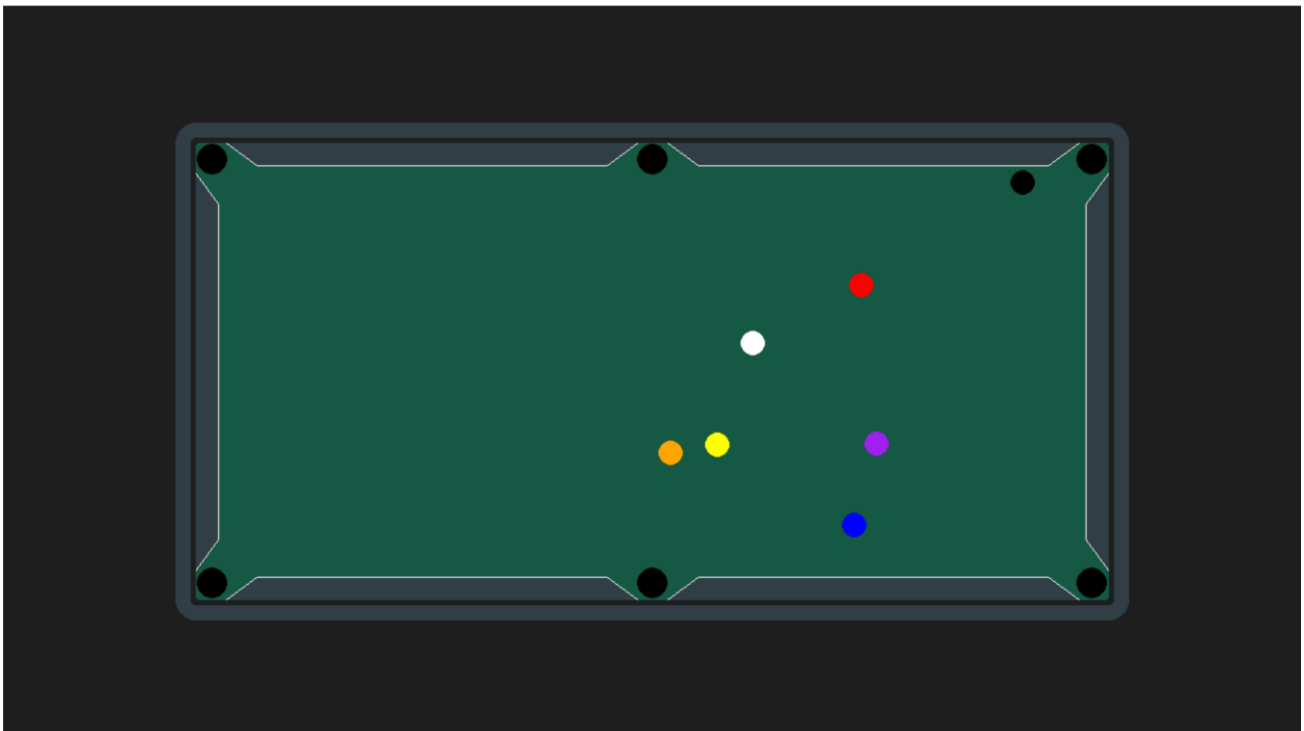
            # Check if colliding
            if distance <= self.radius:
                self.velocity[0] *= -1
                self.velocity[1] *= -1
```

This method in the class ball is called every game tick in the game loop:

```
for ball in ballGroup.sprites():
    ball.checkResolveWallCollision() # checks wall collisions
    ball.draw(screen) # draws each ball
```

This is shown here:

The black ball has just collided with the wall and now has been reflected back.



Now that I know when collisions should be calculated, there must be accurate collisions between the balls and the walls. Currently, the balls completely reflect at 180 degrees. I will now aim to fix this here: This is done here:

This code has largely been copied from the ball-to-ball collision calculation.

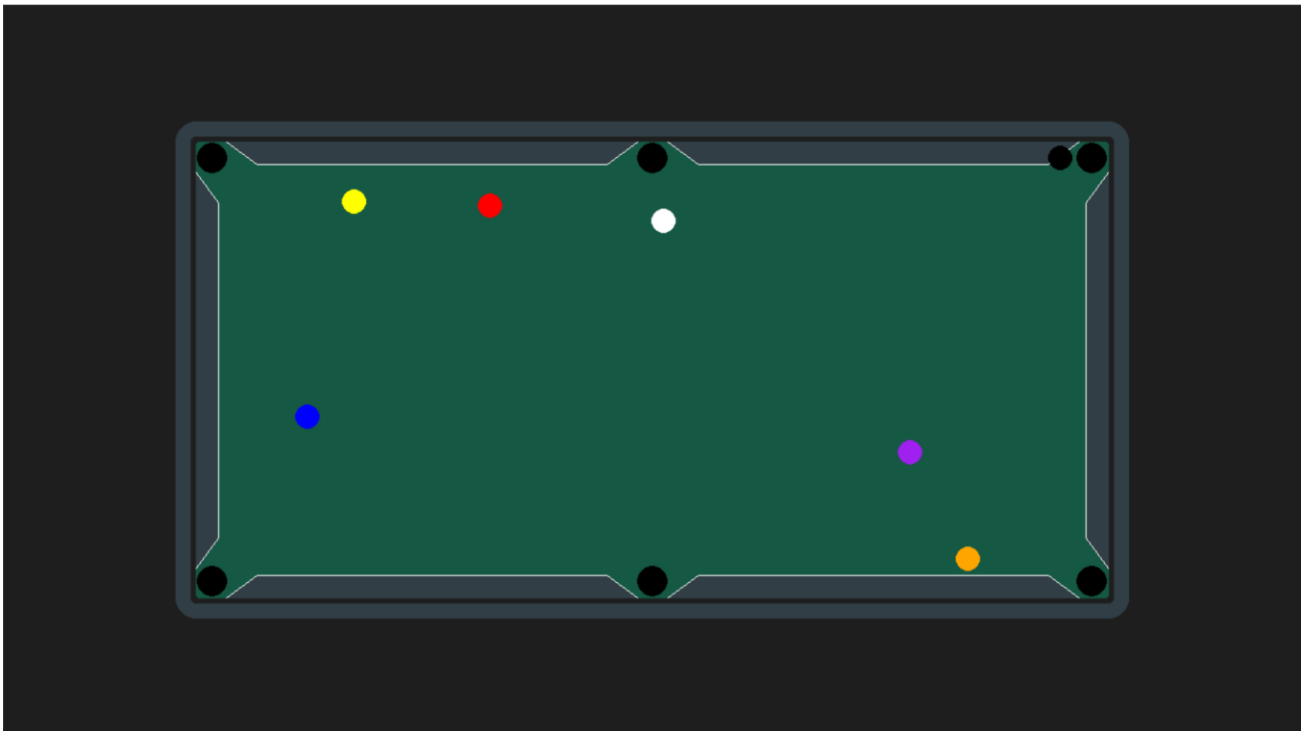
```
# Check if colliding
if distance <= self.radius:

    n = circlePointVec.normalize() # creates a normal vector as a origin for calculations

    impulse = 2 * (n * self.velocity) / (self.mass) # calculates the impulse the ball will get

    self.velocity -= impulse * n # applies impulse to each ball
```

The calculations are correct, given that each ball now accurately reflects off each wall:



However, sometimes a ball will clip into the rail, which obviously needs to be fixed.

This has been done by firstly undoing the update method that is run every tick in the game loop:

```
# Check if colliding
if distance <= self.radius:

    self.pos -= self.velocity # undoes previous update

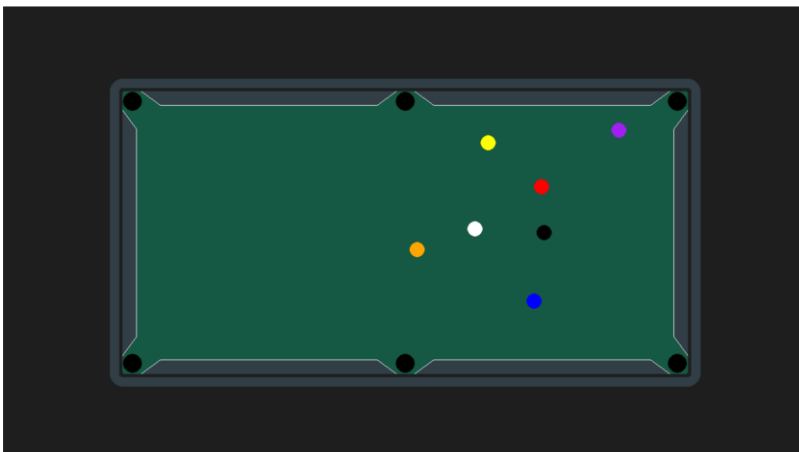
    n = circlePointVec.normalize() # creates a normal vector as a origin for calculations

    impulse = 2 * (n * self.velocity) / (self.mass) # calculates the impulse the ball will get

    self.velocity -= impulse * n # applies impulse to each ball
```

Now all the collisions are accurate.

This is shown with a snapshot of the game here:



To aid with performance improvements later in development, I will add an FPS counter to the top right of the screen.

This code is added has been added to the game loop

```
# FPS Display
currentFPS = str(round(clock.get_fps(), 1))
text = pygame.font.SysFont("dubaimedium", 20).render(
    (str(currentFPS)), True, "white")
screen.blit(text, (5, 5))
```

Implementation shown here with the FPS located in the top left of the screen.



Each ball moves forever, and therefore friction should be implemented to stop this from happening.

Therefore, each game tick, the velocity of the ball should decrease.

In the update method in the class ball, I have changed the velocity so that it decrease over time.

I have defined a variable that will store the value of friction:

```
friction = 0.995
```

```
def update(self): # Updates the position of the ball
    self.pos += self.velocity # moves ball to next position
    self.velocity *= friction # reduces the velocity of the ball
```

Eventually, each ball will reduce their velocities to zero.

I will also add a coefficient of restitution to all objects and their collisions:

```
friction = 0.991
restitutionCoeff = 0.9
```

This has been added at the bottom of the pictures to the code in the resolving collisions method:

```

if distanceSquared < (self.radius + other.radius)**2: # If colliding
    self.pos -= self.velocity # undoes last move, to prevent clipping

    # Calculate Velocities after collision
    n = VEC(posDifference).normalize() # creates a normalised vector as a origin for calculations
    relVel = self.velocity - other.velocity # calculates the relative velocity of the balls
    impulse = 2 * (n * relVel) / (self.mass + other.mass) # calculates the impulse each ball will get

    self.velocity -= impulse * other.mass * n # applies impulse to each ball
    other.velocity += impulse * self.mass * n

    self.velocity *= restitutionCoeff # decreases velocity of both balls
    other.velocity *= restitutionCoeff

```

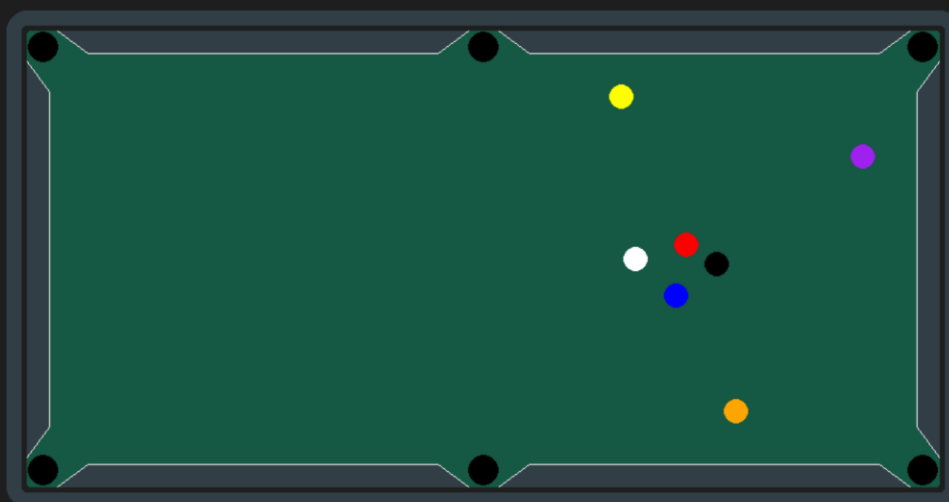
```

# Check if colliding
if distance <= self.radius:
    self.pos -= self.velocity # undoes previous update
    n = circlePointVec.normalize() # creates a normal vector as a origin for calculations
    impulse = 2 * (n * self.velocity) / (self.mass) # calculates the impulse the ball will get
    self.velocity -= impulse * n # applies impulse to each ball
    self.velocity *= restitutionCoeff # decreases velocity

```

After adding a coefficient of restitution and friction, the game eventually comes to a stop after the white ball is played.

61.3



Now I will add all of the balls to the game, following the rules of 8 ball pool.

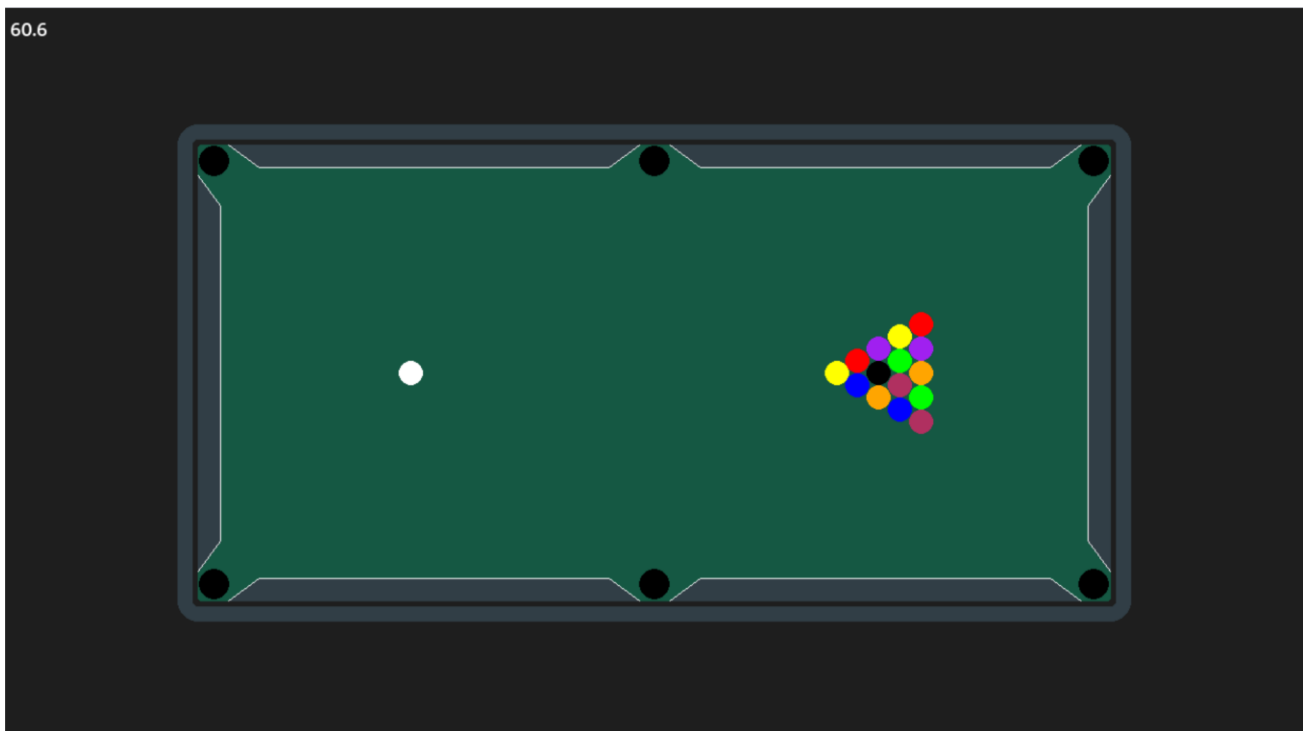
The dictionary, ballStartPos, has been updated to include all the balls in 8 ball pool:

```
# Define where each ball should be relative to the start position
ballStartPos = {
    "white0": (centre[0] - 20*ballRadius, centre[1]),
    "yellow1": (centre[0] + 15*ballRadius, centre[1]),
    "blue2": (centre[0] + 15*ballRadius + 1 * root3 * ballRadius, centre[1] + 1 * ballRadius),
    "red3": (centre[0] + 15*ballRadius + 1 * root3 * ballRadius, centre[1] - 1 * ballRadius),
    "purple4": (centre[0] + 15*ballRadius + 2 * root3 * ballRadius, centre[1] - 2 * ballRadius),
    "orange5": (centre[0] + 15*ballRadius + 2 * root3 * ballRadius, centre[1] + 2 * ballRadius),
    "green6": (centre[0] + 15*ballRadius + 3 * root3 * ballRadius, centre[1] - 1 * ballRadius),
    "maroon7": (centre[0] + 15*ballRadius + 3 * root3 * ballRadius, centre[1] + 1 * ballRadius),
    "black8": (centre[0] + 15*ballRadius + 2 * root3 * ballRadius, centre[1]),
    "yellow9": (centre[0] + 15*ballRadius + 3 * root3 * ballRadius, centre[1] - 3 * ballRadius),
    "blue10": (centre[0] + 15*ballRadius + 3 * root3 * ballRadius, centre[1] + 3 * ballRadius),
    "red11": (centre[0] + 15*ballRadius + 4 * root3 * ballRadius, centre[1] - 4 * ballRadius),
    "purple12": (centre[0] + 15*ballRadius + 4 * root3 * ballRadius, centre[1] - 2 * ballRadius),
    "orange13": (centre[0] + 15*ballRadius + 4 * root3 * ballRadius, centre[1]),
    "green14": (centre[0] + 15*ballRadius + 4 * root3 * ballRadius, centre[1] + 2 * ballRadius),
    "maroon15": (centre[0] + 15*ballRadius + 4 * root3 * ballRadius, centre[1] + 4 * ballRadius),
}
```

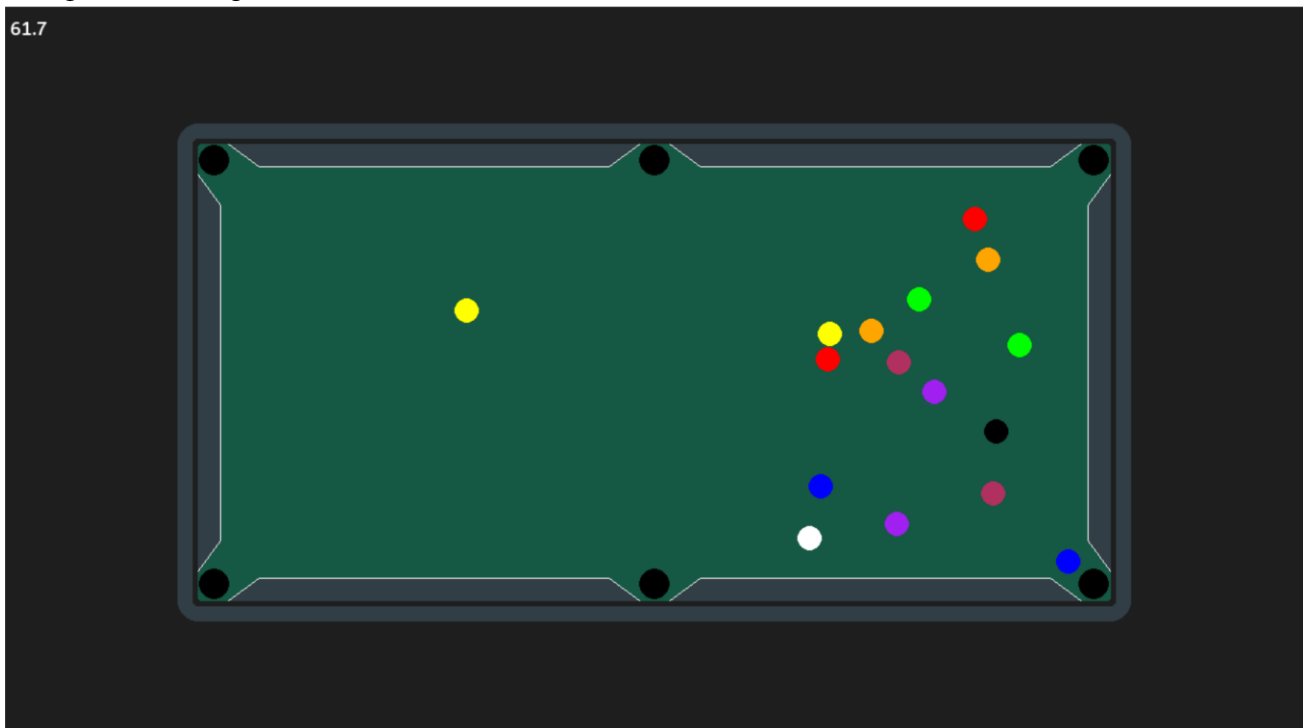
I have added the relevant code to now instantiate each ball:

```
# Instantiate each ball
balls = [
    Ball("white", ballRadius, ballStartPos["white0"], (0, 0), 1), # Whiteball
    Ball("yellow", ballRadius, ballStartPos["yellow1"], (0, 0), 1), # Yellow 1
    Ball("blue", ballRadius, ballStartPos["blue2"], (0, 0), 1), # Blue 2
    Ball("red", ballRadius, ballStartPos["red3"], (0, 0), 1), # Red 3
    Ball("purple", ballRadius, ballStartPos["purple4"], (0, 0), 1), # Purple 4
    Ball("orange", ballRadius, ballStartPos["orange5"], (0, 0), 1), # Orange 5
    Ball("green", ballRadius, ballStartPos["green6"], (0, 0), 1), # Green 6
    Ball("maroon", ballRadius, ballStartPos["maroon7"], (0, 0), 1), # Maroon 7
    Ball("black", ballRadius, ballStartPos["black8"], (0, 0), 1), # Black 8
    Ball("yellow", ballRadius, ballStartPos["yellow9"], (0, 0), 1), # Yellow 9
    Ball("blue", ballRadius, ballStartPos["blue10"], (0, 0), 1), # Blue 10
    Ball("red", ballRadius, ballStartPos["red11"], (0, 0), 1), # Red 11
    Ball("purple", ballRadius, ballStartPos["purple12"], (0, 0), 1), # Purple 12
    Ball("orange", ballRadius, ballStartPos["orange13"], (0, 0), 1), # Orange 13
    Ball("green", ballRadius, ballStartPos["green14"], (0, 0), 1), # Green 14
    Ball("maroon", ballRadius, ballStartPos["maroon15"], (0, 0), 1), # Maroon 15
]
```

When run, this is the game start:



The game working well at the moment:



Up until now, I have been simulating the game without user involvement by initially setting the white ball to have a certain velocity at the start.

Now I will implement a basic cue that will allow me to give the white ball a speed in a direction. This will allow me to test and play the game further.

Firstly, I will implement a check for when all the balls are still. This means that when all the balls are still, the cue should be displayed.

```

ballMoving = False # sets initial case to False
ballGroup.update() # update each ball
for ball in ballGroup.sprites():
    if ball.velocity.magnitude_squared() != 0: # checks if any ball is still moving.
        ballMoving = True # if even one ball is still moving, it is set to True.

```

When testing this, I have realised a problem, the ball velocity is never zero after initially simulating and rather each ball speed is just a very small number.

Here I have printed the balls' speed. As you can see, each number is very small, but not zero.

```

2.6897328621732118e-18
6.26470335849924e-18
6.692911375725989e-19
3.4583995255572593e-18

```

This means that I must set the velocity to zero when the speed reaches a very low number.

This has been done here:

```

def update(self): # Updates the position of the ball
    self.pos += self.velocity # moves ball to next position
    self.velocity *= friction # reduces the velocity of the ball

    if self.velocity.magnitude_squared() < 0.005:
        self.velocity = VEC(0,0)

```

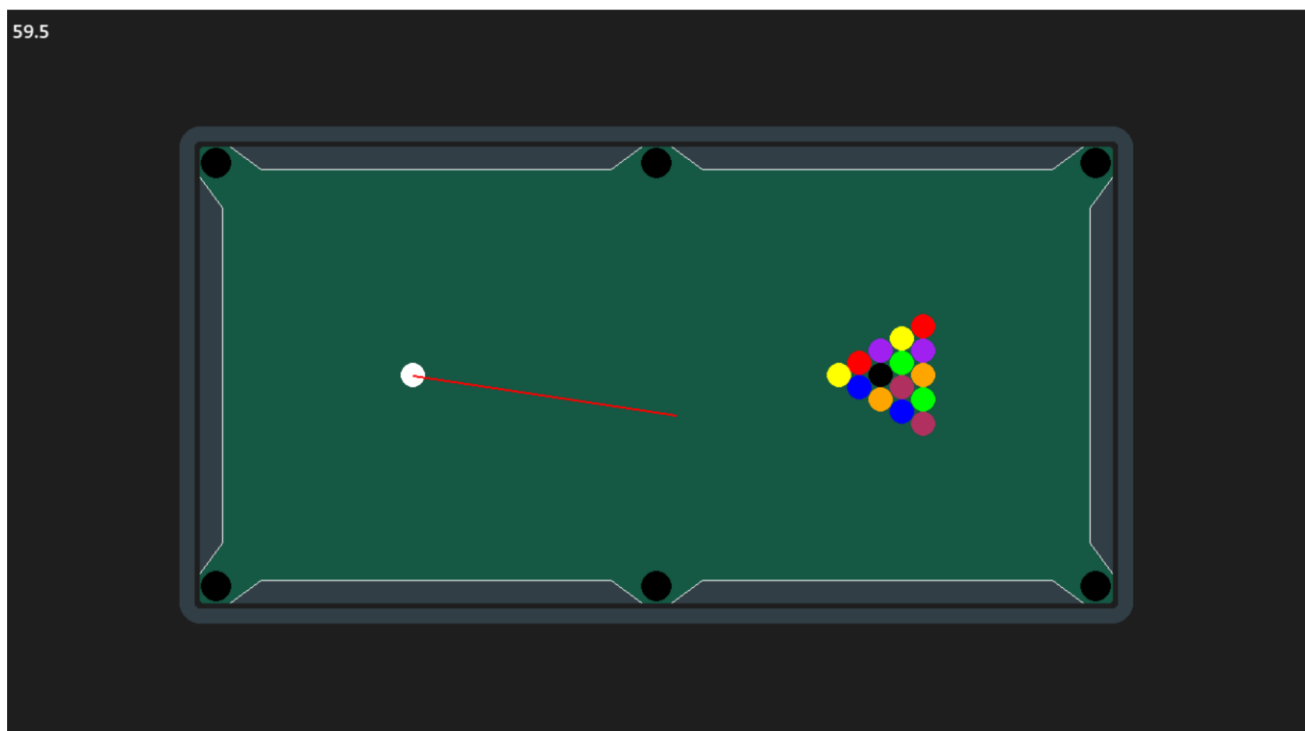
(when the velocity gets low enough, the velocity is set to zero)

Now the Boolean variable, ballMoving, works correctly. I will draw a line that goes from the ball to the mouse pointer and have that act as the cue.

```

if not ballMoving: # when all the balls are still
    pygame.draw.line(screen, "red", balls[0].pos, pygame.mouse.get_pos(), 2) # draw cue

```



Now on click, the white ball should be given a velocity:

In the game loop I have added a checker for when the mouse is clicked:

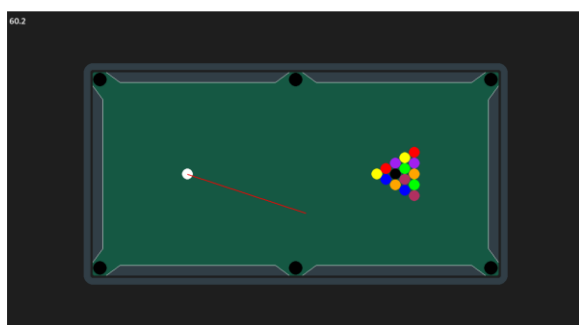
```
# Game Loop
while True:
    mouseDown = False
    for event in pygame.event.get():
        if event.type == pygame.QUIT: # Checks if user closes program
            pygame.quit()
            exit()

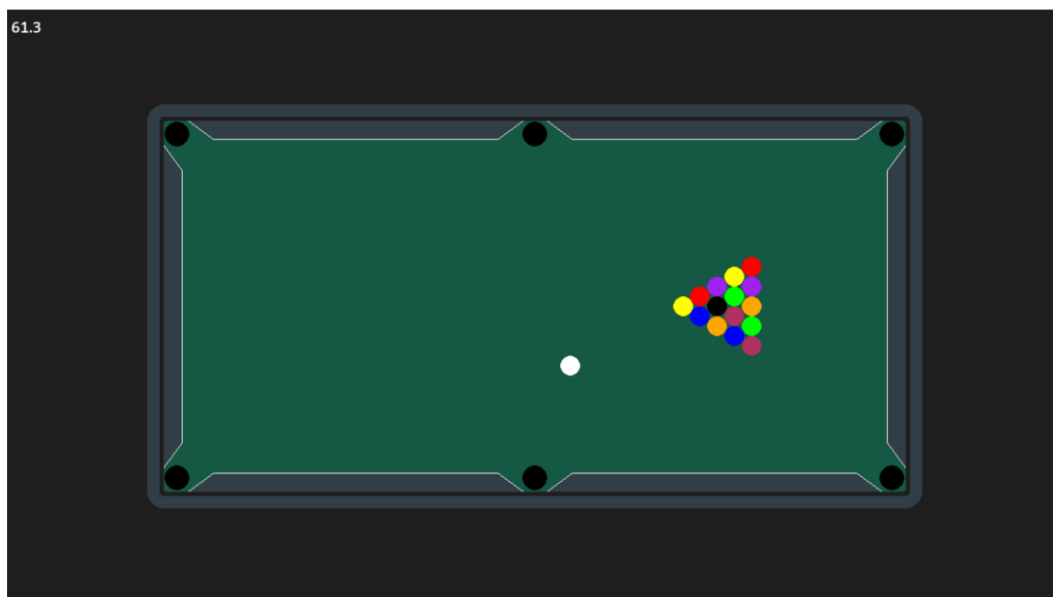
        if event.type == pygame.MOUSEBUTTONDOWN: # on left click
            mouseDown = True
```

Later on in the game loop:

```
if not ballMoving: # when all the balls are still
    pygame.draw.line(screen, "red", balls[0].pos, pygame.mouse.get_pos(), 2) # draw cue
    if mouseDown: # when clicked
        mousePos = pygame.mouse.get_pos() # get mouse position
        mouseVec = VEC(mousePos[0]-balls[0].pos[0],mousePos[1]-balls[0].pos[1]) # vector of line
        balls[0].velocity = mouseVec.normalize()*20 # normalise vector and give ball a magnitude
```

This works as wanted but the ball travels in the opposite direction to what I had wanted.





Therefore, I have swapped the elements in the mouseVec, meaning it travels in the direction I want now:

```
if not ballMoving: # when all the balls are still
    pygame.draw.line(screen, "red", balls[0].pos, pygame.mouse.get_pos(), 2) # draw cue
    if mouseDown: # when clicked
        mousePos = pygame.mouse.get_pos() # get mouse position
        mouseVec = VEC(balls[0].pos[0]-mousePos[0], balls[0].pos[1]-mousePos[1]) # vector of line
        balls[0].velocity = mouseVec.normalize()*20 # normalise vector and give ball a magnitude
```

When testing the collisions further, I have found that if I comment this code out, the ball collisions at higher speeds are more accurate.

This is in the ball class in the checkResolveCollision() method.

The last line shown has been commented out:

```
def checkResolveCollision(self, other): # Check if balls are colliding, then resolves collision
    # Check if balls colliding
    posDifference = self.pos - other.pos # find difference in positions
    distanceSquared = self.pos.distance_squared_to(other.pos) # distance squared between points

    if distanceSquared == 0: # prevents normalisation error
        return

    if distanceSquared < (self.radius + other.radius)**2: # If colliding
        # self.pos -= self.velocity # undoes last move, to prevent clipping
```

This line is redundant as balls clipping in each other is resolved with the next while loop where balls are pushed out:

This shown here:

```
# Push out the colliding balls
while distanceSquared < (self.radius + other.radius)**2:
    self.pos += n # move balls away from each other
    other.pos -= n
    distanceSquared = self.pos.distance_squared_to(other.pos) # distance squared between points
```

I will now add a delete method that will delete each ball that moves into any hole in the pool table. Firstly, I have defined a new method that will remove the instance of the ball.

```
def delete(self):
    self.delete()
```

I will also add another method that will check for collisions between the hole and the ball.

Here I have created a new method that aims to check if each ball is colliding with the holes:

```
def checkHoleCollision(self):
    for i in holePos:
        distance2 = self.pos.distance_squared_to(holePos[i])
        if distance2 <= (holeRadius + ballRadius)**2:
            self.delete()
```

However, when a ball reaches the hole, an error occurs:

```
self.delete()
[Previous line repeated 995 more times]
RecursionError: maximum recursion depth exceeded
```

This has been resolved by changing the code to the following: (self.kill()) instead of self.delete())

```
def delete(self):
    self.kill()
```

```
def checkHoleCollision(self):
    for i in holePos: # for each hole on the table
        distance2 = self.pos.distance_squared_to(holePos[i]) # compute the distance
        if distance2 <= (holeRadius + ballRadius)**2: # if colliding
            return True
    return False
```

In the game loop:

```
ballMoving = False # sets initial case to False
ballGroup.update() # update each ball
for ball in ballGroup.sprites():
    if ball.velocity != (0,0): # checks if the ball is completely still
        ballMoving = True # if even one ball is still moving, it is set to True.
    ball.checkResolveWallCollision() # checks wall collisions
    if ball.checkHoleCollision(): # checks if each ball is in a hole
        ball.delete() # removes the ball from the game
    ball.draw(screen) # draws each ball
```

This now works. However, a logical error occurs when the white ball is potted. This causes the player to not be able to play anymore.

This will be noted and resolved in a later stage (stage 4), but for the moment, the white ball will be immune to being potted. Shown with the code below:

```
def delete(self):  
    if self.colour != "white": # delete later  
        self.kill()
```

Stage 2 Review

What has been done?

- Pool game table set up (holes, rails, rounded edges, all with adjustable sizes).
- Added all balls for 8-ball pool: 16 balls (7 coloured, 7 striped, 8-ball, white ball).
- Balls now bounce off walls and rails realistically.
- Friction and restitution simulate realistic movement and energy loss.
- Clipping and overlap issues have been fixed.
- Cue mechanics, albeit very basic, have been implemented. Allows the user to hit the white ball.
- Balls entering holes are removed (except the white ball).

How is code being tested?

- For each new element/function (e.g., ball movement, wall collisions, ball-to-ball collisions), I add a small bit of functionality and then observe the result, documenting and debugging it accordingly. This incremental approach ensures I follow the computational method of decomposition which ensures for swift development of the solution.
- Bugs (like clipping, stuck balls, or zero-distance errors) are identified and resolved by modifying code of the program, rerunning the simulation, and documenting my errors.
- Comments are used extensively throughout the solution. This helps debugging enormously as I sift through hundreds of lines of code.

Errors that have been fixed:

Error	Fix
The initially placed balls were misaligned.	I adjusted the ball positions to the correct position values
There was no collision detection with table walls, allowing balls to move off the table.	I restricted the ball movement to only be within the table dimensions.
Collision detection worked for straight edges but not for irregularly shaped rails.	I stored polygon coordinates, drew collision lines with each of the lines of the polygon, and then updated the algorithm to detect and resolve collisions with the rails.
Balls were being reflected at 180 degrees instead of an accurate bounce angle.	I adjusted the reflection calculations to use correct physics from ball-to-ball collision code, which made the balls reflect at an accurate angle.
Some balls partially entered the rails before bouncing.	I undid the update step when a collision was detected to correct the ball's position.
After adding friction, ball velocities remained as very small numbers instead of stopping completely.	Set velocity to zero when it falls below a near-zero threshold.
The cue applied force in the opposite direction of the intended shot.	Swapped elements in 'mouseVec' to reverse the ball direction.

The delete method in the ball class caused an error when removing potted balls.	Changed to 'self.kill()' to properly remove the ball from the game, instead of 'self.delete()'.
---	---

Success criteria that has been met:

1	All elements of the game Pool are drawn to the screen.
2	The player should be able to hit the balls with a cue.
3	The game should be played from a singular camera angle from above.
5	The balls should be deleted after they are potted.
8	When playing 8-ball, there should be 15 balls, each with different colours and markings.

Summary of Stage 2

This stage nicely establishes the core gameplay mechanics of the pool game, including realistic table physics, ball collisions, albeit basic cue interaction. This stage saw issues like clipping and movement inaccuracies were fixed, as well overseeing the addition of friction and restitution. This sets the foundation for implementing UI in next stage much easier.

Stage 3: Cue Mechanics and Basic UI

Now I am going to start implementing basic UI to the game:

First, I will add numbers to each of the balls.

In the constructor method, the following has been updated:

I have added two new attributes to each class, number, and the Boolean value isStriped.

```
# Ball Class
class Ball(pygame.sprite.Sprite): # Defines a class for a ball
    def __init__(self, colour, radius, pos, velocity, mass, number, isStriped): # Constructor Method
        super().__init__()
        self.colour = colour
        self.radius = radius
        self.pos = VEC(pos) # Position Vector
        self.velocity = VEC(velocity) # Velocity vector
        self.mass = mass
        self.number = number # stores the ball number, e.g. the 8 ball
        self.isStriped = isStriped # boolean, stores whether the ball is coloured or striped
```

When creating an instance of the class, I have now changed the balls to have the correct values as shown:

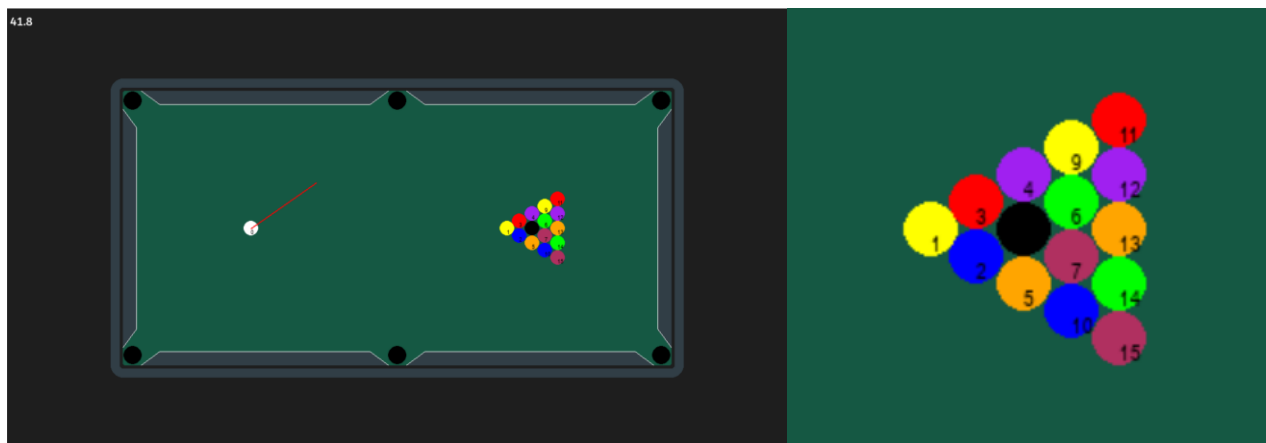
```
# Instantiate each ball
balls = [
    Ball("white", ballRadius, ballStartPos["white0"], (0, 0), 1, 0, False), # Whiteball
    Ball("yellow", ballRadius, ballStartPos["yellow1"], (0, 0), 1, 1, False), # Yellow 1
    Ball("blue", ballRadius, ballStartPos["blue2"], (0, 0), 1, 2, False), # Blue 2
    Ball("red", ballRadius, ballStartPos["red3"], (0, 0), 1, 3, False), # Red 3
    Ball("purple", ballRadius, ballStartPos["purple4"], (0, 0), 1, 4, False), # Purple 4
    Ball("orange", ballRadius, ballStartPos["orange5"], (0, 0), 1, 5, False), # Orange 5
    Ball("green", ballRadius, ballStartPos["green6"], (0, 0), 1, 6, False), # Green 6
    Ball("maroon", ballRadius, ballStartPos["maroon7"], (0, 0), 1, 7, False), # Maroon 7
    Ball("black", ballRadius, ballStartPos["black8"], (0, 0), 1, 8, False), # Black 8
    Ball("yellow", ballRadius, ballStartPos["yellow9"], (0, 0), 1, 9, True), # Yellow 9
    Ball("blue", ballRadius, ballStartPos["blue10"], (0, 0), 1, 10, True), # Blue 10
    Ball("red", ballRadius, ballStartPos["red11"], (0, 0), 1, 11, True), # Red 11
    Ball("purple", ballRadius, ballStartPos["purple12"], (0, 0), 1, 12, True), # Purple 12
    Ball("orange", ballRadius, ballStartPos["orange13"], (0, 0), 1, 13, True), # Orange 13
    Ball("green", ballRadius, ballStartPos["green14"], (0, 0), 1, 14, True), # Green 14
    Ball("maroon", ballRadius, ballStartPos["maroon15"], (0, 0), 1, 15, True), # Maroon 15
]
```

Now, I will use pygame to add number to each ball. This will be done in the draw() method.

```
def draw(self, screen): # Draws the ball to the screen
    pygame.draw.circle(screen, self.colour, self.pos, self.radius) # draws outer circle

    font = pygame.font.SysFont("arial", 10) # specifies font
    text = font.render(str(self.number), True, "black") # renders font
    screen.blit(text, self.pos) # adds font to screen
```

Here the draw() method has been updated and when run, this is the game:



There are two problems:

- The text is not in the correct position.
- The FPS is now very low.

To solve the FPS being low, I have moved the code to initialise the font outside the loop to avoid unnecessary computation.

I have also moved the FPS font outside the game loop:

```
# Fonts
ballFont = pygame.font.SysFont("dubaimedium", 10)
fpsFont = pygame.font.SysFont("dubaimedium", 20)
```

```
def draw(self, screen): # Draws the ball to the screen
    pygame.draw.circle(screen, self.colour, self.pos, self.radius) # draws outer circle

    text = ballFont.render(str(self.number), True, "black") # renders font
    screen.blit(text, self.pos) # adds font to screen
```

Now the FPS is back to 60:



Now I will make the numbers on the balls more aesthetic:

```
def draw(self, screen): # Draws the ball to the screen
    pygame.draw.circle(screen, self.colour, self.pos, self.radius) # draws outer circle

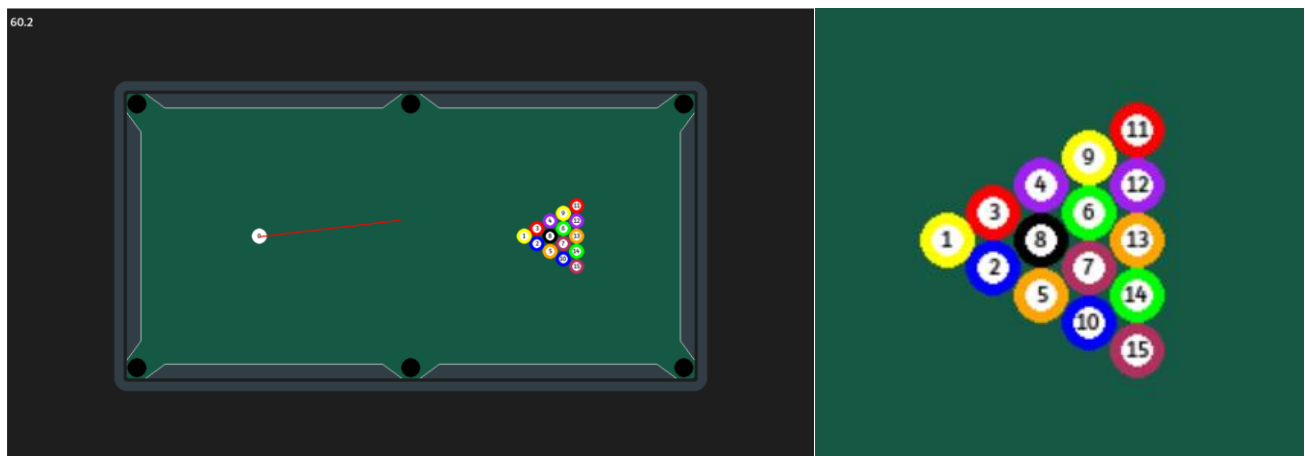
    text = ballFont.render(str(self.number), True, "black") # renders font

    pygame.draw.circle(screen, "white", self.pos, text.get_height()/2.2) # draws inner circle

    fontPos = [ # offsets the font to be central with ball
        self.pos[0] - text.get_width()/2,
        self.pos[1] - text.get_height()/2
    ]

    screen.blit(text, fontPos) # adds font to screen
```

This now gives me this:



I am now going to add more UI elements:

I will add the logo to the right of the screen as seen in the design stage.

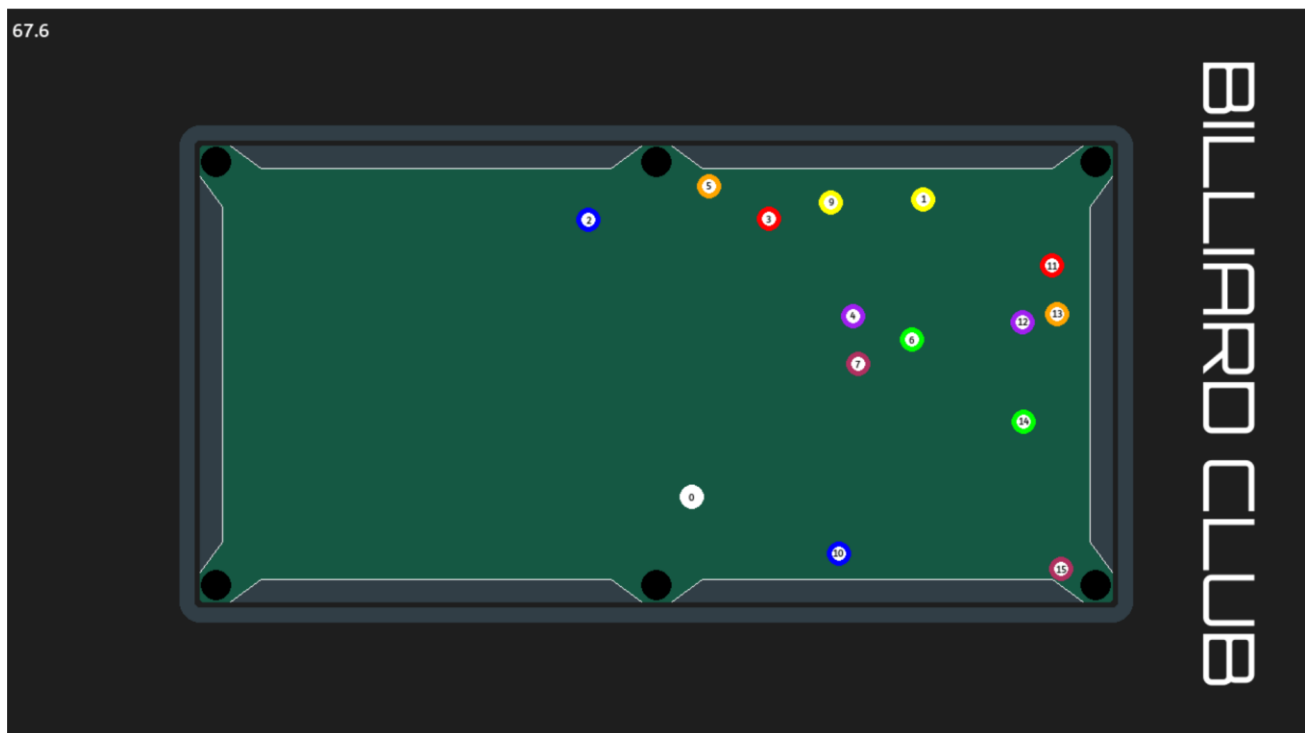
Added the custom font here:

```
# Fonts
ballFont = pygame.font.SysFont("dubaibmedium", int(ballRadius*0.9))
fpsFont = pygame.font.SysFont("dubaibmedium", 20)
logoFont = pygame.font.Font("Project\\other\\Orbitron-VariableFont_wght.ttf", 70)
```

Then in the game loop I have drawn the logo to the screen shown here:

```
# Logo display
logo = logoFont.render(("BILLIARD CLUB"), True, "white")
logo = pygame.transform.rotate(logo, 270)
screen.blit(logo, (resolution[0] - logo.get_width() - 30, centre[1] - logo.get_height()/2))
```

The logo has been added as shown:



Next, I am going to add a slider to allow the user to control the power of the cue:

Since I may use sliders in the menu later in development and the complexity of the slider, I will implement this as a class.

I will create a new file to implement and test this.

Here is the initial code for the new file:

```

import pygame

pygame.init()

# Variables
FPS = 60
resolution = (1000, 600) # Screen dimensions
borderName = "sliders.py" # Displays at the top of the program

# Pygame Functions
pygame.display.set_caption(borderName)
screen = pygame.display.set_mode(resolution) # Sets up the display
clock = pygame.time.Clock()

# Game Loop
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT: # Checks if user closes program
            pygame.quit()
            exit()
    screen.fill((30, 30, 30)) # Fills the screen with a grey colour
    pygame.display.flip() # Updates the display
    clock.tick(FPS) # Caps the frame rate

```

Firstly, I will add a class that will draw a box that the slider will move in:

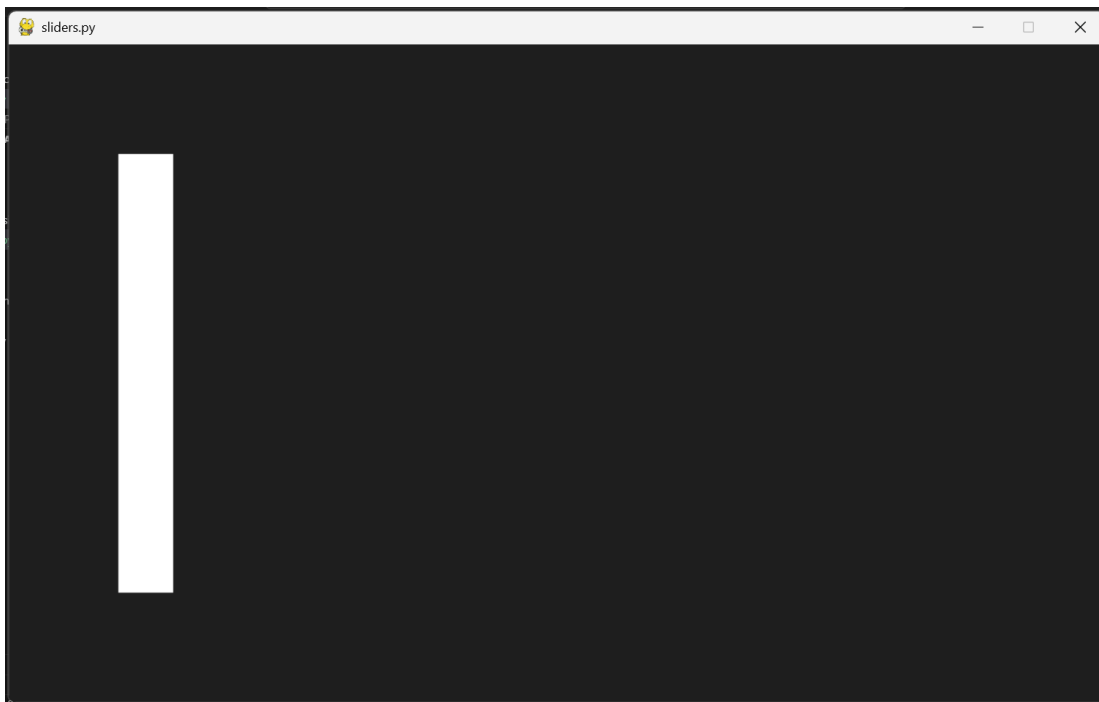
```

class Slider(pygame.sprite.Sprite):
    def __init__(self, colour, position, width, height, startValue):
        super().__init__()
        self.colour = colour # colour of the slider
        self.position = position # (x,y)
        self.width = width # width of the slider
        self.height = height # height of the slider
        self.startValue = startValue # starting value of the slider

    def draw(self, screen): # draw method that draws slider to screen
        pygame.draw.rect(
            screen,
            self.colour,
            pygame.Rect(
                self.position[0],
                self.position[1],
                self.width,
                self.height
            )
        )

slider1 = Slider("white", (100,100), 50, 400, 0) # initialises sprite

```



Now I will draw the actual slider that the user will interact with.

```
def draw(self, screen): # draw method that draws slider to screen
    # Outer Box
    pygame.draw.rect(
        screen,
        self.colour,
        pygame.Rect(
            self.position[0],
            self.position[1],
            self.width,
            self.height
        )
    )

    # Slider
    pygame.draw.circle(
        screen,
        "blue",
        (self.position[0], self.position[1]),
        self.width,
```



The circle is not in the centre therefore this is fixed here:

```
# Slider
pygame.draw.circle(
    screen,
    "blue",
    (self.position[0]+self.width/2, self.position[1]),
    self.width,
)
```

Now I will add an update method that will check if the slider has been clicked

```
# Slider
pygame.draw.circle(
    screen,
    "blue",
    (self.position[0]+self.width/2, self.value * self.height - self.position[1]),
    self.width,
)

def update(self):
    sliderMove = False # set initial case to when user is not interacting with slider
    if event.type == pygame.MOUSEBUTTONDOWN : # when mouse button is clicked
        mousePos = pygame.mouse.get_pos() # get mouse position
        if 0 <= mousePos[0] - self.position[0] <= self.width: # if click is within x bounds
            if 0 <= mousePos[1] - self.position[1] <= self.height: # if click is within y bounds
                distance = mousePos[1]-self.position[1] # finds distance between bottom and mousePos
                self.value = distance/self.height # sets the value of slider to the ratio of distances.
```

This has successfully made a slider but when clicked, the ball does not follow the mouse nor set the ball to the correct position.

This is fixed by changing this method here:

```
def update(self):
    sliderMove = False # set initial case to when user is not interacting with slider
    if event.type == pygame.MOUSEBUTTONDOWN : # when mouse button is clicked
        mousePos = pygame.mouse.get_pos() # get mouse position
        if 0 <= mousePos[0] - self.position[0] <= self.width: # if click is within x bounds
            if 0 <= mousePos[1] - self.position[1] <= self.height: # if click is within y bounds
                distance = mousePos[1]-self.position[1]+self.height/2 # finds distance between bottom and mousePos
                self.value = distance/self.height # sets the value of slider to the ratio of distances.
```

Here it is fixed:



Now I will change the code to make the ball follow the mouse when the click is held.

I will need to add a new attribute to the class:

```
class Slider(pygame.sprite.Sprite):
    def __init__(self, colour, position, width, height, value):
        super().__init__()
        self.colour = colour # colour of the slider
        self.position = position # (x,y)
        self.width = width # width of the slider
        self.height = height # height of the slider
        self.value = value # starting value of the slider
        self.mouseDown = False # for checking if user is interacting with slider
```

And change the following code:

```
def update(self):
    if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # when mouse button is clicked
        self.mouseDown = True

    if event.type == pygame.MOUSEBUTTONUP: # when mouse button is released
        self.mouseDown = False
```

Here I check if the mouse button is pressed, then continually move the slider until the mouse button is released.

This results in weird jolts in the slider because the slider only moves when the mouse is within the bounds.

This I have tried to fix this here:


```
def update(self):

    if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # when mouse button is clicked
        mousePos = pygame.mouse.get_pos() # get mouse position
        if 0 <= mousePos[0] - self.position[0] <= self.width: # if click is within x bounds
            if 0 <= mousePos[1] - self.position[1] <= self.height: # if click is within y bounds
                self.mouseDown = True

    if event.type == pygame.MOUSEBUTTONUP: # when mouse button is released
        self.mouseDown = False

    if self.mouseDown:
        mousePos = pygame.mouse.get_pos() # get mouse position
        distance = mousePos[1]-self.position[1]+self.height/2 # finds distance between bottom and mousePos
        self.value = distance/self.height # sets the value of slider to the ratio of distances.
```

However, this results in this occurring when moving the mouse beyond the bounds shown here:
(flipped to fit on page)



I have now fixed this by changing the value calculations and limiting this ratio to be within 0 and 1:

```
def update(self):

    if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # when mouse button is clicked
        mousePos = pygame.mouse.get_pos() # get mouse position
        if 0 <= mousePos[0] - self.position[0] <= self.width: # if click is within x bounds
            if 0 <= mousePos[1] - self.position[1] <= self.height: # if click is within y bounds
                self.mouseDown = True

    if event.type == pygame.MOUSEBUTTONUP: # when mouse button is released
        self.mouseDown = False

    if self.mouseDown:
        mousePos = pygame.mouse.get_pos() # get mouse position
        distance = mousePos[1]-self.position[1] # finds distance between bottom and mousePos
        self.value = distance/self.height # sets the value of slider to the ratio of distances.

        self.value = max(0, min(1, self.value))
```

Now the slider should output the correct value depending on the position of the blue ball:

```
def returnValue(self):
    print(self.value)
```

0.275



This value will be a multiple for the velocity of the ball, but I need the top of the slider to be the maximum and the bottom to be the minimum.

This is fixed here by just doing 1-value:

```
def returnValue(self):  
    return (1-self.value)
```

0.6074999999999999

Sometimes, this value is returned and therefore I will round the number:

```
def returnValue(self):  
    return round(1-self.value, 3)
```

Now the slider is done, I will change the aesthetics and implement it into the main game file.

Final class code:

```

class Slider(pygame.sprite.Sprite):
    def __init__(self, outerColour, sliderColour, position, width, height, value):
        super().__init__()
        self.outerColour = outerColour # colour of the outer section
        self.sliderColour = sliderColour # colour of the slider
        self.position = position # (x,y)
        self.width = width # width of the slider
        self.height = height # height of the slider
        self.value = value # starting value of the slider
        self.mouseDown = False # for checking if user is interacting with slider

    def draw(self, screen): # draw method that draws slider to screen
        # Outer Box
        pygame.draw.rect(
            screen,
            self.outerColour,
            pygame.Rect(
                self.position[0],
                self.position[1],
                self.width,
                self.height
            ),
            3,
            self.width
        )
        # Slider
        pygame.draw.circle(
            screen,
            self.sliderColour,
            (self.position[0]+self.width/2, self.position[1] + self.value * self.height),
            self.width,
        )

```

```

def update(self): # updates slider position
    if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # when left mouse button clicked
        mousePos = pygame.mouse.get_pos() # get mouse position
        if 0 <= mousePos[0] - self.position[0] <= self.width: # if click is within x bounds
            if 0 <= mousePos[1] - self.position[1] <= self.height: # if click is within y bounds
                self.mouseDown = True
    if event.type == pygame.MOUSEBUTTONUP: # when mouse button is released
        self.mouseDown = False
    if self.mouseDown:
        mousePos = pygame.mouse.get_pos() # get mouse position
        distance = mousePos[1]-self.position[1] # finds distance between bottom and mousePos
        self.value = distance/self.height # sets the value of slider to the ratio of distances.
        self.value = max(0, min(1, self.value))

def returnValue(self): # returns slider position value
    return round(1-self.value, 3)

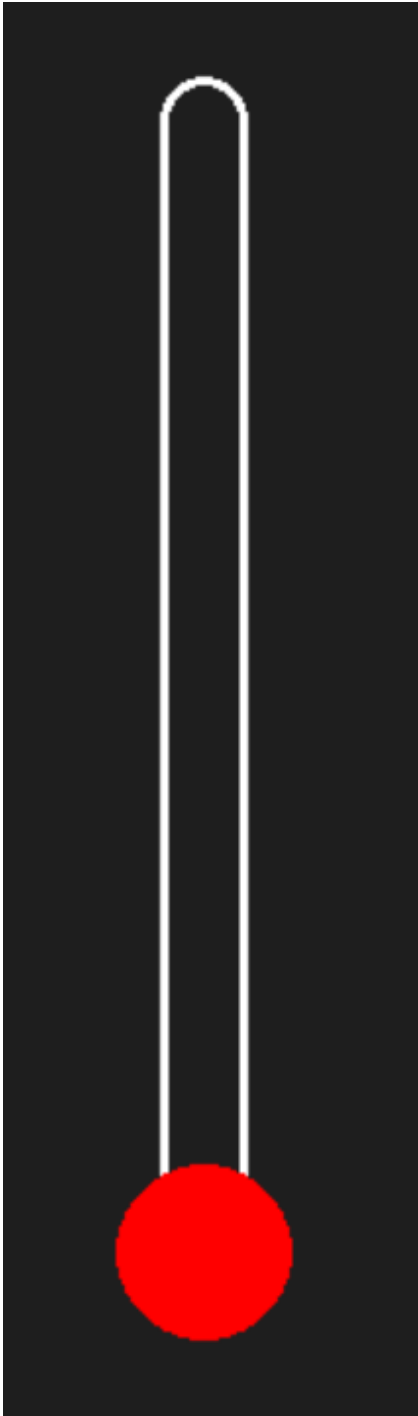
```

```

slider1 = Slider("white", "red", (100,100), 30, 400, 1) # initialises sprite

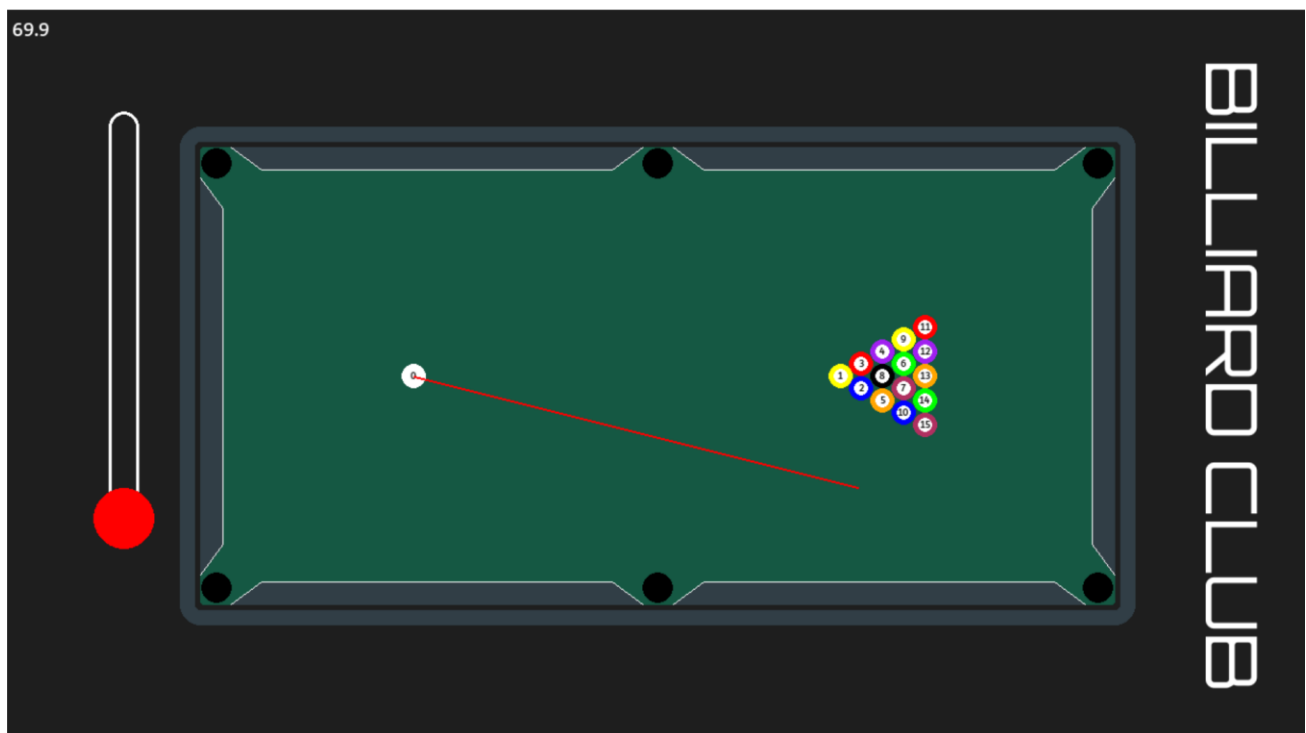
```

Produces a very good-looking slider:



Now I will implement this into the main game file:

```
ballPowerSlider = Slider("white", "red", (100,100), 30, 400, 1) # initialises slider sprite
```

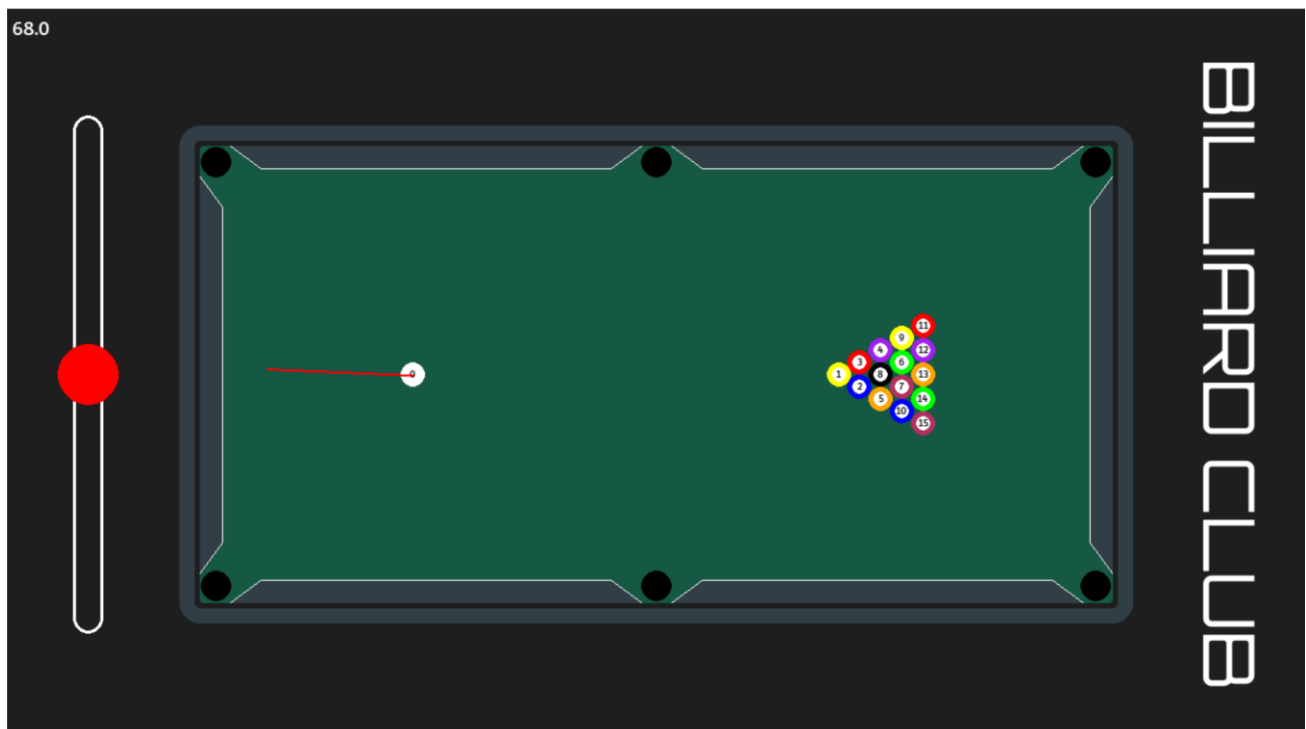


There are a few problems:

- The slider is not centred
- I cannot interact with the slider without triggering the ball

Here I have centred the slider:

```
ballPowerSlider = Slider("white", "red", (tableCorners[0][0]/2 - 30, tableCorners[0][1]-30), 30, tableDimensions[1]+60, 0.5) # initialises slider sprite
```



Now I will change the code so that when the user clicks the pool table only, the ball will be projected in that direction. This will allow the user to change the speed of the ball with the slider before performing a shot.

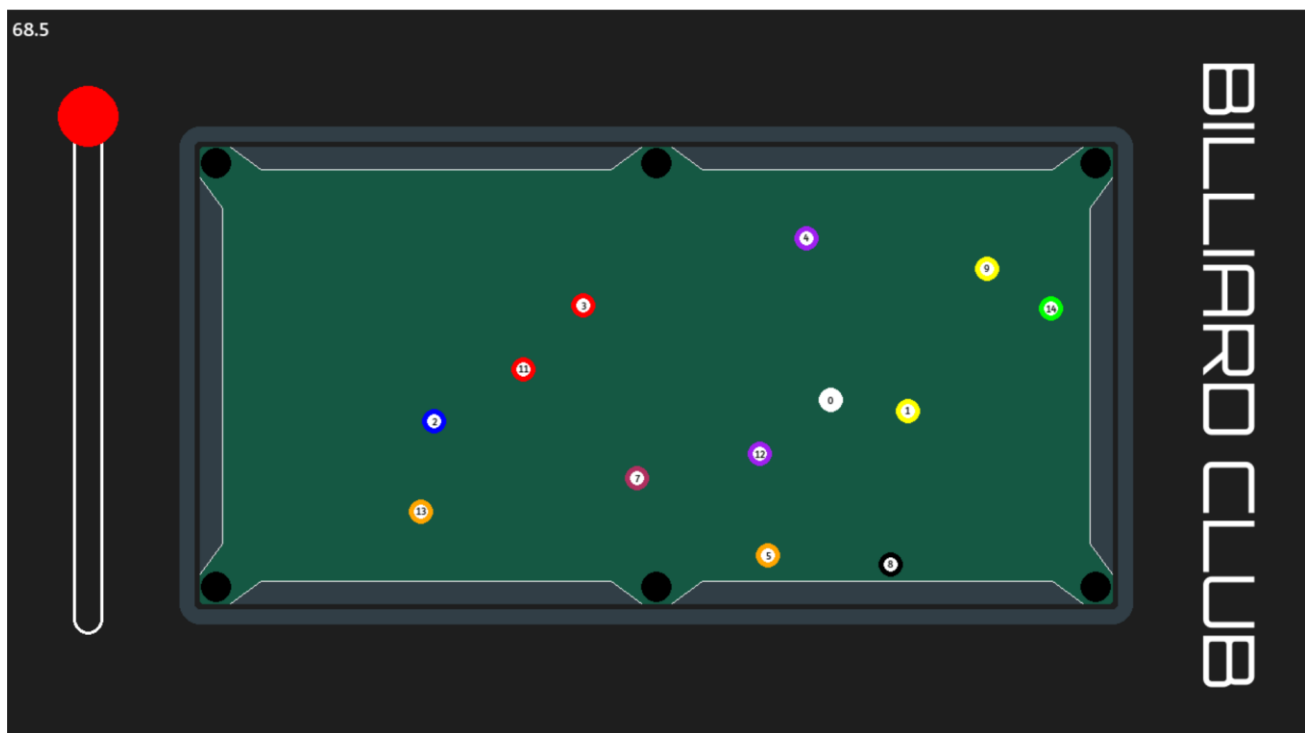
```
if not ballMoving: # when all the balls are still
    pygame.draw.line(screen, "red", balls[0].pos, pygame.mouse.get_pos(), 2) # draw cue
    if mouseDown: # when clicked
        mousePos = pygame.mouse.get_pos() # get mouse position
        if tableCorners[0][0] <= mousePos[0] <= tableCorners[1][0] \
            and tableCorners[0][1] <= mousePos[1] <= tableCorners[2][1]: # if mouse is on the pool table
            mouseVec = VEC(balls[0].pos[0]-mousePos[0], balls[0].pos[1]-mousePos[1]) # vector of line
            balls[0].velocity = mouseVec.normalize()*20 # normalise vector and give ball a magnitude
```

This has been done so that I can test the slider further and later a new cue system will be implemented.

With little change, when clicked, the ball will be given a speed proportional to the slider:

```
if not ballMoving: # when all the balls are still
    pygame.draw.line(screen, "red", balls[0].pos, pygame.mouse.get_pos(), 2) # draw cue
    if mouseDown: # when clicked
        mousePos = pygame.mouse.get_pos() # get mouse position
        if tableCorners[0][0] <= mousePos[0] <= tableCorners[1][0] \
            and tableCorners[0][1] <= mousePos[1] <= tableCorners[2][1]: # if mouse is on the pool table
            mouseVec = VEC(balls[0].pos[0]-mousePos[0], balls[0].pos[1]-mousePos[1]) # vector of line
            balls[0].velocity = mouseVec.normalize()*20*ballPowerSlider.returnValue() # normalise vector and give ball a magnitude
```

This works as shown here:



Now I need to add UI to tell the user which balls are left, and which need to be potted.

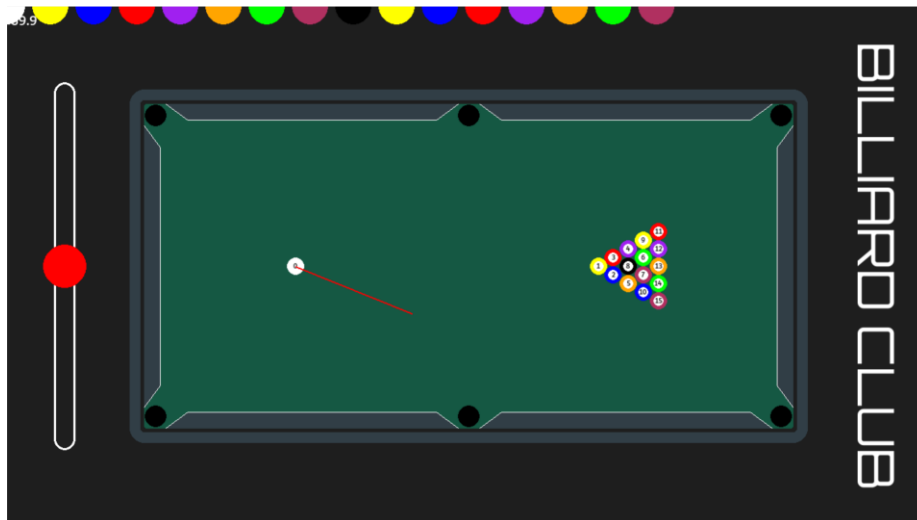
I have preliminarily set up a viewing system here:

```

increment = 0
for ball in ballGroup:
    pygame.draw.circle(screen, ball.colour, (increment, 0), 25)
    increment += 60

```

This code produces this result:



Problem: the balls are in the wrong position.

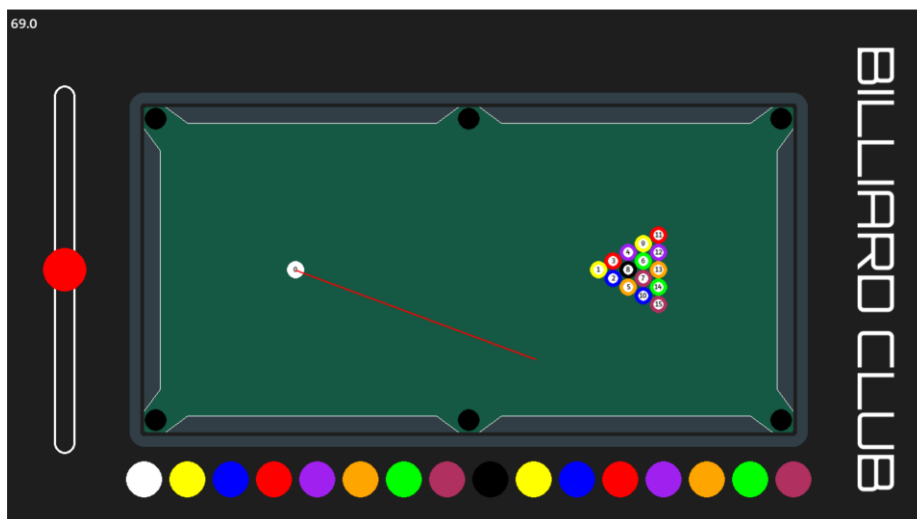
This is fixed here by changing the position of the balls:

```

increment = 0 # adds a value on each time so the balls are not drawn ontop of each other
for ball in ballGroup: # draw each ball
    pygame.draw.circle(screen, ball.colour, (tableCorners[0][0] + increment, tableCorners[2][1]+65), 25)
    increment += 60

```

This code produces this result:



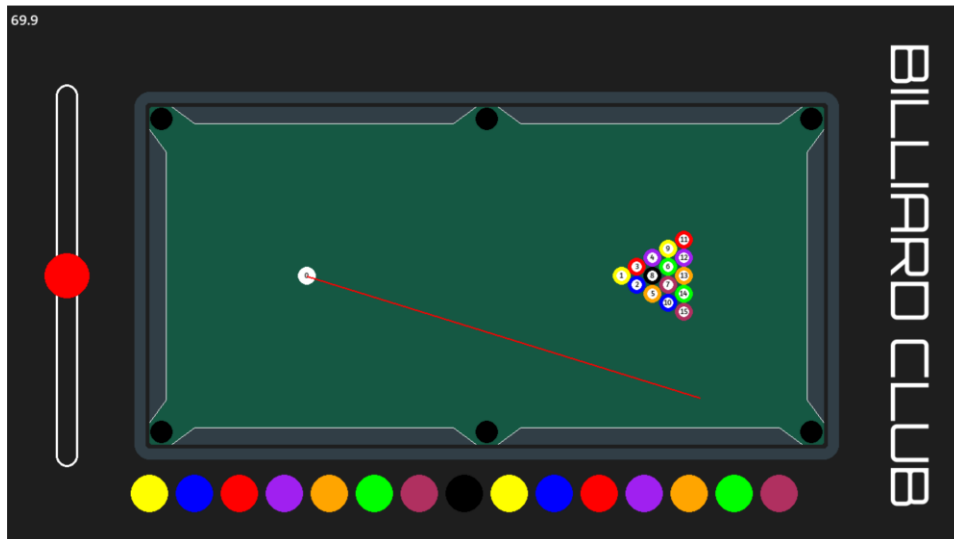
However, the white ball is being drawn, which needs to be removed:

```

increment = 0 # adds a value on each time so the balls are not drawn ontop of each other
for ball in ballGroup: # draw each ball
    if ball.colour != "white": # draws every ball but the white ball
        pygame.draw.circle(screen, ball.colour, (tableCorners[0][0] + increment, tableCorners[2][1]+65), 25)
        increment += 60

```

Here I check if any ball is white, and if the ball is white, the computer will skip it.

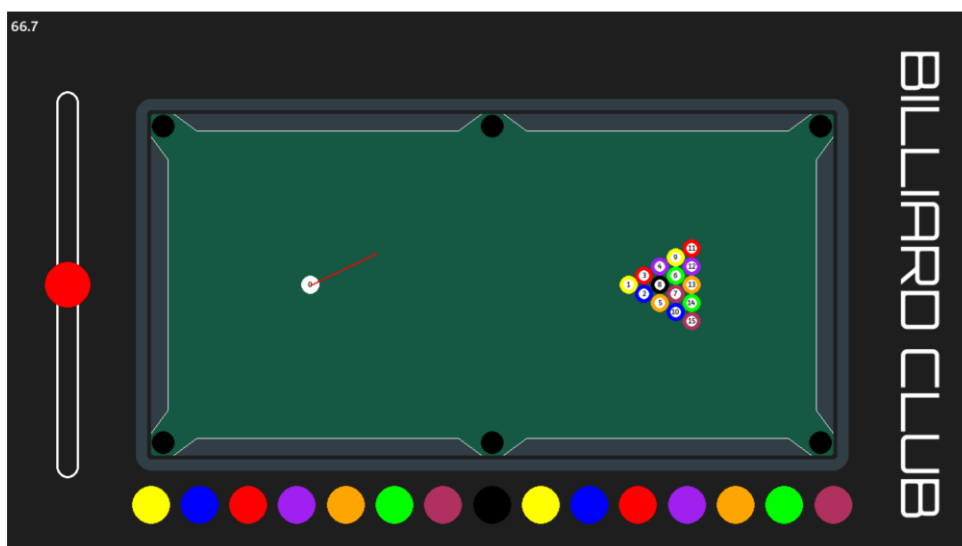


Here I centre the balls:

```

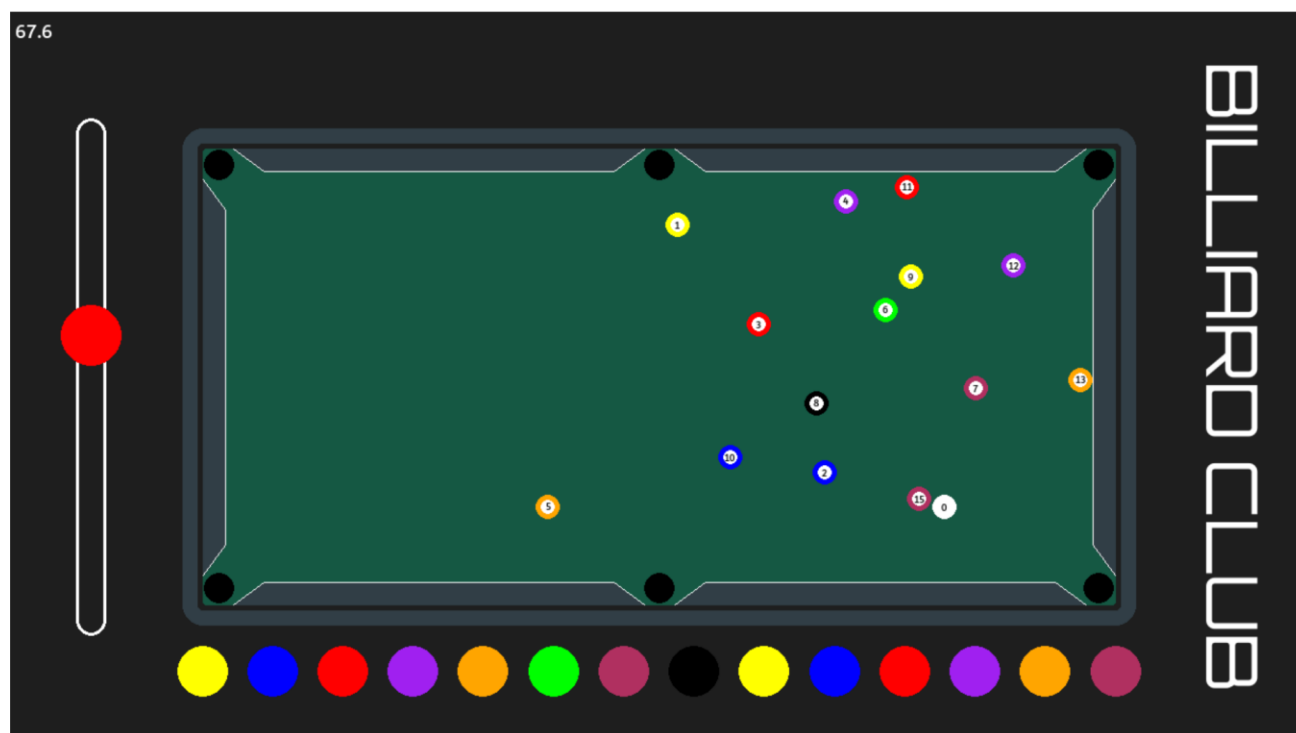
# Draw 15 balls on screen for UI (NOT ON TABLE)
increment = 0 # adds a value on each time so the balls are not drawn on top of each other
distanceBetweenBalls = tableDimensions[0]/(len(ballGroup)-2)
for ball in ballGroup: # draw each ball
    if ball.colour != "white": # draws every ball but the white ball
        pygame.draw.circle(screen, ball.colour, (tableCorners[0][0] + increment, tableCorners[2][1]+65), 25)
        increment += distanceBetweenBalls

```



This so far works well, but when a ball is potted, that ball too is removed from the list, which changes the position of the balls immediately.

This is shown here:



One of the green balls has been potted and therefore the green ball is removed and the ball spacing is increased.

What I want is each ball has a position, and that position does not change, but only deletes when potted.

The better way to do this is to have each ball position to stay the same but have a graphic that overlays each ball to tell the user that it has been potted.

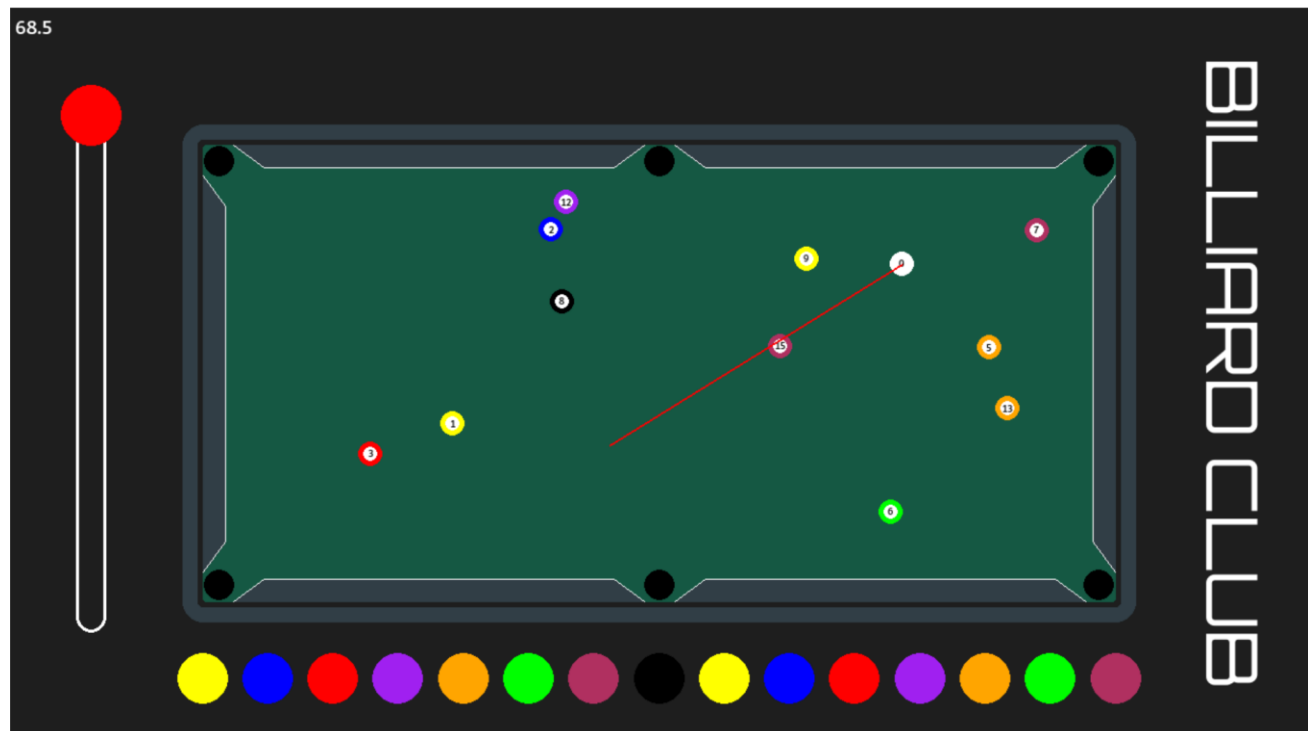
I update this by creating a master tuple that stores the values of each ball:

```
# Ball info
ballInfo = [
    # COLOUR, NUMBER, STRIPED
    ("white", 0, False),
    ("yellow", 1, False),
    ("blue", 2, False),
    ("red", 3, False),
    ("purple", 4, False),
    ("orange", 5, False),
    ("green", 6, False),
    ("maroon", 7, False),
    ("black", 8, False),
    ("yellow", 9, True),
    ("blue", 10, True),
    ("red", 11, True),
    ("purple", 12, True),
    ("orange", 13, True),
    ("green", 14, True),
    ("maroon", 15, True),
]
```

I then change the loop via my loop shown here:

```
# Draw 15 balls on screen for UI (NOT ON TABLE)
increment = 0 # adds a value on each time so the balls are not drawn on top of each other
distanceBetweenBalls = tableDimensions[0]/(len(ballInfo)-2)
for i in range(len(ballInfo)-1): # loops through each ball except white ball
    pygame.draw.circle(screen, ballInfo[i+1][0], (tableCorners[0][0] + increment, tableCorners[2][1]+75), 25)
    increment += distanceBetweenBalls
```

All the balls now are displayed regardless of if they are potted:



Now I will add the numbers and a graphic to them:

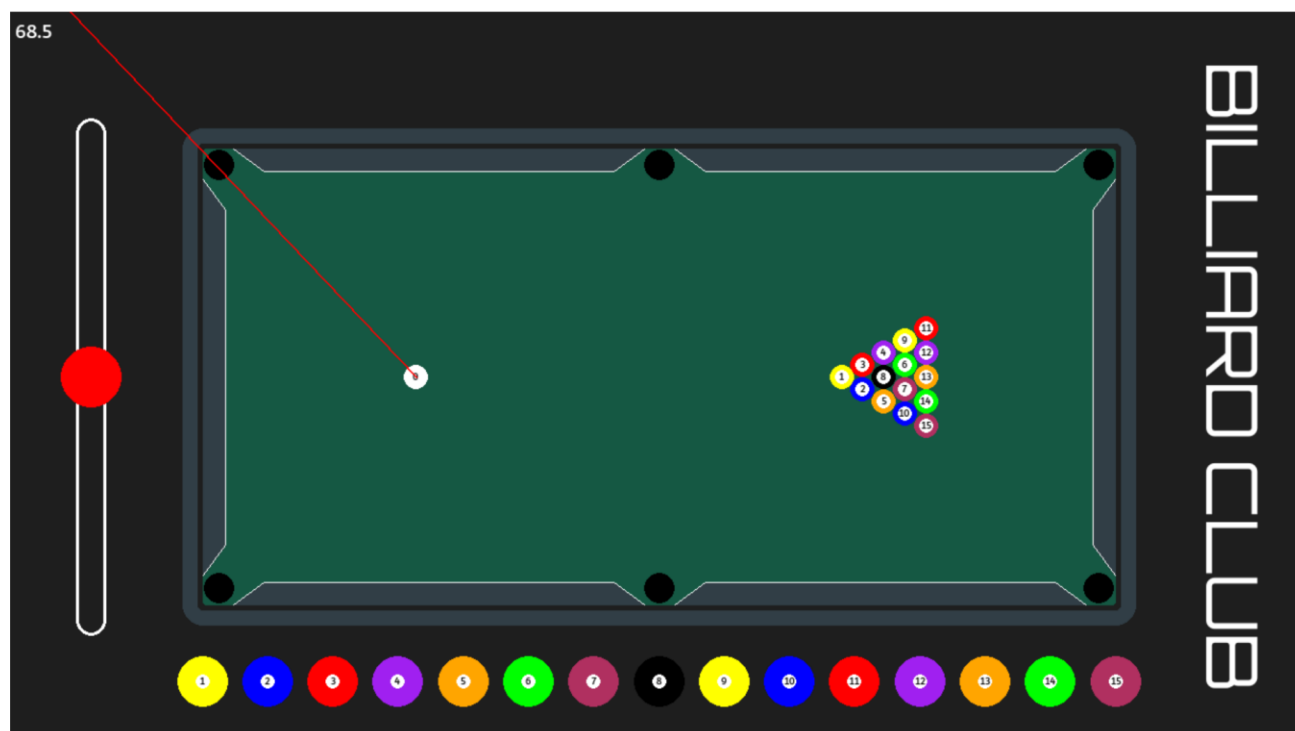
I have reused code that is used in the ball class with the draw() method:

```
# Draw 15 balls on screen for UI (NOT ON TABLE)
increment = 0 # adds a value on each time so the balls are not drawn on top of each other
distanceBetweenBalls = tableDimensions[0]/(len(ballInfo)-2)
for i in range(len(ballInfo)-1): # loops through each ball except white ball
    circleCentre = (tableCorners[0][0] + increment, tableCorners[2][1]+75) # coordinates of each circle

    pygame.draw.circle(screen, ballInfo[i+1][0], circleCentre, 25)
    text = ballFont.render(str(ballInfo[i+1][1]), True, "black") # renders font
    pygame.draw.circle(screen, "white", circleCentre, text.get_height()/2.2) # draws inner circle
    fontPos = [ # offsets the font to be central with ball
        circleCentre[0] - text.get_width()/2,
        circleCentre[1] - text.get_height()/2
    ]
    screen.blit(text, fontPos) # adds font to screen

    increment += distanceBetweenBalls
```

This results in the following solution:



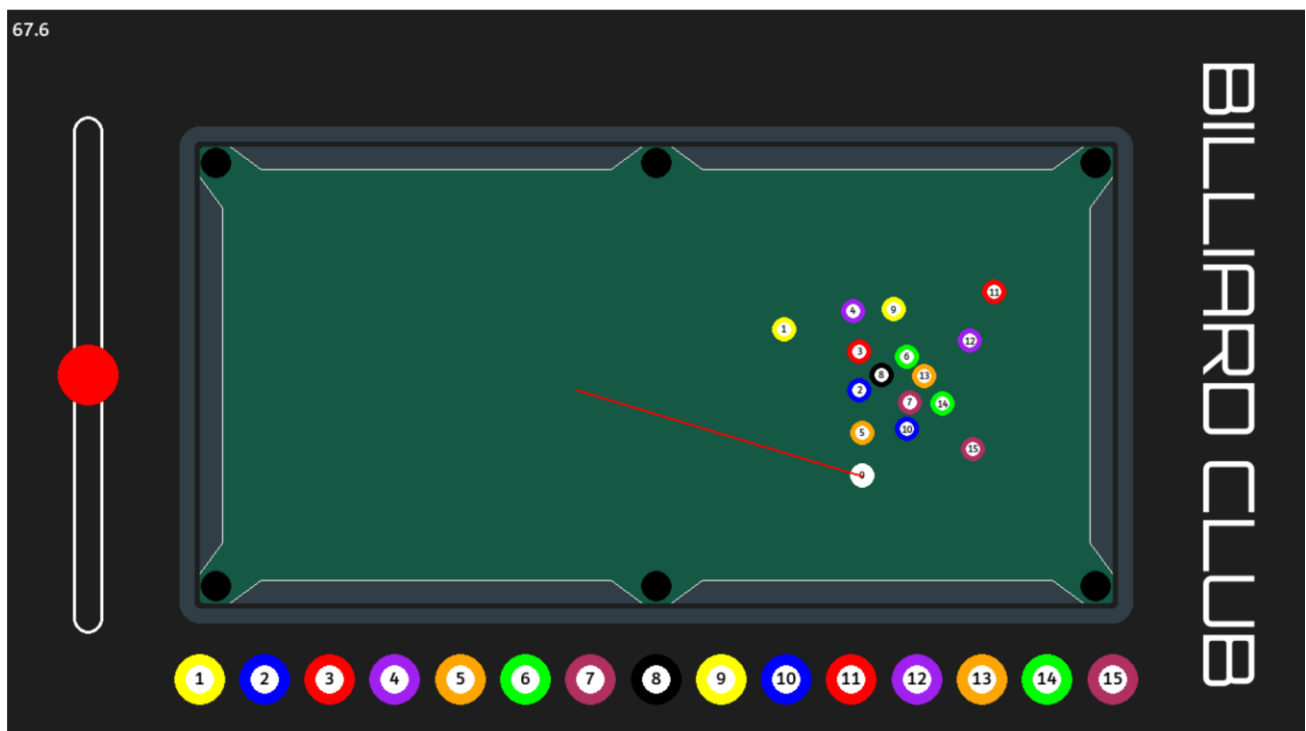
I need to define a new font that will be larger. This is done here and implemented here:

```
ballUIFont = pygame.font.SysFont("dubaimedium", 18)
```

```
# Draw 15 balls on screen for UI (NOT ON TABLE)
increment = 0 # adds a value on each time so the balls are not drawn on top of each other
distanceBetweenBalls = tableDimensions[0]/(len(ballInfo)-2)
for i in range(len(ballInfo)-1): # loops through each ball except white ball
    circleCentre = (tableCorners[0][0] + increment, tableCorners[2][1]+75) # coordinates of each circle

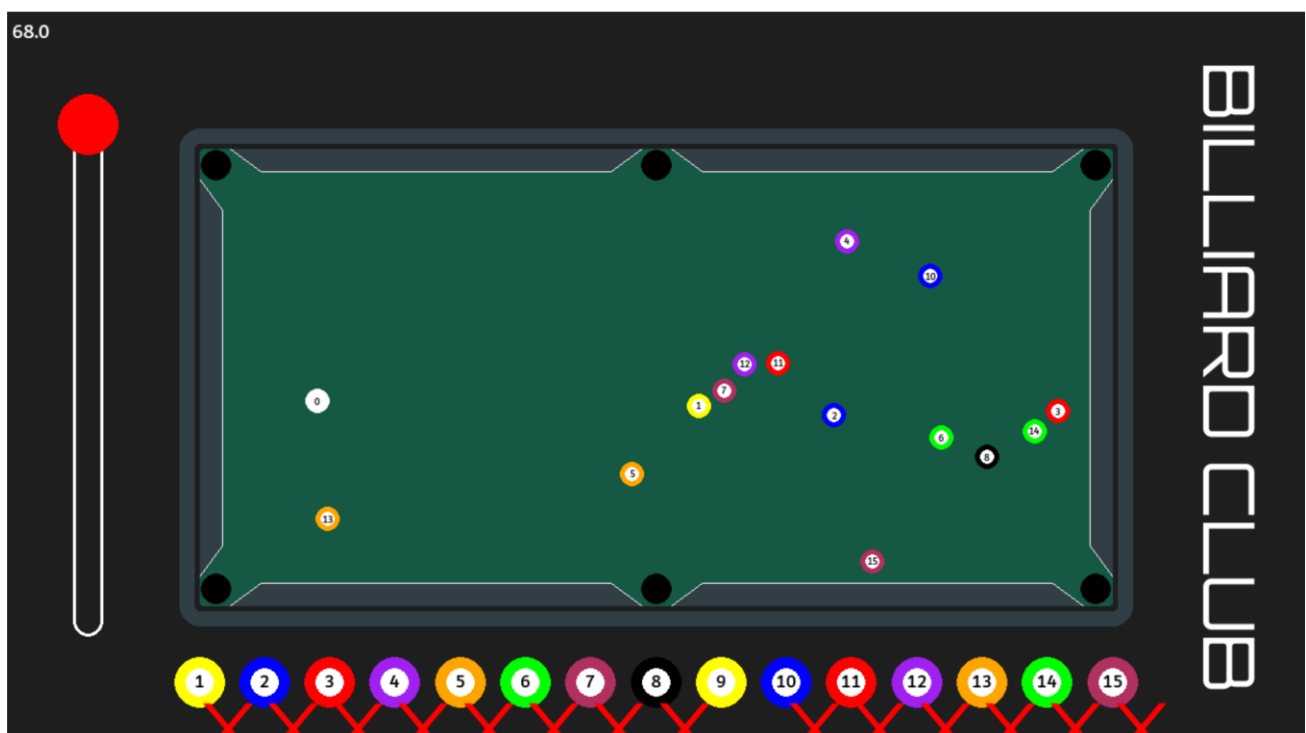
    pygame.draw.circle(screen, ballInfo[i+1][0], circleCentre, 25)
    text = ballUIFont.render(str(ballInfo[i+1][1]), True, "black") # renders font
    pygame.draw.circle(screen, "white", circleCentre, text.get_height()/2.2) # draws inner circle
    fontPos = [ # offsets the font to be central with ball
        circleCentre[0] - text.get_width()/2,
        circleCentre[1] - text.get_height()/2
    ]
    screen.blit(text, fontPos) # adds font to screen

    increment += distanceBetweenBalls
```



Now each time a ball is potted I will draw an X on the ball to show the user it has been potted:

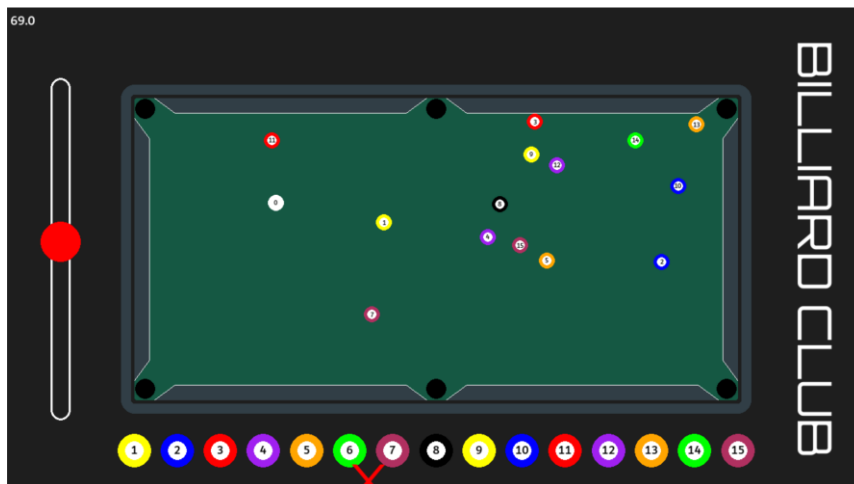
```
for ball in ballGroup:
    if ball.number == ballInfo[i+1][1]:
        screen.blit(ballPottedFont.render("X", True, "red"), circleCentre)
```



There are two problems here, the first is that the "X" is in the incorrect position and the second is that only when a ball is potted does the X disappear. This must be inverted.

One problem has been fixed with the following code:

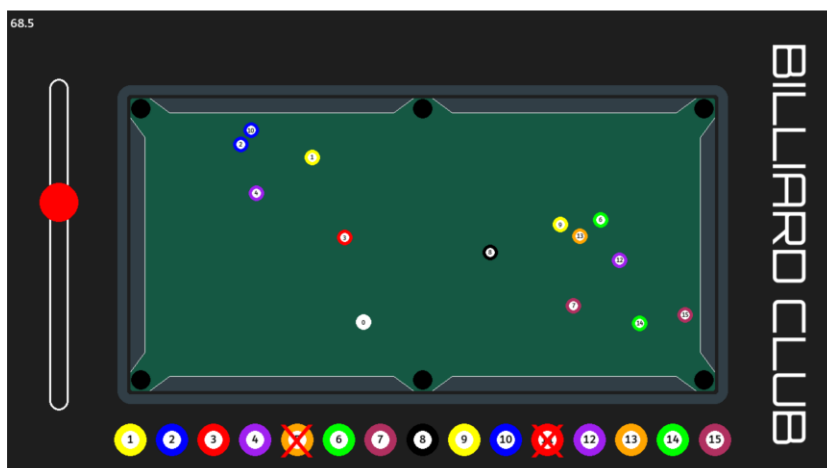
```
potted = True # set initial case to True
for ball in ballGroup:
    if ball.number == ballInfo[i+1][1]: # If the ball still exists
        potted = False # do not draw X
if potted:
    screen.blit(ballPottedFont.render("X",True,"red"),circleCentre) # draw x if ball does not exist
```



The X is then centred with this code:

```
potted = True # set initial case to True
for ball in ballGroup:
    if ball.number == ballInfo[i+1][1]: # If the ball still exists
        potted = False # do not draw X
if potted: # if ball does not exist on table
    xFont = ballPottedFont.render("X",True,"red") # render font
    xFontPos = [circleCentre[0] - xFont.get_width()/2, circleCentre[1] - xFont.get_height()/2] # centre font
    screen.blit(xFont, xFontPos) # draw ball
```

While playing the game: (Xs are shown in the correct position)



Now I will work on the cue mechanics:

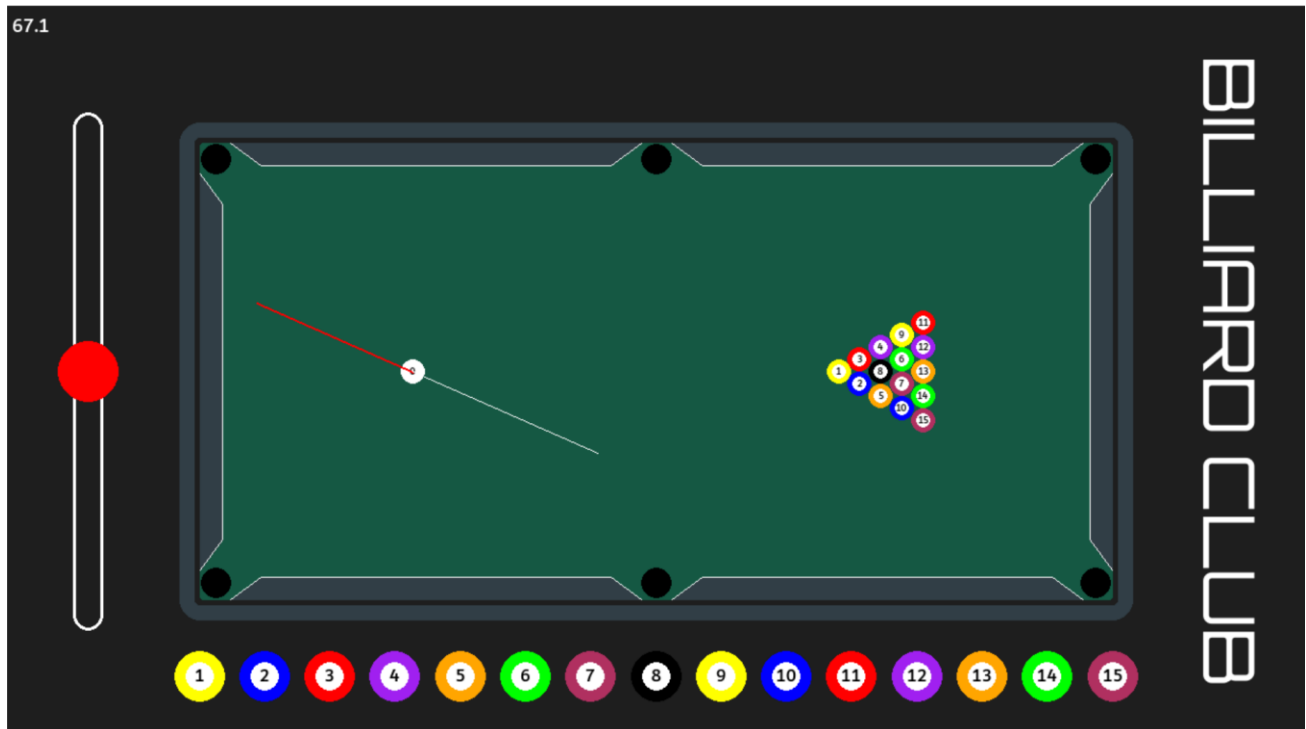
I will draw a line from the white ball to the direction that the user will project the ball to.

In the game loop I have changed the code to the following:

```
# Cue Functions and Draws Cue
if not ballMoving: # when all the balls are still
    mousePos = pygame.mouse.get_pos() # get mouse position
    cueVec = VEC(balls[0].pos[0]-mousePos[0],balls[0].pos[1]-mousePos[1]).normalize() # vector of line
    pygame.draw.line(screen, "red", balls[0].pos, pygame.mouse.get_pos(), 2) # draw cue
    pygame.draw.line(screen, "white", balls[0].pos, balls[0].pos + cueVec*200) # draw projection

    if mouseDown: # when clicked
        if tableCorners[0][0] <= mousePos[0] <= tableCorners[1][0] \
            and tableCorners[0][1] <= mousePos[1] <= tableCorners[2][1]: # if mouse is on the pool table
            balls[0].velocity = cueVec*20*ballPowerSlider.returnValue() # give ball a velocity
```

Which produces this:



This successfully draws a small projection of the ball.

However, I want this to extent further on and be able to reflect off walls accurately and off balls as well.

For now, I will work on projecting off walls.

This means I must create an algorithm that will find the point of intersection of the projection line of the ball and the table rails.

To make my code more maintainable, I will introduce a new function that will input the coordinates of the ball and the vector of the ball projection as well as the rail coordinates and output the coordinates of the point on the table rails that the ball will hit.

```
def calcPOI(railStart, railEnd, ballPos, projectionVec):
    # calculates the projected point of intersection between the ball and the rails

    railVec = VEC(railEnd) - VEC(railStart) # Vector of the rail edge
    railStartToBallVec = VEC(ballPos) - VEC(railStart) # vector of the start of rail to the ball
```

Now I solved a parametric equation to find the point of intersection using vector mechanics:

This involves taking the direction vectors of the ball projection and the table rails and finding the intersection coordinates between the infinite lines.

```
def calcPOI(railStart, railEnd, ballPos, projectionVec):
    # calculates the projected point of intersection between the ball and the rails
    railVec = VEC(railEnd) - VEC(railStart) # Vector of the rail edge
    railStartToBallVec = VEC(ballPos) - VEC(railStart) # vector of the start of rail to the ball

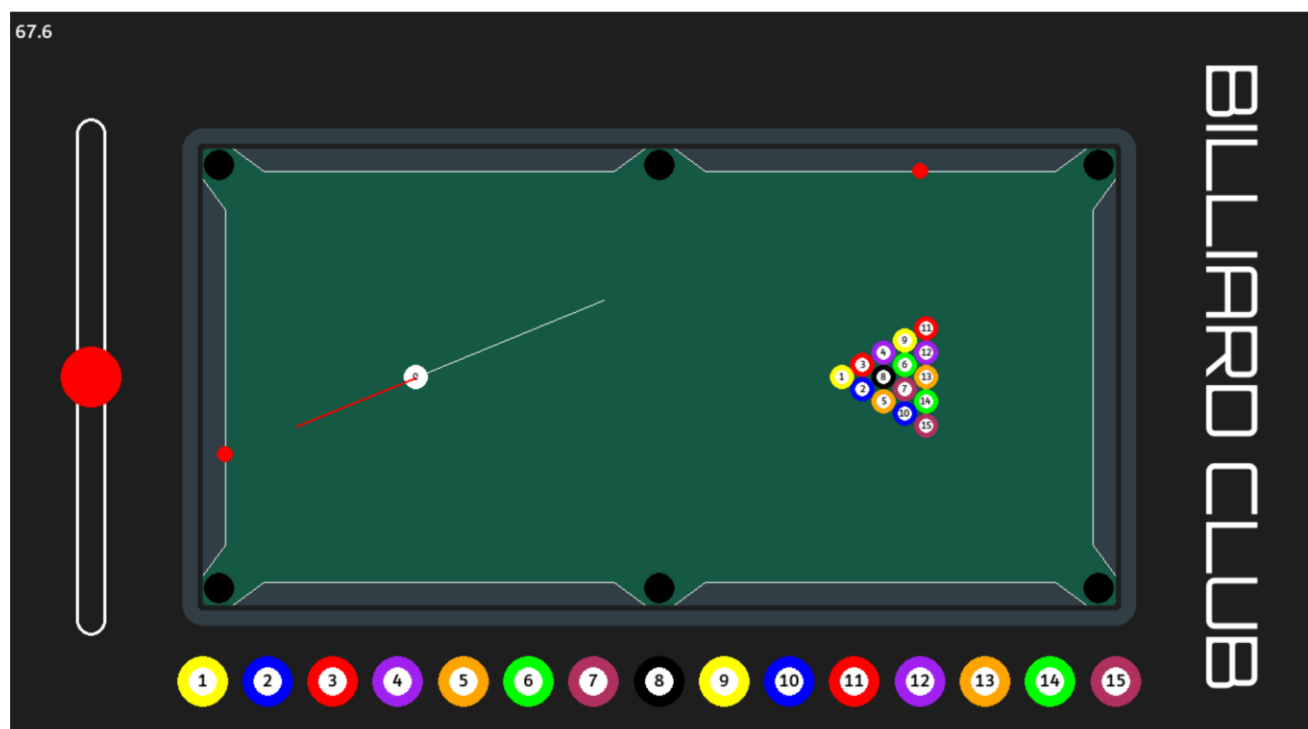
    # solve the intersection between the two vectors parametrically (i.e t should be within 0 and 1 if colliding)
    t = VEC.cross(railStartToBallVec, projectionVec) / VEC.cross(railVec, projectionVec)

    # Check if intersection is within the line segment
    if 0 <= t <= 1: # if t in [0, 1] means it lies on the segment
        return (railStart[0] + t * railVec[0], railStart[1] + t * railVec[1]) # return coordinates of point

# Cue Functions and Draws Cue
if not ballMoving: # when all the balls are still
    mousePos = pygame.mouse.get_pos() # get mouse position
    cueVec = (VEC(balls[0].pos) - VEC(mousePos)).normalize() # vector of line
    for i in tableRails: # loops through every rail (every line segment)
        for j in range(3):
            lineStart = tableRails[i][j] # Coordinates of the Line Start and End
            lineEnd = tableRails[i][j+1] # e.g. [ [x1,y1] , [x2,y2] ]
            intersection = calcPOI(lineStart, lineEnd, balls[0].pos, cueVec) # calculate coordinates of intersection
            if intersection != None: # if there is a coordinate of intersection, draw a circle
                pygame.draw.circle(screen, "red", intersection, 8)
```

With this code, I have set the computer to draw the points of intersection with a red dot.

This produces this result:



The code has worked but, an unforeseen error occurs when there are no points of intersection, or the case when the two vectors are completely parallel. In the case where the vectors are parallel, there is a division by zero in the calculation of the parameter, t .

The error is shown here from the console:

```
t = VEC.cross(railStartToBallVec, projectionVec) / VEC.cross(railVec, projectionVec)
~~~~~^~~~~~
ZeroDivisionError: float division by zero
```

Therefore, I have separated the calculation of t out into a fraction: a numerator and a denominator.

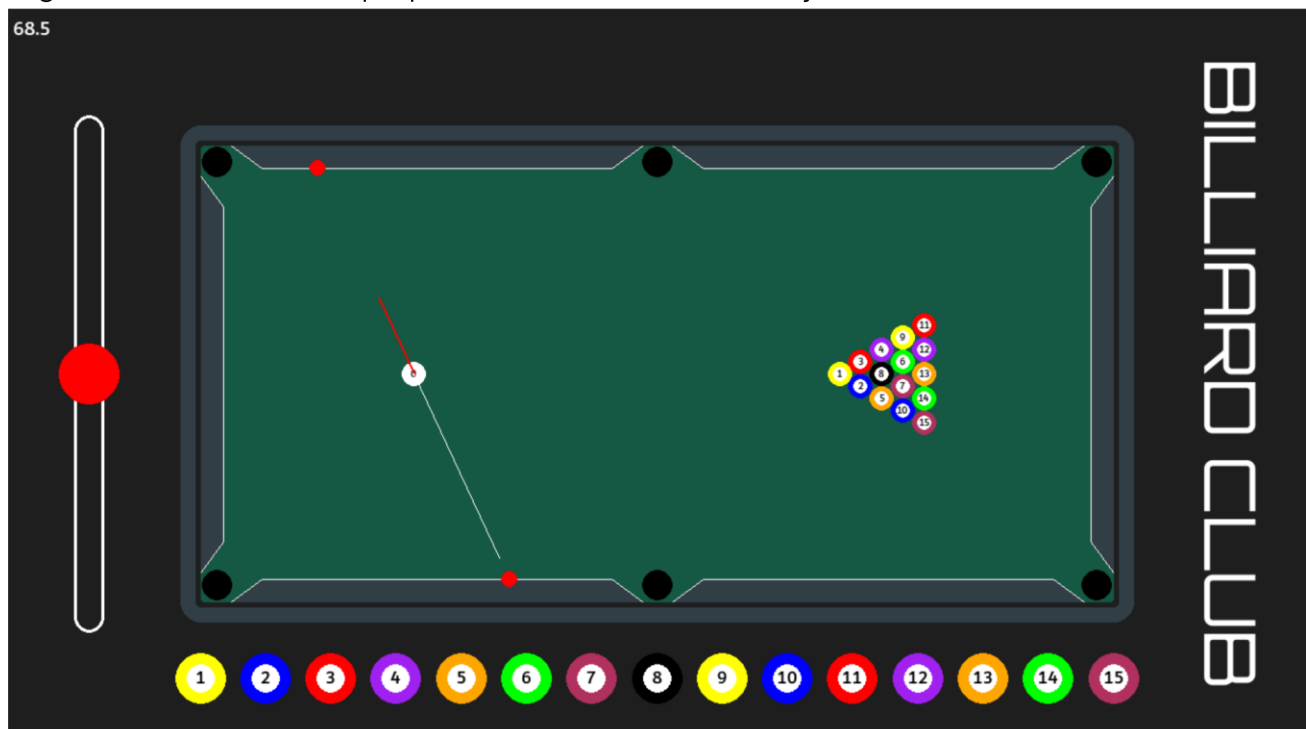
```
def calcPOI(railStart, railEnd, ballPos, projectionVec):
    # calculates the projected point of intersection between the ball and the rails
    railVec = VEC(railEnd) - VEC(railStart) # Vector of the rail edge
    railStartToBallVec = VEC(ballPos) - VEC(railStart) # vector of the start of rail to the ball

    denominator = VEC.cross(railVec, projectionVec) # separates calculation to avoid div by 0

    # solve the intersection between the two vectors parametrically (i.e t should be within 0 and 1 if colliding)
    t = VEC.cross(railStartToBallVec, projectionVec) / denominator

    # Check if intersection is within the line segment
    if 0 <= t <= 1: # if t in [0, 1] means it lies on the segment
        return (railStart[0] + t * railVec[0], railStart[1] + t * railVec[1]) # return coordinates of point
```

The other problem is that since I have used vector calculations with an infinite line and one line segment, there will be multiple points of intersection with many rails. This is shown here:



To fix this, instead of calculating the intersection between two infinite lines and checking if they lie on the line, I will calculate the intersection between two-line segments. Meaning one red dot should disappear (in the diagram above).

Here I have introduced another parameter, s , which is used to check if the rail lies on the projection vector.

This means that if both s and t are within $[0,1]$ (interval notation) then the correct collision has been calculated.

```
def calcPOI(railStart, railEnd, ballPos, projectionVec):
    # calculates the projected point of intersection between the ball and the rails
    railVec = VEC(railEnd) - VEC(railStart) # Vector of the rail edge
    railStartToBallVec = VEC(ballPos) - VEC(railStart) # vector of the start of rail to the ball

    # since projection vector is normalised, the magnitude must increase to become the second line segment
    projectionVec = projectionVec * 100000

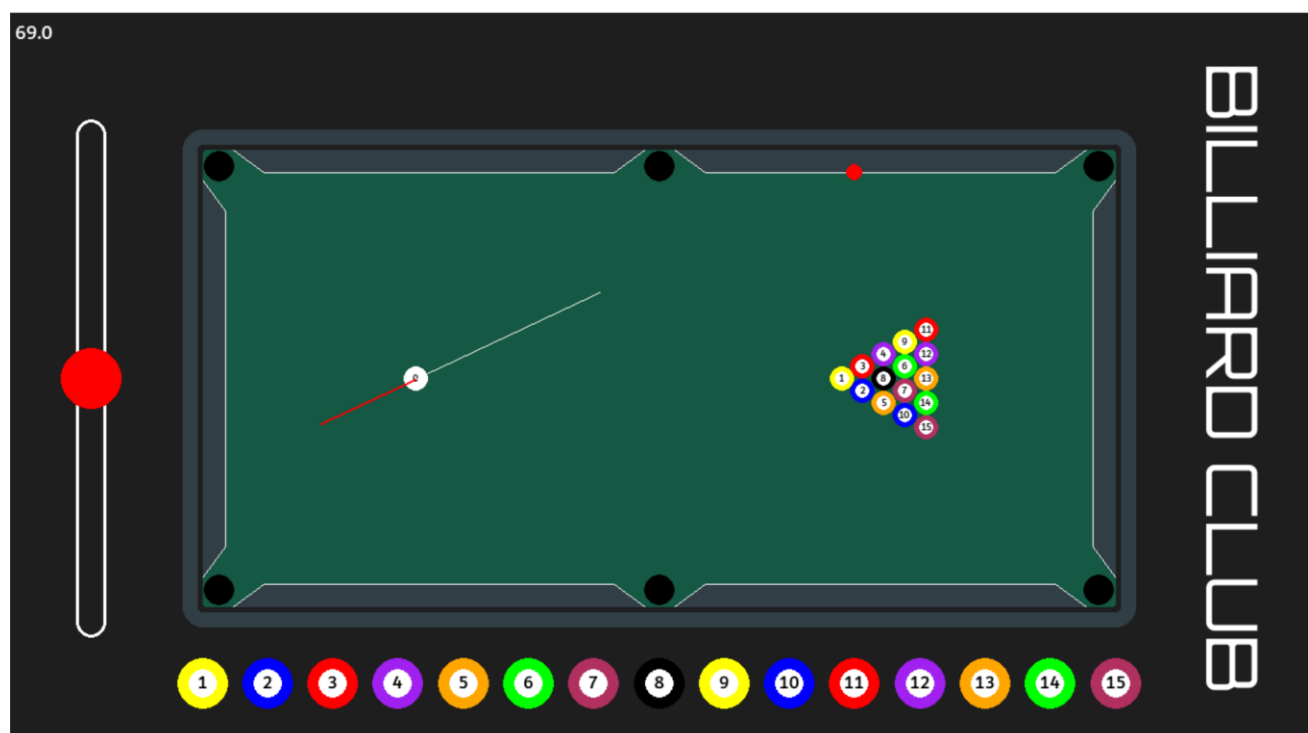
    denominator = VEC.cross(railVec, projectionVec) # separates calculation to avoid div by 0

    if denominator != 0:
        # solve the intersection between the two vectors parametrically
        # (i.e t and s should be within 0 and 1 if line segments are colliding)
        t = VEC.cross(railStartToBallVec, projectionVec) / denominator
        s = VEC.cross(railStartToBallVec, railVec) / denominator

        # Check if intersection is within the two line segments
        if 0 <= t <= 1 and 0 <= s <= 1: # if t and s in [0, 1] means line segments collide
            return (railStart[0] + t * railVec[0], railStart[1] + t * railVec[1]) # return coordinates of point

    return None
```

This works and produces this result:

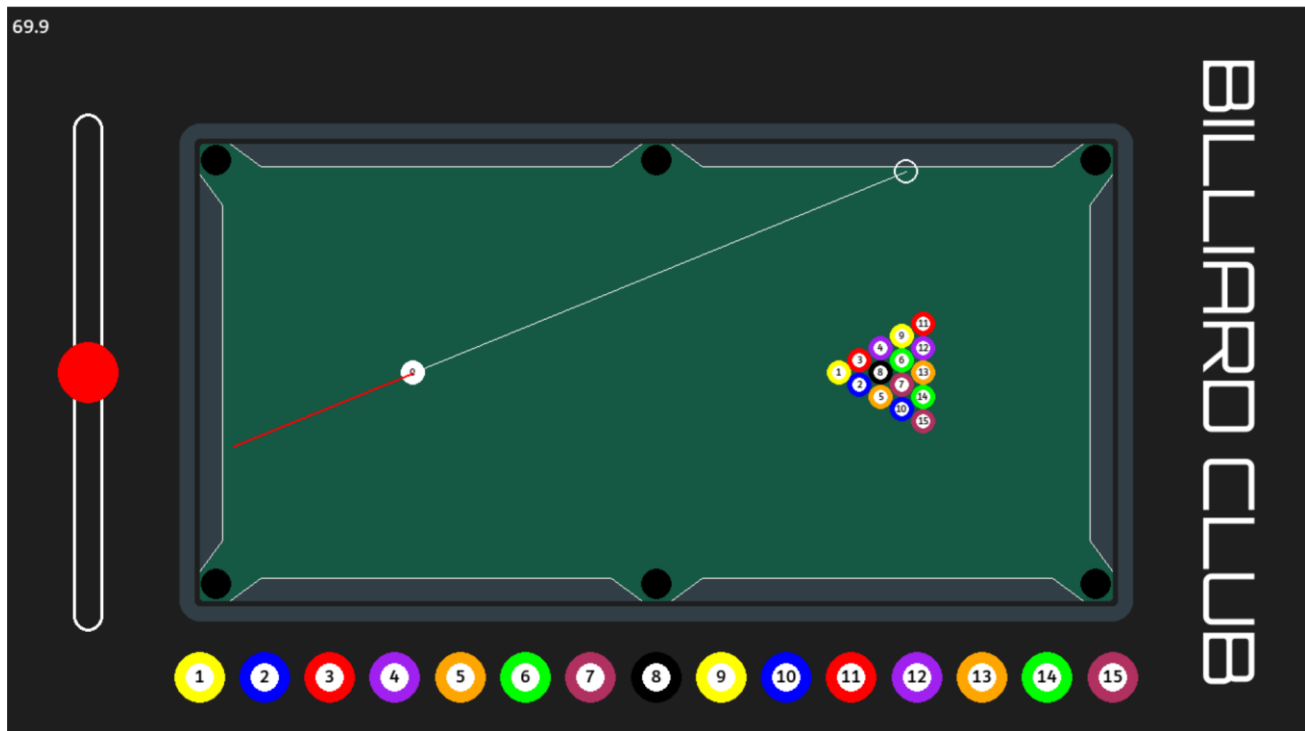


Now I will draw suitable UI to show where the ball will be when it hits the rails. I will call this the virtual ball.

```

# Cue Functions and Draws Cue
if not ballMoving: # when all the balls are still
    mousePos = pygame.mouse.get_pos() # get mouse position
    cueVec = (VEC(balls[0].pos) - VEC(mousePos)).normalize() # vector of line
    for i in tableRails: # loops through every rail (every line segment)
        for j in range(3):
            lineStart = tableRails[i][j] # Coordinates of the Line Start and End
            lineEnd = tableRails[i][j+1] # e.g. [ [x1,y1] , [x2,y2] ]
            intersection = calcPOI(lineStart, lineEnd, balls[0].pos, cueVec) # calculate coordinates of intersection
            if intersection != None: # if there is a coordinate of intersection, draw a circle
                virtualBallPos = intersection - cueVec*ballRadius # sets virtual ball position
                pygame.draw.circle(screen, "white", virtualBallPos, ballRadius, 2) # draw virtual ball
                pygame.draw.line(screen, "white", balls[0].pos, virtualBallPos) # draw projection

```



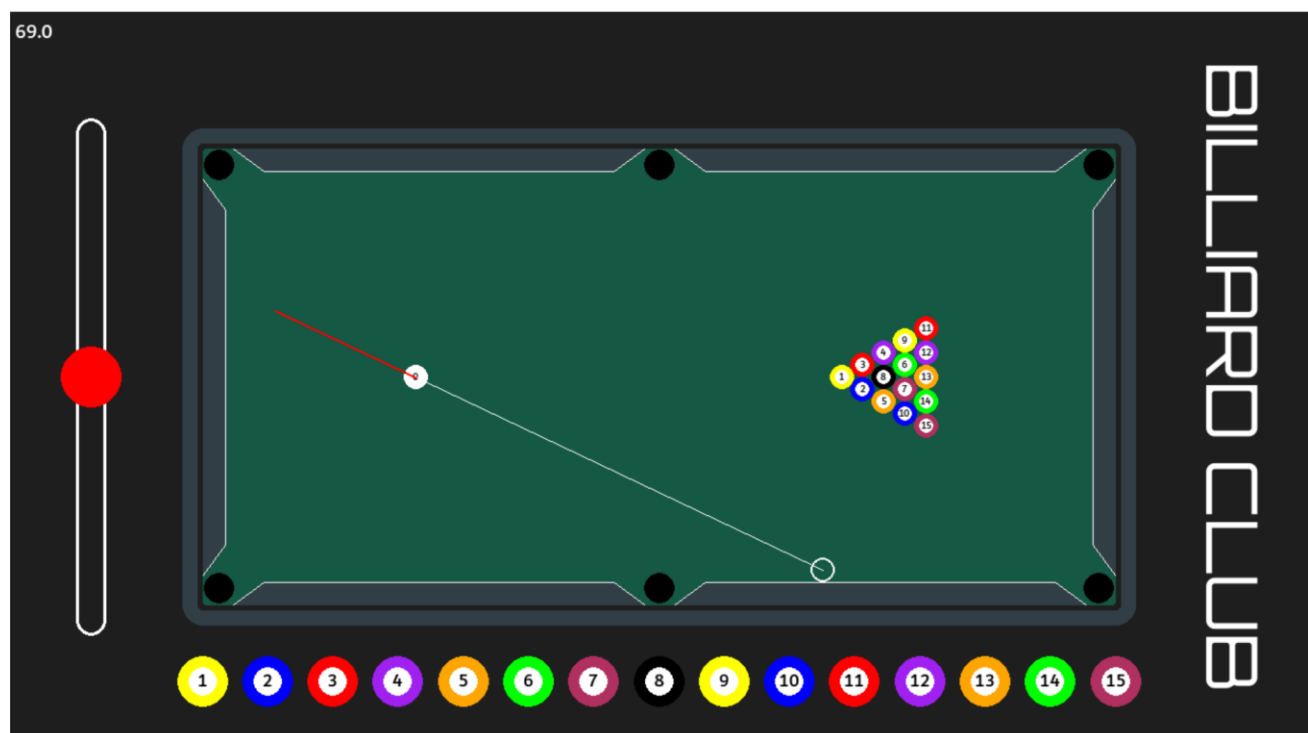
Here you can see that the virtual ball is colliding with the rails. This is a problem and therefore will be fixed.

This has been done using trigonometry to move the virtual ball slightly back so that it is at the point of just touching the rail, therefore showing the user where the ball will land if the ball hits the rail.

```

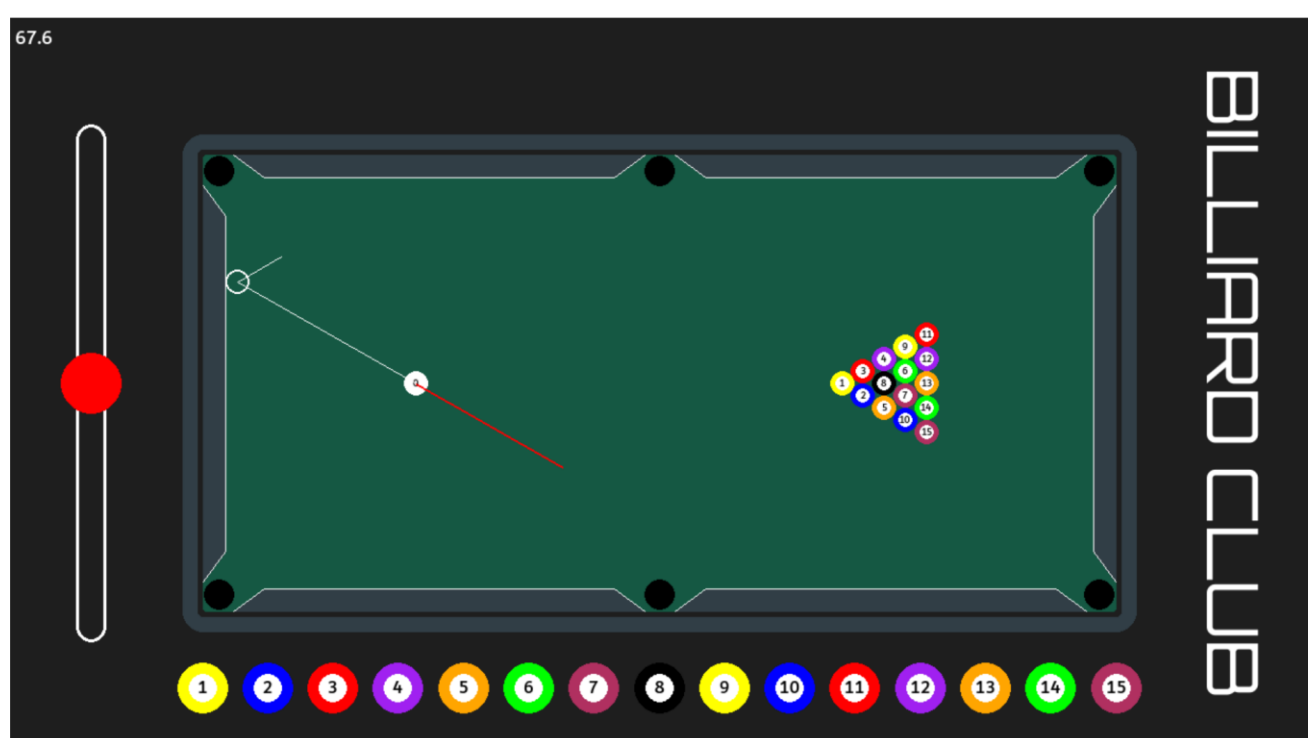
for i in tableRails: # loops through every rail (every line segment)
    for j in range(3):
        lineStart = tableRails[i][j] # Coordinates of the Line Start and End
        lineEnd = tableRails[i][j+1] # e.g. [ [x1,y1] , [x2,y2] ]
        intersection = calcPOI(lineStart, lineEnd, balls[0].pos, cueVec) # calculate coordinates of intersection
        if intersection != None: # if there is a coordinate of intersection, draw a circle
            railVec = VEC(lineEnd) - VEC(lineStart) # calculate vector of rail
            # calculate sin(theta) where theta is the angle between the two vectors
            sinAngleBetweenVectors = VEC.cross(cueVec, railVec)/(cueVec.magnitude()*railVec.magnitude())
            virtualBallPos = intersection - cueVec*abs(ballRadius/sinAngleBetweenVectors) # find projected ball pos
            pygame.draw.circle(screen, "white", virtualBallPos, ballRadius, 2) # draw virtual ball
            pygame.draw.line(screen, "white", balls[0].pos, virtualBallPos) # draw projection

```

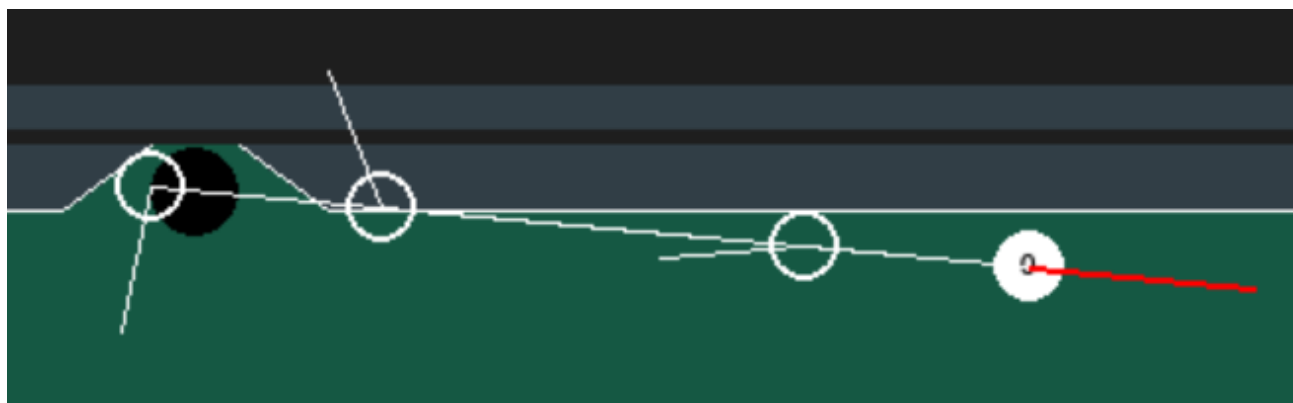


Now I will add a little line on the end of the virtual ball that shows roughly where the ball will go next.

```
# Finds reflected projection vector from the table rails:
reflectedCueVec = VEC(cueVec - 2*(VEC.dot(cueVec, railVec))*railVec).normalize()
# draw projection after rail collision
pygame.draw.line(screen, "white", virtualBallPos, (virtualBallPos-50*reflectedCueVec))
```



While playing the game, I have encountered this error:



Here the line segment is calculating for multiple collisions.

Therefore, I will group all the collisions and only draw the collision which is nearest to the ball.

```
# Cue Functions and Draws Cue
if not ballMoving: # when all the balls are still
    mousePos = pygame.mouse.get_pos() # get mouse position
    cueVec = (VEC(balls[0].pos) - VEC(mousePos)).normalize() # vector of line

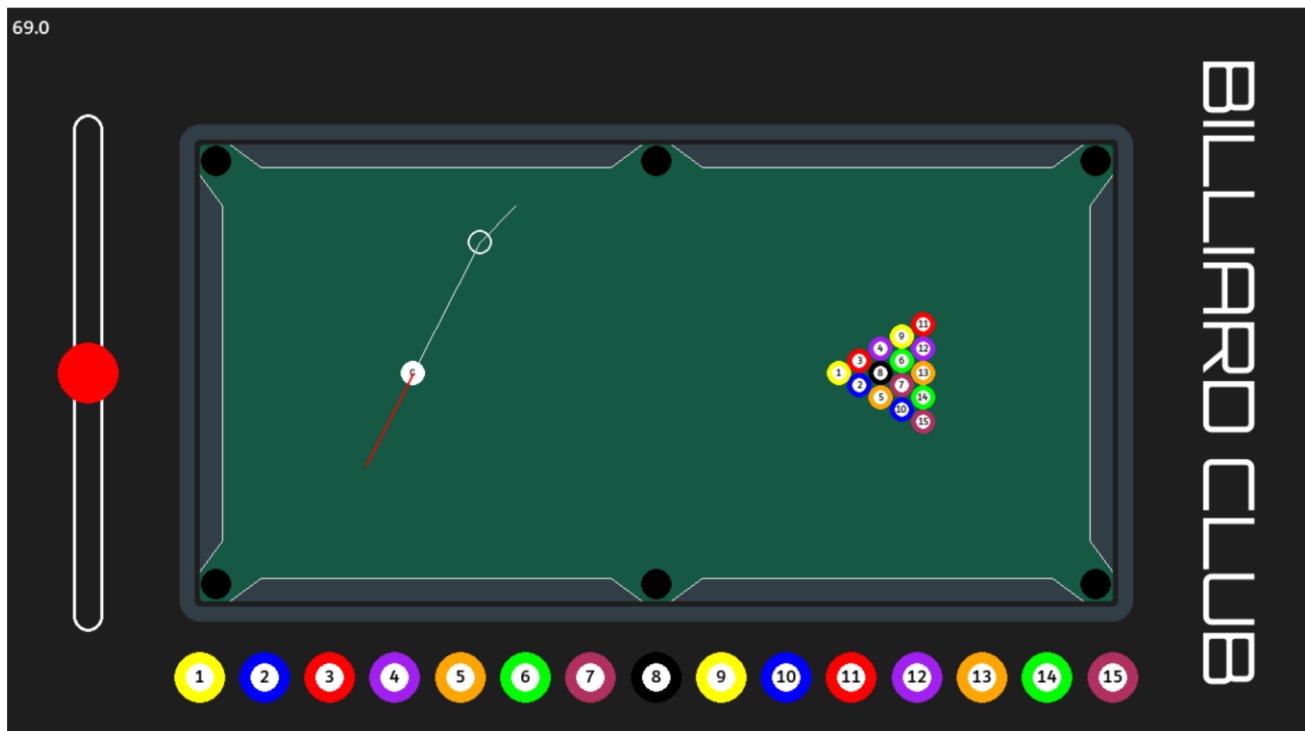
    possibleCollisions = []
    for i in tableRails: # loops through every rail (every line segment)
        for j in range(3):
            lineStart = tableRails[i][j] # Coordinates of the Line Start and End
            lineEnd = tableRails[i][j+1] # e.g. [ [x1,y1] , [x2,y2] ]
            intersection = calcPOI(lineStart, lineEnd, balls[0].pos, cueVec) # calculate coordinates of intersection
            if intersection != None: # if there is a coordinate of intersection, add to list
                possibleCollisions.append((intersection))

    # Get collision which is closest to ball:

    if len(possibleCollisions)>0: # if there is at least one collision
        # loop through all collisions and find the one with the smallest distance
        smallest = possibleCollisions[0]
        for i in range(len(possibleCollisions)-2):
            smallestDistance = smallest.distance_squared_to(balls[0].pos)
            currentDistance = possibleCollisions[i+1].distance_squared_to(balls[0].pos)
            if smallestDistance > currentDistance:
                smallest = possibleCollisions[i+1]
        intersection = smallest
    else: # if no collision, set to None
        intersection = None
```

Here I have added all the possible collisions of the two-line segments into a list, and found the coordinates of the collision which is the smallest distance to the ball

This, however, produces incorrect collision values which is shown on the next picture:



This is because all the railVec values are not stored alongside the coordinates of the collisions. This means that one collision is being calculated with the wrong starting vector, resulting in abnormal calculations shown above.

This is fixed here by storing the vector value alongside the coordinates:

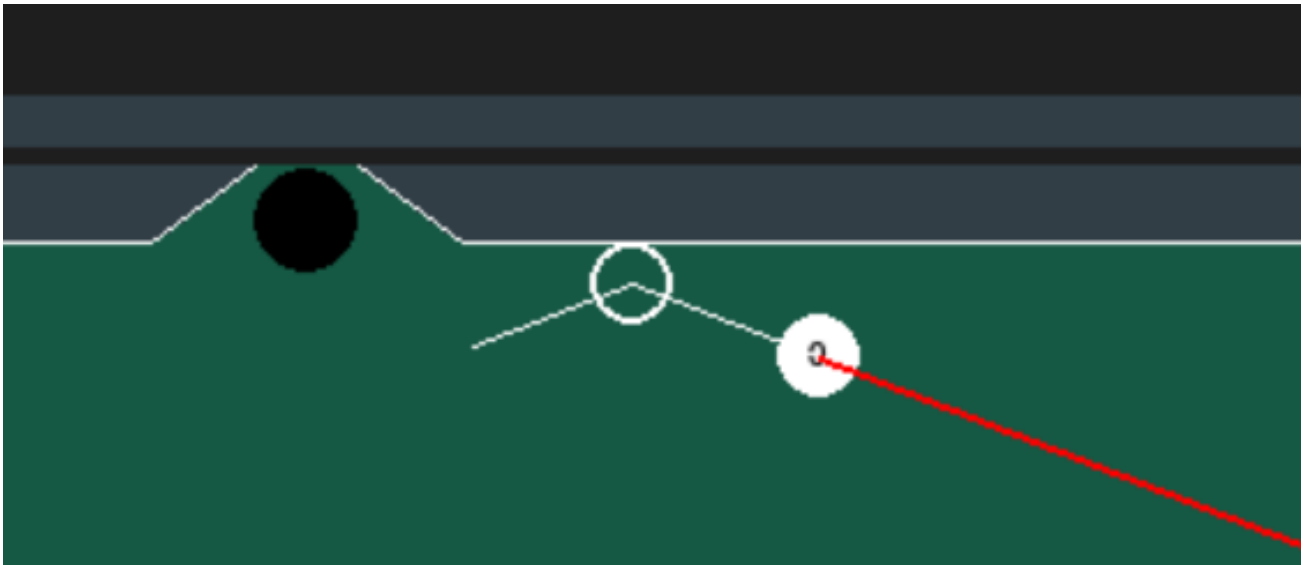
The line “possibleCollisions.append([intersection, railVec])” takes the correct railVec with it and then uses this in the calculation of the ball projection.

```
# Cue Functions and Draws Cue
if not ballMoving: # when all the balls are still
    mousePos = pygame.mouse.get_pos() # get mouse position
    cueVec = (VEC(balls[0].pos) - VEC(mousePos)).normalize() # vector of line

    possibleCollisions = []
    for i in tableRails: # loops through every rail (every line segment)
        for j in range(3):
            lineStart = tableRails[i][j] # Coordinates of the Line Start and End
            lineEnd = tableRails[i][j+1] # e.g. [ [x1,y1] , [x2,y2] ]
            railVec = (VEC(lineEnd) - VEC(lineStart)).normalize() # calculate vector of rail
            intersection = calcPOI(lineStart, lineEnd, balls[0].pos, cueVec) # calculate coordinates of intersection
            if intersection != None: # if there is a coordinate of intersection, add to list
                possibleCollisions.append([intersection, railVec])

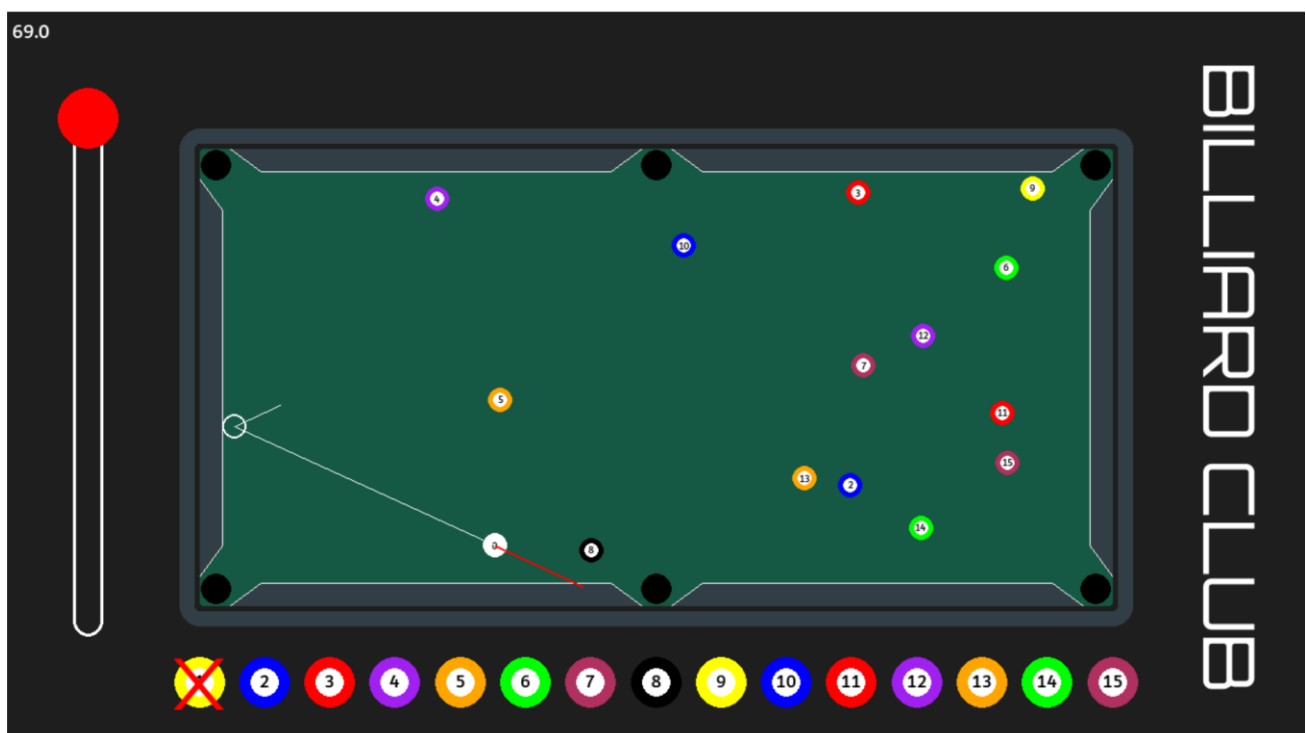
    # Get collision which is closest to ball:

    if len(possibleCollisions)>0: # if there is at least one collision
        # loop through all collisions and find the one with the smallest distance
        smallest = possibleCollisions[0]
        for i in range(len(possibleCollisions)-1):
            smallestDistance = VEC(smallest[0]).distance_squared_to(balls[0].pos)
            currentDistance = VEC(possibleCollisions[i+1][0]).distance_squared_to(balls[0].pos)
            if smallestDistance > currentDistance:
                smallest = possibleCollisions[i+1]
            intersection = VEC(smallest[0])
            railVec = VEC(smallest[1])
        else: # if no collision, set to None
            intersection = None
```

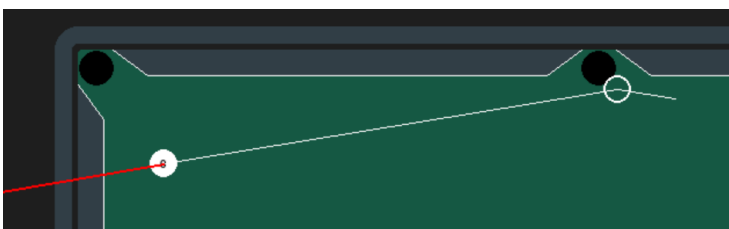


Here you can see it is now fixed, the projection vector would have passed through many lines, but only the one with the smallest distance to the ball is drawn.

The game is now playing very well.



However, while testing the game I have encountered an error in the calculation of the virtual ball position.



This is because of how the collision is calculated.

A line is drawn from the ball along the projection line. This collides with the rail parallel to the table and then using trigonometry and maths move the ball back enough so that the ball is tangential to the line vector. This results in the virtual ball hovering in seemingly mid-air.

One solution would be to calculate the collision by pixels instead of using maths. This would mean the user would always see the correct ball position and then the rebound vector could be calculated with this.

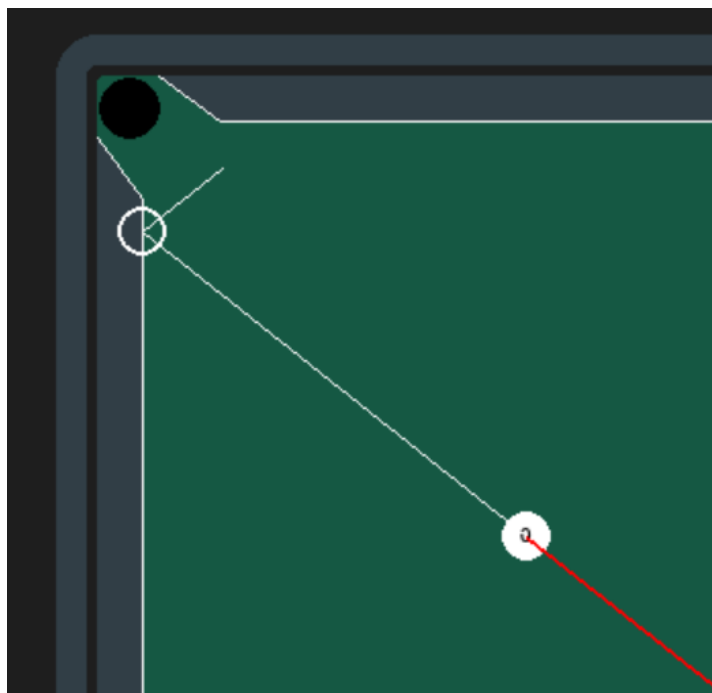
Here I have tried to implement this here.

Firstly, I have removed the code that uses trigonometric calculations to move the ball back.

```
if intersection != None: # if there is a coordinate of intersection, draw a circle
    virtualBallPos = intersection # find projected ball pos
    pygame.draw.circle(screen, "white", virtualBallPos, ballRadius, 2) # draw virtual ball
    pygame.draw.line(screen, "white", balls[0].pos, virtualBallPos) # draw projection

    # Finds reflected projection vector from the table rails:
    reflectedCueVec = VEC(cueVec - 2*(VEC.dot(cueVec, railVec))*railVec).normalize()
    # draw projection after rail collision
    pygame.draw.line(screen, "white", virtualBallPos, (virtualBallPos-50*reflectedCueVec))
```

This code above now displays the ball on the exact coordinates of intersection.



Now I will find the value to position the virtual ball back by incrementally moving the virtual ball back.

Here I will require a function that will move the virtual ball back slightly and then check if they are colliding.

Here I will have a function which inputs the ball position and its radius and check each one of the rails to see if they collide.

```
def checkVirtualCollisionRail(position, radius):
    for i in tableRails:
        for j in range(3):
            lineStart = tableRails[i][j] # Coordinates of the Line Start and End
            lineEnd = tableRails[i][j+1] # e.g. [ [x1,y1] , [x2,y2] ]
            lineVec = VEC(lineEnd[0] - lineStart[0], lineEnd[1] - lineStart[1]) # Vector of the Line
            circleVec = VEC(position[0] - lineStart[0], position[1] - lineStart[1]) # Vector of the circle to the start of the line
            t = max(0, min(1, (circleVec[0] * lineVec[0] + circleVec[1] * lineVec[1]) / lineVec.magnitude_squared()))
            # This creates a parameter that will calculate the s.f of the normal projection that maps the circle onto the line segment
            linePoint = (lineStart[0] + t * lineVec[0], lineStart[1] + t * lineVec[1]) # Calculate closest point on the line to the circle
            circlePointVec = VEC(linePoint[0] - position[0], linePoint[1] - position[1]) # Vector of the circle to the closest point
            distance = circlePointVec.magnitude() # calculates distance between the circle and closest point

            # Check if colliding
            if distance < radius:
                return True

    return False
```

The function has been implemented here:

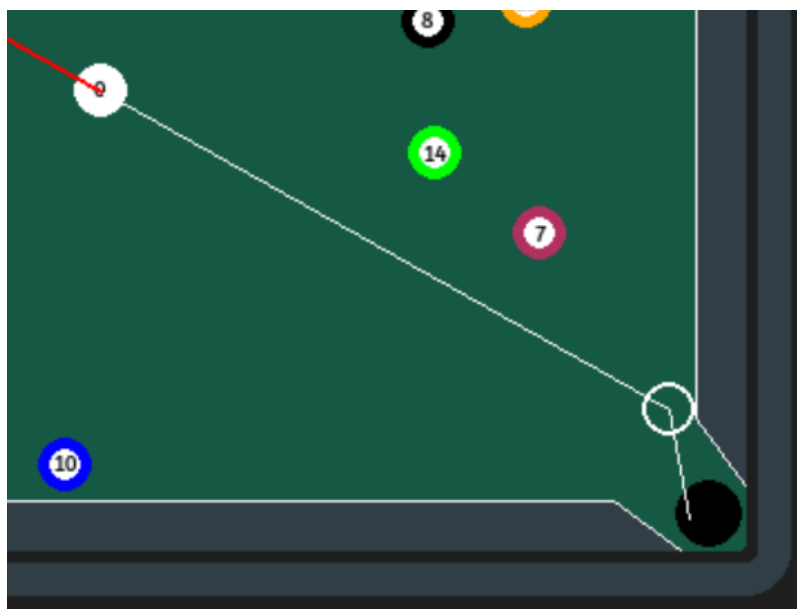
```
if intersection != None: # if there is a coordinate of intersection, draw a circle
    virtualBallPos = intersection # find projected ball pos

    colliding = checkVirtualCollisionRail(virtualBallPos, ballRadius) # check if ball colliding
    while colliding:
        virtualBallPos -= cueVec # adjust ball position until it is not colliding
        colliding = checkVirtualCollisionRail(virtualBallPos, ballRadius) # check if ball colliding again

    pygame.draw.circle(screen, "white", virtualBallPos, ballRadius, 2) # draw virtual ball
    pygame.draw.line(screen, "white", balls[0].pos, virtualBallPos) # draw projection

    # Finds reflected projection vector from the table rails:
    reflectedCueVec = VEC(cueVec - 2*(VEC.dot(cueVec, railVec))*railVec).normalize()
    # draw projection after rail collision
    pygame.draw.line(screen, "white", virtualBallPos, (virtualBallPos-50*reflectedCueVec))
```

This largely works but the projection line after the collision is not accurate.



Considering this and since ball-to-ball collisions still must be calculated, I have decided to redo the virtual ball projection entirely by instead incrementing a vector to the white ball position and checking each time if a collision occurs after every increment.

I have reverted to a previous iteration shown here:


```
# Cue Functions and Draws Cue
if not ballMoving: # when all the balls are still
    mousePos = pygame.mouse.get_pos() # get mouse position
    cueVec = (VEC(balls[0].pos) - VEC(mousePos)).normalize() # vector of line

    pygame.draw.line(screen, "red", balls[0].pos, pygame.mouse.get_pos(), 2) # draw cue

    if mouseDown: # when clicked
        if tableCorners[0][0] <= mousePos[0] <= tableCorners[1][0] \
            and tableCorners[0][1] <= mousePos[1] <= tableCorners[2][1]: # if mouse is on the pool table
            balls[0].velocity = cueVec*20*ballPowerSlider.returnValue() # give ball a velocity
```

This is the only code in the cue section which only allows the player to hit the ball (no projection)

Now I will create a function called distanceToClosestRail():

```
def distanceToClosestRail(intialPos): # returns distance from closest rail
    closestDistance = float("inf") # sets initial value to infinity
    railVec = None
    # Loops through each of the Rails (where collisions should be done)
    for i in tableRails:
        for j in range(3):
            lineStart = tableRails[i][j] # Coordinates of the Line Start and End
            lineEnd = tableRails[i][j+1] # e.g. [ [x1,y1] , [x2,y2] ]
            lineVec = VEC(lineEnd[0] - lineStart[0], lineEnd[1] - lineStart[1]) # Vector of the Line
            pointVec = VEC(intialPos[0] - lineStart[0], intialPos[1] - lineStart[1]) # Vector of the circle to the start of the line
            dot = VEC.dot(pointVec, lineVec) # finds the dot product of the vectors
            # This creates a parameter that will calculate the s.f of the normal projection that maps the circle onto the line segment
            t = max(0, min(1, dot / lineVec.magnitude_squared()))
            linePoint = (lineStart[0] + t * lineVec[0], lineStart[1] + t * lineVec[1]) # Calculate closest point on the line to the circle
            circlePointVec = VEC(linePoint[0] - intialPos[0], linePoint[1] - intialPos[1]) # Vector of the circle to the closest point
            distance = circlePointVec.magnitude() # calculates distance between the circle and closest point

            # if the new distance is smaller, set the new distance to the actual distance
            if distance < closestDistance:
                closestDistance = distance
                railVec = lineVec

    return closestDistance, railVec # returns the distance to the rail and the vector of the rail
```

This will return the distance to the closest rail. The thinking is that the virtual ball will be incremented along the ball projection vector until it collides with a rail.

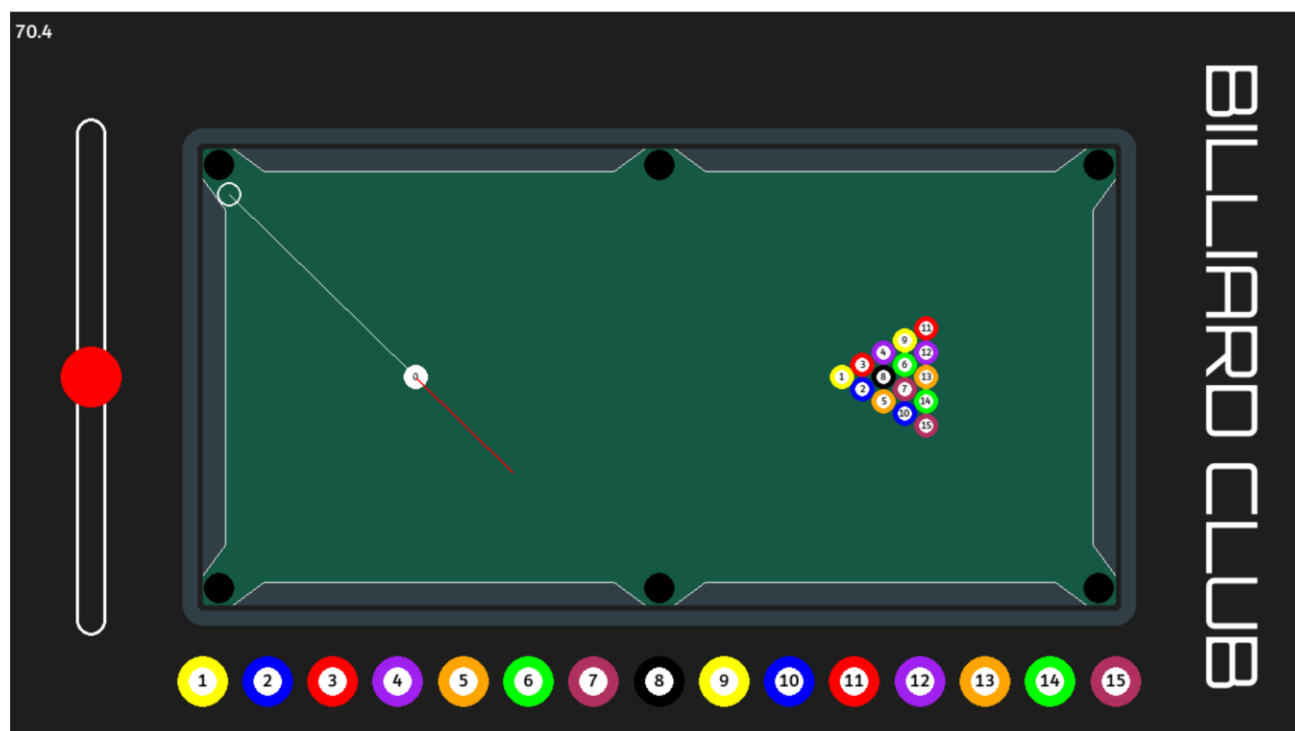
```
# Cue Functions and Draws Cue
if not ballMoving: # when all the balls are still
    mousePos = pygame.mouse.get_pos() # get mouse position
    cueVec = (VEC(balls[0].pos) - VEC(mousePos)).normalize() # vector of line

    virtualBallPos = VEC(balls[0].pos) # sets initial position of virtual ball
    distance, closestRail = distanceToClosestRail(virtualBallPos) # calls function
    while distance > ballRadius: # while the ball is not close to a rail
        virtualBallPos += cueVec # get closer to rail
        distance, closestRail = distanceToClosestRail(virtualBallPos)

    pygame.draw.line(screen, "red", balls[0].pos, pygame.mouse.get_pos(), 2) # draw cue
    pygame.draw.circle(screen, "white", virtualBallPos, ballRadius, 2) # draw virtual ball
    pygame.draw.line(screen, "white", balls[0].pos, virtualBallPos) # draw projection
```

The code then simulates the virtual ball moving, with each displacement checking if the ball is colliding with a rail.

This works well:

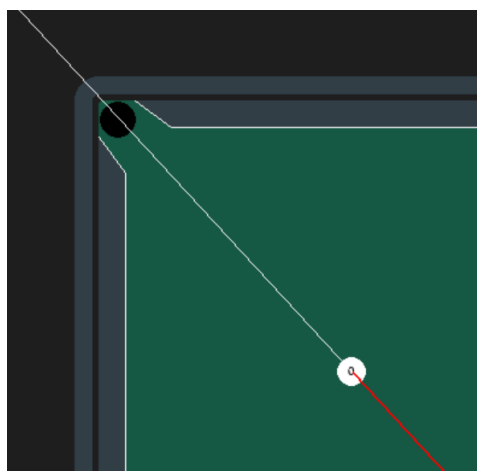


However, once the cue projection does not hit any rail, an infinite loop is created as the program attempts to find a rail forever by incrementally checking if there is a collision between a rail at really large value.

Therefore, I have implemented a maximum iteration variable that will stop the program going into an infinite loop:

```
maxIterations = 1000 # limits the number of iterations the while loop will do
while distance > ballRadius and maxIterations > 0: # while the ball is not close to a rail
    virtualBallPos += cueVec # get closer to rail
    distance, closestRail = distanceToClosestRail(virtualBallPos) # find new distance
    maxIterations -= 1 # decrement counter
```

However, this still results in the ball projection being calculated beyond where there is not a rail shown here:



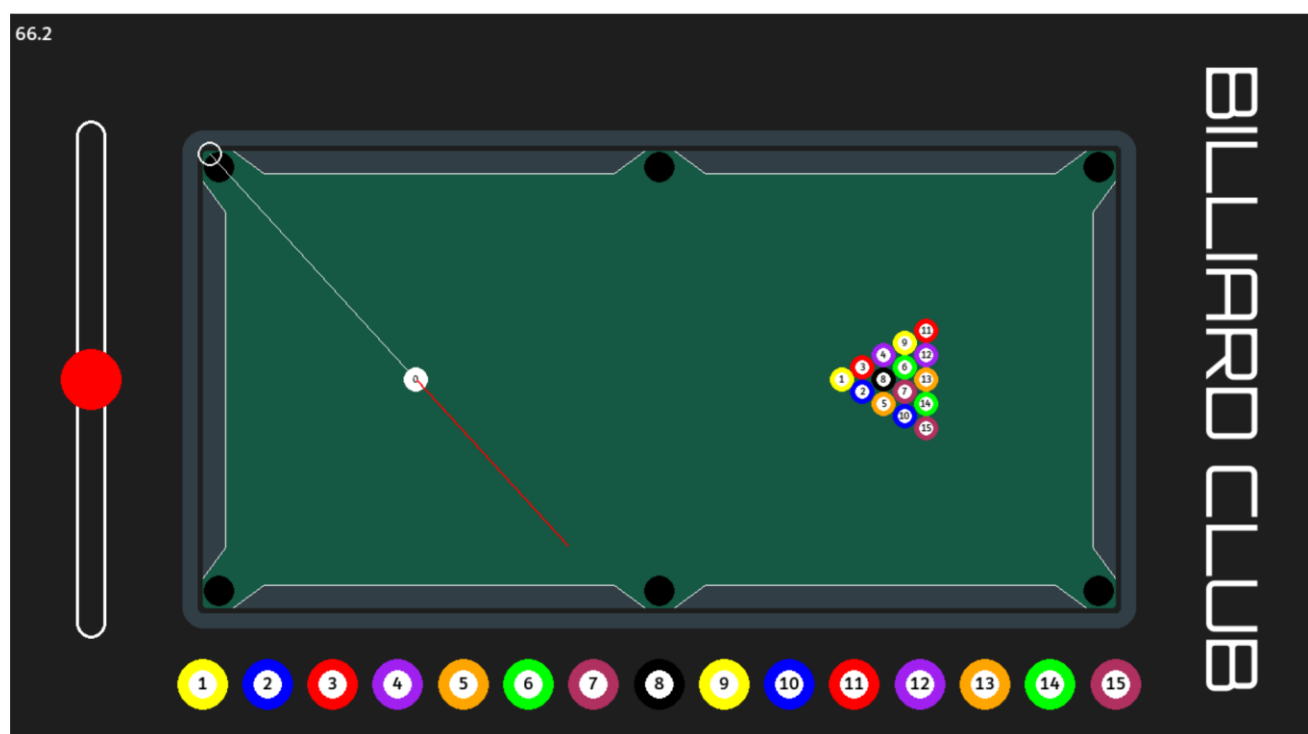
Another problem is that as each increment is very small, it tanks the FPS. This results in low fps when the virtual ball position is far from the actual ball position.

Both of these problems have been fixed with the following code:

```
while distance > ballRadius and maxIterations > 0 and (
    tableCorners[0][0] + ballRadius <= virtualBallPos[0] <= tableCorners[1][0] - ballRadius\
    and tableCorners[0][1] + ballRadius <= virtualBallPos[1] <= tableCorners[2][1] - ballRadius
): # while the ball is not close to a rail (and within the table bounds)
    virtualBallPos += ballRadius*cueVec # get closer to rail
    distance, closestRail = distanceToClosestRail(virtualBallPos) # find new distance
    maxIterations -= 1 # decrement counter
```

Each increment has been changed to a larger value, so the solution performs quicker, fixing the FPS issue.

Now the solution performs well:

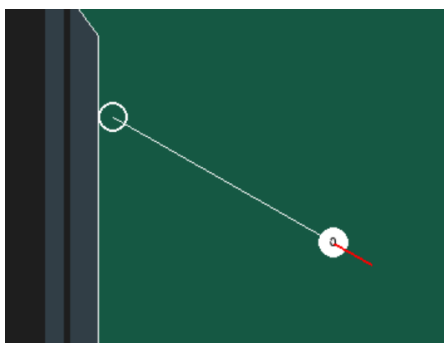


However, this change in increment must also be refined further because the virtual ball position is not in the exact correct place.

This is fixed by noticing that with large increments, the ball always needs to be moved back to fix its position. (therefore, needs very small increments backwards to fix the position).

This has been done here:

```
# finer adjustment
while distance < ballRadius or not (
    tableCorners[0][0] + ballRadius <= virtualBallPos[0] <= tableCorners[1][0] - ballRadius\
    and tableCorners[0][1] + ballRadius <= virtualBallPos[1] <= tableCorners[2][1] - ballRadius
): # while the ball is colliding with a rail or out of the table bounds
    virtualBallPos -= cueVec*0.5 # move the ball back by a very small amount
    distance, closestRail = distanceToClosestRail(virtualBallPos) # find new distance
```

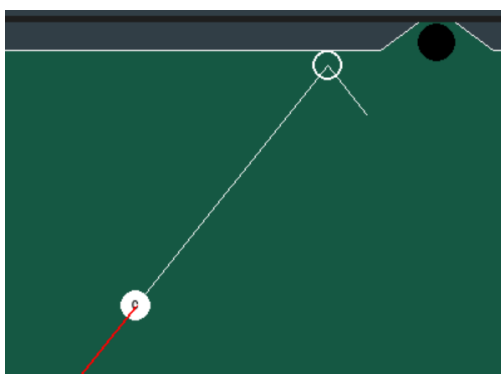


This now works very well!

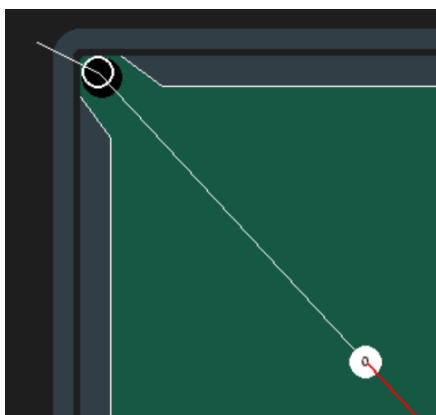
Now I will work on adding a reflected projection line.

```
reflectedCueVec = VEC(cueVec - 2*(VEC.dot(cueVec, closestRail)/VEC.dot(
    closestRail, closestRail))*closestRail).normalize() # calculated reflected vector projection
pygame.draw.line(screen, "white", virtualBallPos, (virtualBallPos-50*reflectedCueVec)) # draw reflected projection
```

Using this formula, I have now calculated the reflected vector:



However, this does not work when the ball is projected to hit a non-white wall:



I have now fixed this here:

I have added a check to say when the virtual ball collides with, he outer wall, the reflected projection vector should not be drawn:

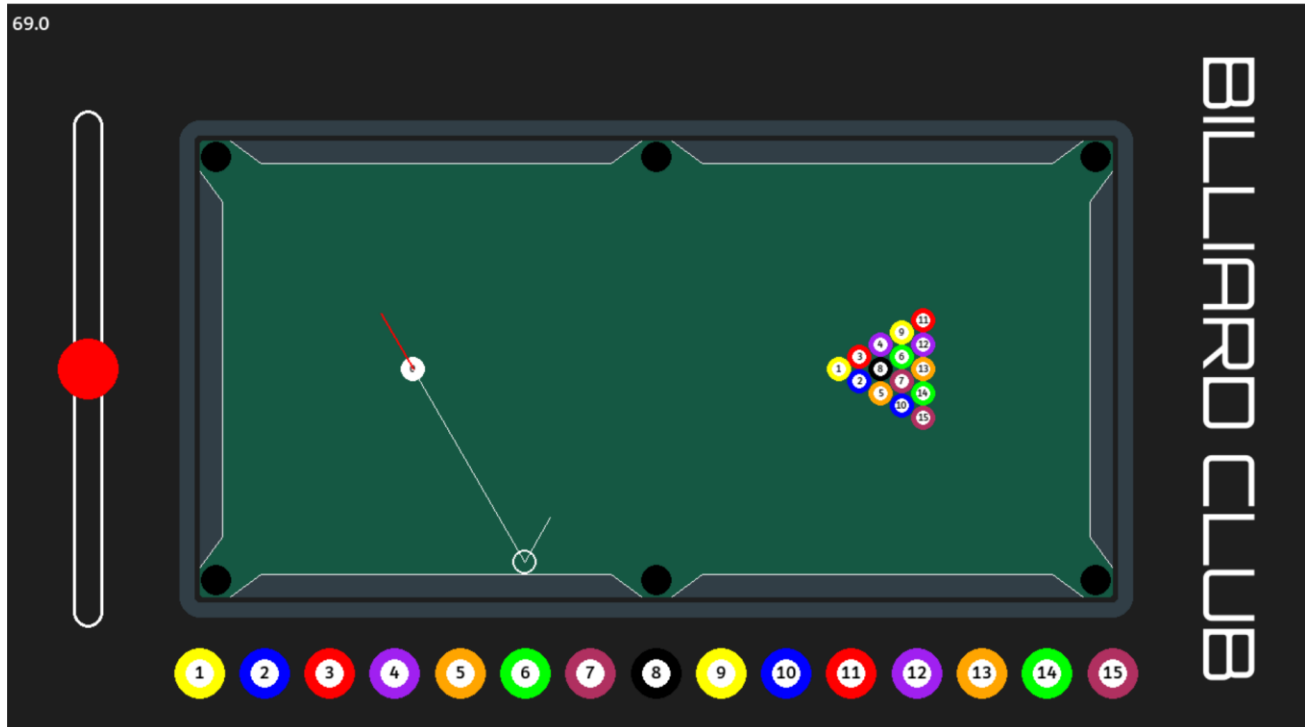
```
# if ball collides with outer wall first, don't draw it.
drawReflectedProjection = True
if not (tableCorners[0][0] + ballRadius <= virtualBallPos[0] <= tableCorners[1][0] - ballRadius\
    and tableCorners[0][1] + ballRadius <= virtualBallPos[1] <= tableCorners[2][1] - ballRadius):
    drawReflectedProjection = False
```

```

if drawReflectedProjection: # only draw if ball did not collide with outer wall
    reflectedCueVec = VEC(cueVec - 2*(VEC.dot(cueVec, closestRail)/VEC.dot(
        closestRail, closestRail))*closestRail).normalize() # calculated reflected vector projection
    pygame.draw.line(screen, "white", virtualBallPos, (virtualBallPos-50*reflectedCueVec)) # draw reflected

```

All the wall collisions have been calculated now:



Now I will add a check for collisions between the virtual ball and the numbered balls.

```

def distanceToClosestBall(intialPos):
    closestDistance = float("inf") # sets initial value to infinity
    otherBall = None # sets initial value to none
    for i in range(len(ballGroup)-1): # loops through every ball (except whiteball)
        distance = VEC(intialPos).distance_to(balls[i+1].pos) # finds the distance between the ball and the whiteball
        if distance < closestDistance: # if the new distance is smaller, set the new distance to the actual distance
            closestDistance = distance
            otherBall = balls[i+1]
    return closestDistance, otherBall # return the distance and then the instance of the other ball

```

This function returns the distance and the instance of the ball which is the closest to the virtual ball.

I have then updated the rough and finer adjustments for the ball collisions here:

```

railDistance, closestRail = distanceToClosestRail(virtualBallPos) # updates distance to closest rail
ballDistance, closestBall = distanceToClosestBall(virtualBallPos) # updates distance to closest ball

```

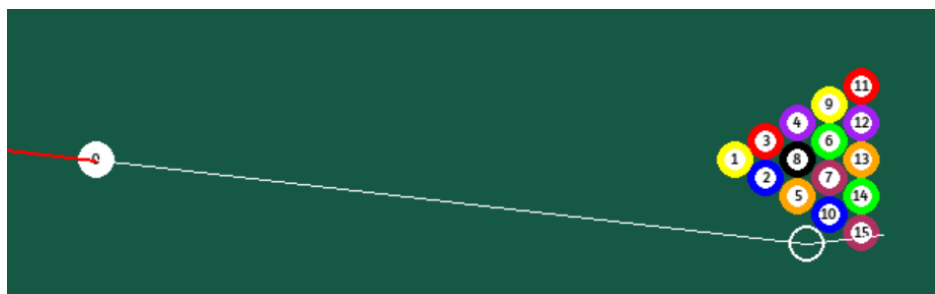
```

# rough check for virtual ball collisions
maxIterations = 1000 # limits the number of iterations the initial while loop will do
while railDistance > ballRadius and ballDistance > 2*ballRadius and maxIterations > 0 and (
    tableCorners[0][0] + ballRadius <= virtualBallPos[0] <= tableCorners[1][0] - ballRadius\
    and tableCorners[0][1] + ballRadius <= virtualBallPos[1] <= tableCorners[2][1] - ballRadius
): # while the ball is not close to a rail (and within the table bounds)
    virtualBallPos += ballRadius*cueVec # get closer to rail
    railDistance, closestRail = distanceToClosestRail(virtualBallPos) # find new distance
    ballDistance, closestBall = distanceToClosestBall(virtualBallPos)
    maxIterations -= 1 # decrement counter

```

```
# finer adjustment for virtual ball collisions
while railDistance < ballRadius or ballDistance < 2*ballRadius or not (
    tableCorners[0][0] + ballRadius <= virtualBallPos[0] <= tableCorners[1][0] - ballRadius\
    and tableCorners[0][1] + ballRadius <= virtualBallPos[1] <= tableCorners[2][1] - ballRadius
): # while the ball is colliding with a rail or out of the table bounds
    virtualBallPos -= cueVec*0.5 # move the ball back by a very small amount
    railDistance, closestRail = distanceToClosestRail(virtualBallPos) # find new distance
    ballDistance, closestBall = distanceToClosestBall(virtualBallPos)
```

This produces the following result:



This is good but the projection for the collision needs to be updated.

This is done here:

After every while loop, I need to determine what the ball is colliding with.

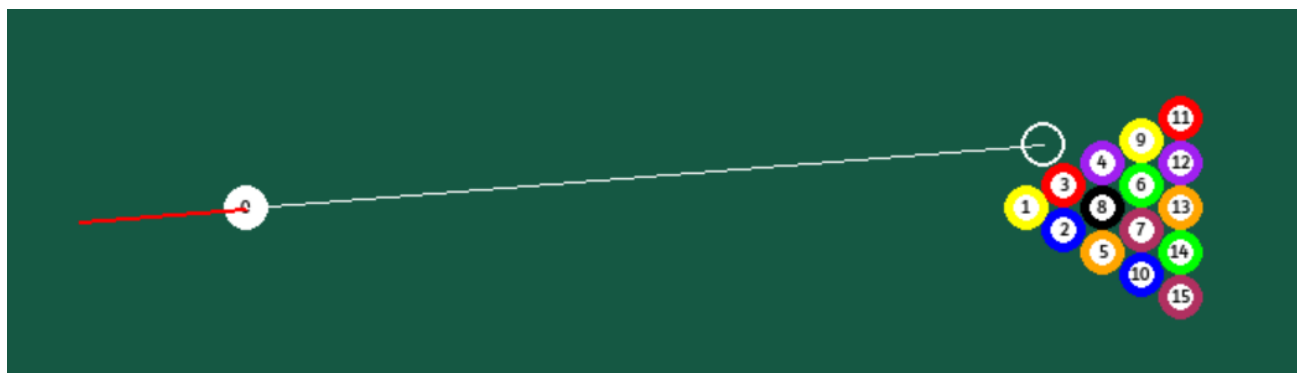
There are three possible things that the ball could be colliding with:

1. Another ball
2. A rail
3. The outer wall

```
# if ball collides with other ball, set this to true
collidingWithBall = False
if ballDistance < 2*ballRadius:
    collidingWithBall = True

# if ball collides with outer wall first or other ball, don't draw projection.
drawReflectedProjection = True
if collidingWithBall or not (tableCorners[0][0] + ballRadius <= virtualBallPos[0] <= tableCorners[1][0] - ballRadius\
    and tableCorners[0][1] + ballRadius <= virtualBallPos[1] <= tableCorners[2][1] - ballRadius):
    drawReflectedProjection = False
```

Firstly, I have set the virtual ball to have no projection vector if it collides with a ball first.



This works as shown.

I have added graphics when the virtual ball collides with another ball:

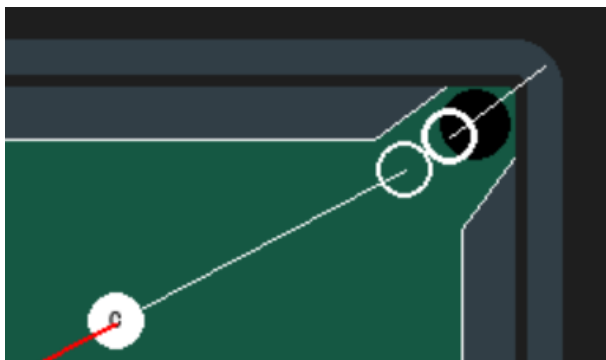
```

if collidingWithBall:
    virtualBallVec = (virtualBallPos - closestBall.pos).normalize() # vector of virtual ball to other ball
    pygame.draw.circle(screen, "white", closestBall.pos, ballRadius, 3) # draw virtual ball
    pygame.draw.line(screen, "white", closestBall.pos, (closestBall.pos-50*virtualBallVec)) # draw reflected projection

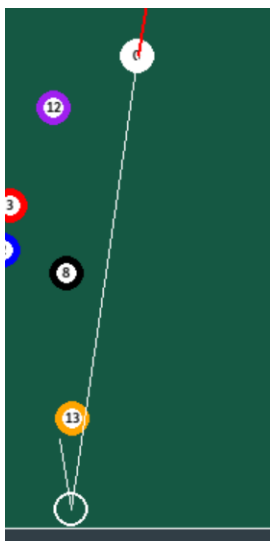
```



While playing the testing the game, I have noticed these errors:



(Game detecting a ball in which has been potted)



(Higher numbered balls are not being detected)

The second error is fixed by changing the following line of code in the function distanceToClosestBall:

```

def distanceToClosestBall(intialPos):
    closestDistance = float("inf") # sets initial value to infinity
    otherBall = None # sets initial value to none
    for i in range(len(ballGroup)-1): # loops through every ball (except whiteball)
        distance = VEC(intialPos).distance_to(balls[i+1].pos) # finds the distance between the ball and the whiteball
        if distance < closestDistance: # if the new distance is smaller, set the new distance to the actual distance
            closestDistance = distance
            otherBall = balls[i+1]
    return closestDistance, otherBall # return the distance and then the instance of the other ball

```

From:

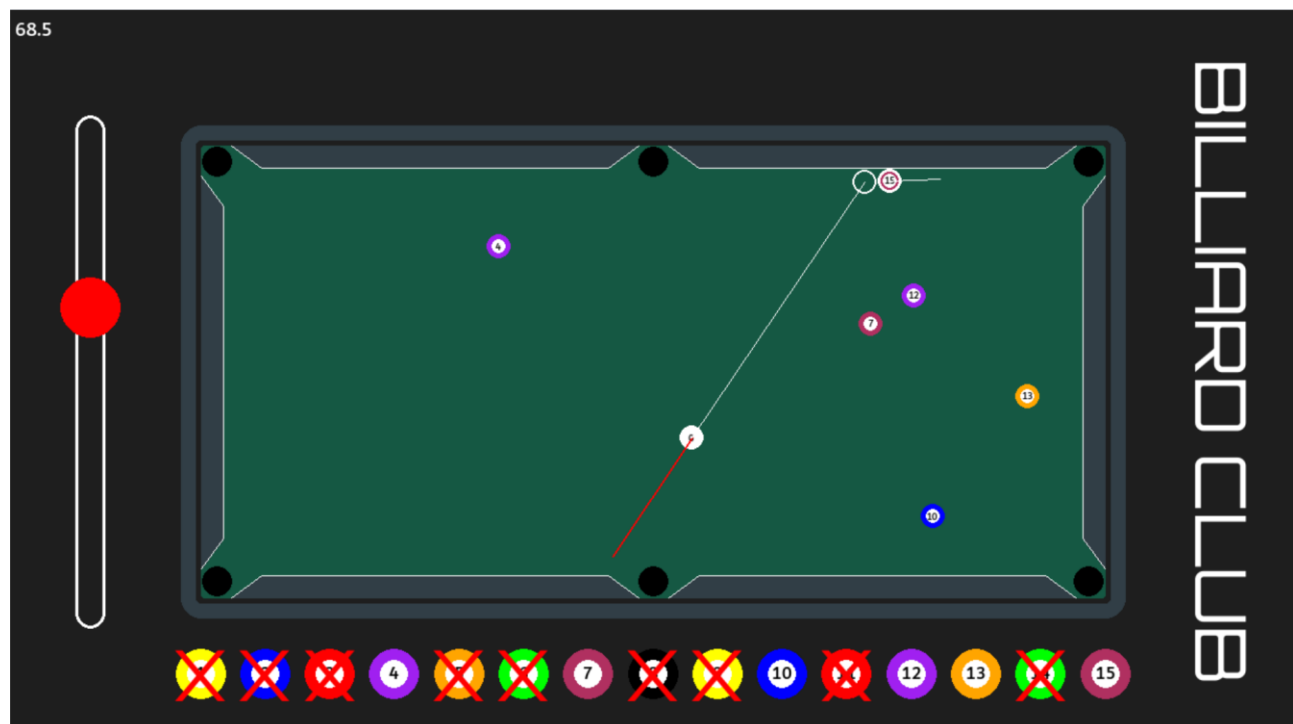
```
for i in range(len(ballGroup)-1):
```

To:

```
for i in range(len(balls)-1): # loops through every ball (except whiteball)
```

Error now fixed as shown:

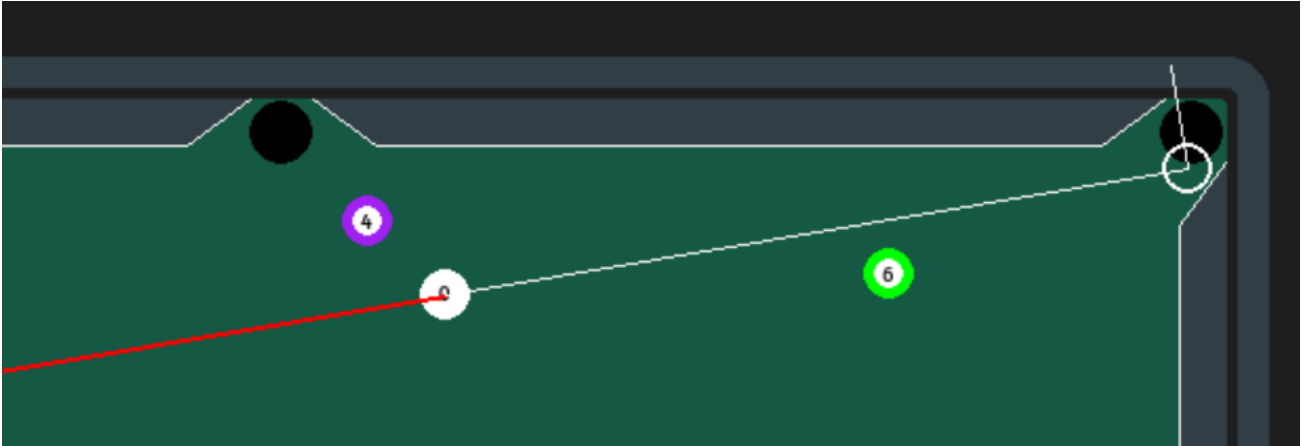
Ball 15 is being detected, and projection is being calculated.



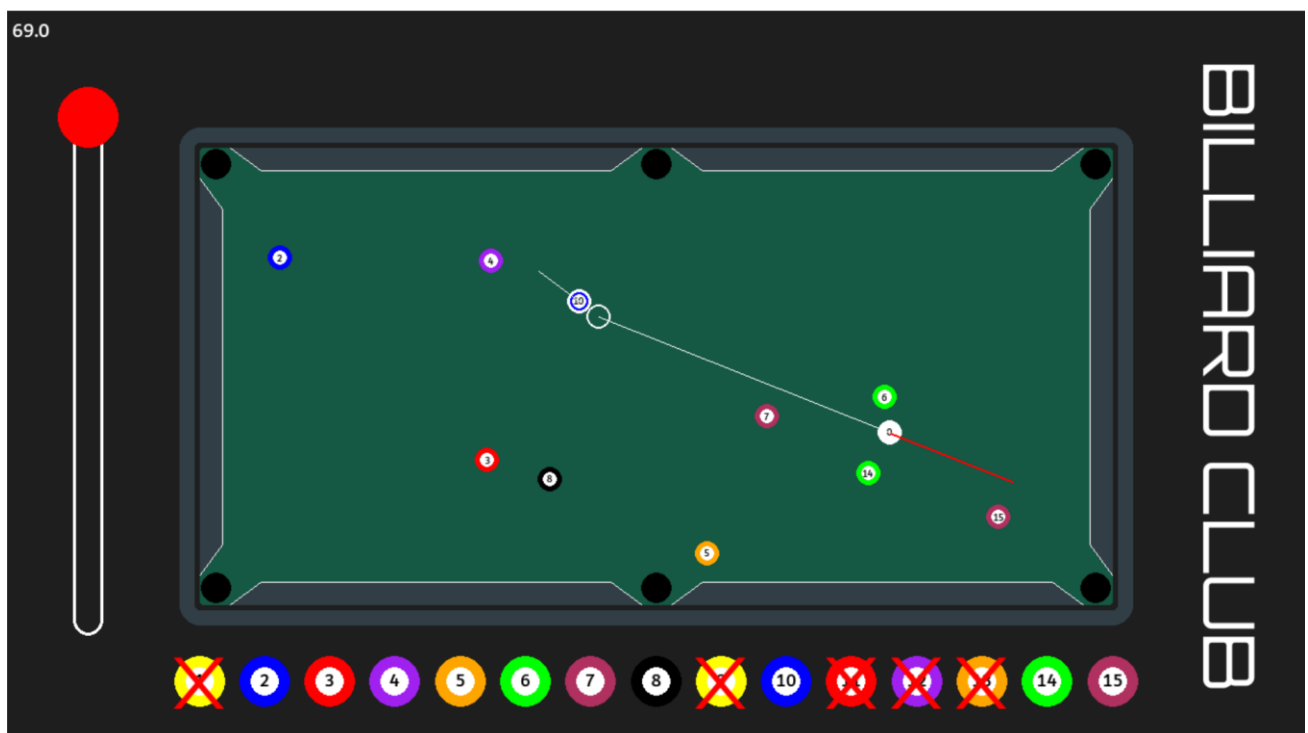
When looking at how to fix the first error, I have realised that the delete method only removes the ball from the ballGroup list. This means it still exists in the Balls list. Therefore, I will undo the previous fix and redo this function. This is the fixed function:

```
def distanceToClosestBall(intialPos):
    closestDistance = float("inf") # sets initial value to infinity
    otherBall = None # sets initial value to none
    for ball in ballGroup.sprites(): # loops through every ball (except whiteball)
        if ball.number != 0: # skip if the ball is itself (white)
            distance = VEC(intialPos).distance_to(ball.pos) # finds the distance between the ball and the whiteball
            if distance < closestDistance: # if the new distance is smaller, set the new distance to the actual distance
                closestDistance = distance
                otherBall = ball
    return closestDistance, otherBall # return the distance and then the instance of the other ball
```


When a ball is potted, the game does not detect any ball in the hole:



Both errors are fixed, and the game now has all its functionality for a complete sandbox game:



Stage 3 Review

What has been done?

- Numbers were added onto the balls. I also adjusted the positions of the text, making it more aesthetic.
- A game logo was added
- A slider for controlling the cue's power was implemented with a reusable class.
- Suitable UI was created to display which balls remain on the table or have not been potted (balls potted have an "X").
- A visual cue has been added which shows users a projection line of the white ball when hit, I have called this the virtual ball.

How is code being tested?

- For each new element/function (e.g., ball movement, wall collisions, ball-to-ball collisions), I add a small bit of functionality and then observe the result, documenting and debugging it accordingly. This incremental approach ensures I follow the computational method of decomposition which ensures for swift development of the solution.
- Bugs (like clipping, stuck balls, or zero-distance errors) are identified and resolved by modifying code of the program, rerunning the simulation, and documenting my errors.
- Comments are used extensively throughout the solution. This helps debugging enormously as I sift through hundreds of lines of code.

Errors that have been fixed:

Error	Fix
The numbers drawn on each ball were not positioned correctly.	I adjusted the draw() method to modify the coordinates for rendering the text so that the numbers appear in the centre of the ball
When drawing the slider, the circle representing the handle was off-centre.	The drawing logic was updated to centre the circle correctly within the box.
On clicking the slider, the slider handle did not follow the mouse movement or position itself correctly.	The update method was revised so that the slider now accurately tracks the mouse when clicked and dragged.
The Ball information UI was positioned incorrectly on the screen.	The positions for the ball icons were recalculated so that they appear in the intended locations.
The white ball was being drawn along with the numbered balls. (It should be omitted)	A condition was added to check if a ball is white and, if so, skip drawing it in the ball list.
When a ball was potted, it was removed from the list and the remaining balls' positions shifted immediately.	Instead of removing the ball from the layout, a master tuple was used to store fixed ball positions while overlaying an "X" to indicate that the ball has been potted.
The "X" (marking potted balls) was not correctly centred and would display on every ball that has not been potted.	The code was adjusted to correctly centre the "X" and invert the logic so that it remains visible for potted balls.

When the projection vector and the rail were parallel, calculating the parameter t led to a division by zero error.	I refactored the calculation of t by separating it into a numerator and denominator, which allowed the code to safely handle cases when the vectors are parallel (when the denominator is zero)
Using infinite line equations led to multiple intersection points being detected between the ball projection and the rails. This is a problem as I only need the nearest one.	I changed the algorithm to compute the intersection between two-line segments. An extra parameter s was introduced so that only intersections with both t and s in the $[0, 1]$ are valid. This ensured that only the nearest collision point was shown.
If the cue projection did not hit any rail, the incremental collision-checking loop create an infinite loop, crashing the program.	I implemented a maximum iteration variable that broke out of the loop when no collision is detected within a reasonable number of steps.
Using very small increments for the virtual ball's movement caused very low FPS.	I first used large increments to get a rough position, then applied very small backward increments to refine the final position to get an accurate value.
The reflected projection vector was calculated incorrectly when the ball was projected toward a non-white outer wall.	A conditional check was added so that if the virtual ball collides with the outer wall, the reflected projection vector is not drawn.
The collision detection function (distanceToClosestBall) was failing to detect higher numbered balls (e.g. balls 14 and 15).	A line of code in the function was corrected, ensuring that all numbered balls are properly detected and factored into the collision calculations.
The delete method was only removing the potted ball from the ballGroup list while it remained in the overall Balls list, causing it to be detected in further collisions, which was causing errors.	I overhauled the delete function so that it removes the ball from all relevant lists, ensuring referential integrity.
When a ball was potted, the game failed to detect that the ball was in the hole.	The detection logic was refined so that potted balls are correctly recognised by the game.

Success criteria that has been met:

6	The system will implement visual aim assistance for the cue.
7	The visual aim assistance will turn red/white depending on the prediction of what ball it will hit.
14	The game should have a power meter to show how powerful each hit with the cue is.

Summary of Stage 3

This stage has been critical for enhancing the player's interaction, visual feedback, and preparing for advanced mechanics, such as user control and strategic gameplay. The game is ready for the next stages of development where the rules and AI will be implemented.

Stage 4: Game Rules and Logic

Before starting the next stage, I have decided to stop and refactor some of the code in order to make the code more maintainable. I have attempted this by doing the following:

I have moved this block of code inside a procedure.

```
# Checks and resolves every collision between every pair of balls
for i in range(len(ballGroup) - 1):
    for j in range(i + 1, len(ballGroup)): # except white ball
        ballGroup.sprites()[i].checkResolveCollision(ballGroup.sprites()[j])
```

To this:

```
def resolveAllCollisions(): # Checks and resolves every collision between every pair of balls
    for i in range(len(ballGroup) - 1):
        for j in range(i + 1, len(ballGroup)): # except white ball
            ballGroup.sprites()[i].checkResolveCollision(ballGroup.sprites()[j])
```

```
resolveAllCollisions() # Checks and resolves every collision between every pair of balls
```

I have also grouped the Ball UI elements into one procedure which will draw the entire Ball UI in one.

Shown here:

```
def drawBallUI(): # Draw 15 balls on screen for UI (NOT ON TABLE)
    increment = 0 # adds a value on each time so the balls are not drawn on top of each other
    distanceBetweenBalls = tableDimensions[0]/(len(ballInfo)-2)
    for i in range(len(ballInfo)-1): # loops through each ball except white ball
        circleCentre = (tableCorners[0][0] + increment, tableCorners[2][1]+75) # coordinates
        pygame.draw.circle(screen, ballInfo[i+1][0], circleCentre, 25)
        numberFont = ballUIFont.render(str(ballInfo[i+1][1]), True, "black") # renders font
        pygame.draw.circle(screen, "white", circleCentre, numberFont.get_height()/2.2) # draws white circle
        numberFontPos = [circleCentre[0] - numberFont.get_width()/2, circleCentre[1] - numberFont.get_height()/2]
        screen.blit(numberFont, numberFontPos) # adds font to screen
        increment += distanceBetweenBalls

    resolveAllCollisions() # Checks and resolves every collision between every pair of balls (only if colliding)
    drawBallUI() # draws the ball UI (15 balls that tells the user which ball to pot next)
```

Overall makes it a lot more readable and easier to manage. This also demonstrates the abstraction computational method as I have removed unnecessary information, making the solution easier to code.

This process has been done again with the Logo and FPS display shown here:

From this:

```
# Logo display
logo = logoFont.render(("BILLIARD CLUB"), True, "white")
logo = pygame.transform.rotate(logo, 270)
screen.blit(logo, (resolution[0] - logo.get_width() - 30, centre[1]-logo.get_height()/2))

# FPS Display
currentFPS = str(round(clock.get_fps(), 1))
text = fpsFont.render((str(currentFPS)), True, "white")
screen.blit(text, (5, 3))
```

To this:

```
def drawLogo(): # Draws the logo
    logo = logoFont.render(("BILLIARD CLUB"), True, "white")
    logo = pygame.transform.rotate(logo, 270)
    screen.blit(logo, (resolution[0] - logo.get_width() - 30, centre[1]-logo.get_height()/2))

def drawFPS(): # FPS Display
    currentFPS = str(round(clock.get_fps(), 1))
    text = fpsFont.render((str(currentFPS)), True, "white")
    screen.blit(text, (5, 3))
```

```
drawLogo() # draws the logo
drawFPS() # draws the FPS
```

This has also been done with the table:

```
def drawTable(): # Draws the table...
```

(Shortened in my IDE)

This has been done repeatedly and now there are all these functions, nicely organised:

```
def checkIfBallMoving(): # Checks if any ball is moving (returns True if any ball is moving) ...
def distanceToClosestBall(intialPos): # returns distance from closest ball ...
def distanceToClosestRail(intialPos): # returns distance from closest rail ...
def resolveAllCollisions(): # Checks and resolves every collision between every pair of balls (only if colliding) ...
def resolveWallCollisions(): # Checks and resolves every collision between every ball and the walls ...
def resolveBallPotted(): # Checks if a ball is potted ...

def drawTable(): # Draws the table ...
def drawBalls(): # Draws all the balls ...
def drawBallUI(): # Draw 15 balls on screen for UI (NOT ON TABLE) ...
def drawLogo(): # Draws the logo ...
def drawFPS(): # FPS Display ...
def drawCue(): # draws the cue ...
```

This means I can organise the game loop nicely, so it now looks like this:

```

# Game Loop
while True:
    mouseDown = False
    for event in pygame.event.get():
        if event.type == pygame.QUIT: # Checks if user closes program
            pygame.quit()
            exit()

        if event.type == pygame.MOUSEBUTTONDOWN: # on left click
            mouseDown = True

    ballPowerSlider.update() # Update Slider
    ballGroup.update() # update each ball

    resolveAllCollisions() # Checks and resolves every collision between every pair of balls (only if colliding)
    resolveWallCollisions() # Checks and resolves every collision between every ball and the walls
    resolveBallPotted() # checks if a ball is potted and deletes it if it is.

    # Draw Everything to Screen:

    screen.fill((30, 30, 30)) # Fills the screen with a grey colour

    drawTable() # Draws the table
    drawBalls() # draws the balls

    if not checkIfBallMoving(): # when all the balls are still
        drawCue() # draw the cue

    ballPowerSlider.draw(screen) # Draws slider

    drawBallUI() # draws the ball UI (15 balls that tells the user which ball to pot next)
    drawLogo() # draws the logo
    drawFPS() # draws the FPS

    pygame.display.flip() # Updates the display
    clock.tick(FPS) # Caps the frame rate

```

This is now done and now I can start work on implementing the game rules to the solution.

In order to fully implement the rules of 8 ball pool, I will work on implementing the game into a 2-player mode.

First, I will implement suitable UI to show which player's go it is.

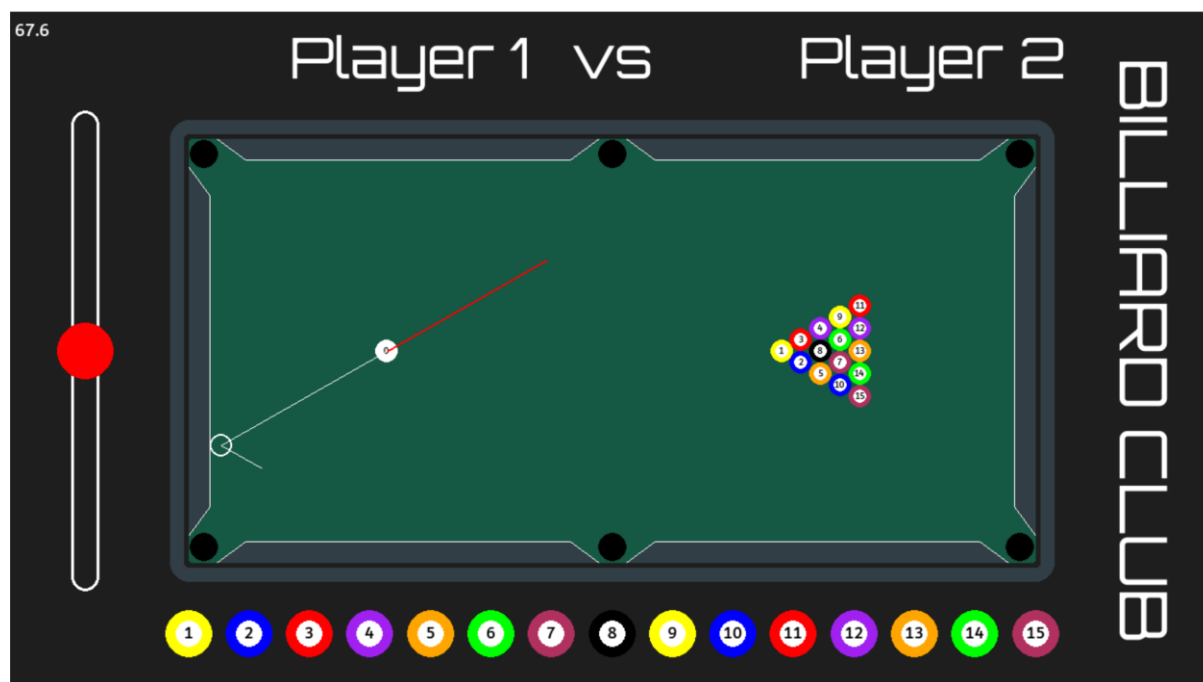
```

def drawPlayerIndicator(): # Draws the player indicator

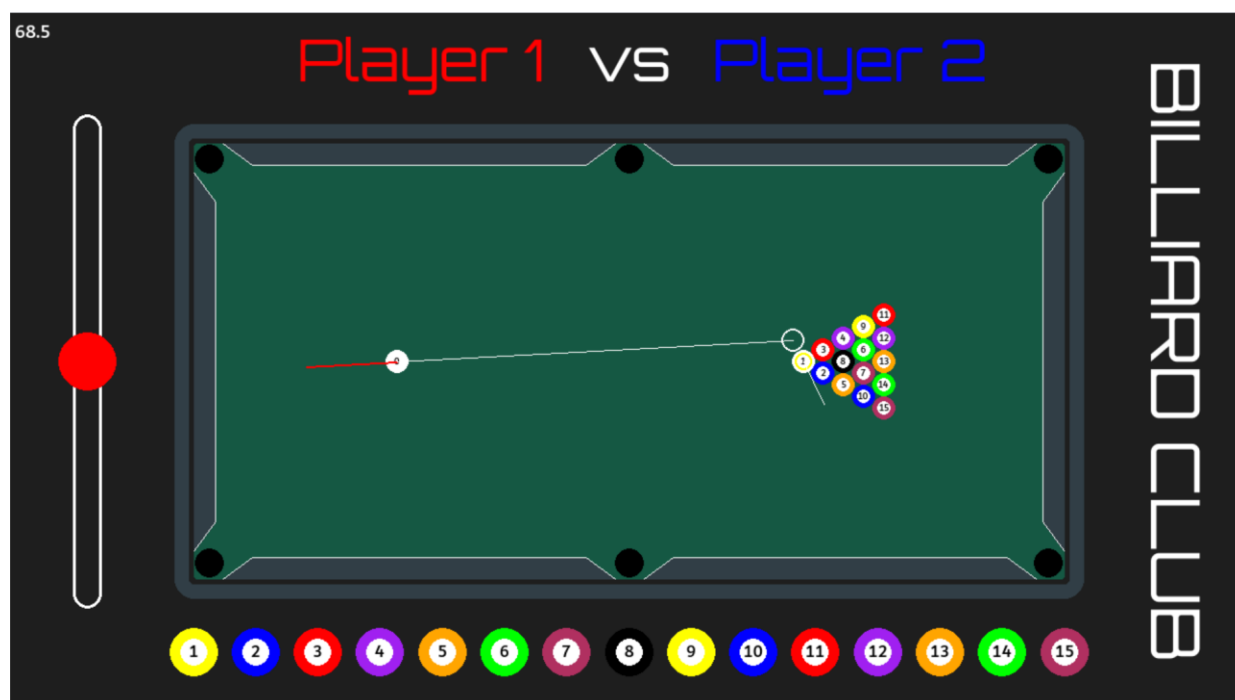
    player1Text = playerIndicatorFont.render(("Player 1"), True, "white")
    vsText = playerIndicatorFont.render(("vs"), True, "white")
    player2Text = playerIndicatorFont.render(("Player 2"), True, "white")

    screen.blit(player1Text, (centre[0] - player1Text.get_width() - vsText.get_width(), 10))
    screen.blit(vsText, (centre[0] - vsText.get_width()/2, 10))
    screen.blit(player2Text, (centre[0] + player2Text.get_width() - vsText.get_width(), 10))

```



Here I have created three separate texts and added them to the screen. However, there is a problem that the player2 text is not centred correctly.



This is now fixed, and I have added a colour to the text.

I have now drawn arrows to the screen:

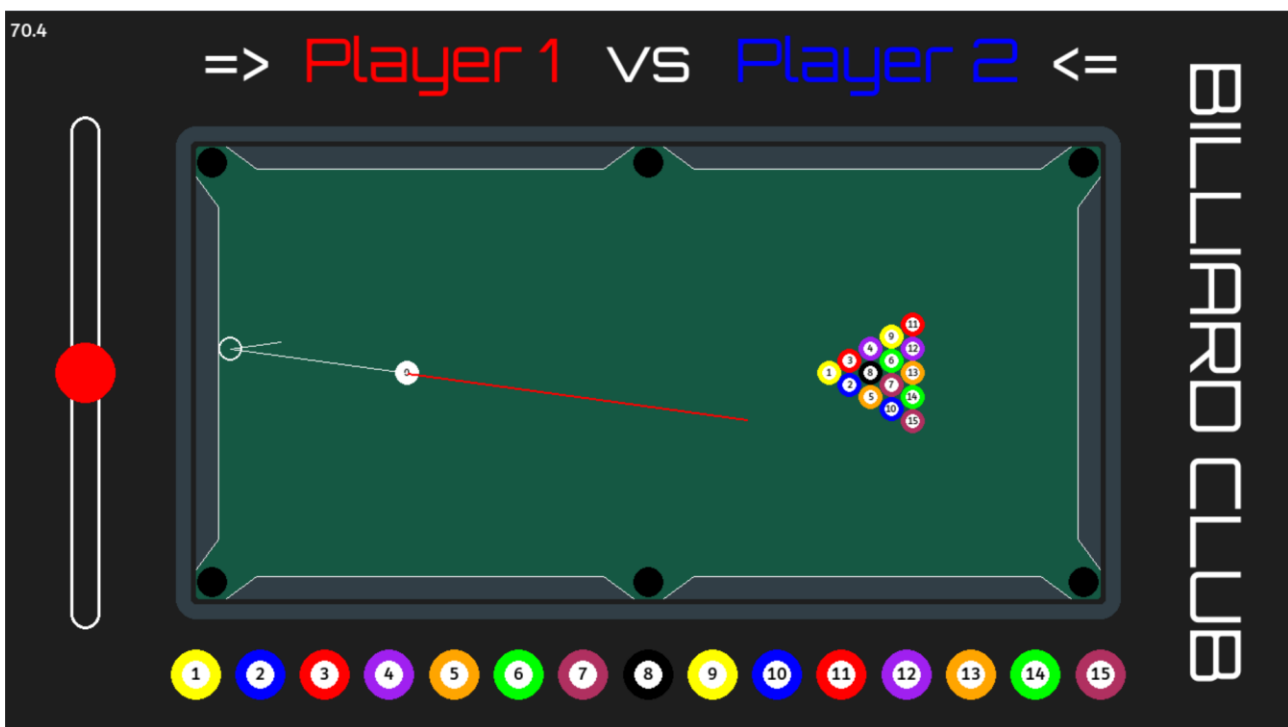

```
def drawPlayerIndicator(): # Draws the player indicator

    # render font
    player1Text = playerIndicatorFont.render(("Player 1"), True, "red")
    vsText = playerIndicatorFont.render(("vs"), True, "white")
    player2Text = playerIndicatorFont.render(("Player 2"), True, "blue")

    player1arrowText = playerIndicatorFont.render(("=>"), True, "white")
    player2arrowText = playerIndicatorFont.render(("<="), True, "white")

    # draw font to screen
    screen.blit(player1Text, (centre[0] - player1Text.get_width() - vsText.get_width(), 10))
    screen.blit(vsText, (centre[0] - vsText.get_width()/2, 10))
    screen.blit(player2Text, (centre[0] + vsText.get_width(), 10))

    screen.blit(player1arrowText, (centre[0] - player1Text.get_width() - vsText.get_width() - player1arrowText.get_width()*1.5, 10))
    screen.blit(player2arrowText, (centre[0] + player2Text.get_width() + vsText.get_width() + player2arrowText.get_width()/2, 10))
```



Now I will add a system to check the player's turn.

Added a function that switches the player turn, stored in a global variable.

```
def drawPlayerIndicator(): # Draws the player indicator

    # render font
    player1Text = playerIndicatorFont.render(("Player 1"), True, "red")
    vsText = playerIndicatorFont.render(("vs"), True, "white")
    player2Text = playerIndicatorFont.render(("Player 2"), True, "blue")

    player1arrowText = playerIndicatorFont.render(("=>"), True, "white")
    player2arrowText = playerIndicatorFont.render(("<="), True, "white")

    # draw font to screen
    screen.blit(player1Text, (centre[0] - player1Text.get_width() - vsText.get_width(), 10))
    screen.blit(vsText, (centre[0] - vsText.get_width()/2, 10))
    screen.blit(player2Text, (centre[0] + vsText.get_width(), 10))

    if player1Turn:
        screen.blit(player1arrowText, (centre[0] - player1Text.get_width() - vsText.get_width() - player1arrowText.get_width()*1.5, 10))
    else:
        screen.blit(player2arrowText, (centre[0] + player2Text.get_width() + vsText.get_width() + player2arrowText.get_width()/2, 10))

def changePlayerTurn(): # Changes the player turn
    player1Turn = not player1Turn
```

At the start of each turn, the variable will switch, causing the indicator to switch as well.

```

if not ballhasbeenstill and not checkIfBallMoving(): # if the balls have just been moving
    changePlayerTurn() # change the player turn
    ballhasbeenstill = True # balls are now still

if not checkIfBallMoving(): # when all the balls are still
    drawCue() # draw the cue
    if checkIfBallMoving(): # if the ball was hit
        ballhasbeenstill = False # when the balls are still next, change go.

```

This code works well.

I have successfully done this but upon reflection, I am going to implement a queue-like structure that will add all the functions that need to be done to a list, and then at the start of each turn, the queue will be emptied. This will hopefully allow me to determine the game state at the end of each turn.

I have decided to just use a simple list to avoid extra complexity than importing the queue module.

```

gameQueue = [] # Queue/List for storing events in between each turn.

```

A new function has been created called 'empty queue,' that will execute all the things in this list when called.

Here I have now implemented the queue into the game loop:

```

if not checkIfBallMoving(): # when all the balls are still

    # execute all functions/procedures in the queue
    for function in gameQueue:
        function() # execute the event
    gameQueue.clear() # clear the queue

    drawCue() # draw the cue and give velocity to the white ball if clicked

    # if the ball has been hit
    if checkIfBallMoving():
        gameQueue.append(changePlayerTurn) # change the turn when balls are next still

```

This code in the game loop checks if the balls are all still, then if they are, the game will execute everything in the queue. The game will then draw the cue and then after the white ball is hit, change player turn is put into the queue so that when the balls are next still, the game will change the player turn.

However, I run into an issue when running this code:

```

player1Turn = not player1Turn
^^^^^^^^^^^^
UnboundLocalError: cannot access local variable 'player1Turn' where it is not associated with

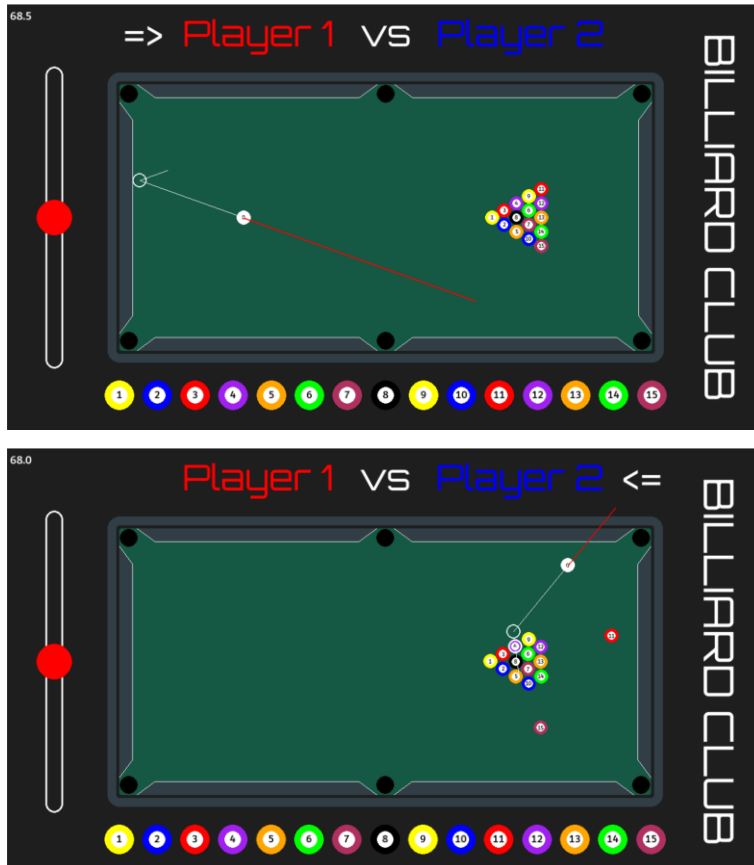
```

Because I have stored the state of the game as a local variable, the queue cannot access the value of the player turn.

This has been fixed by declaring the variable as a global variable here:

```
def changePlayerTurn(): # Changes the player turn
    global player1Turn
    player1Turn = not player1Turn
```

Now the game works well and changes turn every time the balls go still:



The next rule is that you get the balls that are striped or dotted when you first pot one of them.

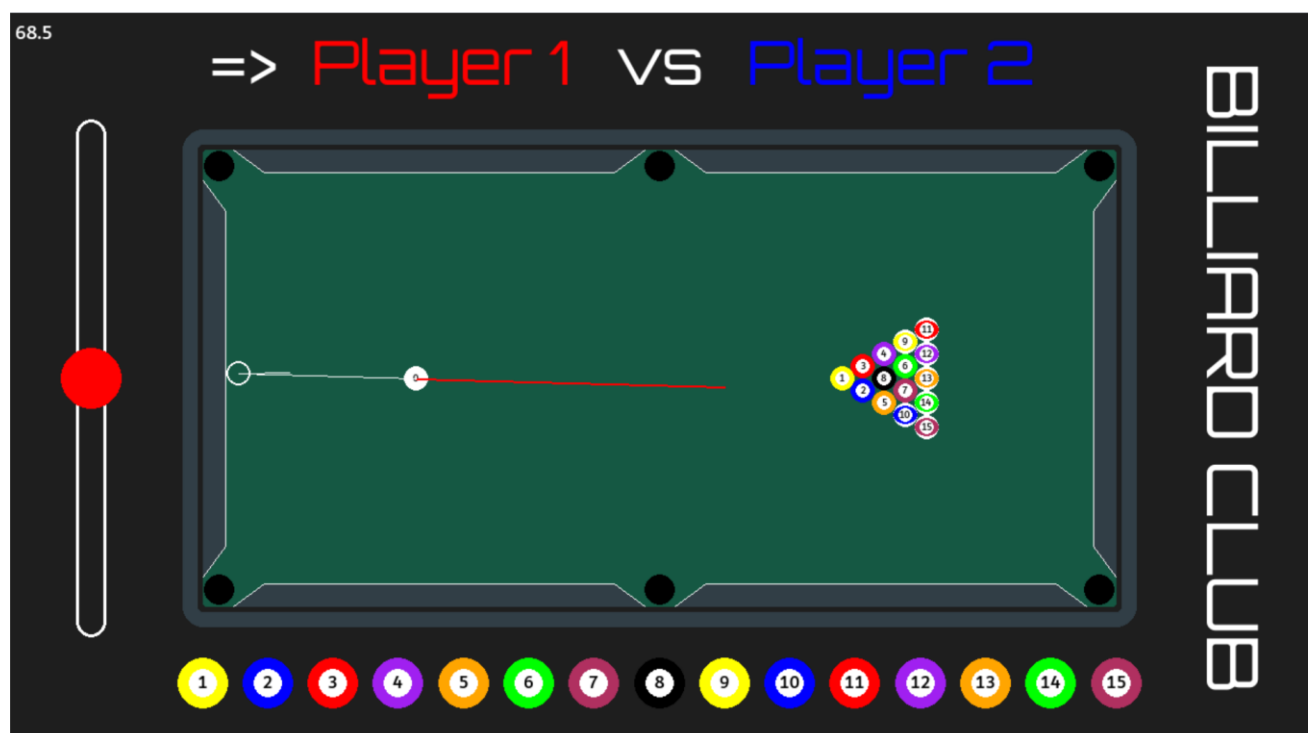
When deciding how I should complete this, I have realised I have not implemented appropriate graphics for the striped balls. This will be done now.

In the draw function of the ball class, I have added a check to see if the ball is striped. Therefore, when the ball is striped, I draw an ellipse to show the ball is striped.

```

if self.isStriped:
    pygame.draw.circle(screen, "white", self.pos, self.radius) # draws outer circle to ov
    # Draw the stripe
    stripe_height = ballRadius / 0.65 # height of the stripe
    pygame.draw.ellipse(
        screen,
        self.colour,
        (
            self.pos[0] - ballRadius,
            self.pos[1] - stripe_height // 2,
            ballRadius * 2,
            stripe_height
        )
    )
)

```



This works well for the playing balls, but the stripes have not carried across to the Ball UI at the bottom of the screen.

This is due to them being completely separate instances (there is no consistent change for both) Therefore, instead of trying to fix this and merge the two so they are congruent, I will save time by just copying code from one to the other.

This is me adding the same code in the drawBallUI() procedure:

```

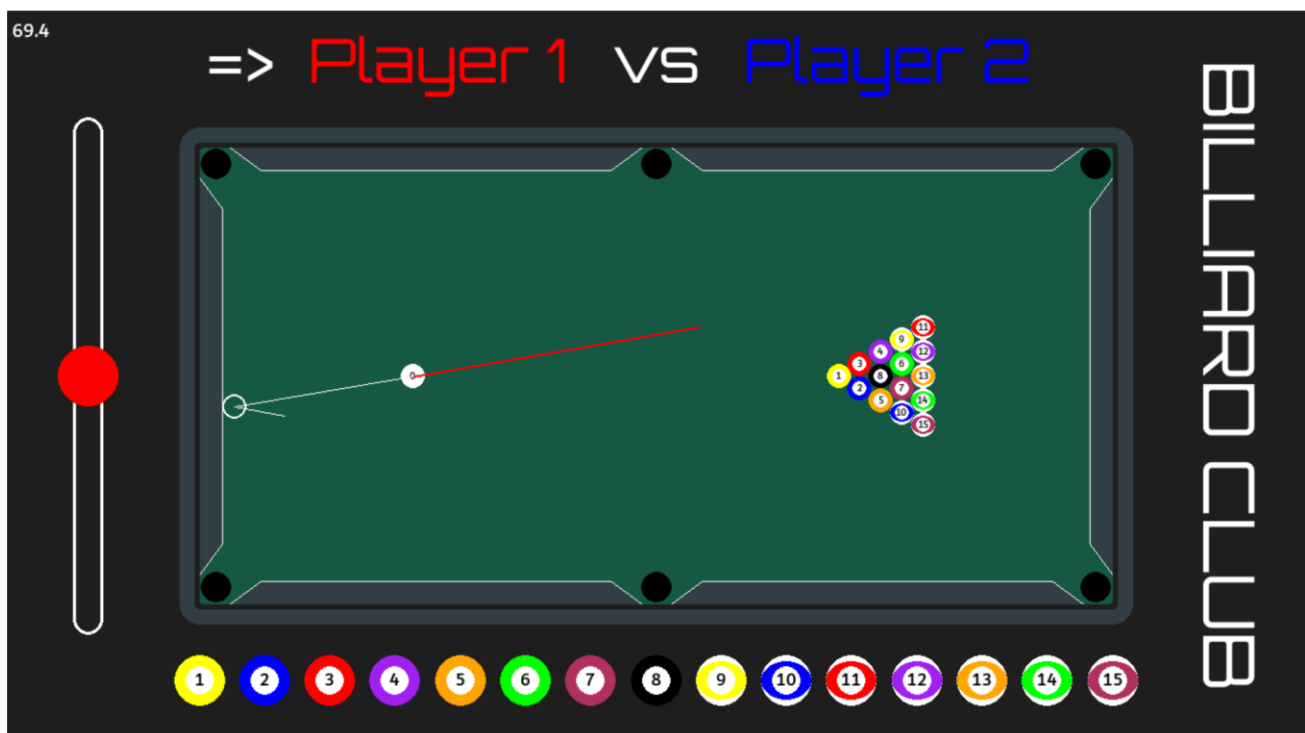
for i in range(len(ballInfo)-1): # loops through each ball except white ball
    circleCentre = (tableCorners[0][0] + increment, tableCorners[2][1]+75) # coordinates of each circle

    pygame.draw.circle(screen, ballInfo[i+1][0], circleCentre, largerBallRadius) # draws outer circle

    if ballInfo[i+1][2]:
        pygame.draw.circle(screen, "white", circleCentre, largerBallRadius) # draws outer circle to overlap stripe
        # Draw the stripe
        stripe_height = largerBallRadius / 0.65 # height of the stripe
        pygame.draw.ellipse(
            screen,
            ballInfo[i+1][0],
            (
                circleCentre[0] - largerBallRadius,
                circleCentre[1] - stripe_height // 2,
                largerBallRadius * 2,
                stripe_height
            )
        )

```

Now both of the sets of balls are drawing the same:



Next, I will address the white ball.

Previously, I set the white ball to be immune to being potted. Now I will fix this to make sure the white ball can be potted.

```

def resolveBallPotted(): # Checks if a ball is potted
    for ball in ballGroup.sprites():
        if ball.checkHoleCollision(): # checks if each ball is in a hole
            if ball.number == 0: # if the white ball is potted
                gameQueue.append(resetWhiteBall) # tells the game to reset the white ball
                balls[0].velocity = VEC(0,0) # resets the velocity of the white ball
                balls[0].pos = VEC(0,0) # resets the position of the white ball

```

Here I check if the white ball has been potted. If it has, I move it out the way of the board, and tell the cue that on the next round, the white ball's position needs to be reset.

This works. Can't prove this with a picture though.

Next is to assign the ball type to the player who pots a ball first.

Here I have set up initial conditions

```
player1BallType = None # Stores the type of balls player 1 is
player2BallType = None # Stores the type of balls player 2 is
```

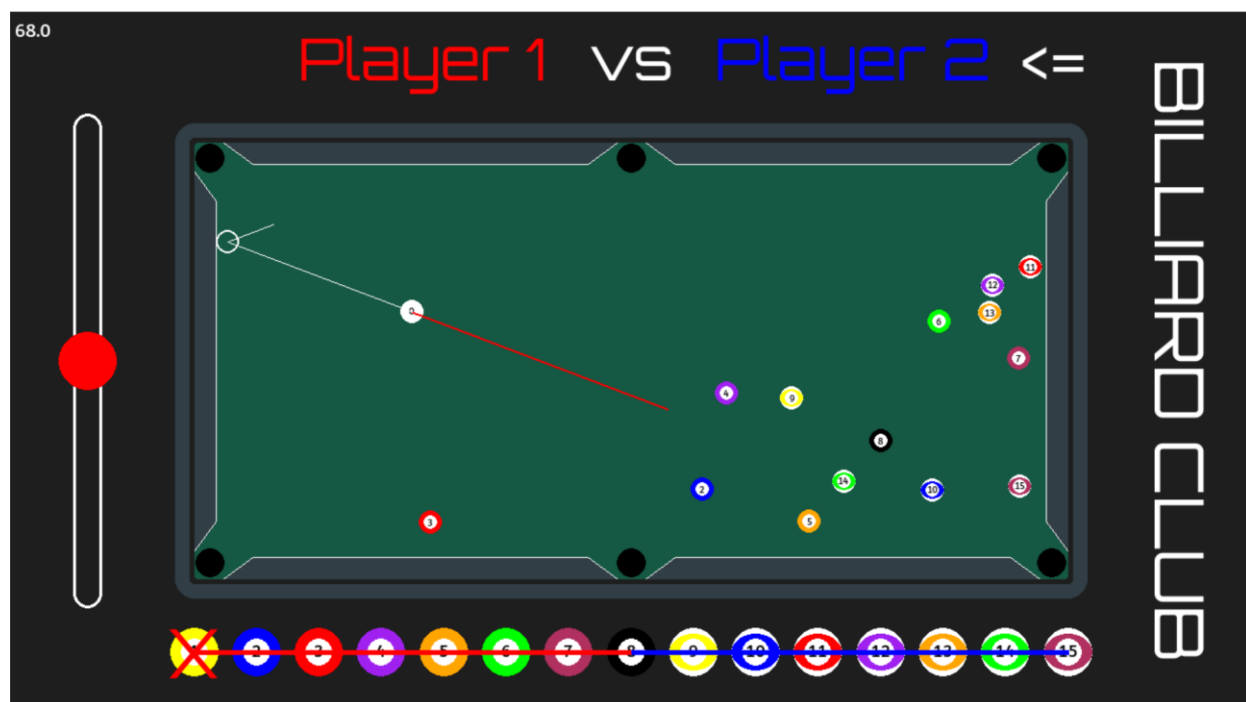
When any ball is potted the following will run which assigns the ball type to each.

```
# On first run, set the ball type for each player
global player1BallType, player2BallType
if player1BallType == None and player2BallType == None:
    if ball.isStriped and player1Turn:
        player1BallType = "striped"
        player2BallType = "coloured"
    else:
        player1BallType = "coloured"
        player2BallType = "striped"
```

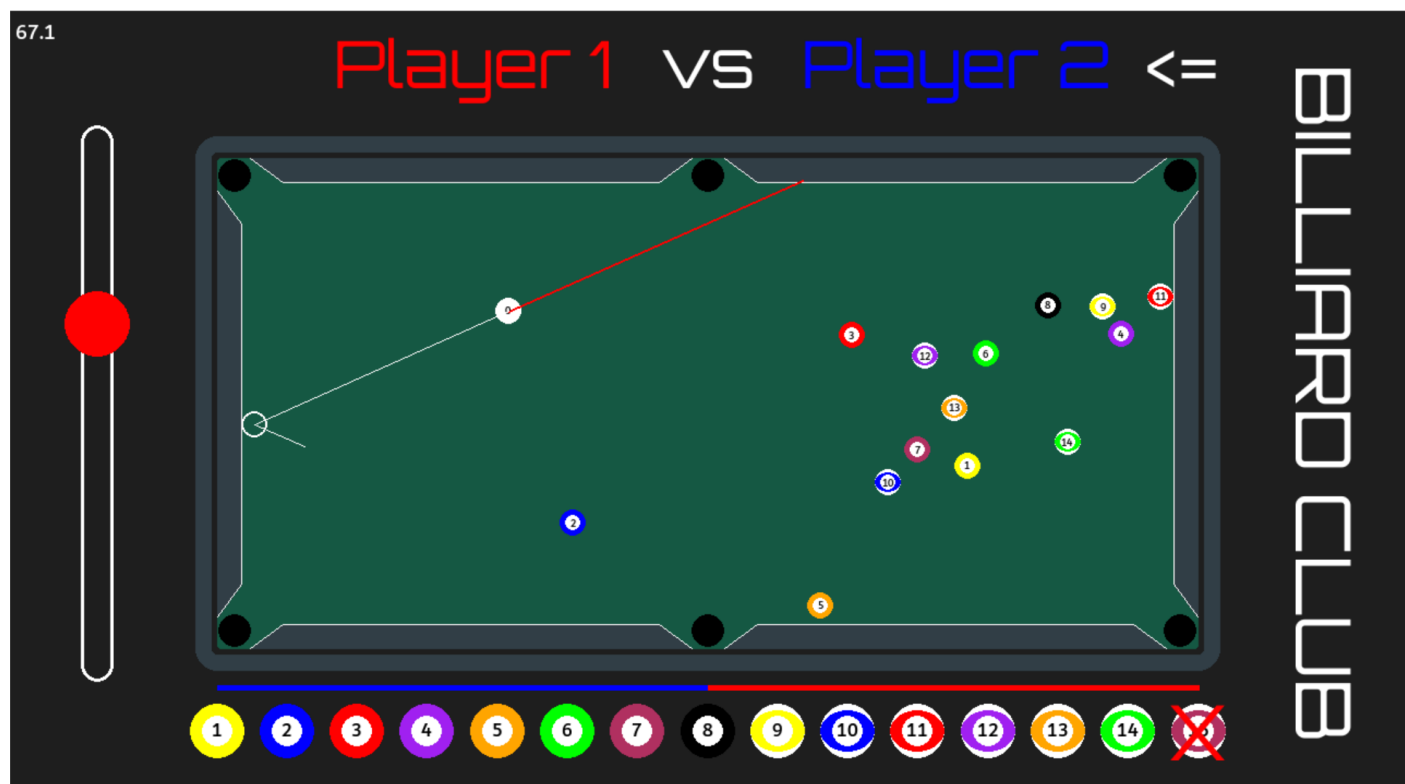
In order to test this, I will create a procedure that will draw a coloured line on the screen, indicating which ball type is each player's ball type.

```
if player1BallType != None and player2BallType != None: # if the ball types have
    if player1BallType == "coloured":
        pygame.draw.line(screen, "red", (tableCorners[0][0], tableCorners[2][1]),
        pygame.draw.line(screen, "blue", (tableCorners[3][0], tableCorners[2][1]),
    if player1BallType == "striped":
        pygame.draw.line(screen, "blue", (tableCorners[0][0], tableCorners[2][1]),
        pygame.draw.line(screen, "red", (tableCorners[3][0], tableCorners[2][1]),
```

Which produces the following:



This is a problem, but it will be fixed by adjusting the position of the line



I would like this to be more obvious and therefore I will add another line and centre all the lines.

Here I have worked on just on the instance where player 1 gets the dotted balls and therefore have set player1balltype to always be coloured.

```

line1start = (tableCorners[0][0] - largerBallRadius, circleCentre[1] - largerBallRadius - 10)
line1end = (centre[0] - distanceBetweenBalls + largerBallRadius, circleCentre[1] - largerBallRadius - 10)

line2start = (tableCorners[3][0] + largerBallRadius, circleCentre[1] - largerBallRadius - 10)
line2end = (centre[0] + distanceBetweenBalls - largerBallRadius, circleCentre[1] - largerBallRadius - 10)

line3start = (tableCorners[0][0] - largerBallRadius, circleCentre[1] + largerBallRadius + 10)
line3end = (centre[0] - distanceBetweenBalls + largerBallRadius, circleCentre[1] + largerBallRadius + 10)

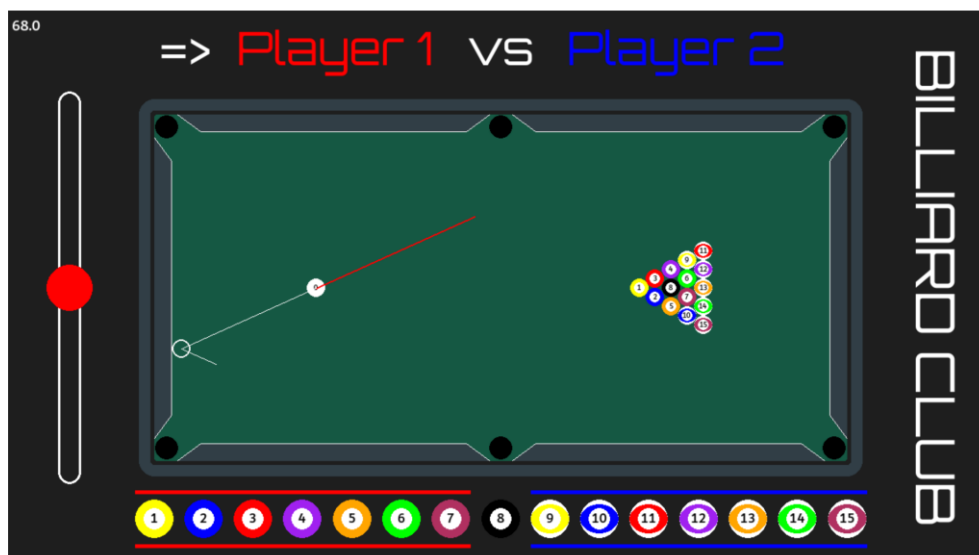
line4start = (tableCorners[3][0] + largerBallRadius, circleCentre[1] + largerBallRadius + 10)
line4end = (centre[0] + distanceBetweenBalls - largerBallRadius, circleCentre[1] + largerBallRadius + 10)

player1BallType = "coloured"
if player1BallType != None or player2BallType != None: # if the ball types have been set
    if player1BallType == "coloured":
        pygame.draw.line(screen, "red", line1start, line1end, 5) # draw lines of player 1
        pygame.draw.line(screen, "red", line3start, line3end, 5) # draw lines of player 1

        pygame.draw.line(screen, "blue", line2start, line2end, 5) # draw line of player 2
        pygame.draw.line(screen, "blue", line4start, line4end, 5) # draw line of player 2

```

This code produces the correct response.



Here I add both instances:

```

line1start = (tableCorners[0][0] - largerBallRadius, circleCentre[1] - largerBallRadius - 10)
line1end = (centre[0] - distanceBetweenBalls + largerBallRadius, circleCentre[1] - largerBallRadius - 10)

line2start = (tableCorners[3][0] + largerBallRadius, circleCentre[1] - largerBallRadius - 10)
line2end = (centre[0] + distanceBetweenBalls - largerBallRadius, circleCentre[1] - largerBallRadius - 10)

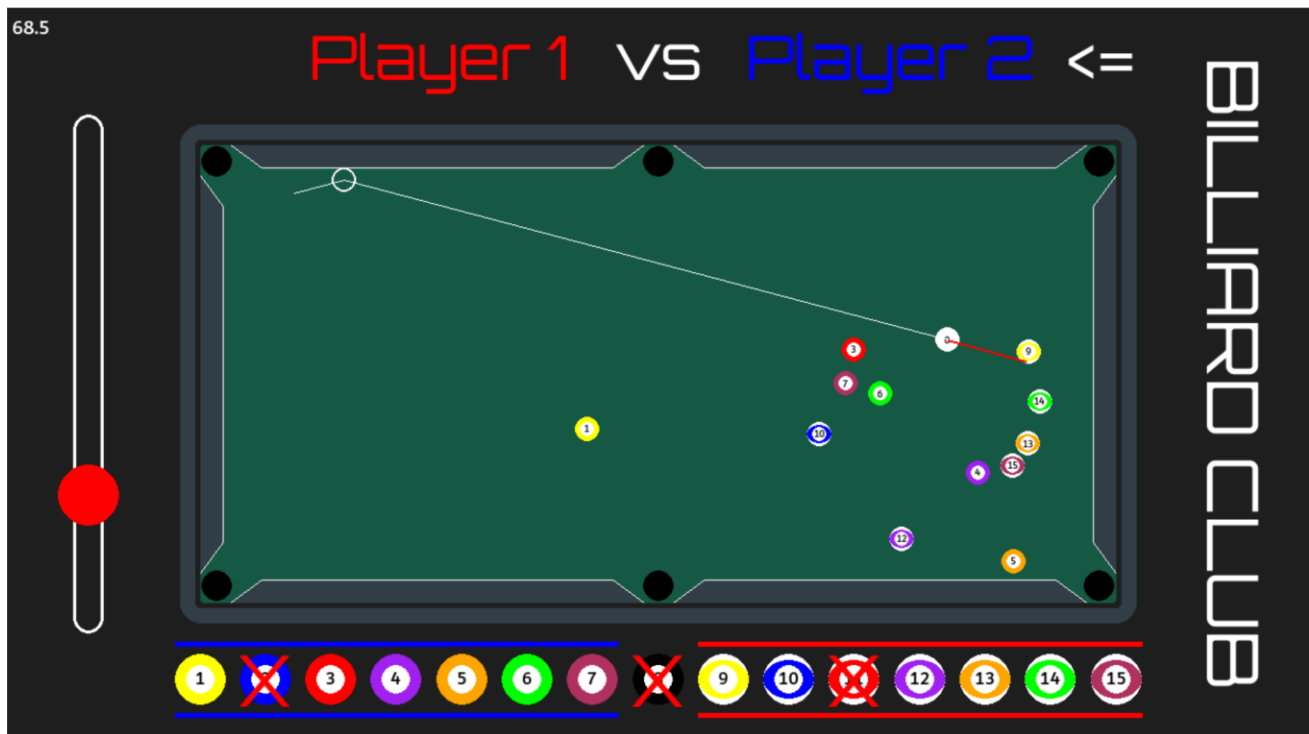
line3start = (tableCorners[0][0] - largerBallRadius, circleCentre[1] + largerBallRadius + 10)
line3end = (centre[0] - distanceBetweenBalls + largerBallRadius, circleCentre[1] + largerBallRadius + 10)

line4start = (tableCorners[3][0] + largerBallRadius, circleCentre[1] + largerBallRadius + 10)
line4end = (centre[0] + distanceBetweenBalls - largerBallRadius, circleCentre[1] + largerBallRadius + 10)

player1BallType = "striped" # sets instance (remove this after)

if player1BallType == "coloured":
    pygame.draw.line(screen, "red", line1start, line1end, 5) # draw lines of player 1
    pygame.draw.line(screen, "red", line3start, line3end, 5) # draw lines of player 1
    pygame.draw.line(screen, "blue", line2start, line2end, 5) # draw line of player 2
    pygame.draw.line(screen, "blue", line4start, line4end, 5) # draw line of player 2
elif player1BallType == "striped":
    pygame.draw.line(screen, "blue", line1start, line1end, 5) # draw lines of player 2
    pygame.draw.line(screen, "blue", line3start, line3end, 5) # draw lines of player 2
    pygame.draw.line(screen, "red", line2start, line2end, 5) # draw line of player 1
    pygame.draw.line(screen, "red", line4start, line4end, 5) # draw line of player 1

```

And after removing the testing line:

```
player1BallType = "striped" # sets instance (remove this after)
```

The game works very well.

Now I will add the game logic that checks the if the player has potted one of their own balls, and if they have given them another go.

```
pottedBalls = [] # List for storing potted balls in whole game
roundPottedBalls = [] # List for storing potted balls in each round
```

Here I have added two variables that will store every ball that has been potted, and which balls have been potted in that round.

Upon more thought, I have decided to move the change player turn from inside the game queue and create a separate function that will run at the end of every round and determine the outcome of the round.

I have removed the procedure changeplayerturn and implemented it into the procedure game logic which will determine all the outcomes of the round in one procedure.

```
# if the ball has been hit
if checkIfBallMoving():
    gameQueue.append(gameLogic) # when the balls are next still, apply game
```

```
def gameLogic(): # determines state of next round on all factors
    global player1Turn
    global pottedBalls
    global roundPottedBalls
    global gameQueue

    if len(pottedBalls) == 0: # if no balls are potted
        player1Turn = not player1Turn # switch turns
```

This now works as before.

Now I need to implement a check to see if the player has potted their own ball and if they have, don't change the go.

To start I have made a few changes here:

```
def resolveBallPotted(): # Checks if a ball is potted
    for ball in ballGroup.sprites():
        if ball.checkHoleCollision(): # checks if each ball is in a hole
            if ball.number == 0: # if the white ball is potted
                gameQueue.append(resetWhiteBall) # tells the game to reset the white ball on the start of the round
                balls[0].velocity = VEC(0,0) # resets the velocity of the white ball
                balls[0].pos = VEC(0,0) # resets the position of the white ball
            else:
                # On first run, set the ball type for each player
                global player1BallType, player2BallType
                if player1BallType == None and player2BallType == None:
                    if ball.isStriped and player1Turn or not ball.isStriped and not player1Turn:
                        player1BallType = "striped"
                        player2BallType = "coloured"
                    elif ball.isStriped and not player1Turn or not ball.isStriped and player1Turn:
                        player1BallType = "coloured"
                        player2BallType = "striped"
                    else:
                        print("Error: Ball type not set")

                pottedBallInfo = [ball.number, ball.isStriped] # collects information about the potted ball

                pottedBalls.append(pottedBallInfo) # adds the ball number to the list of potted balls
                roundPottedBalls.append(pottedBallInfo) # adds the ball number to the list of the round's potted balls

                ball.delete() # removes the ball from the game

    ball.draw(screen) # draws each ball
```

I have collected all the information about the potted ball and then returned it through the lists created: pottedBalls and roundPottedBalls.

I have also explicitly put the conditions for what ball potted triggers which type of ball the player has.

Here I have added the conditions:

```
def gameLogic(): # determines state of next round on all factors
    global player1Turn
    global player1BallType
    global pottedBalls
    global roundPottedBalls
    global gameQueue

    changeTurn = False # set initial case to False

    if len(pottedBalls) == 0: # if no balls are potted
        changeTurn = True # switch turns
    else:
        for i in roundPottedBalls: # loop through each ball potted
            number = i[0]
            isStriped = i[1]

            if player1Turn: # if it is player 1's turn
                if isStriped and player1BallType == "coloured" or not isStriped and player1BallType == "striped": # if the ball is not the correct type
                    changeTurn = True # switch turns
            if not player1Turn: # if it is player 2's turn
                if isStriped and player1BallType == "striped" or not isStriped and player1BallType == "coloured": # if the ball is not the correct type
                    changeTurn = True # switch turns

    if changeTurn: # if the turn is to be changed
        player1Turn = not player1Turn # switch turns

    pottedBalls = [] # resets the list of potted balls
```

However, on playing the game, the computer switched the turns, when I potted a ball of my type.

This has been fixed by amending the code as shown here:

```
def gameLogic(): # determines state of next round on all factors
    global player1Turn
    global player1BallType
    global player2BallType
    global pottedBalls
    global roundPottedBalls
    global gameQueue

    changeTurn = False # set initial case to False

    if len(roundPottedBalls) == 0: # if no balls are potted
        changeTurn = True # switch turns
    else:
        for i in roundPottedBalls: # loop through each ball potted
            number = i[0]
            isStriped = i[1]

            if player1Turn: # if it is player 1's turn
                if player1BallType == "dotted": # if player 1 type is dotted
                    if isStriped: # if the ball is striped
                        changeTurn = True
                elif player1BallType == "striped": # if player 1 type is striped
                    if not isStriped: # if the ball is dotted
                        changeTurn = True
            if not player1Turn: # if it is player 2's turn
                if player2BallType == "dotted": # if player 2 type is dotted
                    if isStriped: # if the ball is striped
                        changeTurn = True
                elif player2BallType == "striped": # if player 2 type is striped
                    if not isStriped: # if the ball is dotted
                        changeTurn = True

    if changeTurn: # if the turn is to be changed
        player1Turn = not player1Turn # switch turns

    roundPottedBalls = [] # resets the list of potted balls
```

The game now functions correctly.

I would like to add a graphic to the drawCue() function that firstly changes the cue colour and also tells the user which ball to hit.

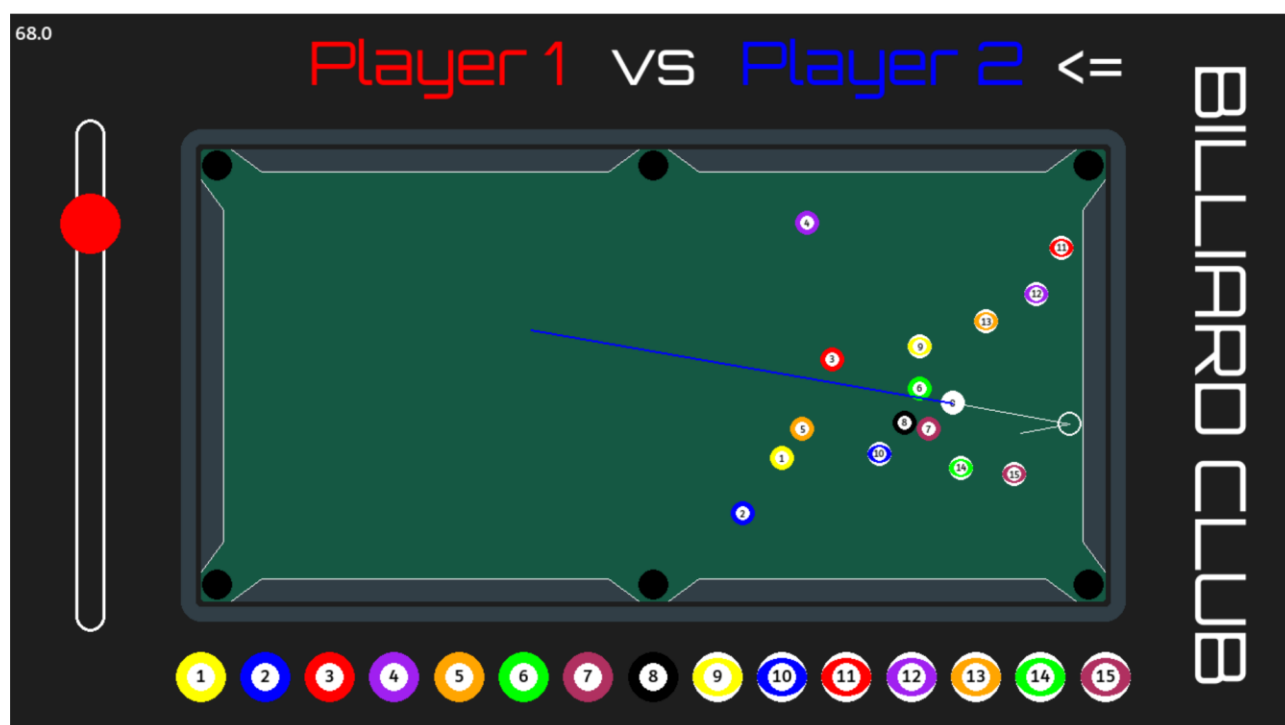
Changing the colour of the cue based on which player's turn it is:

```
player1Colour = "red"
player2Colour = "blue"
```

```
cueColour = "" # set colour of cue
if player1Turn:
    cueColour = player1Colour
else:
    cueColour = player2Colour

pygame.draw.line(screen, cueColour, balls[0].pos, pygame.mouse.get_pos(), 2) # draw cue
```

This works as shown:



Telling the user which ball to hit:

Added this block of code in the drawCue() procedure that will change the colour of the virtual ball to help the user distinguish between what type of ball the player should hit:

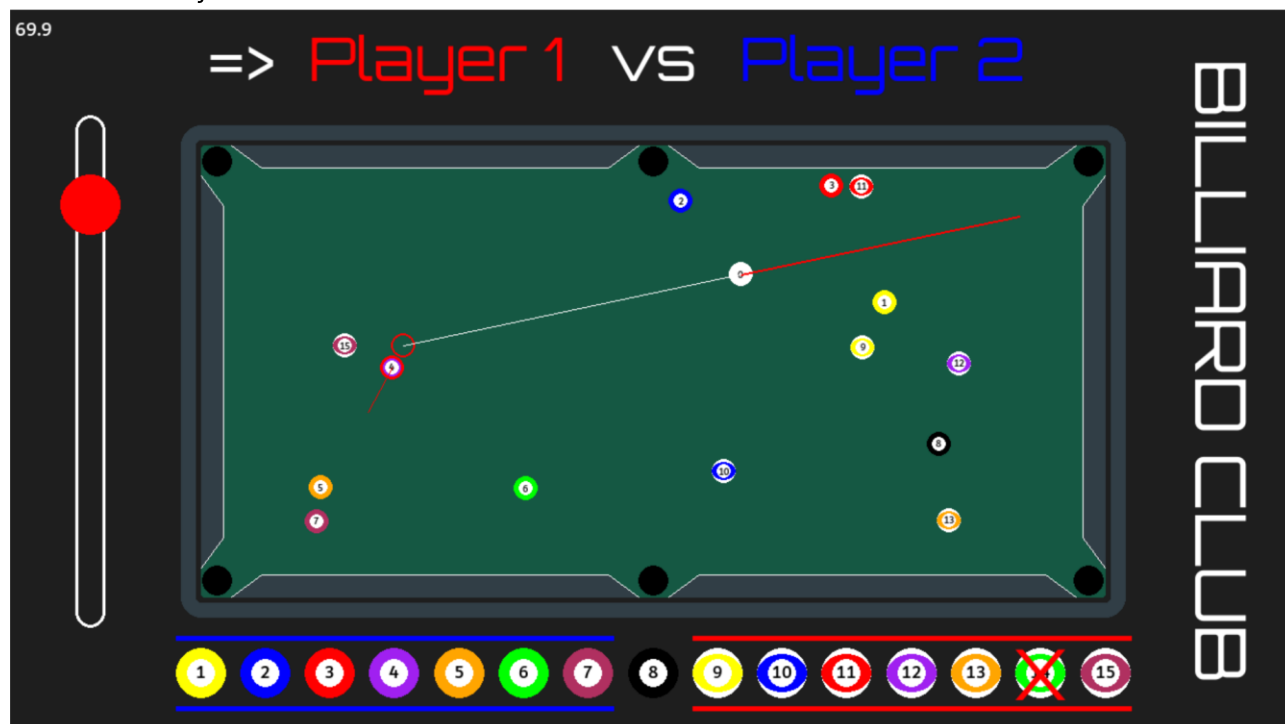
```

virtualBallColour = "white" # set colour of virtual ball
if collidingWithBall:
    if player1Turn: # if it is player 1's turn
        if player1BallType == "dotted": # if player 1 type is dotted
            if closestBall.isStriped: # if the ball is striped
                virtualBallColour = "red" # set colour to red
            elif player1BallType == "striped": # if player 1 type is striped
                if not closestBall.isStriped: # if the ball is dotted
                    virtualBallColour = "red" # set colour to red
        if not player1Turn: # if it is player 2's turn
            if player2BallType == "dotted": # if player 2 type is dotted
                if closestBall.isStriped: # if the ball is striped
                    virtualBallColour = "red" # set colour to red
            elif player2BallType == "striped": # if player 2 type is striped
                if not closestBall.isStriped: # if the ball is dotted
                    virtualBallColour = "red" # set colour to red

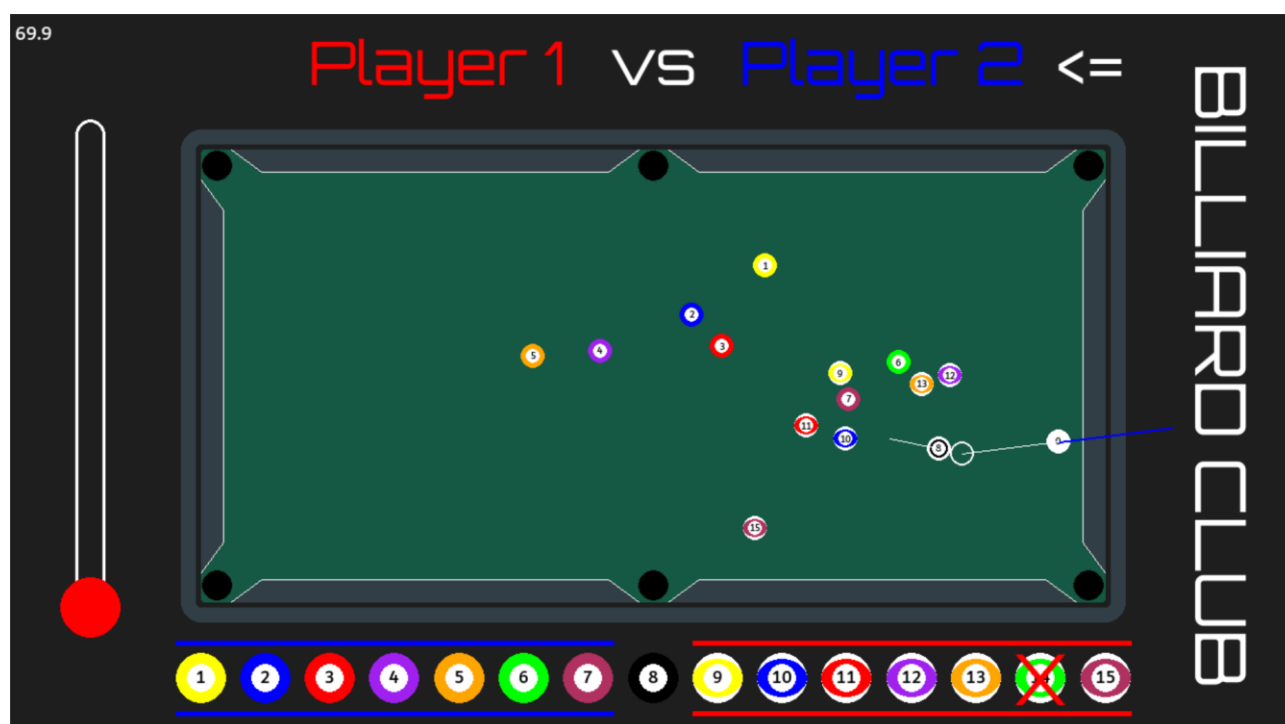
pygame.draw.line(screen, cueColour, balls[0].pos, pygame.mouse.get_pos(), 2) # draw cue
pygame.draw.circle(screen, virtualBallColour, virtualBallPos, ballRadius, 2) # draw virtual ball
pygame.draw.line(screen, "white", balls[0].pos, virtualBallPos) # draw projection

```

This works very well as shown below:



But, when I hover over the 8 ball, it appears red for player 1, but white for player 2.

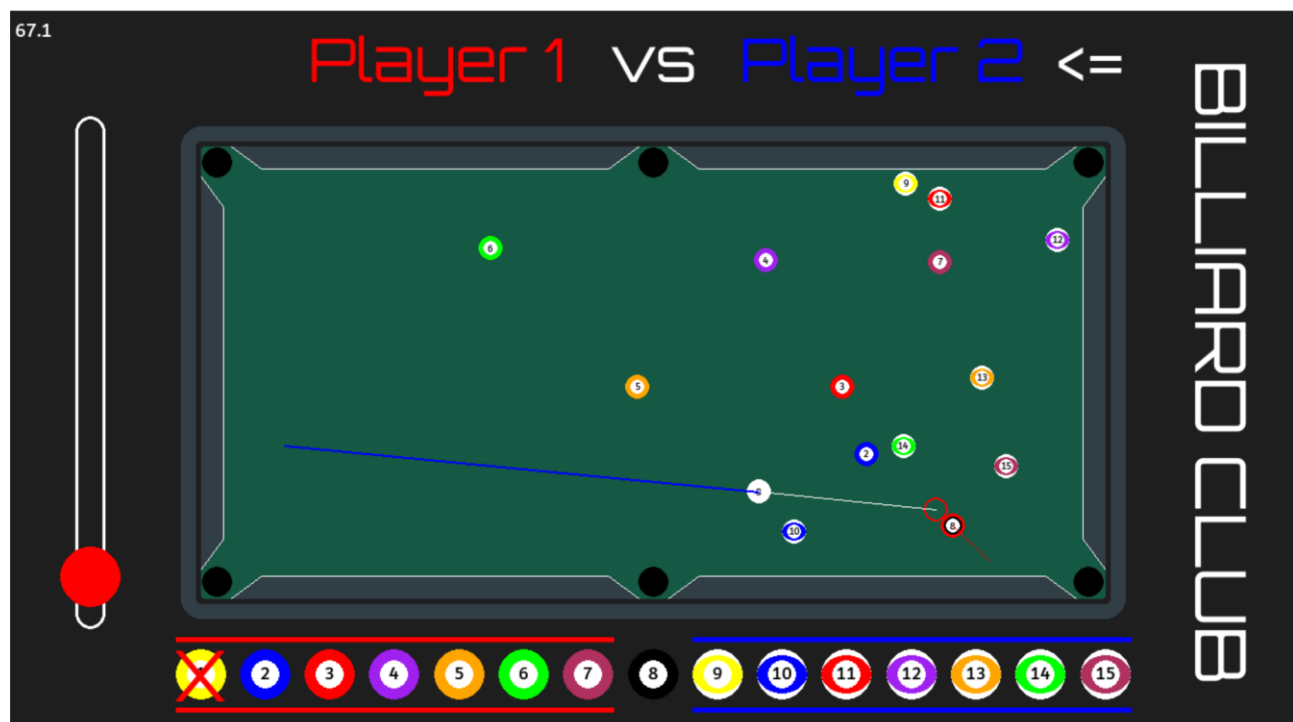
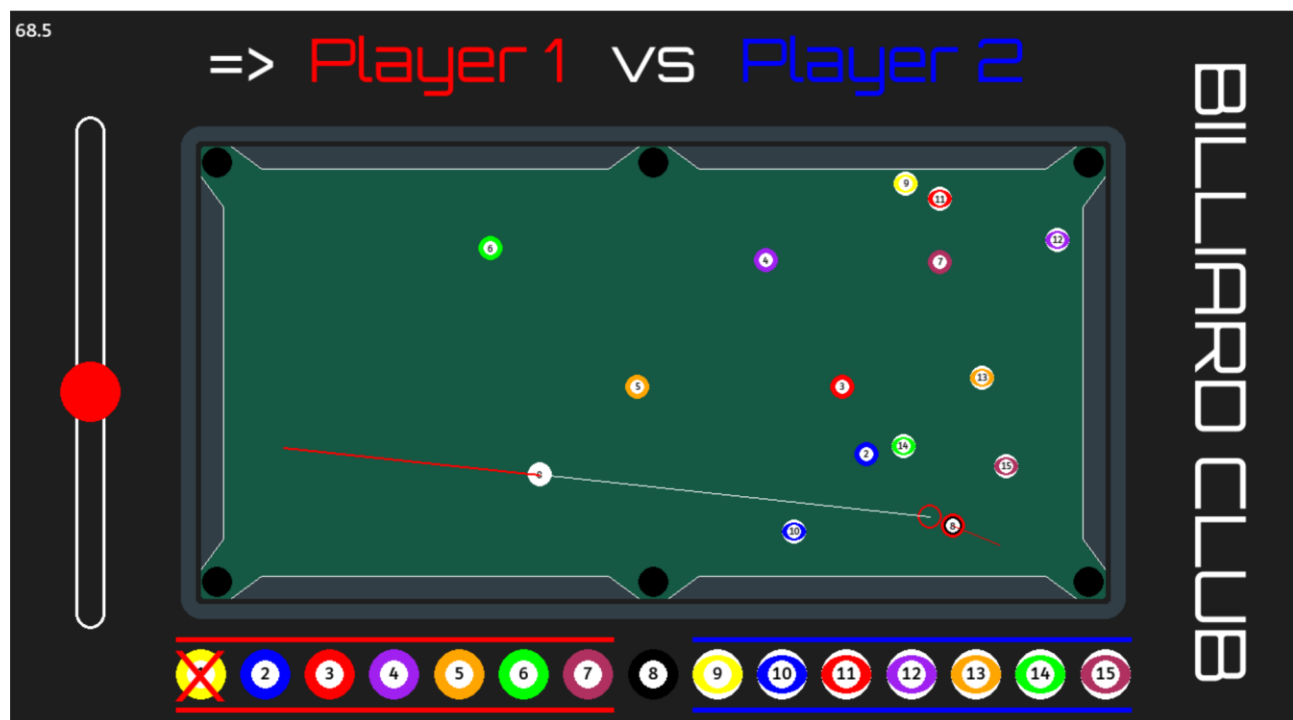


This is likely due to the value of “isStriped” for the 8 ball is set to “None”, which has been equated to False.

This just means I need to add a check for “None” before using Boolean algebra to check if the correct ball is potted.

```
if closestBall.number == 8: # if the ball is the 8 ball
    virtualBallColour = "red" # set colour to black
```

Here I have set the colour to always be red.



This is now working correctly.

Now all that is left is to determine winning conditions.

Here I have added a new Boolean variable called eightBallPotted.

```
if number == 8: # if the 8 ball is potted
    eightBallPotted = True # set case to True
```

Declared a new variable, winner

```
winner = None
```

```
if eightBallPotted: # if the 8 ball is potted
    # check if all other balls are potted.
    dottedPotted = 0
    stripedPotted = 0
    # count the number of potted balls from each type
    for i in roundPottedBalls:
        if i[1]:
            stripedPotted += 1
        else:
            dottedPotted += 1

    if dottedPotted == 7 or stripedPotted == 7:
        # whoever potted the 8 ball wins
        if player1Turn:
            winner = "Player 1"
        else:
            winner = "Player 2"
    else:
        # whoever potted the 8 ball loses
        if player1Turn:
            winner = "Player 2"
        else:
            winner = "Player 1"
```

```
def checkWin(winner): # checks if there is a winner
    if winner != None: # if there is a winner
        winText = logoFont.render(winner + " wins!", True, "white") # renders the font
        winTextPos = [centre[0] - winText.get_width()/2, centre[1] - winText.get_height()/2] # offsets the font to be central
        screen.blit(winText, winTextPos) # adds font to screen
```

```
drawBallUI() # draws the ball UI (15 balls that tells the user which ball to pot next)
drawLogo() # draws the logo
drawFPS() # draws the FPS
drawPlayerIndicator() # draws the player indicator
checkWin(winner) # checks if there is a winner and draws it to the screen
```

As player 1, I have potted the 8 ball, therefore player 2 should win and does, as shown below:



However, when I have played a full game, when player 2 pots the final ball, it shows that player 1 has won.



This has been fixed here:

```
for i in roundPottedBalls:
```

```
for i in pottedBalls:
```

In the gameLogic() loop, I realised that it was looping through the wrong list. The one it should have been looping through was the second list “pottedBalls.” This meant the list was being reset after every round, leading to incorrect results.

Here I had two lists which store the balls potted in one round and the balls potted in all the rounds. I had used one instead of the other which meant that the wrong player was winning.



The game is now working very nicely, and all the relevant rules have been implemented.

Stage 4 Review

What has been done?

- Organised the code by grouping related functionalities into lots of procedures. I have also used the computational method of abstraction to improve the readability and maintainability of the code. This was done by removing redundant information.
- Created more UI elements to indicate the active player's turn, resolved text alignment issues, and added more coloured indicators for each player on the lower screen.
- Introduced a queue structure to manage actions executed at the end of each turn.
- Enhanced ball graphics by implementing striped ball visuals in the game and UI.
- Implemented relevant code to assign ball types (striped or dotted) to players based on the first potted ball.
- Finally, I added winning conditions based on the potting of the 8 ball.

How is code being tested?

- For each new element/function (e.g., ball movement, wall collisions, ball-to-ball collisions), I add a small bit of functionality and then observe the result, documenting and debugging it accordingly. This incremental approach ensures I follow the computational method of decomposition which ensures for swift development of the solution.
- Bugs (like clipping, stuck balls, or zero-distance errors) are identified and resolved by modifying code of the program, rerunning the simulation, and documenting my errors.
- Comments are used extensively throughout the solution. This helps debugging enormously as I sift through hundreds of lines of code.

Success criteria that has been met:

9	The player should be able to play the game against another player.
13	The game should display the winner after every game.
15	The game should display which player is playing and who has the next turn.
16	The game should display which balls should be potted by each player.

Errors that have been fixed:

Error	Fix
When setting up the two-player mode, the text indicating whose turn it is was not centred correctly.	I adjusted the text so that it was displayed in the corrected position.
The player turn state was stored as a local variable, making it inaccessible to the queue function.	I changed the variable storing the player turn to a global variable. This allowed the queue to correctly access and update the game state.

The ball UI did not draw the striped balls.	The code for drawing the stripes was copied from the ball's draw function into drawBallUI() procedure. This meant that the striped graphic were consistent with the table balls graphic.
Initially, the white ball was set to be immune to being potted, which conflicted with the game rules requiring it to be potable.	The code was modified so that when the white ball could be potted.
When assigning ball types to players upon potting a ball, the initial test did not display correctly.	I adjusted the positioning of the indicator line to the centre of the display, making it clearer which ball type was assigned to each player.
When hovering over the 8 ball, the cue appeared red for player 1 but white for player 2.	The code was amended to always set the 8 ball's cue-indicator colour to red, ensuring consistency regardless of the player.
When player 2 potted the final ball, the game incorrectly declared player 1 as the winner.	The game logic was updated to iterate over the correct list (pottedBalls) that stores all the balls potted across rounds.

Summary of Stage 4

The Stage 4 development has involved refactoring the code for maintainability by grouping related functionalities into procedures. I have implemented player turn mechanics with a queue structure and also added UI indicators which enhance ball graphics. I have also implemented code to assign ball types and added the relevant code for winning conditions.

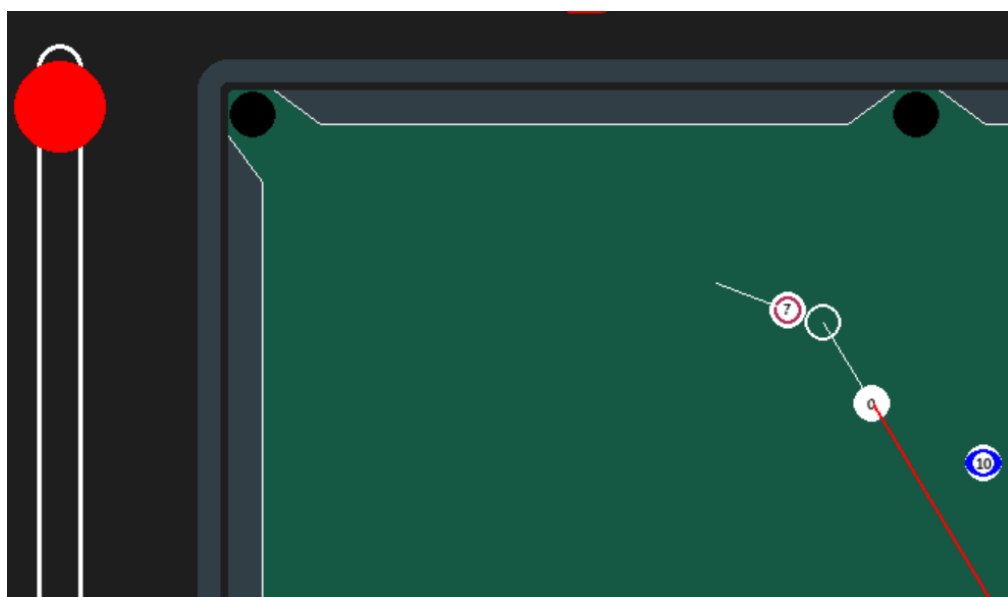
Stage 5: AI Development

Before creating a computer to play the game, I am going to resolve an issue with how the physics are calculated at higher speeds.

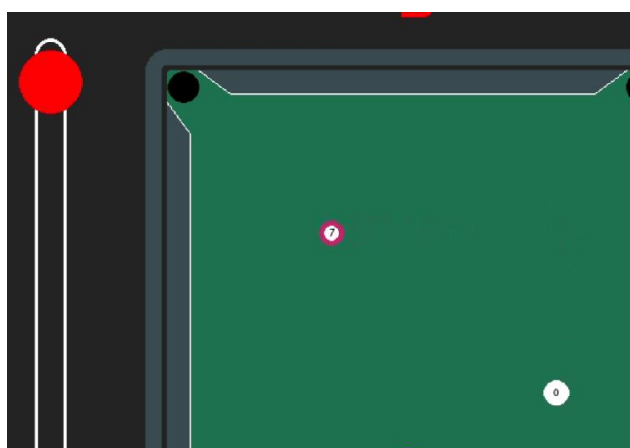
Every frame, the code will update moving balls by applying a vector transformation. However, if this vector transformation directly collides with a ball, the collision will be calculated there and then, with the balls colliding. This therefore results in inaccurate collisions at higher speeds.

To solve this problem, I will implement a similar method to that of calculating the virtual ball. When a collision is calculated, I will repeatedly move the ball back so that the balls are always tangent to each other before calculating the collision. This will be done in both the ball-to-wall collision method and the ball-to-ball method.

Display of the problem:



In the image above, the ball is projected to land somewhere near the hole. However, on simulation:



The ball pretty much moves parallel to the table edge and ends up nowhere near where the projection stated.

There are two functions that I need to fix here, one that resolves wall collisions and the other that resolves ball-to-ball collisions.

The wall collisions have been fixed here:

In the code below, I have moved the distance calculation into an inner function which allows me to call this function repeatedly. This means that when a collision occurs, I incrementally move back the ball and then check if the ball is still colliding. After this, the correct collision is calculated, at the exact point of intersection (where the ball is tangent to the wall).

```
def checkResolveWallCollision(self): # Check if balls are colliding with the walls, then resolves collision
    def checkColliding():
        # Loops through each of the Rails (where collisions should be done)
        for i in tableRails:
            for j in range(3):
                lineStart = tableRails[i][j] # Coordinates of the Line Start and End
                lineEnd = tableRails[i][j+1] # e.g. [ [x1,y1] , [x2,y2] ]
                lineVec = VEC(lineEnd[0] - lineStart[0], lineEnd[1] - lineStart[1]) # Vector of the Line
                circleVec = VEC(self.pos[0] - lineStart[0], self.pos[1] - lineStart[1]) # Vector of the circle to the start of the line
                t = max(0, min(1, (circleVec[0] * lineVec[0] + circleVec[1] * lineVec[1]) / lineVec.magnitude_squared()))
                # This creates a parameter that will calculate the s.f of the normal projection that maps the circle onto the line segment
                linePoint = (lineStart[0] + t * lineVec[0], lineStart[1] + t * lineVec[1]) # Calculate closest point on the line to the circle
                circlePointVec = VEC(linePoint[0] - self.pos[0], linePoint[1] - self.pos[1]) # Vector of the circle to the closest point
                distance = circlePointVec.magnitude() # calculates distance between the circle and closest point
                # Check if colliding
                if distance <= self.radius:
                    return True, circlePointVec
            return False, None

        calculateCollision = False
        isColliding, circlePointVec = checkColliding() # checks if colliding

        if isColliding: # if colliding
            calculateCollision = True # calculate collision at the end
            n = circlePointVec.normalize() # creates a normal vector as a origin for calculations

            while isColliding:
                self.pos -= self.velocity.normalize() # incrementally move the ball back until it is no longer colliding
                isColliding, circlePointVec = checkColliding() # checks if colliding

        # Check if colliding
        if calculateCollision:
            impulse = 2*(n*self.velocity)/(self.mass) # calculates the impulse the ball will get
            self.velocity -= impulse * n # applies impulse to each ball
            self.velocity *= restitutionCoeff # decreases velocity
```

Now I will amend the ball-to-ball collisions here:

```
def checkResolveCollision(self, other): # Check if balls are colliding, then resolves collision
    # Check if balls colliding
    def checkColliding(self, other):
        posDifference = self.pos - other.pos # find difference in positions
        distance = self.pos.distance_to(other.pos) # distance squared between points
        if distance == 0: # prevents normalisation error
            return False, posDifference
        elif distance < (self.radius + other.radius): # If colliding
            return True, posDifference
        return False, posDifference

    colliding, posDifference = checkColliding(self, other) # checks if colliding
    calculateCollision = False # set initial case to False

    if colliding:
        calculateCollision = True # calculate collision at the end
        self.pos -= self.velocity # step back position of both balls
        other.pos -= other.velocity # step back position of both balls
        colliding, posDifference = checkColliding(self, other) # checks if colliding
        while not colliding: # while colliding
            # incrementally move the balls forward until they are tangential
            self.pos += 0.05 * self.velocity
            other.pos += 0.05 * other.velocity
            colliding, posDifference = checkColliding(self, other) # checks if colliding
```

```

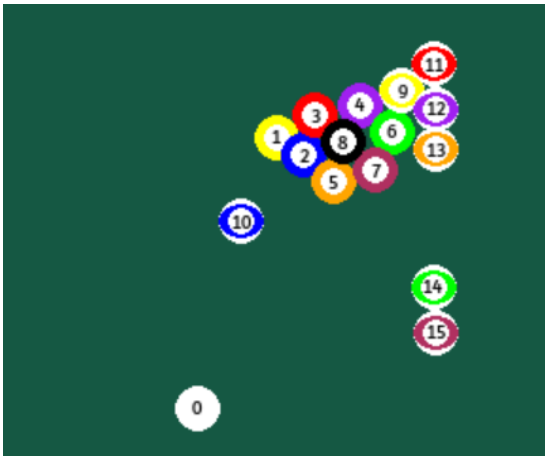
if calculateCollision: # if colliding
    # Calculate Velocities after collision
    n = VEC(posDifference).normalize() # creates a normalised vector as a origin for calculations
    relVel = self.velocity - other.velocity # calculates the relative velocity of the balls
    impulse = 2 * (n * relVel) / (self.mass + other.mass) # calculates the impulse each ball will get

    self.velocity -= impulse * other.mass * n # applies impulse to each ball
    other.velocity += impulse * self.mass * n

    self.velocity *= restitutionCoeff # decreases velocity of both balls
    other.velocity *= restitutionCoeff

```

However, this does not work when the two balls colliding have a velocity of zero. This means that they stay clipped with each other shown here:



This is fixed by adding this block of code to push balls out if they are still colliding when a collision has been calculated:

```

while colliding: # if still colliding
    # push balls apart to prevent sticking
    self.pos += n
    other.pos -= n
    colliding, posDifference = checkColliding(self, other) # checks if still colliding

```

Balls are now behaving correctly, and the projection line now simulates the ball movement accurately.

Now I can move onto developing an AI to play the game.

In order to do this, I will change the whole game to implement an AI playing the game.

Here I have done set the groundwork for implementing a computer to play the game:

```

computerPlaying = True # Set to True if you want the computer to play

```

```

def drawPlayerIndicator(): # Draws the player indicator
    # render font
    player1Text = playerIndicatorFont.render(("Player 1"), True, "red")
    vsText = playerIndicatorFont.render(("vs"), True, "white")
    player2Text = playerIndicatorFont.render(("Computer"), True, "blue")

```

After the game logic has been calculated, this code will run:

It checks whose turn it is and if it is the computer, it executes the shot straight away.

```

if computerPlaying and not player1Turn: # if computer is playing
    computerShoot() # computer finds shot and takes shot
else: # if computer is not playing
    drawCue() # draw the cue and give velocity to the white ball if clicked

```

The computer Shoot algorithm is shown here:

```

def computerShoot(): # computer finds shot and takes shot
    global player2BallType

    if player2BallType == None: # if the ball type is not set
        # find closest ball and hit it
        distance, closestBall = distanceToClosestBall(balls[0].pos, type=None) # find closest ball
        ballVec = (closestBall.pos - balls[0].pos).normalize() # vector of the ball to the white ball
        balls[0].velocity = ballVec*20 # give ball a velocity

    else:
        # find closest ball of that ball type and hit it
        distance, closestBall = distanceToClosestBall(balls[0].pos, type=player2BallType) # find closest ball of correct type
        ballVec = (closestBall.pos - balls[0].pos).normalize() # vector of the ball to the white ball
        balls[0].velocity = ballVec*20 # give ball a velocity

```

This algorithm includes the function of finding the closest ball to the white ball. I therefore had to modify that function so that I can find the closest ball of the correct type. This has been done in the function as shown here:

```

def distanceToClosestBall(initialPos, type): # returns distance from closest ball
    closestDistance = float("inf") # sets initial value to infinity
    otherBall = None # sets initial value to none
    for ball in ballGroup.sprites(): # loops through every ball (except whiteball)
        if (type == "striped" and ball.isStriped) or (type == "dotted" and not ball.isStriped) or type == None: # if the ball is the specified type or not classified
            if ball.number != 0: # skip if the ball is itself (white)
                distance = VEC(initialPos).distance_to(ball.pos) # finds the distance between the ball and the whiteball
                if distance < closestDistance: # if the new distance is smaller, set the new distance to the actual distance
                    closestDistance = distance
                    otherBall = ball
    return closestDistance, otherBall # return the distance and then the instance of the other ball

```

This is working well (I cannot show this as an image but straight after the player's go is done, the computer executes their shot).



While playing, I have noticed that the computer will aim for the 8 ball as the value of this ball is set to "None". This means I will have to implement a manual check to only aim for the 8 ball when there are no other balls of the computer's type to aim for.

Here I have attempted to fix this here:

```
def computerShoot(): # computer finds shot and takes shot
    global player2BallType

    if player2BallType == None: # if the ball type is not set
        # find closest ball and hit it
        distance, closestBall = distanceToClosestBall(balls[0].pos, type=None) # find closest ball
        ballVec = (closestBall.pos - balls[0].pos).normalize() # vector of the ball to the white ball
        balls[0].velocity = ballVec*20 # give ball a velocity

    else:
        # find closest ball of that ball type and hit it

        hitEightBall = True # set initial case to True
        for i in ballGroup: # loop through each ball
            if i.number == 8 or i.number == 0: # if the ball is the 8 ball
                continue # skip the ball
            elif (player2BallType == "striped" and i.isStriped) or (player2BallType == "dotted" and not i.isStriped): # if there is a ball of correct type, hit that ball
                hitEightBall = False # set case to False
            else:
                print("no ball of correct type to hit") # if there is no ball of correct type, print error

        if hitEightBall: # if there is no ball of correct type, hit the 8 ball
            ballVec = (balls[8].pos - balls[0].pos).normalize() # vector of the ball to the white ball
            balls[0].velocity = ballVec*20 # give ball a velocity
        else:
            distance, closestBall = distanceToClosestBall(balls[0].pos, type=player2BallType) # find closest ball of correct type
            ballVec = (closestBall.pos - balls[0].pos).normalize() # vector of the ball to the white ball
            balls[0].velocity = ballVec*20 # give ball a velocity
```

This has not fixed the issue.

After looking at the code, I have determined that because the value of isStriped of the 8 ball is None, this means that the value is calculated as False, meaning another check must be done inside the distance to closest ball:

```
def distanceToClosestBall(intialPos, type): # returns distance from closest ball
    closestDistance = float("inf") # sets initial value to infinity
    otherBall = None # sets initial value to none
    for ball in ballGroup.sprites(): # loops through every ball (except whiteball)
        if (type == "striped" and ball.isStriped and not ball.number == 8) or \
            (type == "dotted" and not ball.isStriped and not ball.number == 8) or \
            (type == None): # if the ball is the specified type or not classified
            if ball.number != 0: # skip if the ball is itself (white)
                distance = VEC(intialPos).distance_to(ball.pos) # finds the distance between the ball and the whiteball
                if distance < closestDistance: # if the new distance is smaller, set the new distance to the actual distance
                    closestDistance = distance
                    otherBall = ball
    return closestDistance, otherBall # return the distance and then the instance of the other ball
```

This now works correctly.

The game now has a functioning AI that the player can play against!

Stage 5 Review

What has been done?

- Resolved inaccuracies in collision calculations at higher speeds.
- Adjusted wall collision handling by recursively identifying the exact point of collision.
- Refined ball-to-ball collisions, preventing stationary balls from clipping after collisions.
- Implemented an AI, allowing a computer to take turns against another player.
- Verified that the collision system and AI function correctly.

How is code being tested?

- For each new element/function (e.g., ball movement, wall collisions, ball-to-ball collisions), I add a small bit of functionality and then observe the result, documenting and debugging it accordingly. This incremental approach ensures I follow the computational method of decomposition which ensures for swift development of the solution.
- Bugs (like clipping, stuck balls, or zero-distance errors) are identified and resolved by modifying code of the program, rerunning the simulation, and documenting my errors.
- Comments are used extensively throughout the solution. This helps debugging enormously as I sift through hundreds of lines of code.

Success criteria that has been met:

25	The player should be able to play against an AI.
----	--

Errors that have been fixed:

Error	Fix
When a moving ball's vector transformation directly collided with a wall or another ball, the collision was calculated immediately, causing inaccurate collision calculation.	The code was refactored by moving the distance calculation into an inner function that ensures the balls are not clipping when the collision is calculated.
In the ball-to-ball collision handling, if the two balls involved had a velocity of zero, the iterative method alone failed to separate them.	Code was added to the collision resolution method to push the balls apart if they are still colliding after the collision has been calculated.
The computer AI was sometimes aiming for the 8-ball as its target, when it should not have.	A manual check was implemented to ensure that the AI only selects the 8-ball as a target when there are no other balls available.

Summary of Stage 5

The collision mechanics now function correctly, ensuring accurate physics at high speeds. The AI can effectively play the game, taking valid shots against the player. The game is now ready for player-versus-computer matches.

Stage 6: Menu Creation

To create a menu interface, I will first create a new file that will allow me to learn the basics of implementing a menu. Here I have created a basic new file:

```
import pygame

# Initialize Pygame
pygame.init()

# Set up display
width, height = 800, 600
screen = pygame.display.set_mode((width, height))
pygame.display.set_caption("Menu")

# Main loop
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # Fill the screen with black
    screen.fill("black")

    # Update the display
    pygame.display.flip()
```

I have now added a class called Menu that will create a menu, when given a bunch of options.

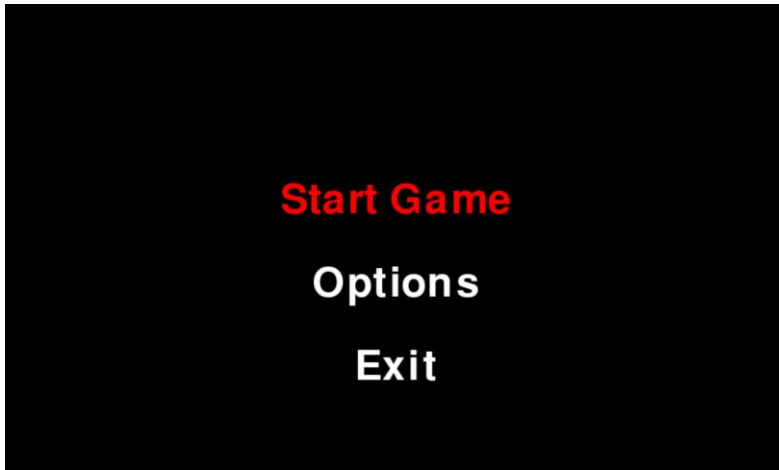
```
class Menu:
    def __init__(self, screen, options):
        # Initialize the Menu class with the screen and options
        self.options = options # List of menu options
        self.menuFont = pygame.font.SysFont(None, 74) # Font for the menu text
        self.selected_option = 0 # Index of the currently selected option
        self.colour = "white" # Colour for non-selected options
        self.selectedColour = "red" # Colour for the selected option

    def draw(self):
        increment = 0 # Variable to increment the position of each menu option
        for option in self.options:
            if self.options[self.selected_option] == option: # Check if the current option is the selected option
                text = self.menuFont.render(option, True, self.selectedColour) # if it is the selected option, draw with red
            else:
                text = self.menuFont.render(option, True, self.colour) # Render non-selected options with the default colour
            # centre font and draw to screen
            textRect = text.get_rect(center=((width // 2), (height // 2) + increment))
            screen.blit(text, textRect)
            increment += 100 # Increment the position for the next menu option

menu = Menu(screen, ["Start Game", "Options", "Exit"]) # create a menu with those options
```

```
menu.draw() # Draw the menu
```

At the bottom of the first image, the last line of code creates a menu with “start game” etc. options, this is shown to work here:



The selected option is also working, but this is not synchronous with the mouse hovering over that option.

Upon a bit of research about creating menus in pygame, I will need to define another class. This is because in order to have the options dynamically respond to the mouse, they must have a rectangle around them which checks for collisions. This means that the best way to approach this is to create a new class which simulates a button.

```
class Button: # Defines Button class
    def __init__(self, text, x, y, width, height):
        self.text = text # Text to display on the button
        self.font = pygame.font.SysFont(None, 70) # Font for the button text
        self.rect = pygame.Rect(x, y, width, height) # Rectangle representing the button's position and size
        self.boxColour = "white" # Button colour
        self.textColour = "black" # Text colour
```

```
def draw(self):
    pygame.draw.rect(screen, self.boxColour, self.rect) # Draw the button rectangle
    textRect = self.textSurface.get_rect(center=self.rect.center) # Get the rectangle for the text surface
    screen.blit(self.textSurface, textRect) # Draw the text on the button
```

Here I have created a new class and have drawn it to the screen using the following code:

```
button = Button("Button Text", 400, 200, 200, 100) # create a button
```



In order to make it more aesthetic, I will change the inputs to only take the size of the font, and the button will be exactly around it.

```
class Button: # Defines Button class
    def __init__(self, text, fontSize, x, y, padding):
        self.text = text # Text to display on the button
        self.font = pygame.font.SysFont(None, fontSize) # Font for the button text
        self.boxColour = "white" # Button colour
        self.textColour = "black" # Text colour
        self.textSurface = self.font.render(self.text, True, self.textColour) # Render the button text
        # Rectangle representing the button's position and size
        self.rect = pygame.Rect(x, y, self.textSurface.get_width() + padding, self.textSurface.get_height() + padding)

    def draw(self):
        pygame.draw.rect(screen, self.boxColour, self.rect) # Draw the button rectangle
        textRect = self.textSurface.get_rect(center=self.rect.center) # Get the rectangle for the text surface
        screen.blit(self.textSurface, textRect) # Draw the text on the button

button = Button("Button Text", 100, 100, 100, 10) # create a button
```



This looks better.

Now I will integrate this class into the Menu class, to create a menu full of buttons.

To do this, I will need functionality to the button.

On hover, the button should change colour.

On click the button should respond.

Here I have implemented a check for when the mouse is over the button:

```
def checkHover(self, mousePos): # checks if the mouse is over the button.
    if self.rect.collidepoint(mousePos): # Check if the mouse position is within the button rectangle
        self.textColour = "red" # highlight button
    else:
        self.textColour = "black" # reverse change
```

```
button.draw() # Draw the button
button.checkHover(pygame.mouse.get_pos()) # checks if mouse is hovering over
```

This is working as when the mouse hovers over, the colour changes:




Here I have cleaned the code up a bit:

```

class Button: # Defines Button class
    def __init__(self, text, fontSize, x, y, padding):
        self.text = text # Text to display on the button
        self.font = pygame.font.SysFont(None, fontSize) # Font for the button text

        self.hovering = False # Set to true if the mouse is hovering over.
        self.clicked = False

        self.boxColour = "white" # Button colour
        self.boxColourClicked = "grey" # Button colour when clicked
        self.textColour = "black" # Text colour when the mouse is not hovering over
        self.textColourHovering = "red" # Text colour when the mouse is hovering over

        self.textSurface = self.font.render(self.text, True, self.textColour) # Render the button text
        # Rectangle representing the button's position and size
        self.rect = pygame.Rect(x, y, self.textSurface.get_width() + padding, self.textSurface.get_height() + padding)

```

```

def draw(self):
    textColour = ""
    boxColour = ""
    if self.hovering: # Check if the button is being hovered over
        textColour = self.textColourHovering # Set textColour to hovering colour
    else:
        textColour = self.textColour # Set textColour to default colour
    if self.clicked: # Check if the button is clicked
        boxColour = self.boxColourClicked # Set boxColour to clicked colour
    else:
        boxColour = self.boxColour # Set boxColour to default colour
    self.textSurface = self.font.render(self.text, True, textColour) # Render the button text
    pygame.draw.rect(screen, boxColour, self.rect) # Draw the button rectangle
    textRect = self.textSurface.get_rect(center=self.rect.center) # Get the rectangle for the text surface
    screen.blit(self.textSurface, textRect) # Draw the text on the button

def checkHover(self, mousePos): # checks if the mouse is over the button.
    if self.rect.collidepoint(mousePos): # Check if the mouse position is within the button rectangle
        self.hovering = True # Sets to True if hovering
        return True # returns true to say the mouse is hovering over
    else:
        self.hovering = False # Sets to False if hovering
        return False # returns False to say the mouse is hovering over

```

Now this is working, I will add more code to the menu class to initiate the buttons from the menu.

Due to buttons now being the main elements of each menu, I have decided to restart the menu class to ensure full compatibility with the buttons.

```

class Menu: # Defines Menu class
    def __init__(self): # Initializes the Menu class
        self.buttons = [] # List to store buttons
        self.create_buttons() # Calls the method to create buttons

    def create_buttons(self): # Method to create buttons
        start = [100, 300] # Starting position for buttons
        increment = 50 # Vertical spacing between buttons
        start_button = Button("Start Game", 30, start[0], start[1] + 1 * increment, 10) # Create Start Game button
        options_button = Button("Options", 30, start[0], start[1] + 2 * increment, 10) # Create Options button
        quit_button = Button("Quit", 30, start[0], start[1] + 3 * increment, 10) # Create Quit button
        self.buttons.extend([start_button, options_button, quit_button]) # Add buttons to the list

    def draw(self): # Method to draw buttons
        for i in self.buttons: # Iterate through each button
            i.draw() # Draw the button

```

This successfully creates a menu!



However, they are not turning red when I hover over them.

This can be fixed by adding an update function that checks each button individually.

```
def update(self): # Method to update button states
    for i in self.buttons: # Iterate through each button
        i.checkHover(pygame.mouse.get_pos()) # Check if the mouse is hovering over the button
```

However, after adding this, the buttons are still not responding.

This was due to me not adding the actual update function in the game loop.

```
menu.update() # update the menu
```

Now the menu is responding well!

Hovering over options:



Here I have cleaned up code and ensured it works as proven above:

```
def create_buttons(self): # Method to create buttons

    start = [100, 300] # Starting position for buttons
    increment = 50 # Vertical spacing between buttons

    for i in range(len(self.options)): # Iterate through the options
        # Create a button for each option and append it to the buttons list
        self.buttons.append(Button(self.options[i], 30, start[0], start[1] + i*increment, 10))
```

```
menu = Menu(["Start", "Options", "Quit"]) # create a menu
```

Now I need to create a way of checking when the user clicks the button.

Here I will implement a system of a callback when the button is clicked.

This will be a function that will be executed when the button is clicked.

```
class Button: # Defines Button class
    def __init__(self, text, fontSize, x, y, padding, callback):
        self.text = text # Text to display on the button
        self.font = pygame.font.SysFont(None, fontSize) # Font for the button text
        self.callback = callback # Function to call when the button is clicked
```

Here I have added a print("Hello") to test if it works:

```
for i in range(len(self.options)): # Iterate through the options
    # Create a button for each option and append it to the buttons list
    self.buttons.append(Button(self.options[i], 60, start[0], start[1] + i*increment, 10, print("Hello")))
```

Now I add the event handler in the Menu class:

```
def handleEvent(self, event): # Method to handle events
    if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # Check if the left mouse button is clicked
        for button in self.buttons: # Iterate through each button
            if button.checkHover(event.pos): # Check if click occurred and if hovering over button
                button.callback # execute function
```

Add the event check in the game loop:

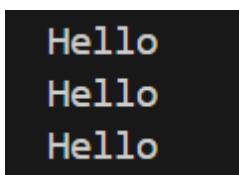
```
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        menu.handleEvent(event) # check if any button is clicked
```

On run:



On click:

The console prints hello:



Now I will add the functions to execute.

I will create a new list that stores the functions to execute that correspond to the buttons when clicked

```
def startGame():  
    print("Starting game...")  
  
def showOptions():  
    print("Showing options...")  
  
def quitGame():  
    global running  
    running = False  
  
menu = Menu(["Start", "Options", "Quit"], [startGame, showOptions, quitGame]) # create a menu
```

This now works well:



I have clicked Start and now it prints:

```
Starting game...
```

When I click Options,

```
Showing options...
```

When I click Quit, the game ends.

This is all the functionality of the Menu now done.

I will now make it more aesthetic.

Here I give the user feedback by changing the colour when actually clicked by setting the `button.clicked` to `True/False`

```
def handleEvent(self, event): # Method to handle events
    if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # Check if the left mouse button is clicked
        for button in self.buttons: # Iterate through each button
            if button.checkHover(event.pos): # Check if click occurred and if hovering over button
                button.clicked = True # Set button as clicked
                button.callback() # Execute the button's callback function
            else:
                button.clicked = False # Set button as not clicked
    elif event.type == pygame.MOUSEBUTTONUP and event.button == 1: # When mouse button is released
        for button in self.buttons: # Iterate through each button
            button.clicked = False # Set button as not clicked
```

Here I had added more functionality to the class:

```
class Menu: # Defines Menu class
    def __init__(self, fontSize, pos, spacing, options, functions): # Initializes the Menu class
        self.fontSize = fontSize # Font size for the menu options
        self.pos = pos # Position of the menu
        self.spacing = spacing # Spacing between menu options
        self.options = options # List of menu options
        self.functions = functions # List of functions corresponding to each menu option
        self.buttons = [] # List to store buttons
        self.create_buttons() # Calls the method to create buttons

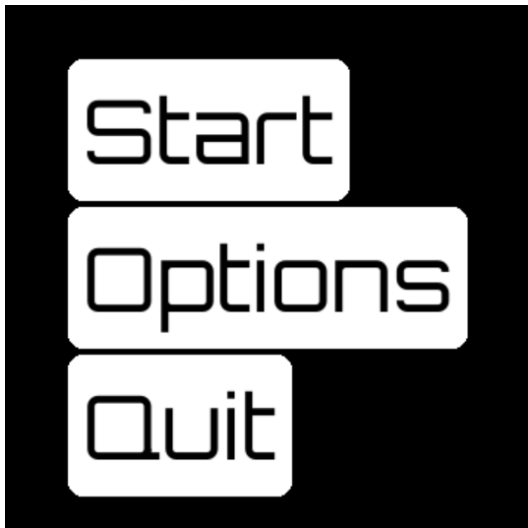
mainMenu = Menu(60, [100,100], 100, ["Start", "Options", "Quit"], [startGame, showOptions, quitGame]) # create a menu
```

This makes it easier to tweak the menu.

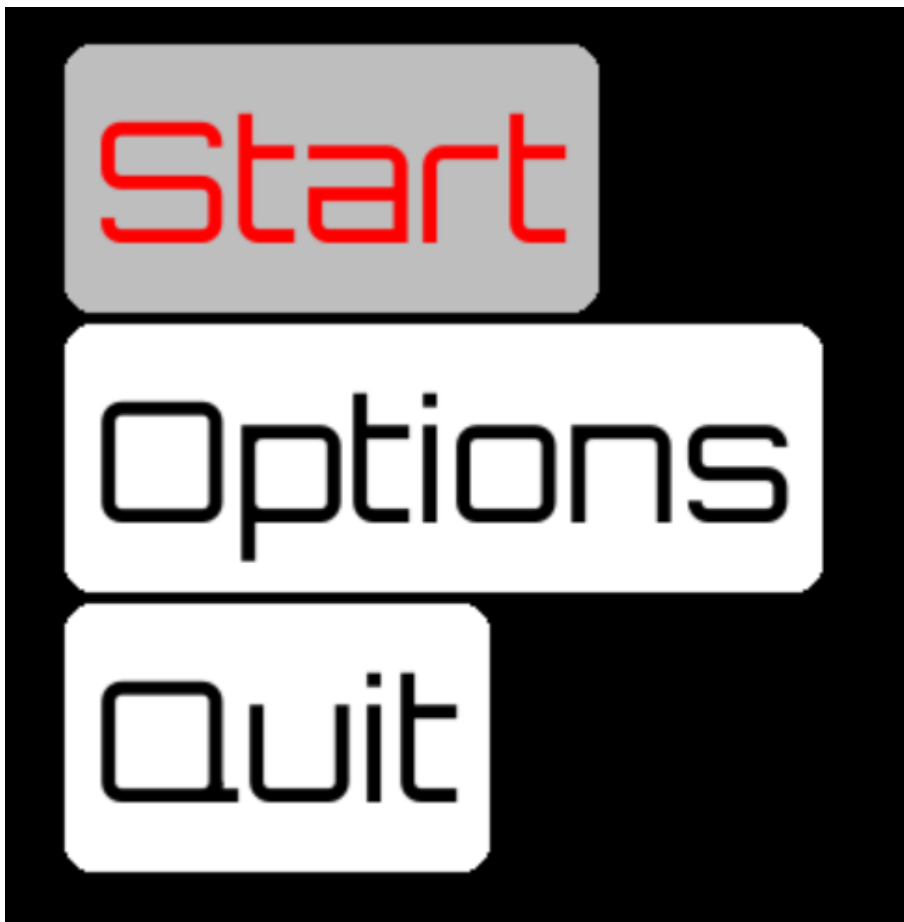
By changing the drawing of the rect in pygame:

```
pygame.draw.rect(screen, boxColour, self.rect, 0, 10) # Draw the button rectangle
```

And by adding the correct font, here is what the menu is looking like:



On click:



Now I need to run the game through the menu.

I will first put the game loop into a function:

```
def newGame():
    global gameQueue
    global pottedBalls
    global roundPottedBalls
    global event
    global mouseDown

    gameQueue = [] # Queue/List for storing events in between each turn.
    pottedBalls = [] # List for storing potted balls in whole game
    roundPottedBalls = [] # List for storing potted balls in each round

    # Game Loop
    while True:
        mouseDown = False

        for event in pygame.event.get():
            if event.type == pygame.QUIT: # Checks if user closes program
                pygame.quit()
                exit()

            if event.type == pygame.MOUSEBUTTONDOWN: # on left click
                mouseDown = True

        ballPowerSlider.update() # Update Slider
        ballGroup.update() # update each ball
```

Then I will import the whole file into the menu file.

```
from main import *
```

Now from the menu file, when the Start button is clicked, the code will execute this function.

```
mainMenu = Menu(60, [100,100], 100, ["Start", "Options", "Quit"], [runGame, showOptions, quitGame]) # create a menu
```

```
def runGame():
    newGame()
```

This, however, returns an error message.

```
if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # when left mouse button clicked
    ^^^^^
NameError: name 'event' is not defined
```

At first, I thought that the variables event in the two game loops were colliding. I tried to fix this here by changing the menu.py event to menuEvent.

```
for menuEvent in pygame.event.get():
    if menuEvent.type == pygame.QUIT:
        running = False
    mainMenu.handleEvent(menuEvent) # check if any button is clicked
```

But this is returning the same error.

```
if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # when left mouse button clicked
    ^^^^^
NameError: name 'event' is not defined
```

For the ball slider, the each pygame event needed to be checked in accordance with the slider. And since event was not defined in this scope, it is defined by passing event as a parameter in the update method.

```
def update(self, event): # updates slider position
    if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # when left mouse button clicked
        mousePos = pygame.mouse.get_pos() # get mouse position
        if 0 <= mousePos[0] - self.position[0] <= self.width: # if click is within x bounds
            if 0 <= mousePos[1] - self.position[1] <= self.height: # if click is within y bounds
```

The next thing was to indent this line of code:

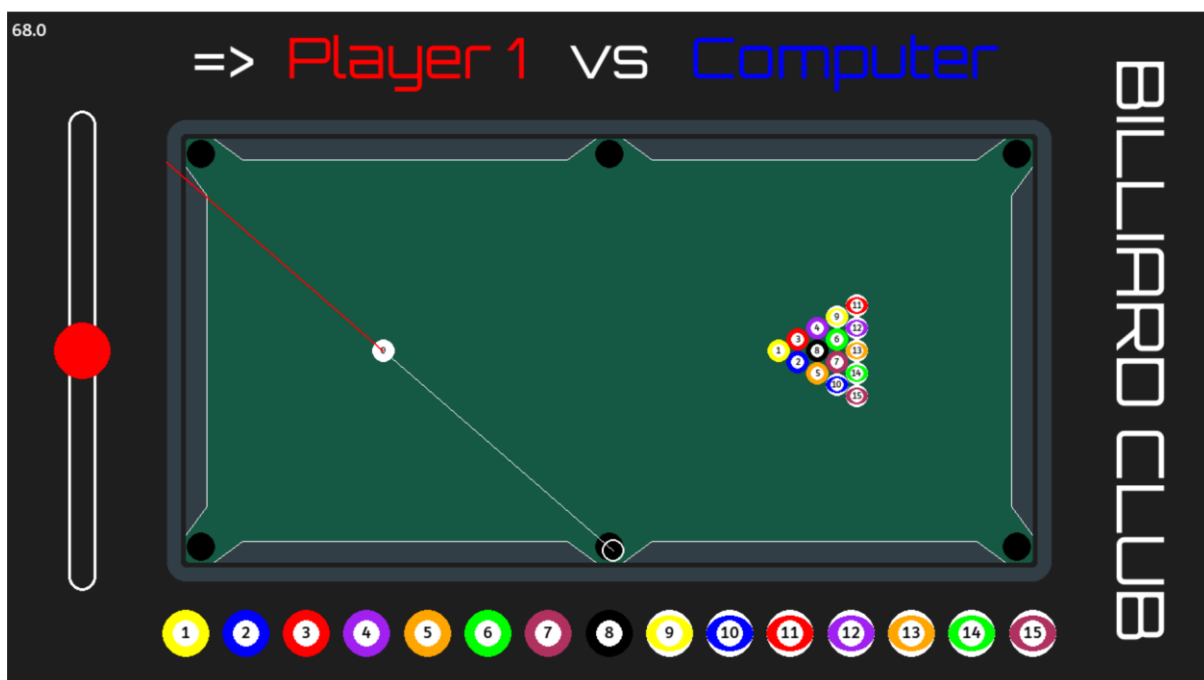
```
815     while True:
816         mouseDown = False
817
818         for event in pygame.event.get():
819             if event.type == pygame.QUIT: # Checks if user closes program
820                 pygame.quit()
821                 exit()
822
823             if event.type == pygame.MOUSEBUTTONDOWN: # on left click
824                 mouseDown = True
825
826                 ballPowerSlider.update(event) # Update Slider
827                 ballGroup.update() # update each ball
```

It turns out that line 826 needed to be indented.

Now the game is functioning correctly with the menu:



When start is clicked, the game moves instantly to the Game Screen:



Now I am going to add multiple screens for options, and whether the player wants to play against a computer or another player.

I am going to separate screens by using multiple while loops embedded in functions that are run when they need to be.

```
def mainMenuScreen():
    # Main loop
    running = True
    while running:
        for menuEvent in pygame.event.get(): # Iterate through the events
            if menuEvent.type == pygame.QUIT: # Check if the quit event is triggered
                running = False # Exit the loop
            mainMenu.handleEvent(menuEvent) # Check if any button is clicked

        screen.fill("black") # Fill the screen with black color
        mainMenu.draw() # Draw the menu
        mainMenu.update() # Update the menu
        pygame.display.flip() # Update the display
```

Here I have added some more screens: with the same code as the main menu screen.

```
def mainMenuScreen(): # Main loop for the main menu...

def gameModeScreen(): # Main loop for the game mode selection screen...

def optionsScreen(): # Main loop for the options screen...

def runGame(): # Function to start a new game...

def showOptions(): # Function to show the options screen...

def quitGame(): # Function to quit the game...

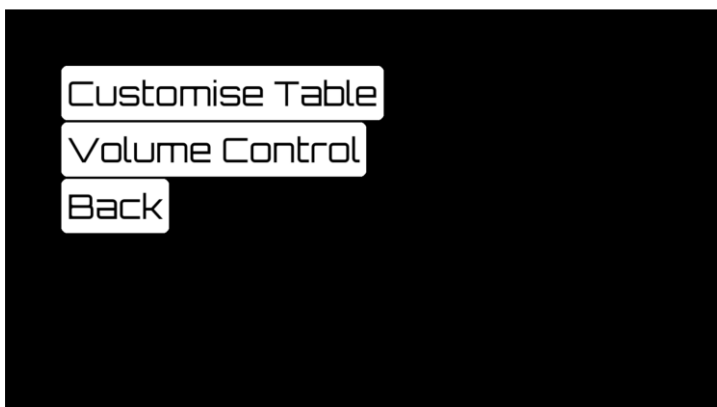
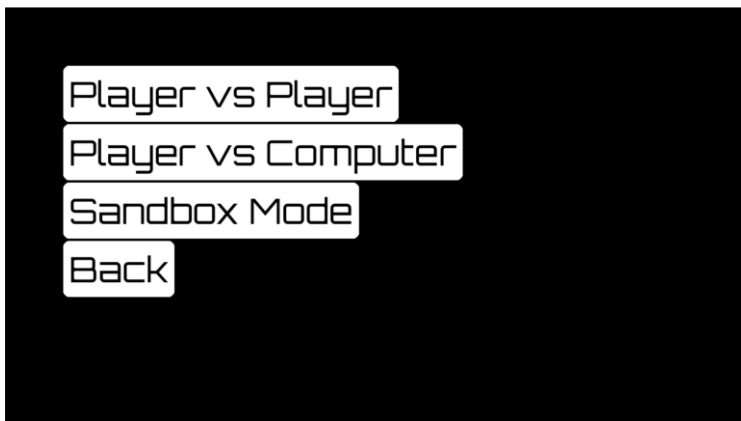
mainMenu = Menu(60, [100,100], 100, ["Start", "Options", "Quit"], [gameModeScreen, showOptions, quitGame]) # create a menu
gameModeMenu = Menu(60, [100,100], 100, ["Player vs Player", "Player vs Computer", "Back"], [runGame, runGame, mainMenuScreen]) # create a menu
optionsMenu = Menu(60, [100,100], 100, ["Options", "Back"], [showOptions, mainMenuScreen]) # create a menu
```

There is a problem, as the callback function is called before the button can update, this means that when you navigate back to that menu, the button you previously clicked appears grey. This is fixed here by calling the function after the button is released.

```
def handleEvent(self, event): # Method to handle events
    if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # Check if the left mouse button is clicked
        for button in self.buttons: # Iterate through each button
            if button.checkHover(event.pos): # Check if click occurred and if hovering over button
                button.wasClicked = True # Set button as clicked
            else:
                button.wasClicked = False # Set button as not clicked
    elif event.type == pygame.MOUSEBUTTONUP and event.button == 1: # When mouse button is released
        for button in self.buttons: # Iterate through each button
            if button.wasClicked:
                button.wasClicked = False
                button.callback() # Execute the button's callback function
            button.wasClicked = False # Set button as not clicked
```

Updated menu screens:

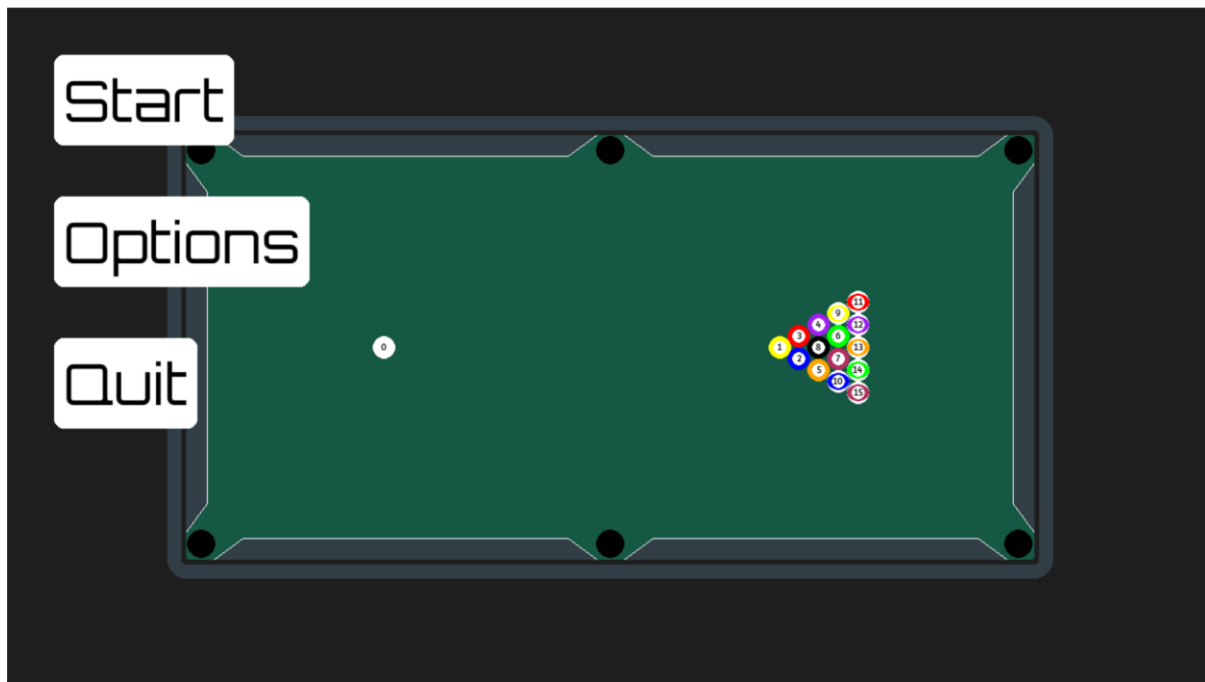
```
mainMenu = Menu(60, [100,100], 100, ["Start", "Options", "Quit"], [gameModeScreen, showOptions, quitGame]) # create a menu
gameModeMenu = Menu(60, [100,100], 100, ["Player vs Player", "Player vs Computer", "Sandbox Mode", "Back"], [runGame, runGame, runGame, mainMenuScreen]) # create a menu
optionsMenu = Menu(60, [100,100], 100, ["Customise Table", "Volume control", "Back"], [mainMenuScreen, mainMenuScreen, mainMenuScreen]) # create a menu
```



To make the menus a bit more interesting and aesthetic, I will add a pool table in the background:

```
screen.fill(□(30, 30, 30)) # Fills the screen with a grey colour  
drawTable() # Draws the table on the screen  
drawBalls() # Draws the balls on the screen
```

Looks better now:



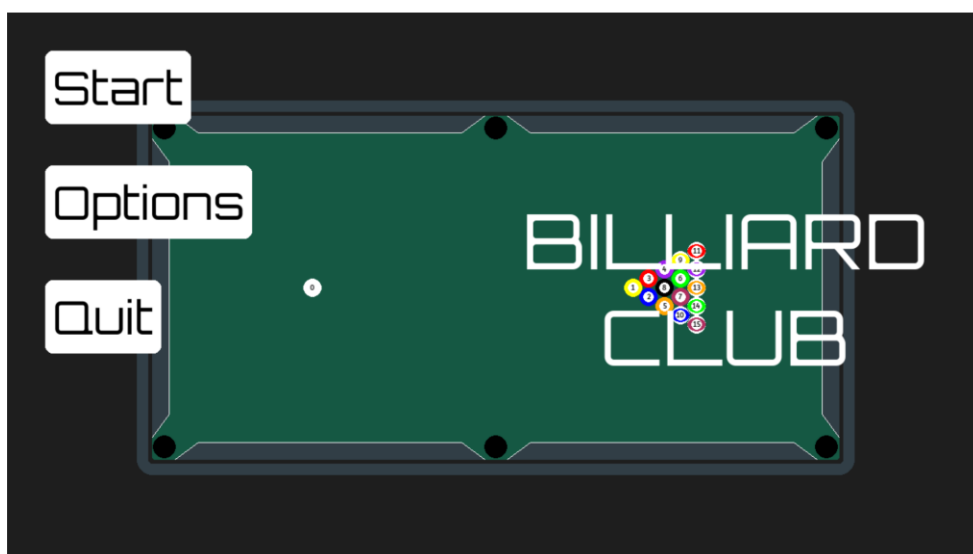
Now I will draw the logo:

```
def drawMenuLogo():
    # Render the menu logo text
    menuLogo1 = MenuLogoFont.render(("BILLIARD"), True, "white")
    menuLogo2 = MenuLogoFont.render(("CLUB"), True, "white")

    # Calculate the position for the logos
    logoPos = (centre[0] + 300, centre[1])
    logo1Pos = (logoPos[0] - menuLogo1.get_width() // 2, logoPos[1] - menuLogo1.get_height())
    logo2Pos = (logoPos[0] - menuLogo2.get_width() // 2, logoPos[1])

    # Draw the logos on the screen
    screen.blit(menuLogo1, logo1Pos)
    screen.blit(menuLogo2, logo2Pos)
```

This now looks better:



I will also add a black outline to this logo to make it pop:

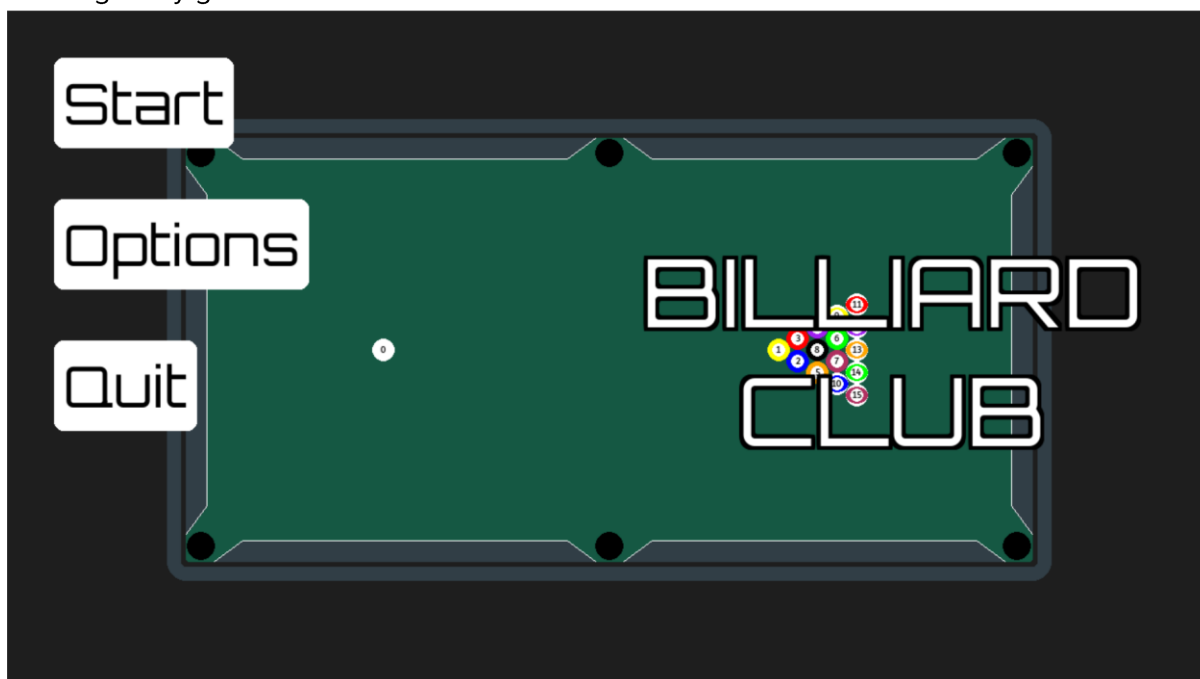
```
def drawMenuLogo():
    # Render the menu logo text
    menuLogo1 = MenuLogoFont.render(("BILLIARD"), True, "white")
    menuLogo2 = MenuLogoFont.render(("CLUB"), True, "white")
    menuLogo1OUTLINE = MenuLogoFont.render(("BILLIARD"), True, "black")
    menuLogo2OUTLINE = MenuLogoFont.render(("CLUB"), True, "black")

    # Calculate the position for the logos
    logoPos = (centre[0] + 300, centre[1])
    logo1Pos = (logoPos[0] - menuLogo1.get_width() // 2, logoPos[1] - menuLogo1.get_height())
    logo2Pos = (logoPos[0] - menuLogo2.get_width() // 2, logoPos[1])

    # Draw the outline of the logo
    for i in range(-4,5):
        for j in range(-4,5):
            screen.blit(menuLogo1OUTLINE, (logo1Pos[0]+i, logo1Pos[1]+j))
            screen.blit(menuLogo2OUTLINE, (logo2Pos[0]+i, logo2Pos[1]+j))

    # Draw the logos on the screen
    screen.blit(menuLogo1, logo1Pos)
    screen.blit(menuLogo2, logo2Pos)
```

Looking really good now



Now I will add the functionality for the Player Game modes:

```
def newGame(computerPlaying, sandbox):
    global gameQueue
    global pottedBalls
    global roundPottedBalls
    global mouseDown

    gameQueue = [] # Queue/List for storing events in between each turn.
```

```
# if the ball has been hit
if checkIfBallMoving() and not sandbox:
    gameQueue.append(gameLogic) # when the balls are next still, apply game logic to determine the next round.

if not sandbox:
    drawPlayerIndicator() # draws the player indicator
    checkWin(winner) # checks if there is a winner and draws it to the screen
```

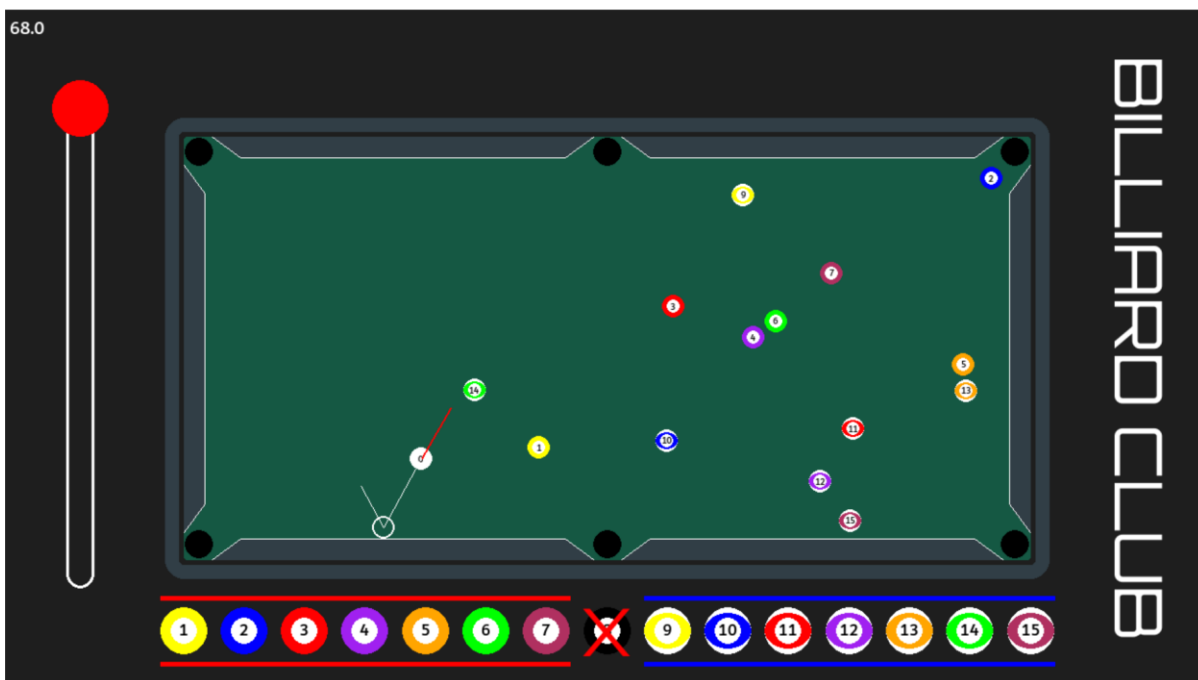
In order to implement the feature above, parameters must be passed through the run game function. Therefore, I have had to change the way the solution handles the callbacks.

```
mainMenu = Menu(60, [50,50], 150, ["Start", "Options", "Quit"], ["gameModeScreen()", "showOptions()", "quitGame()"]) # create a menu
gameModeMenu = Menu(60, [50,50], 150, ["Player vs Player", "Player vs AI", "Sandbox Mode", "Back"], ["runGame(False, False)", "runGame(True, False)", "runGame(False, False)", "runGame(True, False)"]) # create a menu
optionsMenu = Menu(60, [50,50], 150, ["Customise Table", "Volume control", "Back"], ["mainMenuScreen()", "mainMenuScreen()", "mainMenuScreen()"]) # create a menu
```

Here I have changed the functions to text and then in the button class, when clicked, it will execute that string as if it were a line of code via this here:

```
eval(button.callback) # Execute the button's callback function
```

While in sandbox mode:



This has been fixed here:

```
if not isSandbox: # don't draw this if in sandbox mode
    line1start = (tableCorners[0][0] - largerBallRadius, circleCentre[1] - largerBallRadius - 10)
    line1end = (centre[0] - distanceBetweenBalls + largerBallRadius, circleCentre[1] - largerBallRadius - 10)

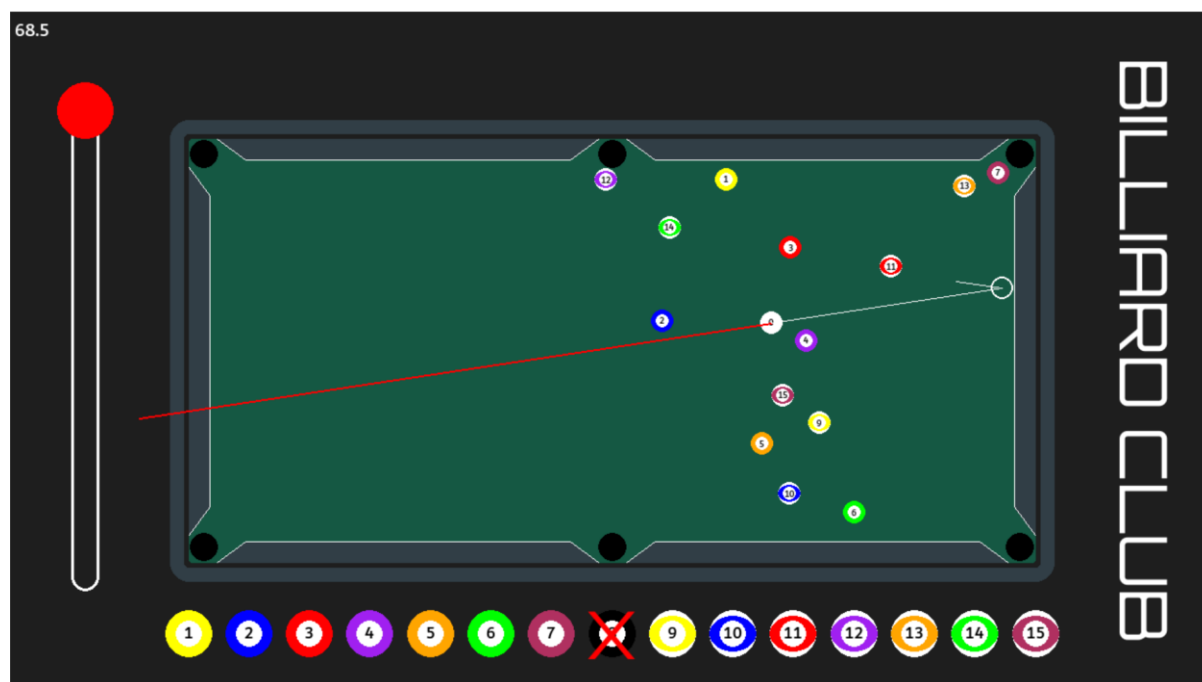
    line2start = (tableCorners[3][0] + largerBallRadius, circleCentre[1] - largerBallRadius - 10)
    line2end = (centre[0] + distanceBetweenBalls - largerBallRadius, circleCentre[1] - largerBallRadius - 10)

    line3start = (tableCorners[0][0] - largerBallRadius, circleCentre[1] + largerBallRadius + 10)
    line3end = (centre[0] - distanceBetweenBalls + largerBallRadius, circleCentre[1] + largerBallRadius + 10)

    line4start = (tableCorners[3][0] + largerBallRadius, circleCentre[1] + largerBallRadius + 10)
    line4end = (centre[0] + distanceBetweenBalls - largerBallRadius, circleCentre[1] + largerBallRadius + 10)

    if player1BallType == "dotted":
```

Now working:



While in Player vs AI mode, all seems fine with the following changes:

```
if computerPlaying and not player1Turn: # if computer is playing
    computerShoot() # computer finds shot and takes shot
else: # if computer is not playing
    drawCue() # draw the cue and give velocity to the white ball if clicked
```

While in Player vs Player Mode:

I am able to play as both players, but Player 2 is labelled as "Computer".

```
if computerPlaying:
    player2Name = "Computer"
```

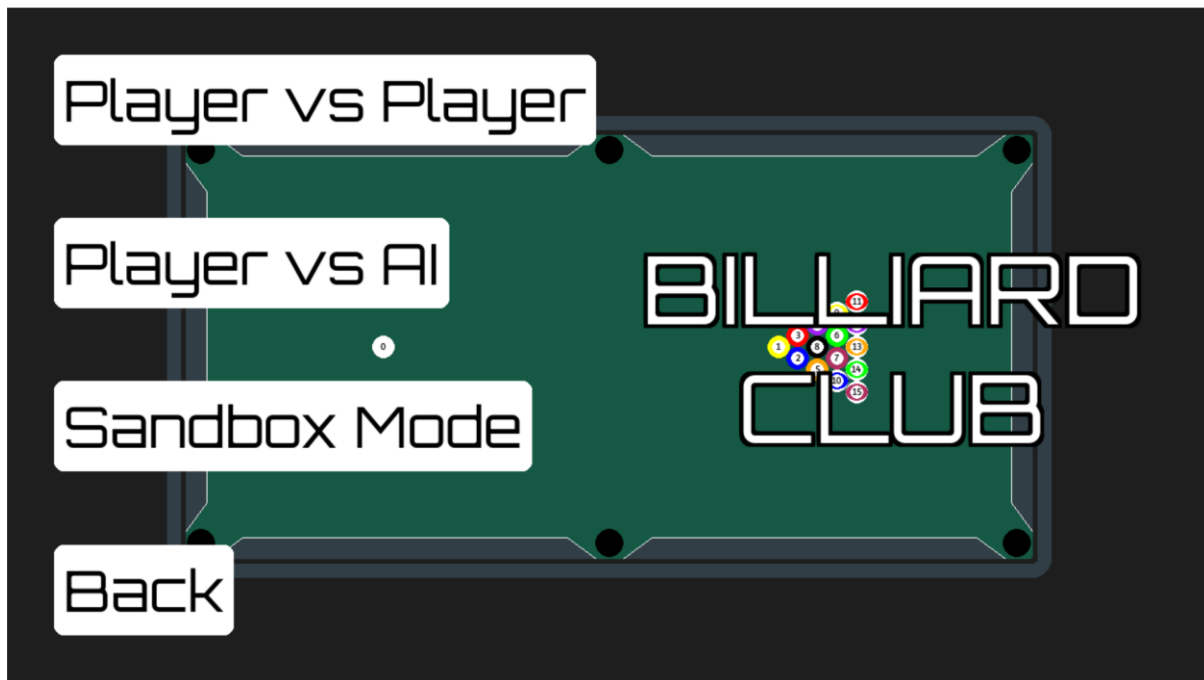
Defined two new variables for player Names as shown above

This now works:

=> **Player 1** vs **Player 2**

I have also centred the menus with this line of code:

```
increment = (resolution[1]-200)/(len(self.options)-1) # Vertical spacing between buttons
```



Customisation Screen:

Here I have created a customisation menu:

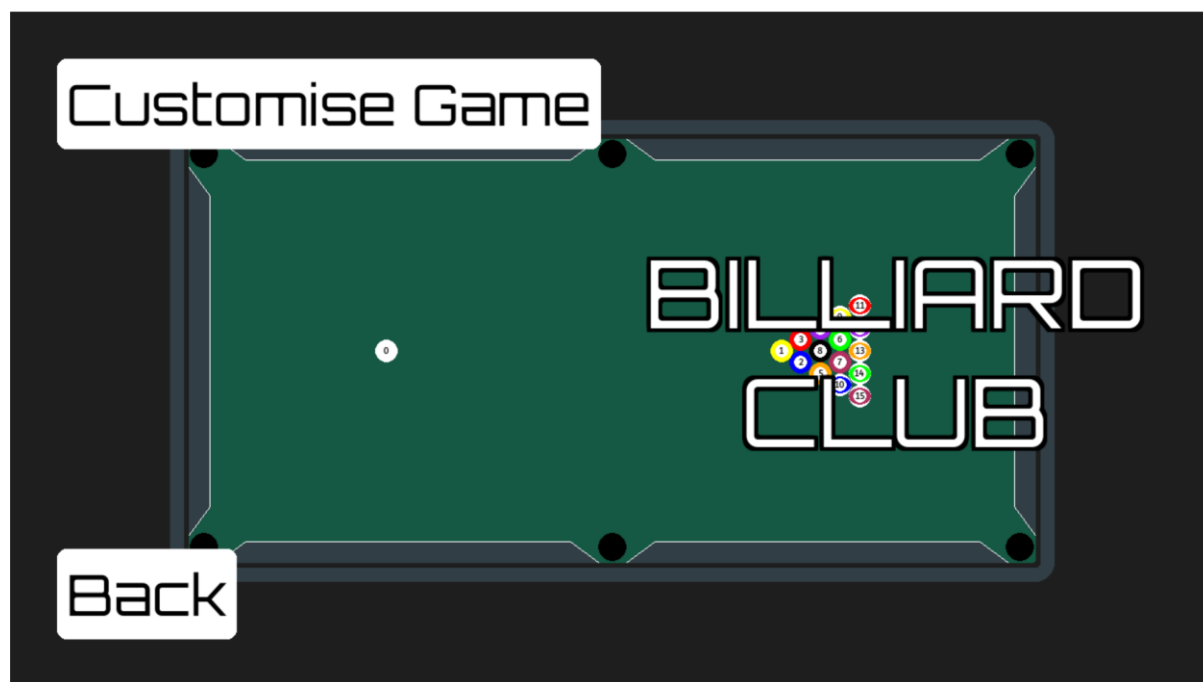
```
customisationMenu = Menu(60, [50,50], ["Customise Game", "Back"], ["customisationScreen()", "mainMenuScreen()"]) # create a menu

def customisationScreen(): # Main loop for the customisation screen
    running = True
    while running:
        for menuEvent in pygame.event.get(): # Iterate through the events
            if menuEvent.type == pygame.QUIT: # Check if the quit event is triggered
                running = False # Exit the loop
            customisationMenu.handleEvent(menuEvent) # Check if any button is clicked

        screen.fill(□(30, 30, 30)) # Fills the screen with a grey colour
        drawTable()
        drawBalls()
        drawMenuLogo()

        customisationMenu.draw() # Draw the menu
        customisationMenu.update() # Update the menu
        pygame.display.flip() # Update the display
```

When the code is run:



Now I will add the ability for the player to change their Name displayed in the game as well as each player's colour.

This will require two text boxes:

This therefore means I will create another class that will draw a text box that a user can input text into.

```
class TextBox: # Defines TextBox class
    def __init__(self, x, y, initialText, fontSize): # Initializes the TextBox class
        self.text = initialText # Initial text to display in the text box
        self.font = pygame.font.Font("Project/other/Orbitron-VariableFont_wght.ttf", fontSize) # Font for the text box text

        self.textColour = "black" # Text colour
        self.boxColour = "white" # Box colour

        self.active = False # Indicates if the text box is active (clicked)

        # Colours for the text box when active and passive
        self.colourActive = "red"
        self.colourPassive = "black"

        self.textSurface = self.font.render(self.text, True, self.textColour) # Render the text
        # Rectangle representing the text box's position and size
        self.rect = pygame.Rect(x, y, self.textSurface.get_width() + 10, self.textSurface.get_height() + 10)
```

Now I will add a draw and handle event functions:

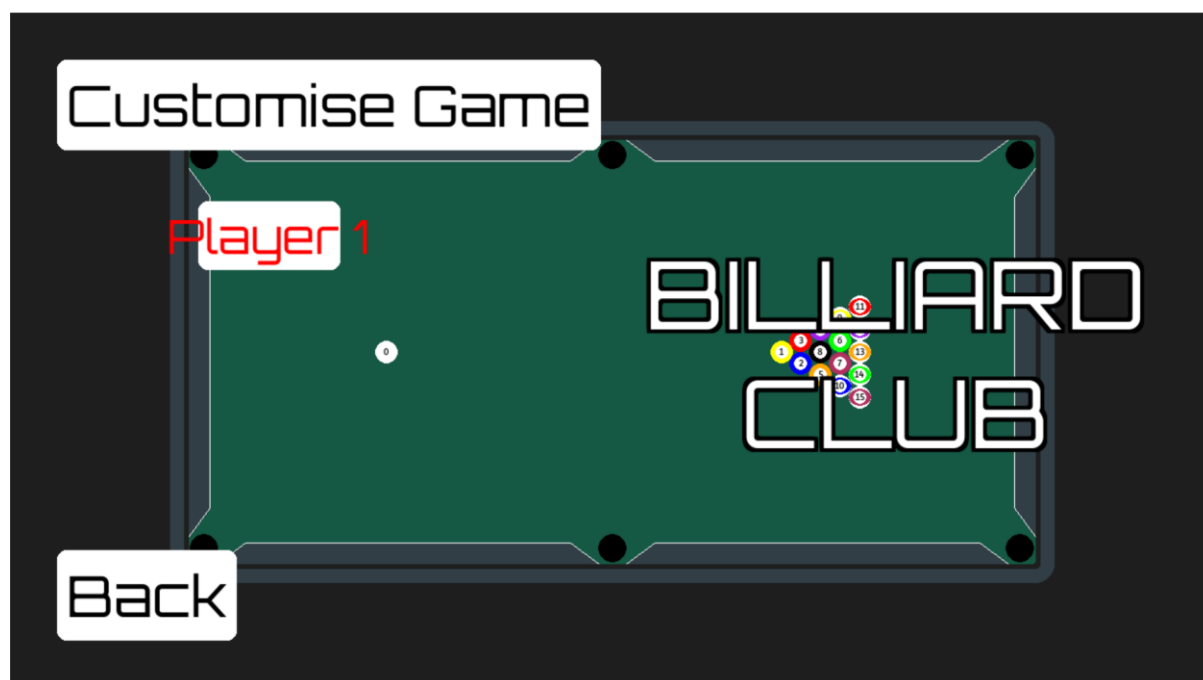
```

def draw(self): # Method to draw the text box
    if self.active: # Check if the text box is active
        self.textColour = self.colourActive # Set text colour to active colour
    else:
        self.textColour = self.colourPassive # Set text colour to passive colour
    self.textSurface = self.font.render(self.text, True, self.textColour) # Render the text
    pygame.draw.rect(screen, self.boxColour, self.rect, 0, 10) # Draw the text box rectangle
    textRect = self.textSurface.get_rect(center=self.rect.center) # Get the rectangle for the text surface
    screen.blit(self.textSurface, textRect) # Draw the text on the text box

def handle_event(self, event): # Method to handle events
    if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # Check if the left mouse button is clicked
        if self.rect.collidepoint(event.pos): # Check if the text box was clicked
            self.active = True # Set text box as active
        else:
            self.active = False # Set text box as inactive
    if event.type == pygame.KEYDOWN: # Check if a key is pressed
        if self.active: # Check if the text box is active
            if event.key == pygame.K_RETURN: # Check if the Enter key is pressed
                self.text = "" # Clear the text
            elif event.key == pygame.K_BACKSPACE: # Check if the Backspace key is pressed
                self.text = self.text[:-1] # Remove the last character
            else:
                self.text += event.unicode # Add the character to the text
        self.textSurface = self.font.render(self.text, True, self.textColour) # Re-render the text

```

This is working well, but I forgot to re-render the text so that the box's width changes with the width of the text.



Changed this:

Here I re render the text surface's width: (last line)

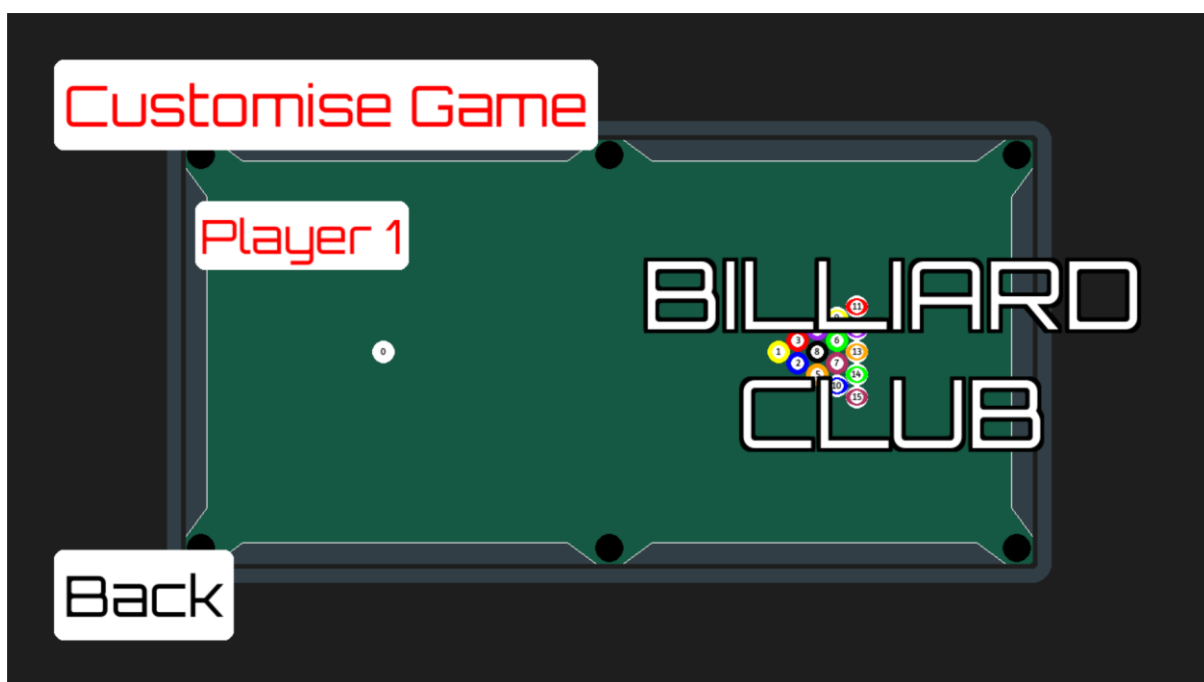
```

def draw(self): # Method to draw the text box
    if self.active: # Check if the text box is active
        self.textColour = self.colourActive # Set text colour to active colour
    else:
        self.textColour = self.colourPassive # Set text colour to passive colour

    self.rect.w = self.textSurface.get_width() + 10 # Update the width of the text box rectangle based on the text width

```

Now the text surface correctly changes depending on the width

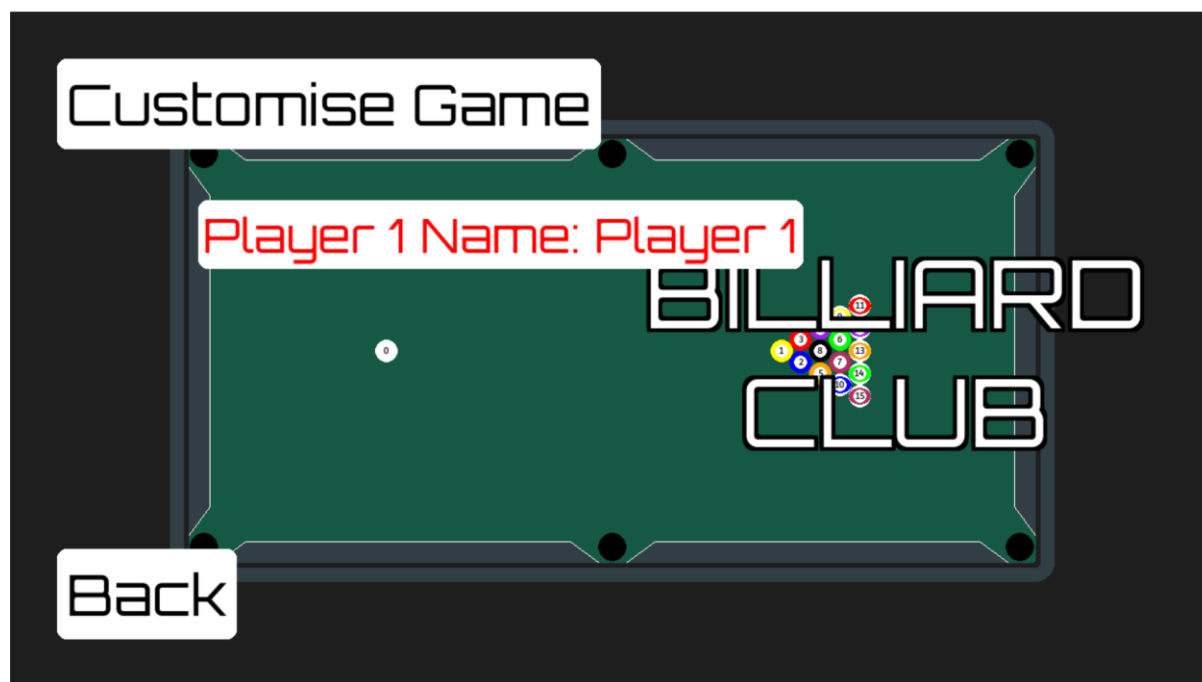


Now I will add some permanent Text in the input box to label the input field appropriately.

```
self.permanentText = permanentText # permanent Text that can't be deleted by user in the text field
```

```
self.textSurface = self.font.render(permanentText + self.text, True, self.textColour) # Render the text
```

```
TextBox1 = TextBox(200, 200, "Player 1 Name: ", "Player 1", 50)
```

This is now working really well.

I will add another one and then link these values to the actual names of the players.

```
TextBox1 = TextBox(70, 175, "Player 1 Name: ", player1Name, 30)
TextBox2 = TextBox(70, 250, "Player 2 Name: ", player2Name, 30)
```

```
TextBox1.handle_event(menuEvent)
TextBox2.handle_event(menuEvent)
```

```
newGame(computerPlaying, sandbox, player1Name, player2Name, player1Colour, player2Colour)
```

```
TextBox1.draw() # Draw the first text box
TextBox2.draw() # Draw the second text box

player1Name = TextBox1.text # Update player 1 name with the text from the first text box
player2Name = TextBox2.text # Update player 2 name with the text from the second text box
```

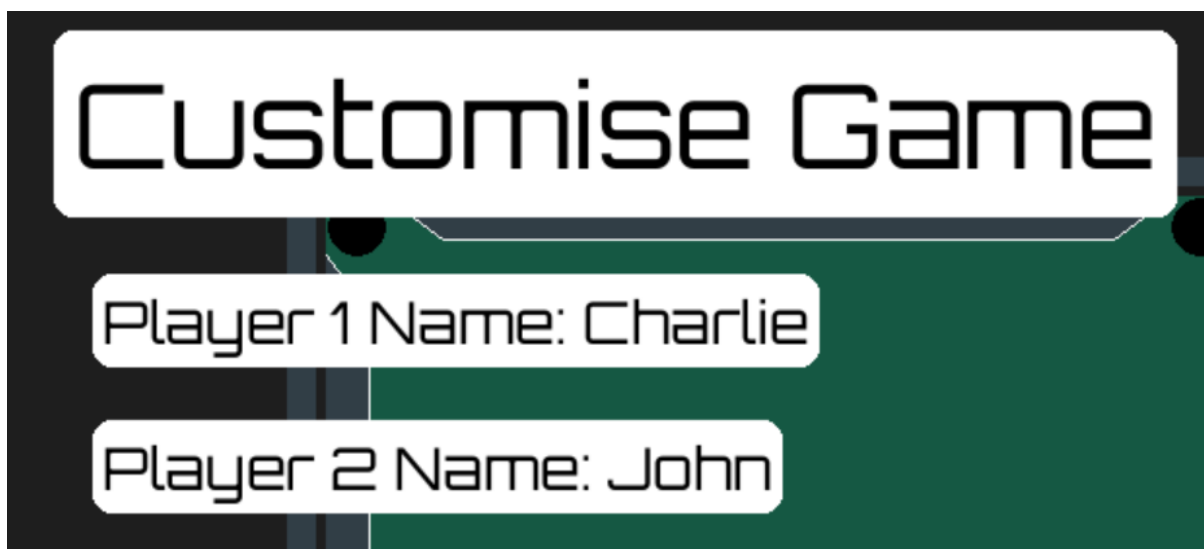
Made some changes here so that the menu.py file could update both the colours and the names of the users playing.

```
def newGame(computerPlaying, sandbox, player1Nametemp, player2Nametemp, player1Colourtemp, player2Colourtemp):
    global gameQueue
    global pottedBalls
    global roundPottedBalls
    global mouseDown

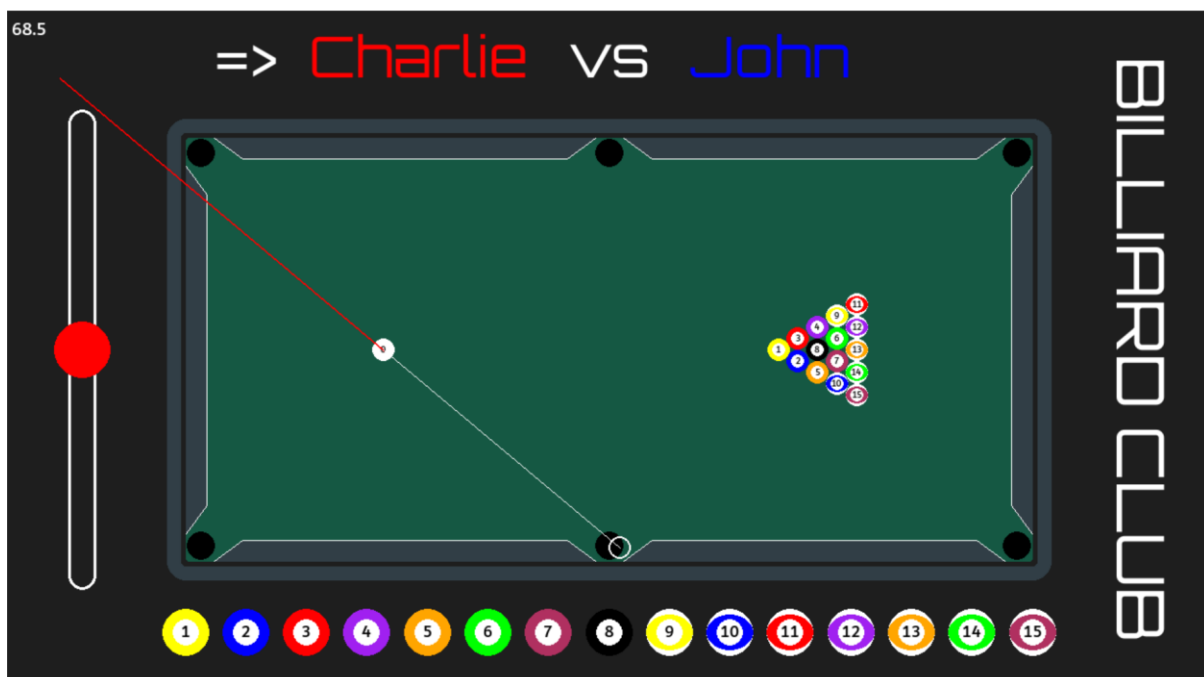
    global player1Name
    global player2Name
    global player1Colour
    global player2Colour

    player1Name = player1Nametemp
    player2Name = player2Nametemp
    player1Colour = player1Colourtemp
    player2Colour = player2Colourtemp
```

All working now:



Updated here. Now going to play a game:



All working now.

Now I will add a colour slider for each player:

Here I define a new class:

```
# Slider Class
class HorizontalSlider(pygame.sprite.Sprite):
    def __init__(self, x, y, width, height, startValue, sliderColour, outerColour):
        self.outerColour = outerColour # Colour of the outer section
        self.sliderColour = sliderColour # Colour of the slider
        self.x = x # X position of the slider
        self.y = y # Y position of the slider
        self.width = width # Width of the slider
        self.height = height # Height of the slider
        self.value = startValue # Starting value of the slider [0,1]
        self.rect = pygame.Rect(x, y, width, height) # Rectangle representing the slider's position and size

    def draw(self):
        pygame.draw.rect(screen, self.sliderColour, self.rect) # Draw the slider rectangle
        circlePos = [self.x + self.value * self.width, self.y + self.height / 2] # Calculate the position of the slider circle
        pygame.draw.circle(screen, "white", circlePos, self.height * 0.6) # Draw the slider circle
```

Then I test it works by doing the following:

```
slider1 = HorizontalSlider(500,500, 200,50, 0, "orange", "white")
slider1.draw()
```

This produces this result:



This is good, but now I need to add more colours. Here I have added a check to see if the slider should be a colour slider. If it is, it will draw a spectrum with the following code:

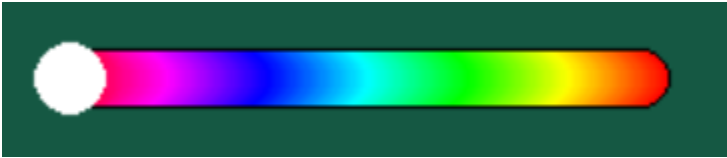
```
def draw(self):
    pygame.draw.rect(screen, self.sliderColour, self.rect, 0, 10) # Draw the slider rectangle

    if self.sliderType == "colourPicker": # Check if the slider type is a colour picker
        for i in range(self.width): # Iterate through the width of the slider
            colour = pygame.Color(0) # Create a new colour object
            colour.hsla = (int(360 * i / self.width), 100, 50, 100) # Set the colour based on the position
            pygame.draw.rect(screen, colour, (self.x, self.y, self.width - i, self.height), 0, 10) # Draw the colour rectangle

    pygame.draw.rect(screen, self.outerColour, self.rect, 1, 10) # Draw the outer slider rectangle
    circlePos = [self.x + self.value * self.width, self.y + self.height / 2] # Calculate the position of the slider circle
    pygame.draw.circle(screen, "white", circlePos, self.height * 0.6) # Draw the slider circle
```

This is working well as shown.

```
slider1 = HorizontalSlider(500,500, 200, 20, 0, "orange", "black", "colourPicker")
```



Made a small change to get red to appear first:

```
def draw(self):
    pygame.draw.rect(screen, self.sliderColour, self.rect, 0, 10) # Draw the slider rectangle
    if self.sliderType == "colourPicker": # Check if the slider type is a colour picker
        for i in range(self.width-18): # Iterate through the width of the slider
            colour = pygame.Color(0) # Create a new colour object
            colour.hsla = (int(360 * i / self.width), 100, 50, 100) # Set the colour based on the position
            pygame.draw.rect(screen, colour, (self.x + i, self.y, self.width-i, self.height), 0, 10) # Draw the colour rectangle
        pygame.draw.rect(screen, self.outerColour, self.rect, 1, 10) # Draw the outer slider rectangle
    pygame.draw.circle(screen, "white", self.circlePos, self.height * 0.6) # Draw the slider circle
```



Now I calculate the correct colour and draw the correct colour for the colour of the circle.

```
def draw(self):
    pygame.draw.rect(screen, self.sliderColour, self.rect, 0, 10) # Draw the slider rectangle
    if self.sliderType == "colourPicker": # Check if the slider type is a colour picker
        for i in range(self.width-18): # Iterate through the width of the slider
            colour = pygame.Color(0) # Create a new colour object
            colour.hsla = (int(360 * i / self.width), 100, 50, 100) # Set the colour based on the position
            pygame.draw.rect(screen, colour, (self.x+i, self.y, self.width - i, self.height), 0, 10) # Draw the colour rectangle
        pygame.draw.rect(screen, self.outerColour, self.rect, 1, 10) # Draw the outer slider rectangle

    # calculate selected colour
    selectedColour = pygame.Color(0)
    selectedColour.hsla = (int(self.value * self.width), 100, 50, 100)

    pygame.draw.circle(screen, "black", self.circlePos, self.height * 0.6 + 1) # Draw outer circle
    pygame.draw.circle(screen, selectedColour, self.circlePos, self.height * 0.6) # Draw the slider circle
```

I have also added a black border around the circle.

```
def update(self):
    if self.clicked: # when clicked
        mousePos = pygame.mouse.get_pos() # get mouse position
        self.circlePos[0] = min(max(self.x, mousePos[0]), self.x + self.width) # update position
        self.value = (self.circlePos[0] - self.x) / self.width # update value (between 0 and 1)
```

This is now fully functional:



Changed code so the class is compatible with both a colour picker and a normal slider:

```
def draw(self):
    pygame.draw.rect(screen, self.sliderColour, self.rect, 0, 10) # Draw the slider rectangle
    if self.sliderType == "colourPicker": # Check if the slider type is a colour picker
        for i in range(self.width-18): # Iterate through the width of the slider
            colour = pygame.Color(0) # Create a new colour object
            colour.hsla = (int(360 * i / self.width), 100, 50, 100) # Set the colour based on the position
            pygame.draw.rect(screen, colour, (self.x+i, self.y, self.width - i, self.height), 0, 10) # Draw the colour rectangle

    pygame.draw.rect(screen, self.outerColour, self.rect, 1, 10) # Draw the outer slider rectangle

    pygame.draw.circle(screen, self.outerColour, self.circlePos, self.height * 0.6 + 1) # Draw outer circle

    if self.sliderType == "colourPicker":
        pygame.draw.circle(screen, self.getColour(), self.circlePos, self.height * 0.6) # Draw the slider circle
    else:
        pygame.draw.circle(screen, self.circleColour, self.circlePos, self.height * 0.6) # Draw the slider circle
```

```
def getColour(self):
    # calculate selected colour
    selectedColour = pygame.Color(0)
    selectedColour.hsla = (int(self.value * (self.width + 30)), 100, 50, 100)
    return selectedColour

def returnValue(self):
    self.value = (self.circlePos[0]-self.x)/self.width # update value (between 0 and 1)
    return self.value
```

Next, I have set up the menu screen with the following code:

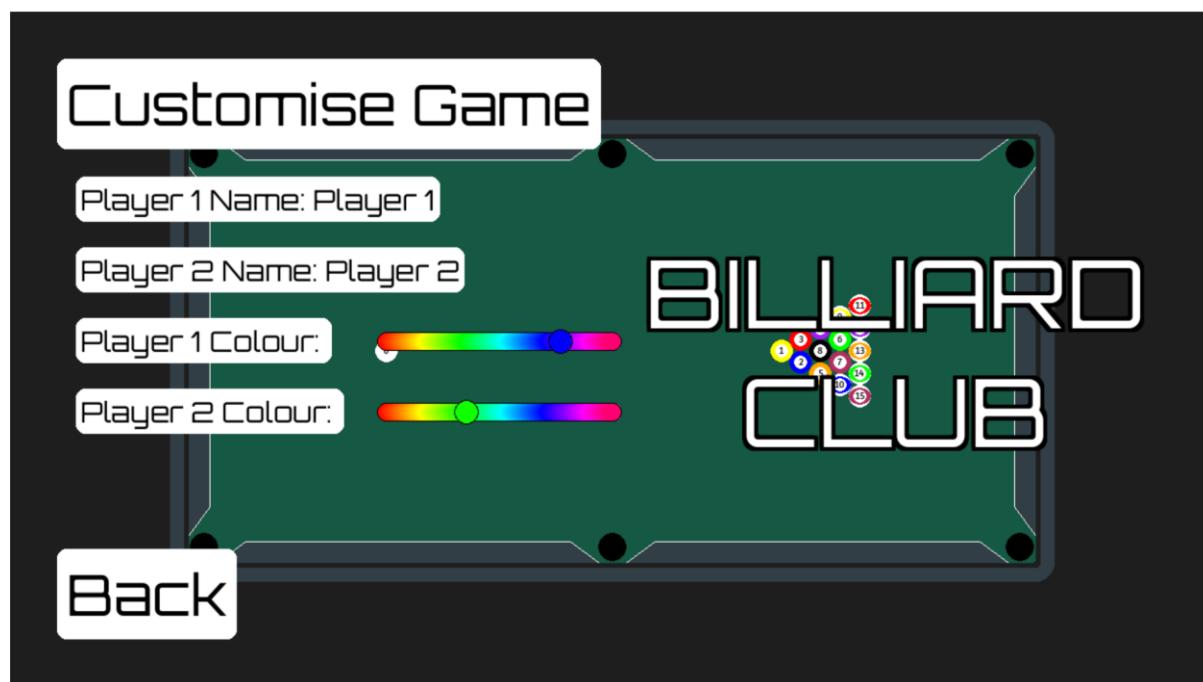
```
def customisationScreen(): # Main loop for the customisation screen
    global player1Name
    global player2Name

    if player2Name == "Computer":
        player2Name == "Player 2"

    TextBox1 = TextBox(70, 175, "Player 1 Name: ", player1Name, 30)
    TextBox2 = TextBox(70, 250, "Player 2 Name: ", player2Name, 30)

    TextBox3 = TextBox(70, 325, "Player 1 Colour: ", "", 30)
    TextBox4 = TextBox(70, 400, "Player 2 Colour: ", "", 30)

    colourSlider1 = HorizontalSlider(390, 340, 260, 20, 0, "orange", "black", "white", "colourPicker")
    colourSlider2 = HorizontalSlider(390, 415, 260, 20, 0, "orange", "black", "white", "colourPicker")
```



This is looking good now.

Small change swapped the order of the elements:

```

TextBox1 = TextBox(70, 175, "Player 1 Name: ", player1Name, 30)
TextBox3 = TextBox(70, 250, "Player 1 Colour: ", "", 30)
TextBox2 = TextBox(70, 325, "Player 2 Name: ", player2Name, 30)
TextBox4 = TextBox(70, 400, "Player 2 Colour: ", "", 30)
colourSlider1 = HorizontalSlider(390, 265, 260, 20, 0, "orange", "black", "white", "colourPicker")
colourSlider2 = HorizontalSlider(390, 415, 260, 20, 0.7, "orange", "black", "white", "colourPicker")

```

Now

looks like this:



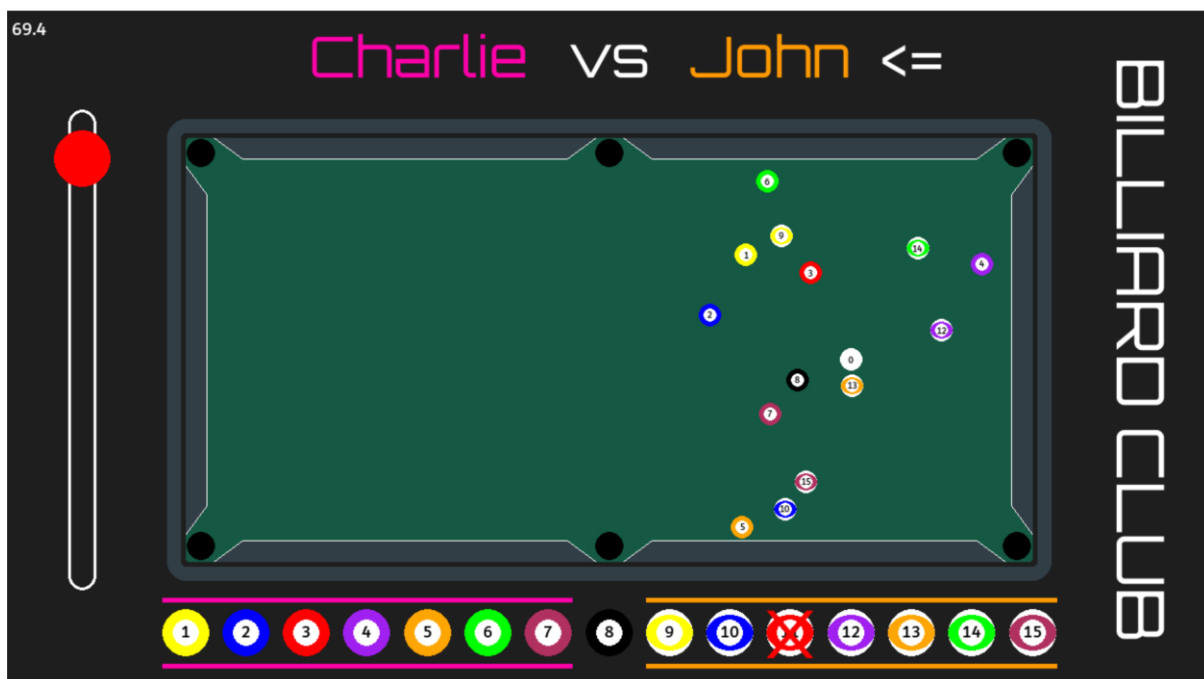
Now to update the colour when a new game is played:

```
def drawPlayerIndicator(): # Draws the player indicator
    # render font
    player1Text = playerIndicatorFont.render((player1Name), True, player1Colour)
    vsText = playerIndicatorFont.render(("vs"), True, "white")
    player2Text = playerIndicatorFont.render((player2Name), True, player2Colour)
```

Testing this, Updated the following information:



On player vs player:



The colours and names are all successfully there!

The next task is to implement a Pause Screen for when the user is in a game or has finished a game.

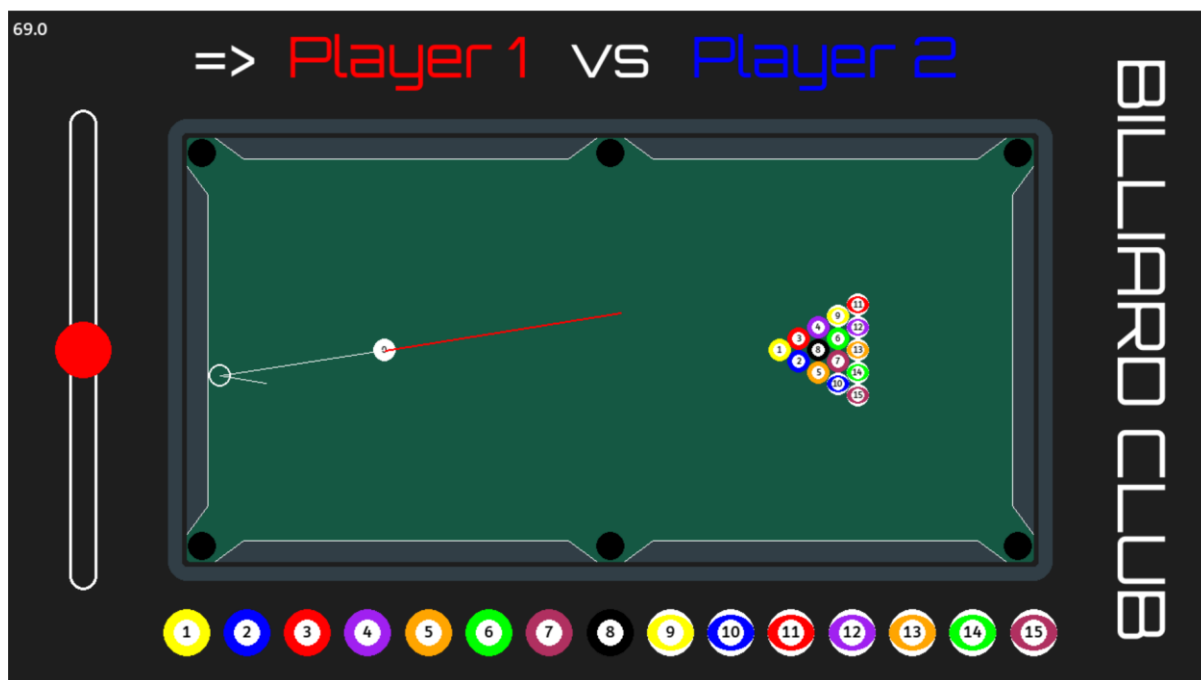
```
def pauseScreen():
    running = True
    while running:
        for event in pygame.event.get(): # Iterate through the events
            if event.type == pygame.QUIT: # Check if the quit event is triggered
                running = False # Exit the loop
            if event.type == pygame.KEYUP: # Check if a key is released
                if event.key == pygame.K_ESCAPE: # Check if the key is ESCAPE
                    running = False # Exit the loop

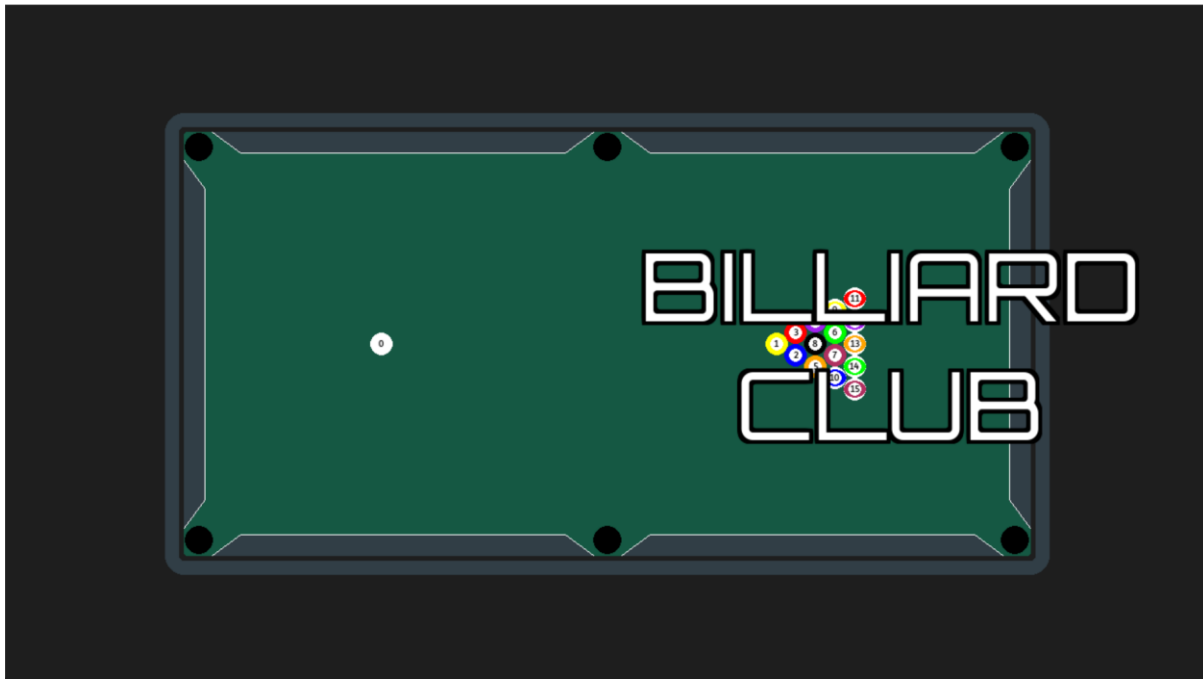
        screen.fill((30, 30, 30)) # Fills the screen with a grey colour
        drawTable() # Draw the table
        drawBalls() # Draw the balls
        drawMenuLogo() # Draw the menu logo
        pygame.display.flip() # Update the display
```

Updated the main game loop:

```
# Check if a key is released
if event.type == pygame.KEYUP:
    if event.key == pygame.K_ESCAPE: # Check if the key is ESCAPE
        pauseScreen() # Call the pause screen function
```

Now on press of the escape key, a menu shows:





This now works.

I will now populate this with options:

When trying to populate the screen, I have run into some errors as I have tried to carry out circular imports. This is due to the code being separated into two files. This can simply be fixed by importing all the functions into one python file.

This has been done here:

```

import pygame
import math

pygame.init()

# Variables
FPS = 70
resolution = (1280, 720) # Screen dimensions
borderName = "Billiard Club" # Displays at the top of the program
VEC = pygame.math.Vector2
root3 = math.sqrt(3)
friction = 0.991
restitutionCoeff = 0.95

# Pygame Functions and Setup
pygame.display.set_caption(borderName)
screen = pygame.display.set_mode(resolution) # Sets up the display
clock = pygame.time.Clock()

# Table Variables and Setup
primaryTableColour = "#155843"
secondaryTableColour = "#313e46"
outerTableColour = "#313e46"
tableDimensions = (900, 450)
centre = (resolution[0]//2, resolution[1]//2)
holeRadius = 15
ballRadius = 12

# Coordinates for the Table Corners.
tableCorners = (...)

# Coordinates for Hole Positions
holePos = (...)

# Coordinates for Table Rails
tableRails = (...)

# Ball info
ballInfo = ( # COLOUR, NUMBER, STRIPED...)

# Define where each ball should be relative to the start position
ballStartPos = (...)

# Some Variables
computerPlaying = True # Set to True if you want the computer to play
player1Turn = True # Initially player 1's turn
playerBallType = None # Stores the type of balls player 1 is putting
player2BallType = None # Stores the type of balls player 2 is putting
winner = None # Stores the winner of the game
playerName = "Player 1"
player2Name = "Player 2"
playerColour = "red"
player2Colour = "blue"

# Fonts
ballFont = pygame.font.SysFont("bahmunicode", int(ballRadius*0.9))
ballInfoFont = pygame.font.SysFont("bahmunicode", 18)
fpsFont = pygame.font.SysFont("bahmunicode", 20)
logoFont = pygame.font.Font("ProjectOther/Orbitron-VariableFont_wght.ttf", 70)
ballInfoOffsetFont = pygame.font.Font("ProjectOther/Orbitron-VariableFont_wght.ttf", 70)
playerIndicatorFont = pygame.font.Font("ProjectOther/Orbitron-VariableFont_wght.ttf", 60)
menuLogoFont = pygame.font.Font("ProjectOther/Orbitron-VariableFont_wght.ttf", 100)

# Classes
class Ball(pygame.sprite.Sprite): # Defines a class for a ball...
class VerticalSlider(pygame.sprite.Sprite): # Vertical Slider Class...
class HorizontalSlider(pygame.sprite.Sprite): # Horizontal Slider Class...
class Button: # Defines Button class...
class Menu: # Defines Menu class...
class Textbox: # Defines Textbox class...

# Initialize Menu
mainMenu = Menu(0, [0,0], ["Start", "Options", "Quit"], ["gameHoleScreen()", "optionsScreen()", "quitGame()"]) # create a menu
gameHoleMenu = Menu(0, [0,0], ["Player vs Player", "Player vs AI", "Settings Menu", "Back"], ["random(False, False)", "random(True, False)", "random(False, True)", "mainMenuScreen()"]) # create a menu
optionsMenu = Menu(0, [0,0], ["Customize Game", "Volume Control", "Back"], ["customisationScreen()", "soundScreen()", "mainMenuScreen()"]) # create a menu
customisationMenu = Menu(0, [0,0], ["Customize Game", "Back"], ["customisationScreen()", "optionsScreen()"]) # create a menu
soundMenu = Menu(0, [0,0], ["Game Paused", "Exit Game"], ["print()", "mainMenuScreen()"]) # create a menu

# Initialize Slider sprite
ballPowerSlider = VerticalSlider("white", "red", (tableCorners[0][0]//2 - 30, tableCorners[0][1]-30), 30, tableDimensions[1] + 60, 8.5)

# Initialize each ball
balls = []

ballGroup = pygame.sprite.Group() # Groups all balls for easier maintenance

for i in balls: # adds each ball to the group...

# Some Functions
def checkIfBallMoving(): # Checks if any ball is moving (returns True if any ball is moving)...
def distanceToClosestBall(initialPos, type): # Returns distance from closest ball...
def distanceToClosestBall(initialPos): # Returns distance from closest rail...
def resolveAllCollisions(): # Checks and resolves every collision between every pair of balls (only if colliding)...
def resolveWallCollisions(): # Checks and resolves every collision between every ball and the walls...
def resolveBallPotted(): # Checks if a ball is potted...
def resetWhiteBall(): # Resets the white ball...
def gameLogic(): # determines state of next round on all factors...
def checkWinLose(): # checks if there is a winner...
def computerShoot(): # Computer finds what and takes what...
def drawEndLogo():
def quitGame(): # Function to quit the game...

# Draw Elements to Screen
def drawTable(): # Draw the table...
def drawBalls(): # Draw all the balls...
def drawBallUI(isGadshu): # Draw 33 balls on screen for UI (NOT ON TABLE)...
def drawLogo(): # Draw the logo...
def drawFPS(): # FPS Display...
def drawCue(): # Draw the cue...
def drawPlayerIndicator(): # Draw the player indicator...

# Screens
def mainMenuScreen(): # Main loop for the main menu...
def gameHoleScreen(): # Main loop for the game hole selection screen...
def optionsScreen(): # Main loop for the options screen...
def customisationScreen(): # Main loop for the customisation screen...
def pauseScreen(): # Main loop for the pause screen...

# Run game
def runGame(computerPlaying, random): # Function to start a new game...

def endGame(computerPlaying, random, player1Name, player2Name, player1Colour, player2Colour):
mainMenuScreen()

```

Now I am able to create a menu for the pause screen.

```
def pauseScreen(): # Main loop for the pause screen
    running = True
    while running:
        for event in pygame.event.get(): # Iterate through the events
            if event.type == pygame.QUIT: # Check if the quit event is triggered
                running = False # Exit the loop
            if event.type == pygame.KEYDOWN: # Check if a key is released
                if event.key == pygame.K_ESCAPE: # Check if the key is ESCAPE
                    running = False # Exit the loop
                pauseMenu.handleEvent(event)

        screen.fill((30, 30, 30)) # Fills the screen with a grey colour
        drawTable() # Draw the table
        drawBalls() # Draw the balls
        drawMenuLogo() # Draw the Logo for the menu screen

        pauseMenu.draw() # draw the menu
        pauseMenu.update() # Update the menu
        pygame.display.flip() # Update the display
```

I then ensure this works by adding this menu when the escape key is pressed:

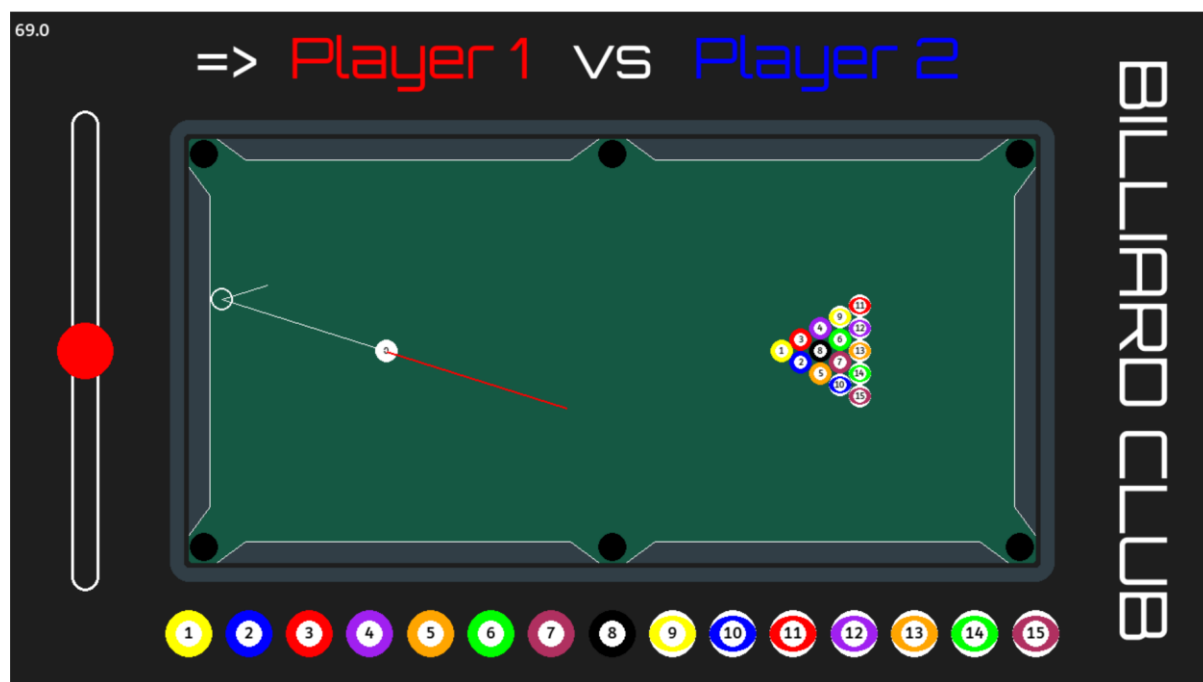
```
# Game Loop
while True:
    mouseDown = False

    for event in pygame.event.get():
        if event.type == pygame.QUIT: # Checks if user closes program...

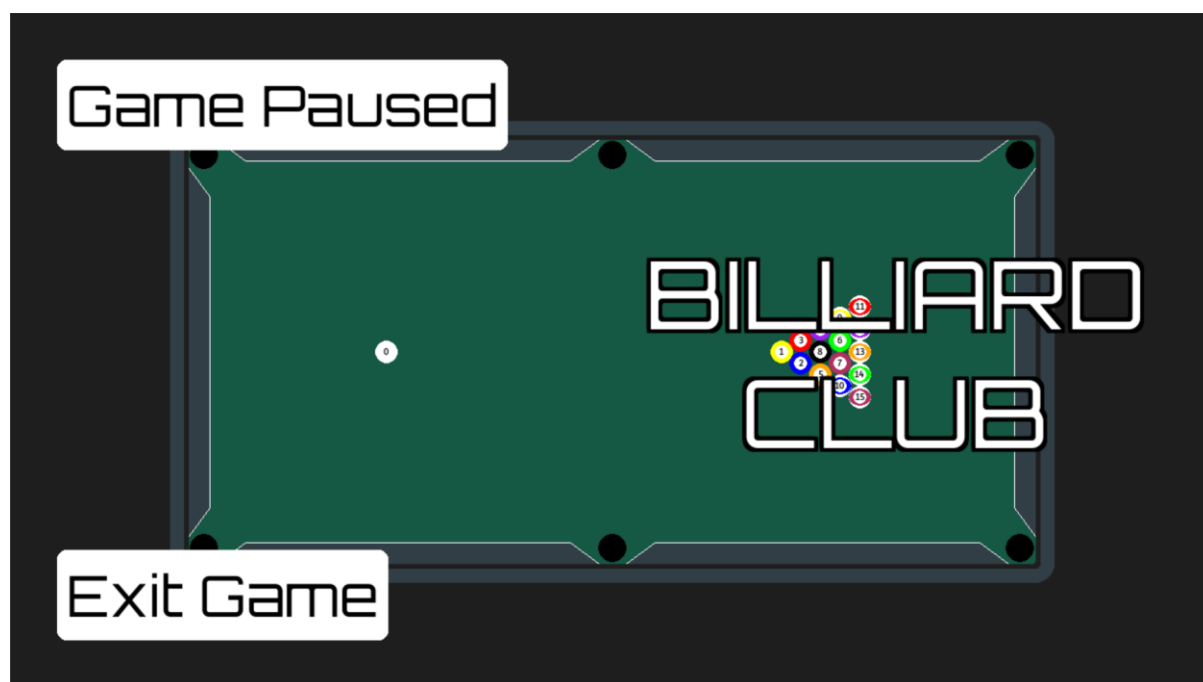
        if event.type == pygame.MOUSEBUTTONDOWN: # on left click...

        if event.type == pygame.KEYDOWN: # Check if any key is pressed
            if event.key == pygame.K_ESCAPE: # Check if the key is ESCAPE
                pauseScreen() # Call the pause screen function
```

This now works as shown:



On press of the escape key, this pause screen is shown:

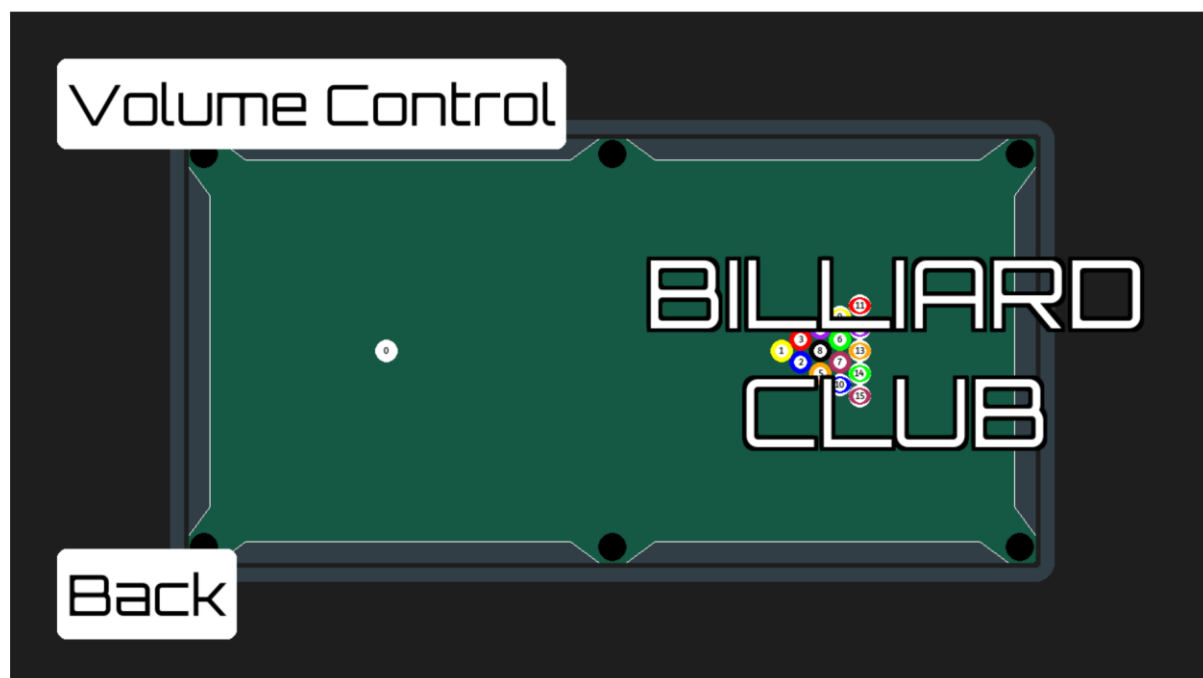


Now I will add a volume control screen:

```
volumeMenu = Menu(60, [50,50], ["Volume Control", "Back"], ["volumeControlScreen()", "optionsScreen()"]) # create a menu
```

```
def volumeControlScreen(): # Main loop for the volume control screen
    running = True
    while running:
        for event in pygame.event.get(): # Iterate through the events
            if event.type == pygame.QUIT: # Check if the quit event is triggered
                running = False # Exit the loop
            volumeMenu.handleEvent(event)
        screen.fill((30, 30, 30)) # Fills the screen with a grey colour
        drawTable() # Draw the table
        drawBalls() # Draw the balls
        drawMenuLogo() # Draw the Logo for the menu screen
        volumeMenu.draw() # draw the menu
        volumeMenu.update() # Update the menu
        pygame.display.flip() # Update the display
```

This code produces the following result:



Now I will add a few sliders that will control the sounds for the music, ball sounds, and master volume.

This required a slight tweak in the horizontal slider class as before the slider was always drawing a spectrum.

```
def draw(self):
    pygame.draw.rect(screen, self.outerColour, self.rect, 0, 10) # Draw the slider rectangle
    if self.sliderType == "colourPicker": # Check if the slider type is a colour picker
        for i in range(self.width-18): # Iterate through the width of the slider
            colour = pygame.Color(0) # Create a new colour object
            colour.hsla = (int(360 * i / self.width), 100, 50, 100) # Set the colour based on the position
            pygame.draw.rect(screen, colour, (self.x+i, self.y, self.width - i, self.height), 0, 10) # Draw the colour rectangle
    else:
        pygame.draw.rect(screen, self.sliderColour, (self.x, self.y, self.width*self.value, self.height), 0, 10) # Draw the colour rectangle
    pygame.draw.rect(screen, self.outerColour, self.rect, 1, 10) # Draw the outer slider rectangle
    pygame.draw.circle(screen, self.outerColour, self.circlePos, self.height * 0.6 + 1) # Draw outer circle
    if self.sliderType == "colourPicker":
        pygame.draw.circle(screen, self.getColour(), self.circlePos, self.height * 0.6) # Draw the slider circle
    else:
        pygame.draw.circle(screen, self.circleColour, self.circlePos, self.height * 0.6) # Draw the slider circle
```

Here I initialised the volume sliders.

```
# Initialise Sliders
masterVolumeSlider = HorizontalSlider(390, 265, 260, 20, 1, "green", □(30, 30, 30), "white", "normal")
musicVolumeSlider = HorizontalSlider(390, 415, 260, 20, 1, "green", □(30, 30, 30), "white", "normal")
ballVolumeSlider = HorizontalSlider(390, 575, 260, 20, 1, "green", □(30, 30, 30), "white", "normal")
```

Handled all the events:

```
masterVolumeSlider.handleEvent(event)
musicVolumeSlider.handleEvent(event)
ballVolumeSlider.handleEvent(event)
```

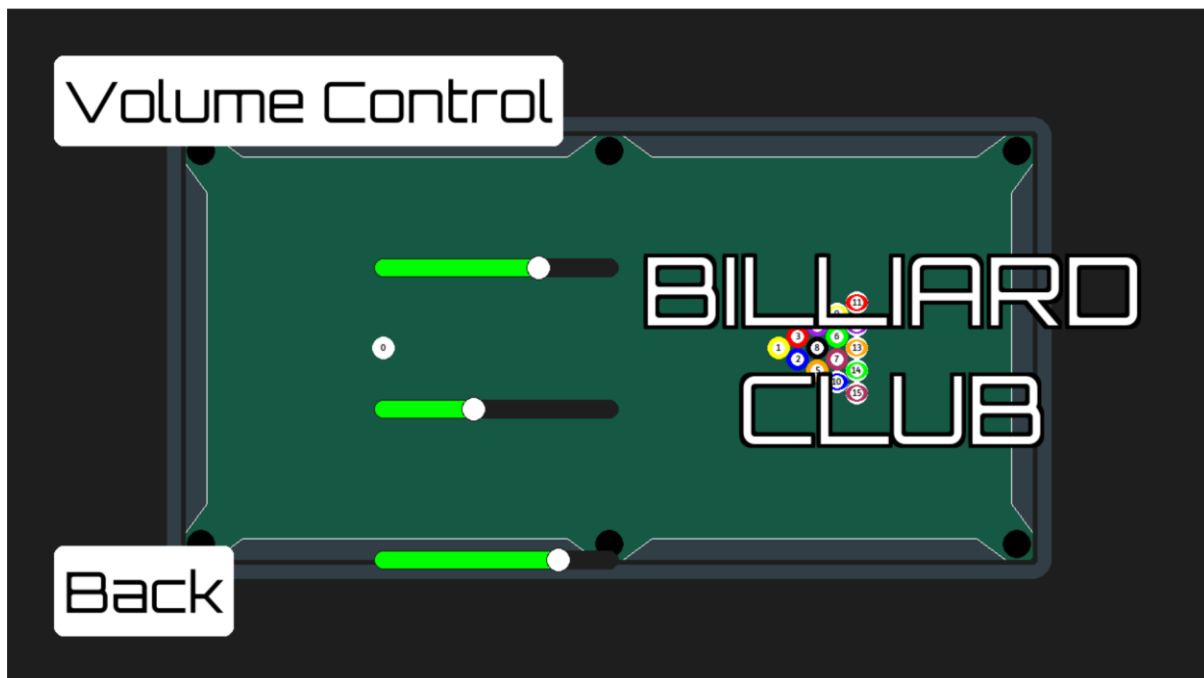
Then drew and updated all the sliders to the screen:

```
masterVolumeSlider.update() # Update master volume slider
masterVolumeSlider.draw() # Draw master volume slider

musicVolumeSlider.update() # Update music volume slider
musicVolumeSlider.draw() # Draw music volume slider

ballVolumeSlider.update() # Update ball volume slider
ballVolumeSlider.draw() # Draw ball volume slider
```

This successfully draws the sliders to the screen:



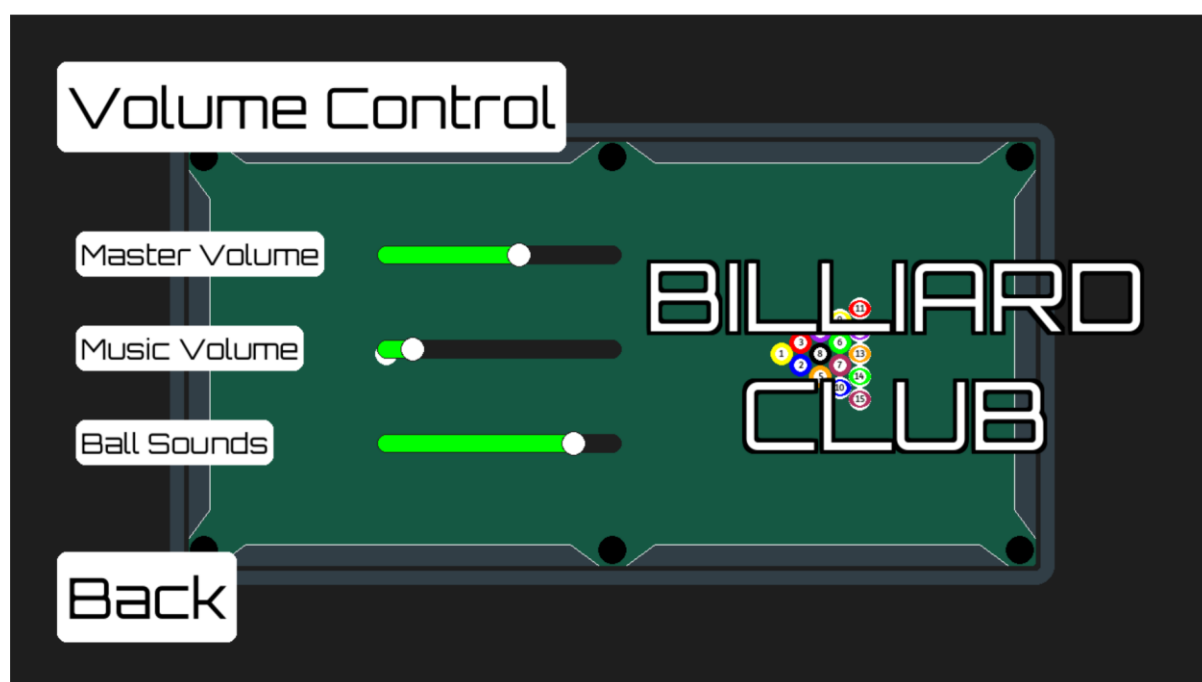
Next, I will add labels to the screen, e.g. Master Volume etc.

```
# Initialise Sliders
masterVolumeSlider = HorizontalSlider(390, 245, 260, 20, 1, "green", □(30, 30, 30), "white", "normal")
musicVolumeSlider = HorizontalSlider(390, 345, 260, 20, 1, "green", □(30, 30, 30), "white", "normal")
ballVolumeSlider = HorizontalSlider(390, 445, 260, 20, 1, "green", □(30, 30, 30), "white", "normal")

# Initialise Text Boxes
TextBox1 = TextBox(70, 230, "Master Volume", "", 30)
TextBox2 = TextBox(70, 330, "Music Volume", "", 30)
TextBox3 = TextBox(70, 430, "Ball Sounds", "", 30)
```

```
TextBox1.draw() # Draw the Master Volume text box
TextBox2.draw() # Draw the Music Volume text box
TextBox3.draw() # Draw the Ball sounds text box
```

Working really well now:



Now all the menu elements have been implemented and are working, this stage is complete.

Stage 6 Review

What has been done?

- Added a Button class to display a button which also handles mouse interactions/events.
- Added a Menu class to group Button classes together into a menu.
- Cleaned up code and integrated buttons into the Menu class.
- Introduced multiple screens (main menu, options, choose game, etc.) and ensured proper navigation between them.
- Added visual improvements for the menus: a pool table background, logo, and customisations for options menu like colour changes and text boxes for player names.
- Implemented menus for different game modes (Player vs Computer, Player vs Player)
- Created a pause screen
- Final Stage: Integrated all menus and functionalities (buttons, game modes, volume control) into a complete, interactive game interface.

How is code being tested?

- For each new element/function (e.g., ball movement, wall collisions, ball-to-ball collisions), I add a small bit of functionality and then observe the result, documenting and debugging it accordingly. This incremental approach ensures I follow the computational method of decomposition which ensures for swift development of the solution.
- Bugs (like clipping, stuck balls, or zero-distance errors) are identified and resolved by modifying code of the program, rerunning the simulation, and documenting my errors.
- Comments are used extensively throughout the solution. This helps debugging enormously as I sift through hundreds of lines of code.

Success criteria that has been met:

4	The game should have an intuitive user interface, where the user can navigate to all menus.
24	The game is visually aesthetic/appealing.
26	The game should include a practice mode

Errors that have been fixed:

Error	Fix
When the menu was created, the selected option did not update in sync with the mouse hover.	A new button class was introduced that creates a collision rectangle around each option, which allowed the menu to update.
The buttons were not changing colour when the mouse was hovering over them.	I added an update function to the button class that individually checks the hover state of each button and updates its colour appropriately.
A line of code was not indented properly, causing a syntax error.	The line was correctly indented.

The callback function linked to the button click is being called before the button could fully update, meaning the button briefly appears grey.	The callback execution was modified to occur only after the button is released, ensuring that the button state resets correctly.
In the Player vs Player mode, Player 2 was renamed to "Computer"	New variables for player names were defined, meaning no errors are now present.
The text box where the player enters their name was not resizing according to the text length.	The text surface was re-rendered every time the text input changed so that the text box's width updates correctly.
When creating the pause screen menu, errors occurred because of circular imports.	All the code was put into one file, which removes the issue of circular imports.
The horizontal slider component was always drawing a colour spectrum.	The slider class was tweaked to check whether it should render as a colour slider, ensuring that the spectrum is only drawn when a continuous spectrum is called.

Summary of Stage 6

Stage 6 has focused on creating an interactive menu system with buttons, multiple game modes, and customisation options. It includes features like a pause screen, volume control sliders, and smooth navigation between menus, all with aesthetic improvements for a dynamic user interface.

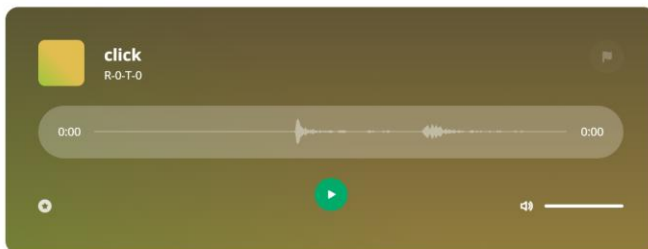
Stage 7: Audio

Now that the majority of the game has been completed and implemented, I am going to implement some audio for the game. This will include ball sounds and game music. Here is a preliminary list of game audios that I will implement.



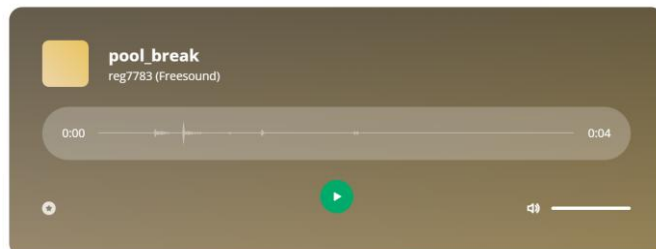
AI Generated Music, Song, No copyright music music. Free for use.

<https://pixabay.com/music/beats-chill-house-mellow-moments-251693/>



Click, Mouse, Clicker sound effect. Free for use.

<https://pixabay.com/sound-effects/click-151673/>



Break, Cue, Billiards sound effect. Free for use.

<https://pixabay.com/sound-effects/pool-break-105353/>



Billiard, Pool, Billiards sound effect. Free for use.

<https://pixabay.com/sound-effects/billiard-hole-288419/>

What are you allowed to do with Content?

Subject to the Prohibited Uses (see below), the Content License allows users to:

- ✓ Use Content for free
- ✓ Use Content without having to attribute the author (although giving credit is always appreciated by our community!)
- ✓ Modify or adapt Content into new works

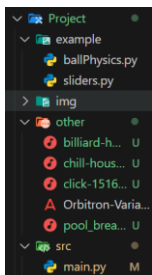
What are you not allowed to do with Content?

We refer to these as Prohibited Uses which include:

- ✗ You cannot sell or distribute Content (either in digital or physical form) on a Standalone basis. Standalone means where no creative effort has been applied to the Content and it remains in substantially the same form as it exists on our website.
- ✗ If Content contains any recognisable trademarks, logos or brands, you cannot use that Content for commercial purposes in relation to goods and services. In particular, you cannot print that Content on merchandise or other physical products for sale.
- ✗ You cannot use Content in any immoral or illegal way, especially Content which features recognisable people.
- ✗ You cannot use Content in a misleading or deceptive way.
- ✗ You cannot use any of the Content as part of a trade-mark, design-mark, trade-name, business name or service mark.

I have downloaded all the following sounds from the website <https://pixabay.com/>. I have also ensured that each sound is free to use under the following rules. Due to the nature of the project being non-commercial, I am free to use, distribute and edit these sound files.

I have uploaded each audio file into the game directory as shown here:



Next, I have used the pygame mixer module to play the background music indefinitely.

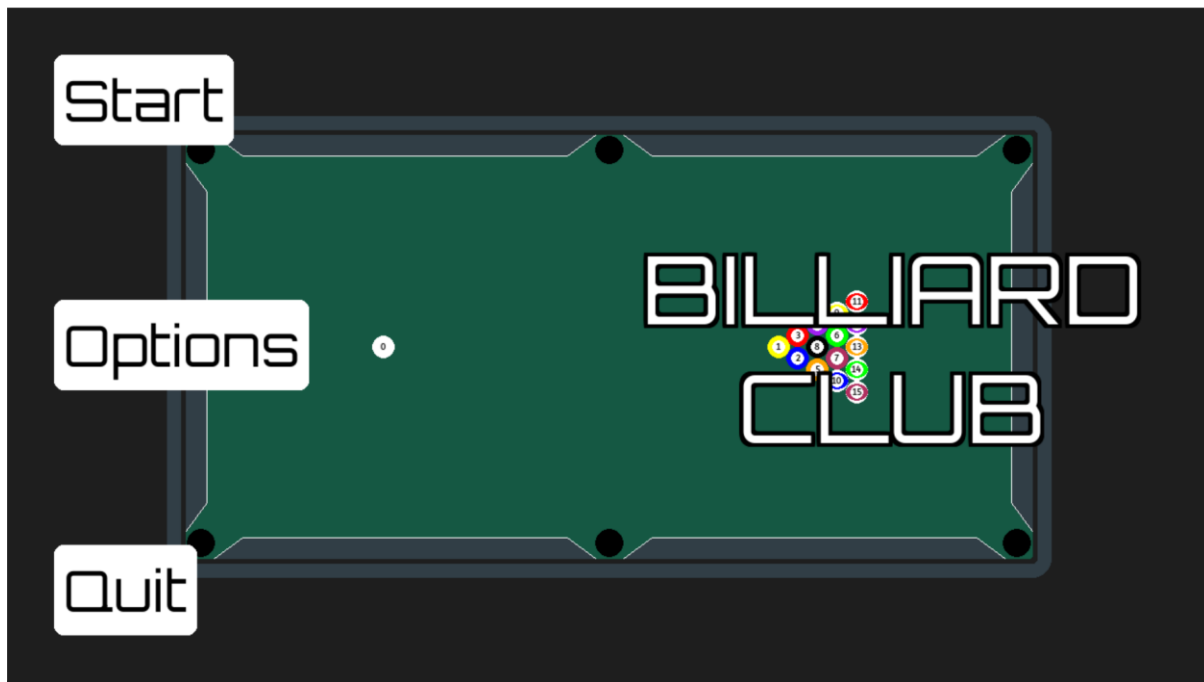
And I have stored the shorter sound effects as variables so that they can be played elsewhere in the game loop.

```
pygame.init() # Initialize all imported pygame modules

# Play Game Music in background indefinitely
pygame.mixer.music.load("Project/other/chill-house-sunset-groove-251697.mp3")
pygame.mixer.music.play(-1,0.0)

# Initialise Music/Sounds
clickSound = pygame.mixer.Sound("Project/other/click-151673.mp3")
ballCollideSound = pygame.mixer.Sound("Project/other/pool_break-105353.mp3")
ballPottedSound = pygame.mixer.Sound("Project/other/billiard-hole-288419.mp3")
```

On run of the game, the game plays the background music here.



Now I will play the other sound effects in the correct locations.

In the ball class, when a collision is calculated, the game will play the colliding sound:

```
if calculateCollision: # if colliding
    ballCollideSound.play() # plays sound when balls collide
```

In the menu class, when a button is clicked, the game will play the click sound:

```
def handleEvent(self, event): # Method to handle events
    if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # Check if the left mouse button is clicked
        for button in self.buttons: # Iterate through each button
            if button.checkHover(event.pos): # Check if click occurred and if hovering over button
                clickSound.play() # plays click sound
                button.wasClicked = True # Set button as clicked
```

When a ball reaches the hole, the game will play the ball potted sound.

```
def resolveBallPotted(): # Checks if a ball is potted
    for ball in ballGroup.sprites():
        if ball.checkHoleCollision(): # checks if each ball is in a hole
            ballPottedSound.play() # plays sound when ball potted
```

On a quick test, this each sound has successfully worked.

Now I need to implement a way of changing the volume of each of the sounds via the slider I implemented in the previous stage and also hook this up to a master volume slider.

While trying to implement this, I have found an error that occurs from the previous section:

```
def volumeControlScreen(): # Main loop for the volume control screen

    # Initialise Sliders
    masterVolumeSlider = HorizontalSlider(390, 245, 260, 20, 1, "green", □(30, 30, 30), "white", "normal")
    musicVolumeSlider = HorizontalSlider(390, 345, 260, 20, 1, "green", □(30, 30, 30), "white", "normal")
    ballVolumeSlider = HorizontalSlider(390, 445, 260, 20, 1, "green", □(30, 30, 30), "white", "normal")
```

Here the function to display the screen will keep initialising the volume sliders every time the screen is displayed. This results in the values resetting each time. To fix this, I will move the initialisation of the sliders outside the procedure.

This is done by moving the sliders to the class initialisation section of code:

```
# Initialise Slider sprite
ballPowerSlider = VerticalSlider("white", "red", (tableCorners[0][0]/2 - 30,tableCorners[0][1]-30), 30, tableDi

# Initialise Sliders
masterVolumeSlider = HorizontalSlider(390, 245, 260, 20, 1, "green", □(30, 30, 30), "white", "normal")
musicVolumeSlider = HorizontalSlider(390, 345, 260, 20, 1, "green", □(30, 30, 30), "white", "normal")
ballVolumeSlider = HorizontalSlider(390, 445, 260, 20, 1, "green", □(30, 30, 30), "white", "normal")
```

Now the value of the slider does not reset after exiting the procedure.

Now I must link this value to the volume of the sounds throughout the game.

In the volume screen method, I have set the volumes of the sounds to the values of the sliders on the screen. (initially the value of the sliders are all 1)

```
# Updates the volumes of the Sound
clickSound.set_volume(masterVolumeSlider.returnValue()*ballVolumeSlider.returnValue())
ballCollideSound.set_volume(masterVolumeSlider.returnValue()*ballVolumeSlider.returnValue())
ballPottedSound.set_volume(masterVolumeSlider.returnValue()*ballVolumeSlider.returnValue())

# Updates Background Music Volume
pygame.mixer.music.set_volume(masterVolumeSlider.returnValue()*musicVolumeSlider.returnValue())
```

The game now dynamically responds by changing the volume when the sliders are used:



However, on play, when the white ball is potted, the game will continually play the ball potted sound.

This will be fixed here:

Here I have noticed that this section of code is being run multiple times which plays the sound.

```

654 def resolveBallPotted(): # Checks if a ball is potted
655 |     toDelete = [] # deletes balls after looped through list
656 |     for ball in ballGroup.sprites():
657 |         if ball.checkHoleCollision(): # checks if each ball is in a hole
658 |             ballPottedSound.play() # plays sound when ball potted
659 |             if ball.number == 0: # if the white ball is potted
660 |                 print("if this is printed multiple times then somethings wrong")
661 |                 gameQueue.append(resetWhiteBall) # tells the game to reset the white ball on the start of
662 |                 ball.velocity = VEC(0,0) # resets the velocity of the white ball
663 |                 ball.pos = VEC(0,0) # resets the position of the white ball
664 |             else:
665 |                 # On first run, set the ball type for each player

```

PROBLEMS 18 OUTPUT DEBUG CONSOLE TERMINAL 18

▼ TERMINAL

```

if this is printed multiple times then somethings wrong
if this is printed multiple times then somethings wrong
if this is printed multiple times then somethings wrong
if this is printed multiple times then somethings wrong
if this is printed multiple times then somethings wrong
if this is printed multiple times then somethings wrong

```

What is happening is that the ball seems to be colliding with a hole even after the ball has been moved off the screen, causing the sound to be played over and over.

After much thinking, the cause of this problem was that the white ball was getting updated to be back on the screen even after moving it. This meant that the ball was back in the hole, causing a sound to be played, and then moving again, this caused a loop until the ball was reset to its original position.

This has been fixed with the following code in the update method of the ball class:

```

# Create exception for white ball:
if self.pos == VEC(0,0):
    return None

```

Now when the white ball is potted, the sounds are not repeated.

Another problem I am having is the idea of the ball collide sound being constant regardless of the speed of collision. Here I will make the volume of the collision dynamically change based on the velocities of the balls.

```

# Calculate volume of ball collision sound
collisionSpeed = VEC(relVel).magnitude() # obtains rough value of speed
volume = masterVolumeSlider.returnValue()*ballVolumeSlider.returnValue()*(collisionSpeed/2)
ballCollideSound.set_volume(max(0.1,min(1,volume))) # updates volume of sound
ballCollideSound.play() # plays sound when balls collide

```

This is now working well!

All the audio is now correctly functioning.

Stage 7 Review

What has been done?

- I have downloaded and integrated free-to-use audio files from the website Pixabay, for background music and pool sound effects.
- I used the pygame mixer module to manage and play background music indefinitely on launch.
- Assigned sound effects to ball collisions, menu button clicks, and ball potting events.
- Gave volume sliders functionality to adjust sound levels.
- Implemented the ball sound volume to dynamically change based on speed of collision
- Successfully tested and confirmed all audio features function correctly.

How is code being tested?

- For each new element/function (e.g., ball movement, wall collisions, ball-to-ball collisions), I add a small bit of functionality and then observe the result, documenting and debugging it accordingly. This incremental approach ensures I follow the computational method of decomposition which ensures for swift development of the solution.
- Bugs (like clipping, stuck balls, or zero-distance errors) are identified and resolved by modifying code of the program, rerunning the simulation, and documenting my errors.
- Comments are used extensively throughout the solution. This helps debugging enormously as I sift through hundreds of lines of code.

Success criteria that has been met:

18	The game will have a music track to be played in the background.
19	A relevant sound will be played if the player pots a ball
20	A relevant sound will be played when balls collide.
21	When balls collide, the system should play a sound with a suitable volume, depending on the speed of collision.
22	The game should play a click sound every time a menu is clicked.

Errors that have been fixed:

Error	Fix
Every time the screen was displayed, the volume sliders were reinitialised.	The initialisation of the sliders was moved out of the display function and placed into the initialisation section of code (ensures saved values are kept).

When the white ball was potted, the game continuously played the ball potted sound.	The ball's update method was modified to prevent the repeated triggering of the ball potted sound via a check.
---	--

Summary of Stage 7

Stage 7 saw Audio implemented using Pygame mixer with background music and sound effects for collisions, clicks, and ball potting. Volume sliders were updated to retain volume values and now dynamically adjust sound levels. A looping potting sound bug was resolved, and collision volume now varies with speed. All audio features were successfully implemented and tested.

Stage 8: Final Additions

There are a few things that are needed to be added to reach all the must-haves for the success criteria. These will be addressed here by adding them to the game. I will also add other elements that will increase the aesthetics and quality of the overall game.

To improve the quality of the game, I will reset the power slider and change the colour of the slider, so it matches the player's colour.

A set method for the vertical power slider:

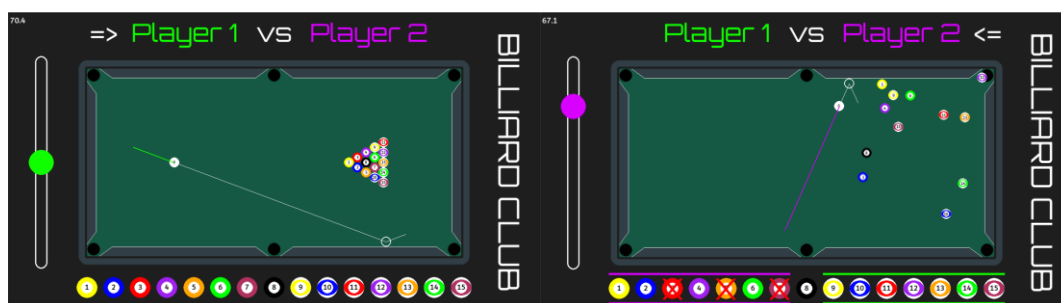
```
def updateColour(self, newColour): # updates the colour of the slider
    self.sliderColour = newColour
```

When the player's turn changes, the colour of the slider will be updated and then the position of the slider is also set to 0.5 times power.

```
if changeTurn: # if the turn is to be changed
    player1Turn = not player1Turn # switch turns
    if player1Turn:
        ballPowerSlider.updateColour(player1Colour) # update the colour of the power slider
    else:
        ballPowerSlider.updateColour(player2Colour) # update the colour of the power slider

    ballPowerSlider.value = 0.5 # reset the power slider
```

This is shown to be working:



Next, I am going to improve the aesthetics of the menu screens by adding a blurring function to the screen.

After doing a bit of research, a gaussian blur is the best approach to this.

Here I use the python library OpenCV to apply a gaussian blur to the array of pixels collected from the game.

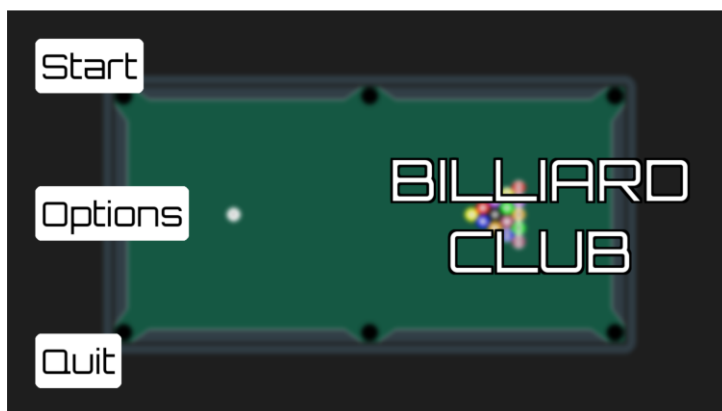
```
def gaussianBlur(screen, radius):
    # uses cv2 to apply a gaussian blur to the screen
    screenArray = np.transpose(pygame.surfarray.array3d(screen), [1, 0, 2]) # Convert screen to array and transpose axes for cv2
    blurredSurface = np.transpose(cv2.GaussianBlur(screenArray, (radius, radius), 0), [1, 0, 2]) # Apply Gaussian blur and transpose back to original format
    blurredSurface = pygame.surfarray.make_surface(blurredSurface) # Convert blurred array back into a pygame surface
    screen.blit(blurredSurface, [0, 0]) # Draw the blurred surface onto the screen
```

This function is then used in each screen method to blur the previous contents that have been drawn to the screen.

```
# Screens
def mainMenuScreen(): # Main loop for the main menu
    running = True
    while running:
        for menuEvent in pygame.event.get(): # Iterate through the events...

            screen.fill((30, 30, 30)) # Fills the screen with a grey colour
            drawTable() # Draws the table on the screen
            drawBalls() # Draws the balls on the screen
            gaussianBlur(screen, 25) # Apply a gaussian blur to the screen
            drawMenuLogo() # Draws the menu logo on the screen
            mainMenu.draw() # Draw the menu
            mainMenu.update() # Update the menu
            pygame.display.flip() # Update the display
```

This is shown to have work:



The screen is now blurred.

Next, I am going to add a proper win screen. At the moment, the win screen looks like this:



I am going to add a new method for the win screen, which is shown here:

This will be done by creating a new menu class:

```
winMenu = Menu(60, [50,50], ["Play Again", "Exit to Menu"], ["print()", "mainMenuScreen()"]) # create a menu
```

Then on win, the winScreen function will be called, passing the winner as a parameter.

```
def checkWin(winner): # checks if there is a winner
    if winner != None: # if there is a winner
        winScreen(winner) # display the win screen
```

This will make the game run the win screen main loop.

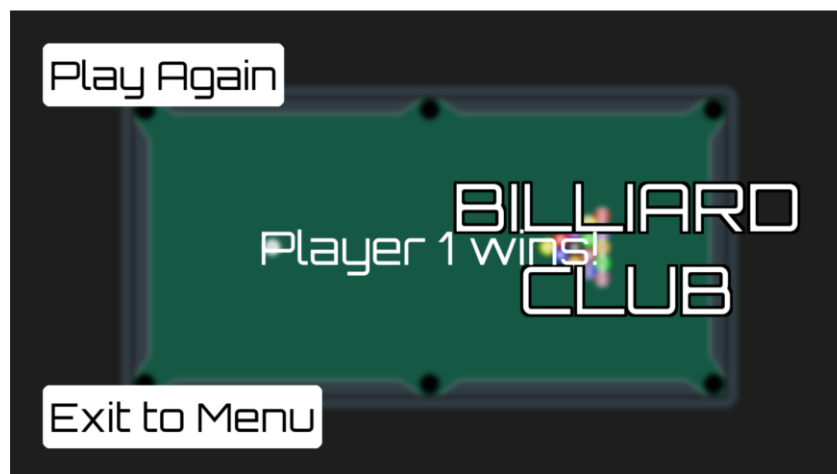
```
def winScreen(winner): # Main loop for the win screen
    running = True
    while running:
        for event in pygame.event.get(): # Iterate through the events
            if event.type == pygame.QUIT: # Check if the quit event is triggered
                running = False # Exit the loop
            if event.type == pygame.KEYDOWN: # Check if a key is released
                if event.key == pygame.K_ESCAPE: # Check if the key is ESCAPE
                    running = False # Exit the loop
            winMenu.handleEvent(event)

        screen.fill((30, 30, 30)) # Fills the screen with a grey colour
        drawTable() # Draw the table
        drawBalls() # Draw the balls
        gaussianBlur(screen, 25) # Apply a gaussian blur to the screen
        drawMenuLogo() # Draw the Logo for the menu screen

        winText = logoFont.render(winner + " wins!", True, "white") # renders the font
        winTextPos = [centre[0] - winText.get_width()/2, centre[1] - winText.get_height()/2] # offsets the font to be central
        screen.blit(winText, winTextPos) # adds font to screen

        winMenu.draw() # draw the menu
        winMenu.update() # Update the menu
        pygame.display.flip() # Update the display
```

On win, the program displays the following:



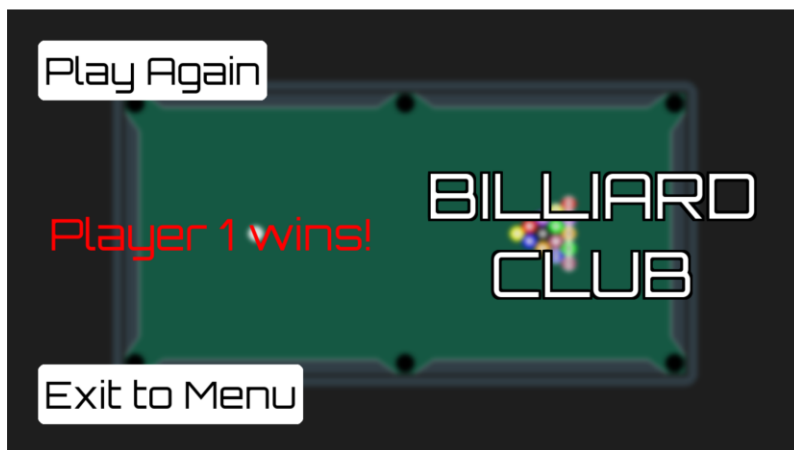
I need to centre the winner and maybe put the winner in their chosen colour.

This is done here:

```
def winScreen(winner, colour): # Main loop for the win screen
    running = True
    while running:
        for event in pygame.event.get(): # Iterate through the events
            if event.type == pygame.QUIT: # Check if the quit event is triggered
                running = False # Exit the loop
            if event.type == pygame.KEYDOWN: # Check if a key is released
                if event.key == pygame.K_ESCAPE: # Check if the key is ESCAPE
                    running = False # Exit the loop
            winMenu.handleEvent(event)

        screen.fill((30, 30, 30)) # Fills the screen with a grey colour
        drawTable() # Draw the table
        drawBalls() # Draw the balls
        gaussianBlur(screen, 25) # Apply a gaussian blur to the screen
        drawMenuLogo() # Draw the Logo for the menu screen
        winText = logoFont.render(winner + " wins!", True, colour) # renders the font
        winTextPos = [centre[0] - winText.get_width()*1.1, centre[1] - winText.get_height()/2] # offsets the font to be central
        screen.blit(winText, winTextPos) # adds font to screen

        winMenu.draw() # draw the menu
        winMenu.update() # Update the menu
        pygame.display.flip() # Update the display
```

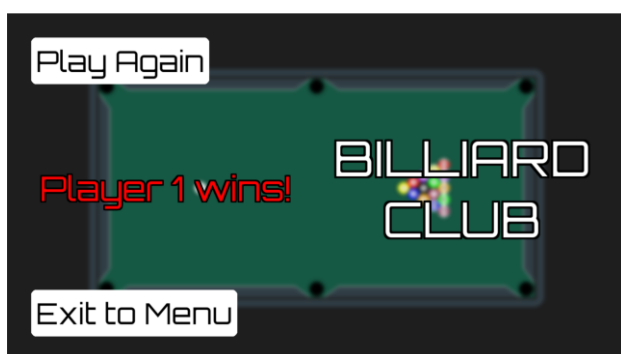


To make this more aesthetic, I will add an outline to this.

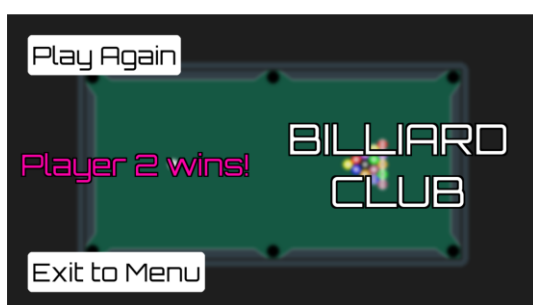
```
winText = logoFont.render(winner + " wins!", True, colour) # renders the font
winOutlineText = logoFont.render(winner + " wins!", True, "black") # renders the font

winTextPos = [centre[0] - winText.get_width()*1.1, centre[1] - winText.get_height()/2] # offsets the font to be central
# Draw the outline of the logo
for i in range(-3,4):
    for j in range(-3,4):
        screen.blit(winOutlineText, (winTextPos[0]+i, winTextPos[1]+j))

screen.blit(winText, winTextPos) # adds font to screen
```



This now works for both players:



At the moment, the play again button does not work, here is the code to ensure the play again button functions:

```
winMenu = Menu(60, [50,50], ["Play Again", "Exit to Menu"], ["gameModeScreen()", "mainMenuScreen()"]) # create a menu
```

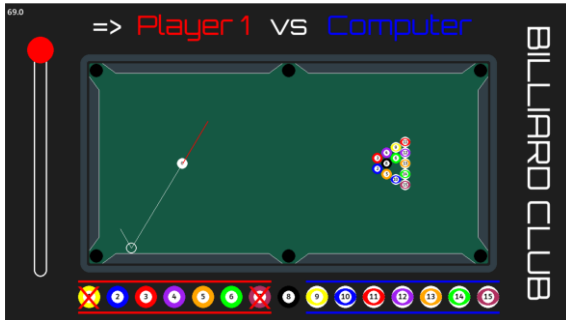
One final problem is that when the user has played a game, when the player wants to play another game, the ball positions are not reset and therefore the player is met with the previous win screen.

This will be fixed here.

When the new game method is called, all the ball positions are reset.

```
# reset all ball positions
for ball in ballGroup.sprites():
    ball.pos = ballStartPos[ball.colour + str(ball.number)] # reset ball position
    ball.velocity = VEC(0, 0) # set ball velocity to zero
```

This is proved to be working, but if the user has potted balls, then those balls have been deleted. Therefore, they are not able to be put back on the table and therefore there are missing balls on the table:



To fix this, I will delete each ball and then re-instantiate each ball on the table. This means I will not need to reset the ball positions and therefore can delete the previous code.

```
ballGroup = pygame.sprite.Group() # Groups all balls for easier maintenance

def initialiseBalls():
    # Initialise each ball
    balls = [
        Ball("white", ballRadius, ballStartPos["white0"], (0, 0), 1, 0, None), # Whiteball
        Ball("yellow", ballRadius, ballStartPos["yellow1"], (0, 0), 1, 1, False), # Yellow 1
        Ball("blue", ballRadius, ballStartPos["blue2"], (0, 0), 1, 2, False), # Blue 2
        Ball("red", ballRadius, ballStartPos["red3"], (0, 0), 1, 3, False), # Red 3
        Ball("purple", ballRadius, ballStartPos["purple4"], (0, 0), 1, 4, False), # Purple 4
        Ball("orange", ballRadius, ballStartPos["orange5"], (0, 0), 1, 5, False), # Orange 5
        Ball("green", ballRadius, ballStartPos["green6"], (0, 0), 1, 6, False), # Green 6
        Ball("maroon", ballRadius, ballStartPos["maroon7"], (0, 0), 1, 7, False), # Maroon 7
        Ball("black", ballRadius, ballStartPos["black8"], (0, 0), 1, 8, None), # Black 8
        Ball("yellow", ballRadius, ballStartPos["yellow9"], (0, 0), 1, 9, True), # Yellow 9
        Ball("blue", ballRadius, ballStartPos["blue10"], (0, 0), 1, 10, True), # Blue 10
        Ball("red", ballRadius, ballStartPos["red11"], (0, 0), 1, 11, True), # Red 11
        Ball("purple", ballRadius, ballStartPos["purple12"], (0, 0), 1, 12, True), # Purple 12
        Ball("orange", ballRadius, ballStartPos["orange13"], (0, 0), 1, 13, True), # Orange 13
        Ball("green", ballRadius, ballStartPos["green14"], (0, 0), 1, 14, True), # Green 14
        Ball("maroon", ballRadius, ballStartPos["maroon15"], (0, 0), 1, 15, True), # Maroon 15
    ]

    for i in balls: # adds each ball to the group
        ballGroup.add(i)

def deleteAllBalls():
    for ball in ballGroup.sprites(): # loops through all balls
        ball.delete() # deletes each ball

initialiseBalls() # call function straight away to initialise balls on table
```

Then when the main game loop is run, the balls are deleted then initialised back again to prevent any errors:

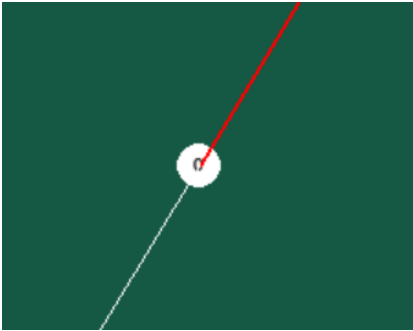
```
deleteAllBalls() # Deletes all balls
initialiseBalls() # Then Initialise all balls to ensure all balls on table
```

This is all now working but on re-run, the player's ball type must be reset, which has not been done. Therefore, this is done here:

```
winner = None # so there is no winner
player1BallType = None # Reset player 1 ball type
player2BallType = None # Reset player 2 ball type
```

Now the user can play as many games as they like.

One final change that needs to be made is the number of the ball is being drawn as 0 on the white ball.



This has been fixed here with the following code in the draw method in the ball class.

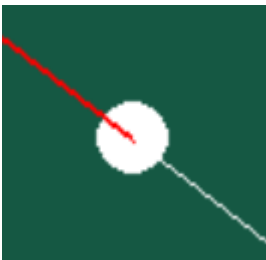
```
def draw(self, screen): # Draws the ball to the screen
    pygame.draw.circle(screen, self.colour, self.pos, self.radius) # draws outer circle

    if self.isStriped: ...

    if self.number != 0: # draws a number on every ball except white
        text = ballfont.render(str(self.number), True, "black") # renders font
        pygame.draw.circle(screen, "white", self.pos, text.get_height()/2.2) # draws inner circle

        fontPos = [self.pos[0] - text.get_width()/2, self.pos[1] - text.get_height()/2] # offsets the font to be central with ball

        screen.blit(text, fontPos) # adds font to screen
```



Everything seems now to be working.

Stage 8 Review

What has been done?

- The power slider now resets at each player's turn, updates its colour to match the current player and positions itself at half the power value at the start of every turn.
- A blurring function was used in the menu screens using a Gaussian blur function from an import.
- A new win screen was created which displays the winner in a more aesthetic way.
- On game reset, balls are deleted and then re-instantiated, ensuring referential integrity.

How is code being tested?

- For each new element/function (e.g., ball movement, wall collisions, ball-to-ball collisions), I add a small bit of functionality and then observe the result, documenting and debugging it accordingly. This incremental approach ensures I follow the computational method of decomposition which ensures for swift development of the solution.
- Bugs (like clipping, stuck balls, or zero-distance errors) are identified and resolved by modifying code of the program, rerunning the simulation, and documenting my errors.
- Comments are used extensively throughout the solution. This helps debugging enormously as I sift through hundreds of lines of code.

Success criteria that has been met:

10	After every game, the ball positions should be reset so a new game can be played after another.
17	The size of the game should be no more than 200MB
23	The game should run at 60fps or more for good performance.

Errors that have been fixed:

Error	Fix
The "Play Again" button was not functioning.	The code for the play again button was revised, and relevant code was added to ensure functionality.
When starting a new game, the ball positions were not properly reset.	The new game method was modified to reset all ball positions when a game is restarted.
If balls had been potted in the previous game, they were permanently deleted, meaning when a new game starts, some balls are missing.	I updated the code to delete each ball and then re-instantiate them on the table, ensuring all balls are on the table.
Upon restarting the game, the player's ball type was not being reset, potentially causing inconsistencies in gameplay.	Additional code was implemented to reset the player's ball type on game re-run, ensuring that the game state is properly refreshed, and players can continue playing seamlessly.

The 0 on the white ball was still being displayed.	I updated the code to add an exception that does not draw the number if the ball is the white ball.
--	---

Summary of Stage 8

Stage 8 saw the game's functionality and aesthetics improving significantly via resetting and recolouring the power slider every turn, adding a Gaussian blur to menu and pause screens, and creating a better win screen for players. The play again buttons now function correctly (with re-instantiating each ball).

Evaluation

Testing the final solution

In order to test the final solution, I will firstly conduct tests based on the test plan outlined in the design stage of this project. I will also conduct stress testing which will test for boundaries and ensure the system is robust and functions as intended. Although code has already been tested throughout development as mentioned previously in the stages of development, it is important that the whole system is tested as a whole, to ensure the program works as intended.

Testing against the Test Plan

In this section I will iterate through the test plan and outline how each test has been conducted and explain any issues or tests that failed. I will copy each section of the test plan, showing what the test is, how the test should be conducted, the expected result and the actual result of the program.

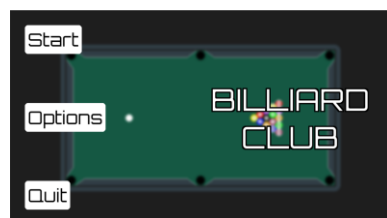
#	Item Tested	Purpose of Test	Method of Test	Expected Result	Link to Success Criteria
---	-------------	-----------------	----------------	-----------------	--------------------------

Above shows the relevant column titles of each of the test plans, which helps to outline which box contains the test to conduct, purpose, method, and result of the test.

Test 1

1	Game Start	Ensures game starts correctly	Open the executable file for the game	Game starts	1
---	------------	-------------------------------	---------------------------------------	-------------	---

When I open the executable file, I am greeted with this menu:

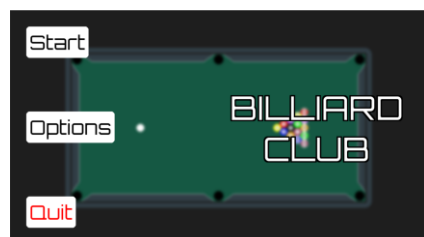


Therefore, this test is a success.

Test 2

2	Game Exit	Ensures the game starts	Press exit game in the menu	Game exits	4
---	-----------	-------------------------	-----------------------------	------------	---

When I press the quit button here, the game exits successfully.

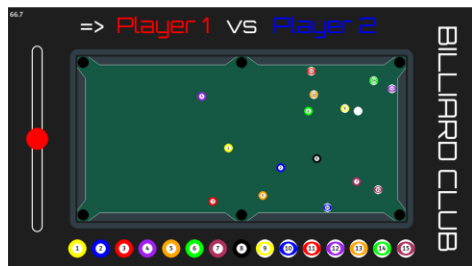


Therefore, this test is a success.

Test 3

3	Balls are given a velocity	Ensures the balls move on the table	Play the game, see if balls move.	Balls should move on table if playing	2
---	----------------------------	-------------------------------------	-----------------------------------	---------------------------------------	---

The balls are moving on the table after using the cue, shown here:



Therefore, this test is a success.

Test 4

4	Friction applied to balls	Ensures balls eventually stop moving	Play game and see if balls eventually stop	Balls should eventually stop moving	2
---	---------------------------	--------------------------------------	--	-------------------------------------	---

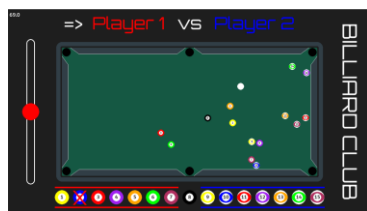
When playing the game, all balls are shown to slowly decrease their velocity and eventually come to a stand-still, due to a friction being applied to the balls. This is difficult to show with an image, but this has been tested.

Therefore, this test is a success.

Test 5

5	Ball-to-Ball collisions	Ensures ball-to-ball collisions are recognised and resolved	Play game and hit one ball into another	Balls should accurately collide and reflect off each other	11
---	-------------------------	---	---	--	----

When using the cue, I have shot a ball into the side of another ball, for an accurate collision, the ball must go left due to where the ball hits. This has been done in this image:



Therefore, this test is a success.

Test 6

6	Ball-to-Wall Collisions	Ensures ball-to-wall collisions are recognised and resolved	Play game and hit one ball into a wall/rail	Balls should accurately reflect off walls and rails	12
---	-------------------------	---	---	---	----

Here I shoot the ball into the wall, and it reflects off the wall accurately:

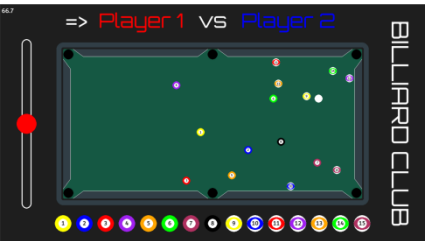


Therefore, this test is a success.

Test 7

7	Balls should remain within table	Ensures all the balls are interactable and game functions	Play game and after check balls after one round	All balls that are not potted, should still be on the table.	1
---	----------------------------------	---	---	--	---

All balls as shown are within the table bounds and continue to be throughout the game:



Therefore, this test is a success.

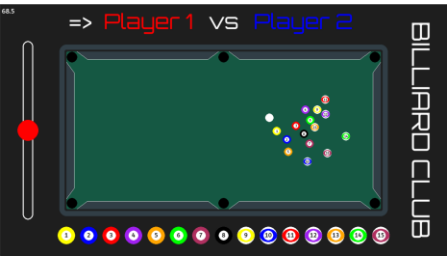
Test 8

8	Balls should handle multiple collisions	Ensures game calculates physics correctly	Break in the game, watch all the collisions occurring	Ensure the break is accurate.	11
---	---	---	---	-------------------------------	----

To test multiple collisions, I have reverted back to the beginning. I shoot the cue into the balls as shown:



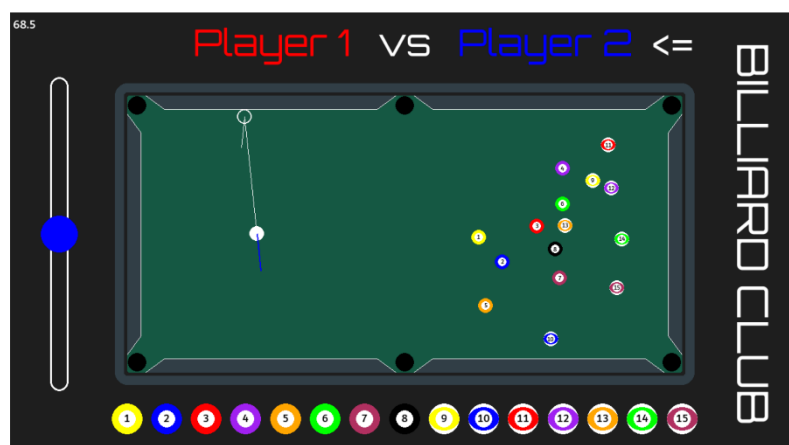
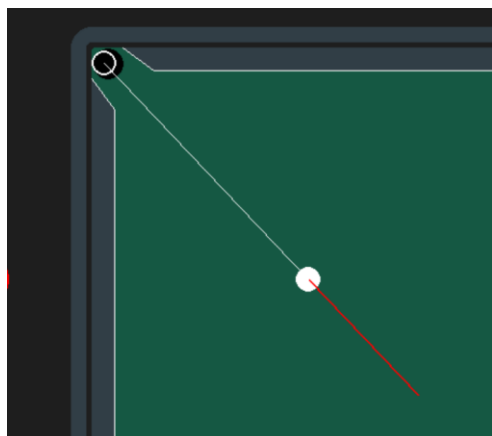
I then get the balls accurately moving away from each other as shown in this image:



Therefore, this test is a success.

Test 9

9	Potting White Ball	Ensures white ball is not deleted when potted	Pot the white ball	White ball should reappear after potted in the next round	5
---	--------------------	---	--------------------	---	---



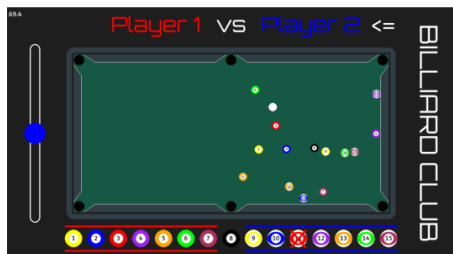
Here I pot the white ball and when I pot it, the game fouls player 1 and gives the turn to player 2.

Therefore, this test is a success.

Test 10

10	Pot any ball	Ensures any ball (except white) is deleted.	Pot any ball except the white ball	Ball should be deleted	5
----	--------------	---	------------------------------------	------------------------	---

Here I pot the number 11 ball, the ball is deleted.

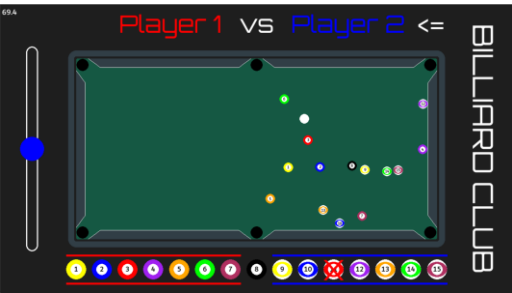


Therefore, this test is a success.

Test 11

11	Setting player type	Ensures a player is given a ball type on potting a ball	Pot a ball when no player types have been decided	Player should have that type of ball	15
----	---------------------	---	---	--------------------------------------	----

When player 2 pots the number 11 ball, the ball types have been decided, shown by the graphic on the bottom of the screen:

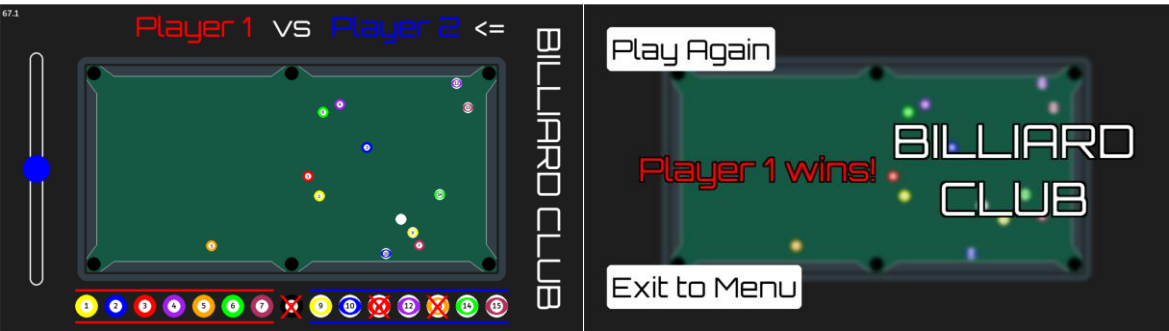


Therefore, this test is a success.

Test 12

12	Potting 8 ball	Ensures the game gives a win to someone	Pot the 8 ball	Player should either win/lose depending on if they potted all other balls	5
----	----------------	---	----------------	---	---

Here player 2 pots the 8 ball:

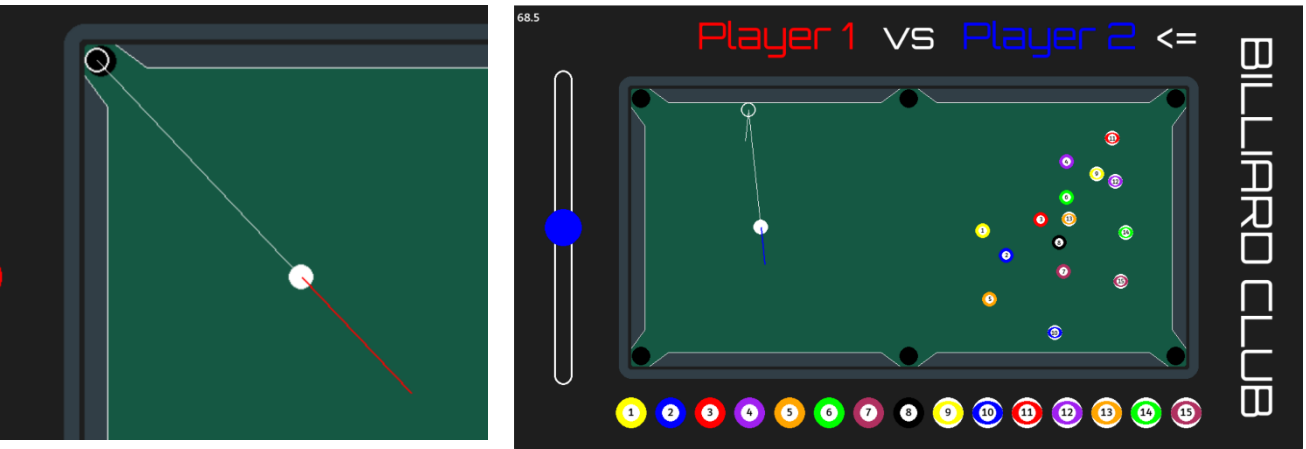


Then the game declares Player 1 the winner.

Therefore, this test is a success.

Test 13

13	Foul Detection	Ensures player turn switches if player fouls	Commit a foul (e.g. pot another player's ball)	Should switch turn	8
----	----------------	--	--	--------------------	---



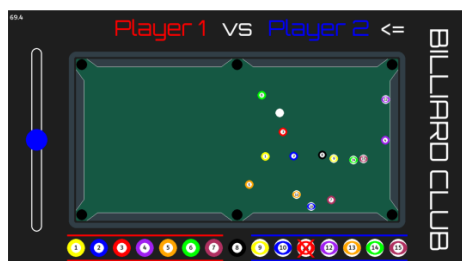
Here I pot the white ball and when I pot it, the game fouls player 1 and gives the turn to player 2.

Therefore, this test is a success.

Test 14

14	Ball Pocket Detection	Ensures balls are detected to be potted	Pot a ball	Ball should be deleted	5
----	-----------------------	---	------------	------------------------	---

Here I pot the number 11 ball, the ball is deleted.

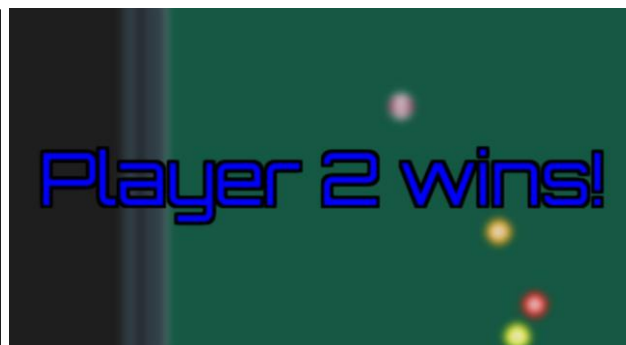
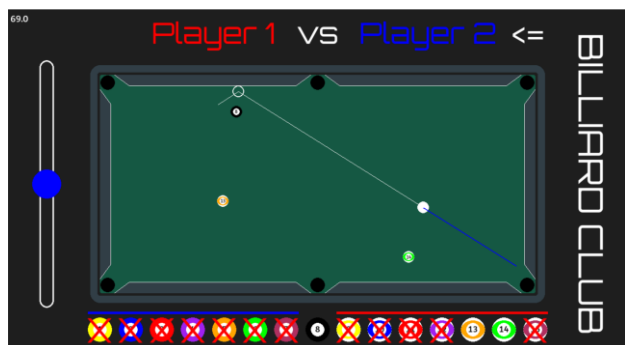


Therefore, this test is a success.

Test 15

15	Game End	When all balls have been potted, the game should end	Pot all the balls in the game	Game should give the win to the correct player.	8
----	----------	--	-------------------------------	---	---

Here, Player 2 has pot all but the 8 ball, therefore when player 2 pots the 8 ball, this win message is received:



Therefore, this test is a success.

Test 16

16	Win screen display on win	Ensures a winner is determined at the end of each game	Pot the 8 ball	Game should display the correct winner in an appropriate message	13
----	---------------------------	--	----------------	--	----

When player 2 pots all of the relevant balls, this win message should be received:

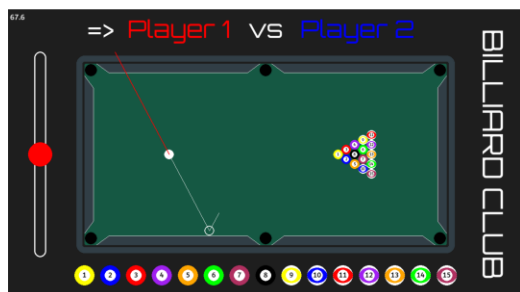


Therefore, this test is a success.

Test 17

17	Game initialisation	Ensures at the start of each game, all the balls are put in the correct place	Start a new game after another	Game should reset and a new game should start	10
----	---------------------	---	--------------------------------	---	----

Here I have started a new game after another, and all the balls are in the correct position:

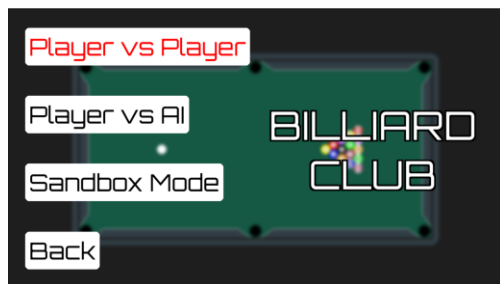


Therefore, this test is a success.

Test 18

18	Main Menu Start button	Ensures that when "start" is pressed, the game mode selection launches	Press "start" in the main menu	Directs user to game mode selection	4
----	------------------------	--	--------------------------------	-------------------------------------	---

On pressing start, I am navigated to this menu:

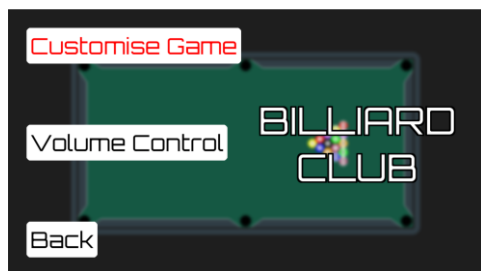


Therefore, this test is a success.

Test 19

19	Main menu Options button	Ensures that when options are pressed, the options screen launches	Press "options" in the main menu	Launches options screen	4
----	--------------------------	--	----------------------------------	-------------------------	---

On press of "options," I am brought to this screen:

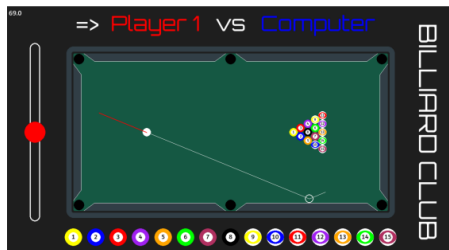


Therefore, this test is a success.

Test 20

20	Game Mode Selection	Ensures that each game mode is launched when pressed	Launch each game mode.	The correct game mode is launched	4
----	---------------------	--	------------------------	-----------------------------------	---

When I press each of the game modes, I am brought to the correct option, here I show that when I press “Player vs Computer,” I am brought to the game where I can verse the computer:



Therefore, this test is a success.

Test 21

21	Options Screen	Ensures each menu in the options screen is functional	Press each button in the options screen	Each button should link to the correct screen.	4
----	----------------	---	---	--	---

Each button on the menu screen is functional, as shown by the successful navigation to the volume screen:

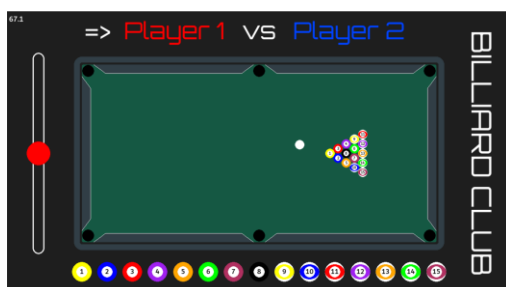


Therefore, this test is a success.

Test 22

22	Cue is drawn when user's go.	Ensures no cue is drawn when simulating	Run game and use cue.	After confirmation of shot, the cue should disappear	2
----	------------------------------	---	-----------------------	--	---

When the player has given the white ball speed, the cue is not drawn to the screen, and the game is left to simulate all the balls until they go still.

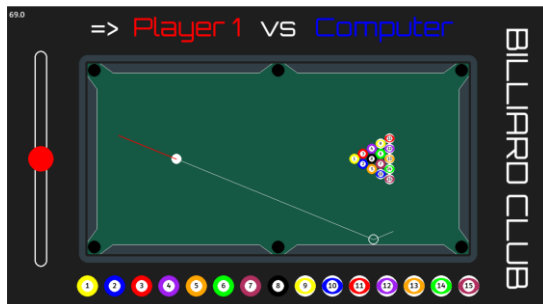


Therefore, this test is a success.

Test 23

23	Virtual Ball UI	When aiming, the virtual ball should be shown to the user	On player's turn verify the ball projection is drawn	Ball projection is drawn to screen	6
----	-----------------	---	--	------------------------------------	---

Here the game shows a virtual projection of the ball if the user were to hit the ball there and then.

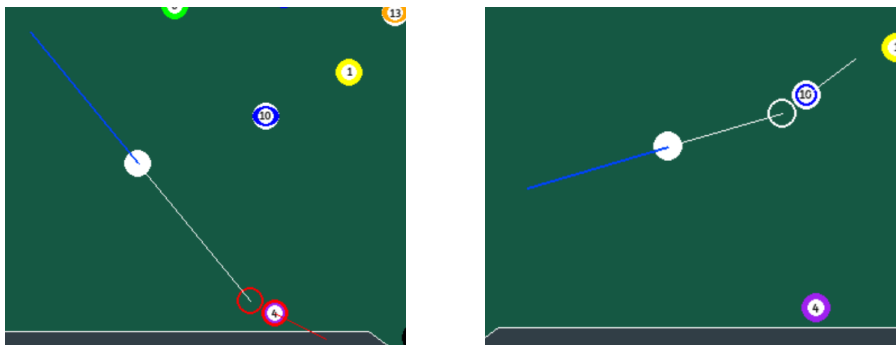


Therefore, this test is a success.

Test 24

24	Virtual Ball Colour Indicator	When aiming to another ball, the colour should change depending on the user's ball type	Aim the cue to a ball of the player's type and another not of their type.	Should indicate which ball not to hit	7
----	-------------------------------	---	---	---------------------------------------	---

Here I show that when the ball types have been set, the virtual ball indicator changes colour if the ball is going to hit a ball not of its own type.

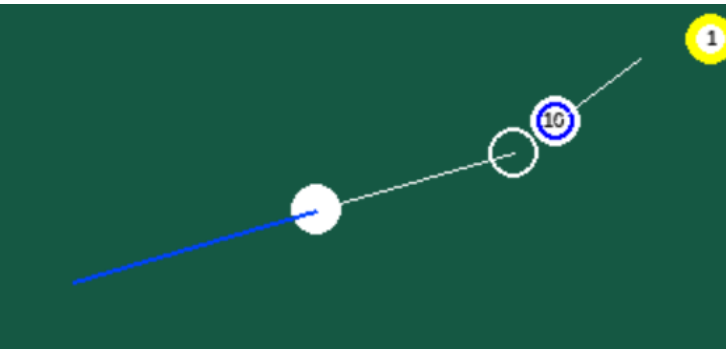


Therefore, this test is a success.

Test 25

25	Virtual ball projection on another ball	Ensures the user knows where the collision will project the ball to be hit.	Hover the cue over the ball	The projection should appear and be accurate	6
----	---	---	-----------------------------	--	---

As shown below, the virtual ball projection shows a rough direction for where the number 10 ball will go after the collision. This is accurate to the physics of real world and of the game.

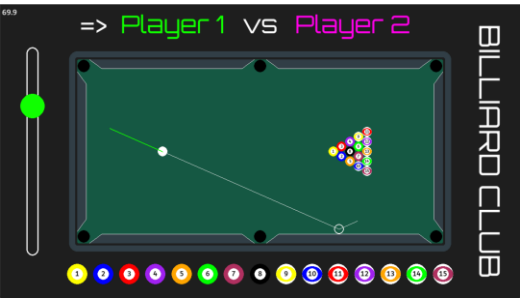


Therefore, this test is a success.

Test 26

26	The cue colour should match with the player's colour	This ensures each player knows whose turn it is	Observe the cue colour	The cue colour is the same as whoever's turn it is.	24
----	--	---	------------------------	---	----

Here I have changed the colour of player 1's cue, and it has updated to that colour:

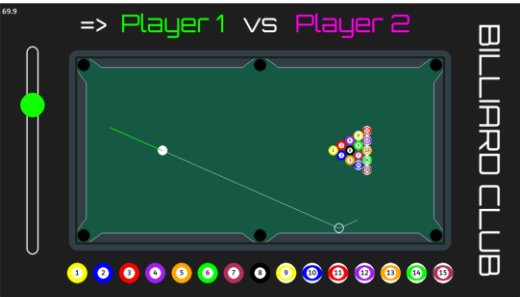


Therefore, this test is a success.

Test 27

27	The power slider's colour should match with the player's colour	This ensures each player knows whose turn it is	Observe the power slider's colour	The cue colour is the same as whoever's turn it is.	24
----	---	---	-----------------------------------	---	----

Here I have changed the colour of player 1's cue, and the slider has also updated to that colour:

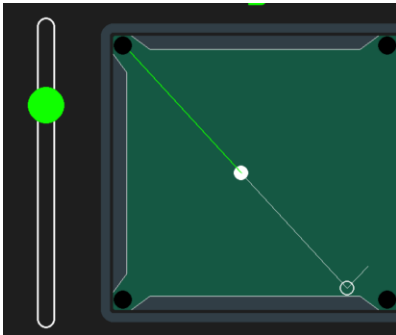


Therefore, this test is a success.

Test 28

28	Smooth power slider	Ensures the slider is easy and smooth to use	Use the power slider and decide if it is smooth	Slider should be responsive and smooth to use	24
----	---------------------	--	---	---	----

The slider is very easy to use and updates to the y coordinates of the mouse, therefore responsive, smooth, and easy to use.

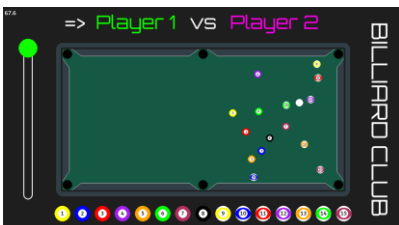


Therefore, this test is a success.

Test 29

29	Fluid ball motion	Ensures ball movement is smooth and accurate	Observe balls when moving	Balls should have accurate and fluid motion	24
----	-------------------	--	---------------------------	---	----

In my opinion, the high frame rate ensures the ball movement is smooth and ensures aesthetic gameplay.

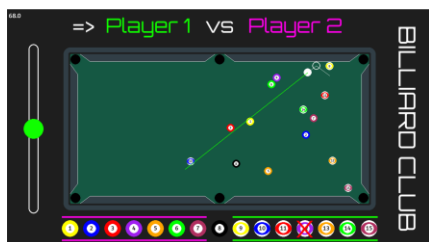


Therefore, this test is a success.

Test 30

30	All table elements should be drawn to the screen	Ensures all functionality and data is displayed to the user	Observe table	All elements/balls have been drawn	1
----	--	---	---------------	------------------------------------	---

As shown in the image below, all the relevant elements have been drawn to the screen: player indicator, player names, slider, logo, cue, virtual ball projection, all non-potted balls, Ball potted UI, UI for showing which balls have been potted and all of the table elements.



Therefore, this test is a success.

Test 31

31	Ball numbers should be rendered on each ball	Ensures that the user can tell which ball to aim for.	Observe the balls	All balls except for the white ball should have a number	1
----	--	---	-------------------	--	---

As shown in the image, each ball has a number rendered to it:



Therefore, this test is a success.

Test 32

32	Balls should be either striped or dotted	Ensures balls can be differentiated from each other for the user	Observe the balls	All balls should have a graphic which displays them as dotted or striped	1
----	--	--	-------------------	--	---

As you can see, striped balls are shown to have white top and bottoms, while dotted balls have a single dot on them. These are easily differentiable.



Therefore, this test is a success.

Test 33

33	Player turn indicator	Ensures the user knows which player's turn it is	Observe the indicator	The indicator should be clearly shown to be whoever's turn it is	1
----	-----------------------	--	-----------------------	--	---

The image below shows the indicator that shows it is player 1's turn. This swaps when needed.



Therefore, this test is a success.

Test 34

34	FPS is consistent and at least 60	Ensures the game is smooth to play and has good performance	Observe the FPS display	FPS should fluctuate above 60	23
----	-----------------------------------	---	-------------------------	-------------------------------	----

In the top left corner of the screen, there is an FPS display. This is consistently above 60 when playing all the games (as shown in previous tests).

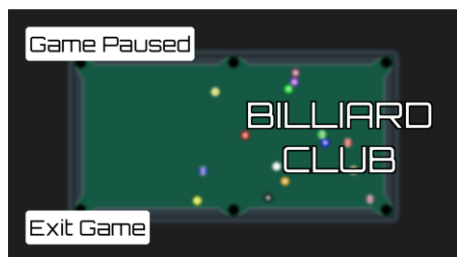


Therefore, this test is a success.

Test 35

35	Pause Screen Activation	Ensures the user can pause the game at any point in the game	Press the ESC key while playing a game	Should direct the user to a pause screen	3
----	-------------------------	--	--	--	---

When I press the ESC key, the game shows the user a pause screen.

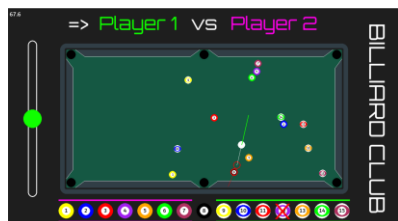


Therefore, this test is a success.

Test 36

36	Pause Screen Exit	Ensures the user can replay the game after pause	Press the ESC key while paused	Should return the user back to the game	3
----	-------------------	--	--------------------------------	---	---

When I press the ESC again, the game shows the user the game again.

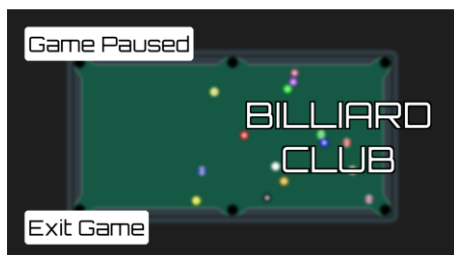


Therefore, this test is a success.

Test 37

37	No game interaction while paused	Ensures the user can interfere with the game while paused	Press keys and attempt to break the game while paused	The game will only return when the ESC key is pressed	3
----	----------------------------------	---	---	---	---

Here I press every key on the keyboard except the esc key. Nothing happens, until the esc key is pressed.

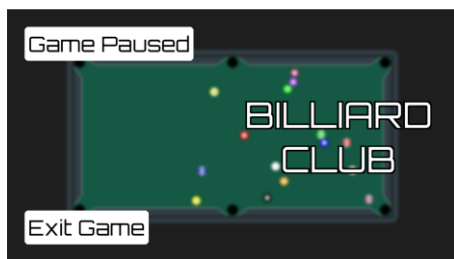


Therefore, this test is a success.

Test 38

38	The game is blurred in the background while paused	Ensures the pause screen is aesthetic.	Observe the blur in the pause screen	Game is blurred in background	24
----	--	--	--------------------------------------	-------------------------------	----

As shown in the image below, the background is blurred for added aesthetics.



Therefore, this test is a success.

Test 39

39	The game should be blurred in all menus	Ensures all menu screens are aesthetic.	Observe the blur in menus screen	Game is blurred in background	24
----	---	---	----------------------------------	-------------------------------	----

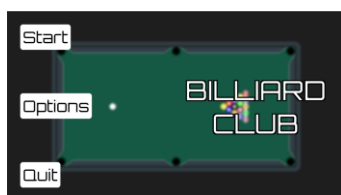


Therefore, this test is a success.

Test 40

40	Each button in the menu should be clickable	Ensures each button has functionality	Click various buttons in menus	They should all have functionality	4
----	---	---------------------------------------	--------------------------------	------------------------------------	---

Each button in the menu is indeed working and each have a separate functionality. I have tested this by clicking each button in the menu.

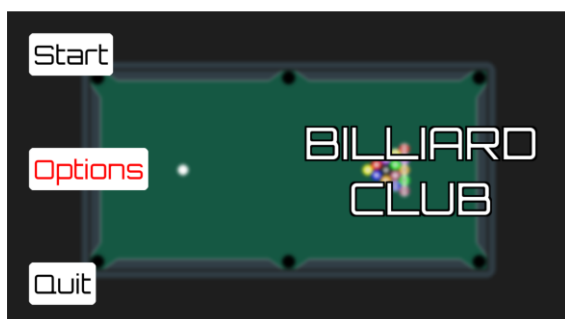


Therefore, this test is a success.

Test 41

41	Each button should change colour when hovering	Ensures user knows that box is a button	Hover over button	Should change colour	4
----	--	---	-------------------	----------------------	---

When I hover over each button in the menu, the text changes colour. Here I am hovering over the button called Options, and the text changes to red.

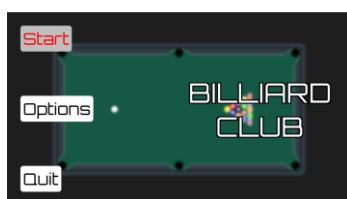


Therefore, this test is a success.

Test 42

42	Each button should change colour when clicked	Ensures user knows that button has been clicked	Click button	Should change colour	4
----	---	---	--------------	----------------------	---

Here I have clicked the start button, and the colour of the box has changed colour:



Therefore, this test is a success.

Test 43

43	Each textbox should change colour when clicked	Ensures the user knows when to start typing	Click a text box	Changes colour	4
----	--	---	------------------	----------------	---

For the change name screen, the text box has turned red when clicked, indicating the user can now enter text.



Therefore, this test is a success.

Test 44

44	Each textbox should allow the user to type	Ensures text can be input into a text box	Click and start typing in a text box	Characters pressed show up in textbox	4
----	--	---	--------------------------------------	---------------------------------------	---

Here I am able to use the keyboard to enter characters into the text box and the box updates to fit all the text in.



Therefore, this test is a success.

Test 45

45	When delete is pressed in a textbox, the letter is deleted	Ensures that characters can be removed in the text box if the user mistypes	Press delete in the text box	Last character is deleted	4
----	--	---	------------------------------	---------------------------	---

When I press backspace, the last character in the text box is removed. This allows the user to remove all text and completely change their name.



Therefore, this test is a success.

Test 46

46	Back button in menus	Ensures the user can go to the previous menu	Press back in menus	Game should revert to previous menu	4
----	----------------------	--	---------------------	-------------------------------------	---

When I press back on every menu, it redirects me back to the previous menu as intended, shown in these pictures:

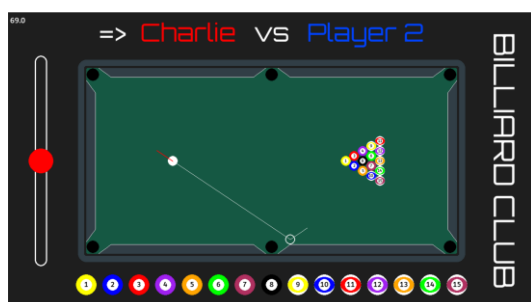


Therefore, this test is a success.

Test 47

47	When the user types a new name, it updates to the game	Ensures the user can change their name	Change name, observe it in a new game	New name is updated to game, observable in game	4
----	--	--	---------------------------------------	---	---

Before I had updated the name to "Charlie". Now when I press play "Player vs Player", "Charlie" is replaced with "Player 1".

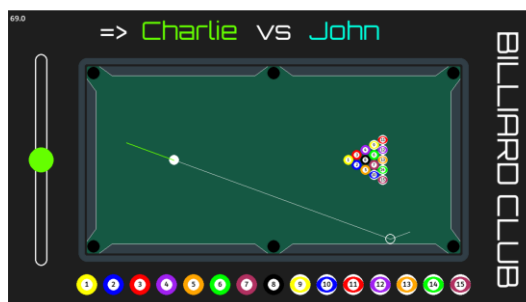


Therefore, this test is a success.

Test 48

48	Colour picker	Ensures the user can pick a colour for their indicator	Use the colour picker to pick any colour	That colour should be updated, is observable in a game	4
----	---------------	--	--	--	---

Here I have updated the colours for both players, and when I press play game, I am met with this screen, which confirms that the colours are changeable.



Therefore, this test is a success.

Test 49

49	Colour Slider is smooth and easy to user	Ensures the user can operate the slider well and is easy to use	Use the slider and determine if easy to use	Slider is easy to use	4
----	--	---	---	-----------------------	---

The slider is very easy to use, it matches the mouse's x coordinate when pressed and therefore guarantees a smooth and responsive feel.



Therefore, this test is a success.

Test 50

50	Master Volume Slider	Ensures user can change the volume for all elements in the game	Use slider, check before and after	Slider should correspond to how loud every sound is	4
----	----------------------	---	------------------------------------	---	---

When I adjust the slider, the volume for the whole game is changed depending on how far the slider is from the left most point on the slider, just as intended.



Therefore, this test is a success.

Test 51

51	Music Volume Slider	Ensures user can change the volume for only the music	Use slider, check before and after	Slider should correspond to how loud the background music is	4
----	---------------------	---	------------------------------------	--	---

When I adjust the slider, the volume for the just the background music is changed depending on how far the slider is from the left most point on the slider, just as intended.



Therefore, this test is a success.

Test 52

52	Ball sound Volume Slider	Ensures user can change the volume every ball sound effect	Use slider, check before and after	Slider should correspond to how loud ball collisions and potting is	4
----	--------------------------	--	------------------------------------	---	---

When I adjust the slider, the volume for just the ball sounds is changed depending on how far the slider is from the left most point on the slider, just as intended.

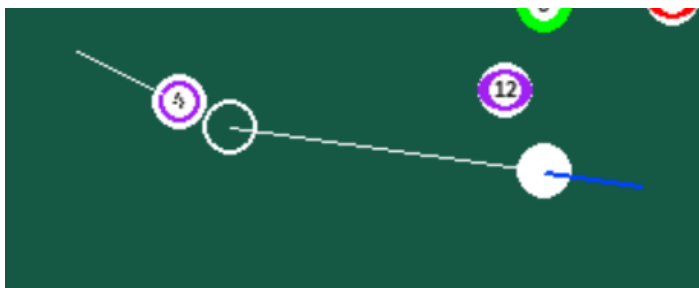


Therefore, this test is a success.

Test 53

53	Sound Played on cue hitting white ball	Ensures a response to the user after they have hit the ball	Hit the white ball with a cue	Sound Is heard	20
----	--	---	-------------------------------	----------------	----

When I executed the shot below, I heard a sound that resembled the cue hitting the white ball. This gives a clear indication of the white ball being hit.

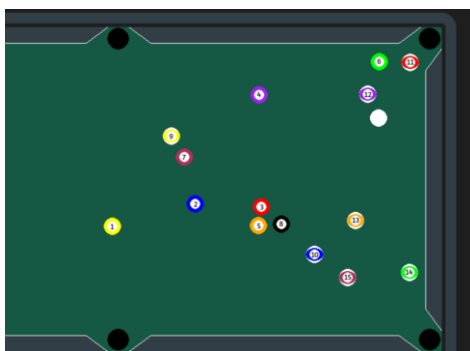


Therefore, this test is a success.

Test 54

54	Sound played on ball collisions	Ensures user knows when a ball collision occurs	Project one ball into another	Sound is heard	20
----	---------------------------------	---	-------------------------------	----------------	----

When two balls collide, a sound is played to notify the user of a collision.



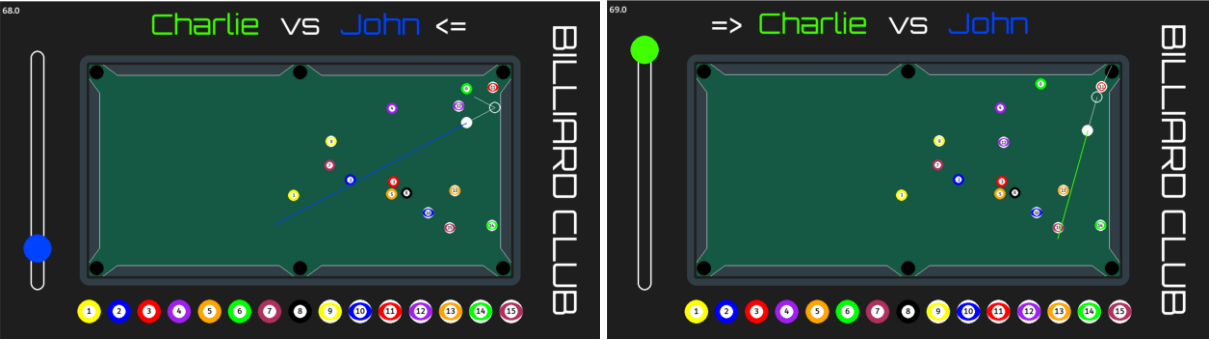
Therefore, this test is a success.

Test 55

55	Sound volume should depend on speed of collision	Ensures the user is met with a dynamic volume for more immersion	Project one ball into another at a slow speed, then at a	Sound at higher speed is louder than at a lower speed	21
----	--	--	--	---	----

			higher speed then compare		
--	--	--	------------------------------	--	--

Firstly, I completed a collision with low power as shown on the left. I found that the sound was very quiet. Next, I completed a second collision, on the right, and found the sound of balls colliding was overall much louder, definitely adding more immersion.

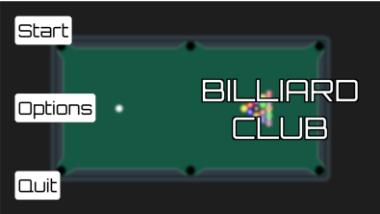


Therefore, this test is a success.

Test 56

56	Background music is heard everywhere on game start	Ensures music plays	Open game and check if music playing	Background music playing	18
----	--	---------------------	--------------------------------------	--------------------------	----

On game start, the music plays.

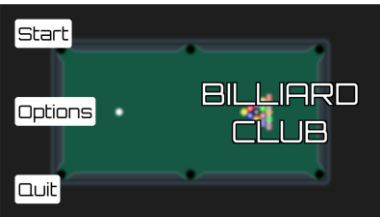


Therefore, this test is a success.

Test 57

57	Background music should replay after it ends.	Ensures the music plays indefinitely	Leave the game for 20 mins, and check if music still playing	Music continues to play	18
----	---	--------------------------------------	--	-------------------------	----

Here I have left the game on the menu screen for over 20 minutes and the music is still playing, confirming the music will play for a long time.

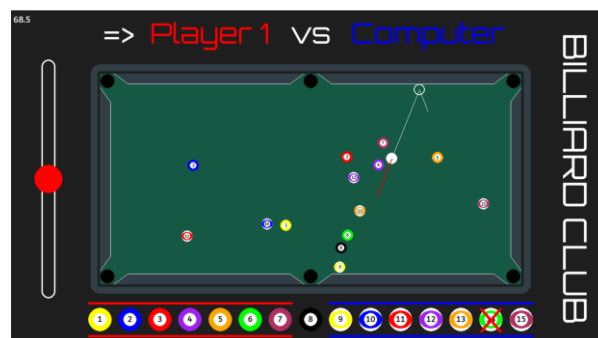
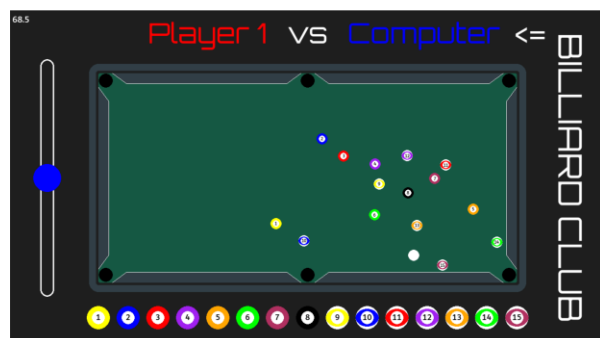


Therefore, this test is a success.

Test 58

58	Computer calculates next shot on turn	Ensures the user can play against a computer	Select Player vs AI and start playing	When it is the computer's turn, they will calculate a shot	25
----	---------------------------------------	--	---------------------------------------	--	----

Here I started a Player vs AI game and then failed to pot a ball, when my go was over, the AI immediately took over and performed a shot and potted the number 14 ball.

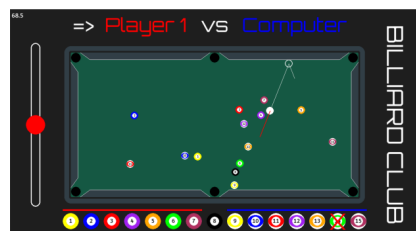


Therefore, this test is a success.

Test 59

59	The computer should calculate the shot instantly	Ensures the user is not waiting around for a long time	Give the computer the turn, time how long they take	Should be near instant	25
----	--	--	---	------------------------	----

As mentioned in the previous test, as soon as Player 1's turn is finished, the AI will take the shot to avoid unnecessary wait times.

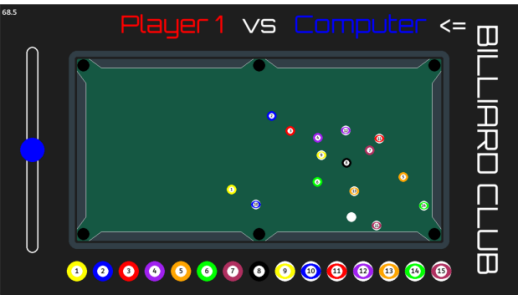


Therefore, this test is a success.

Test 60

60	The user's cue should not be drawn when the computer is playing	Ensures there is no redundant information in the UI when playing	Give the computer the turn	The cue should not be drawn here	1
----	---	--	----------------------------	----------------------------------	---

As shown in the image below, when the computer is playing, the cue is not drawn because it does not need to be. Therefore, it prevents redundant UI from displaying.

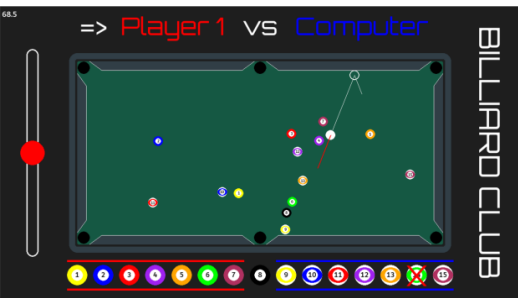


Therefore, this test is a success.

Test 61

61	When balls are potted, the UI updates to show the user which have been potted	Ensures the user knows which ball are potted and which need to be potted	Pot a ball	Ball is shown to be potted in UI	4
----	---	--	------------	----------------------------------	---

The computer has potted ball number 14. Here, the game was used appropriate UI to display that ball number 14 is not on the table and has been deleted, from the giant "X".

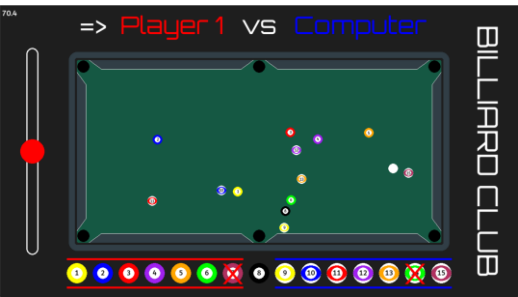


Therefore, this test is a success.

Test 62

62	A sound should be played when a ball is potted	Ensures the user knows when someone pots a ball	Pot a ball	Sound is heard	19
----	--	---	------------	----------------	----

After potting the number 7 ball, a sound was played that resembled a ball being potted into a hole.



Therefore, this test is a success.

Summary of Testing against the Test Plan

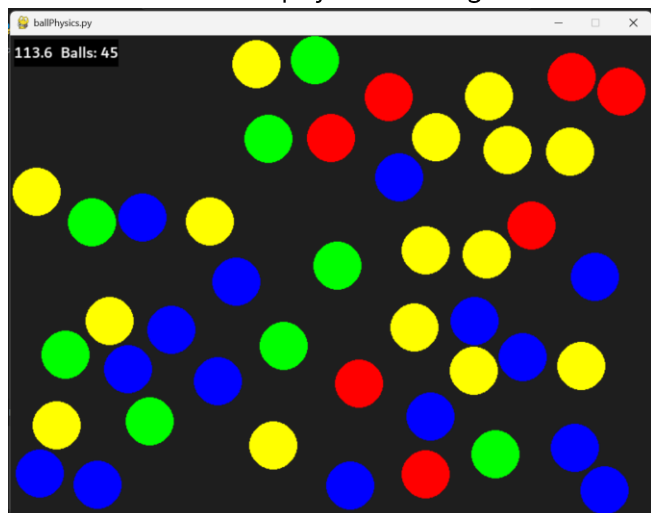
In conclusion, all the tests have been passed. This is no surprise, as the approach to implementation was to test all the functions while implementing them, ensuring they work before finishing each stage. As mentioned before, these modular stages were then combined together to form a complete and well coded solution, requiring little end maintenance, given all the tests have been passed. Therefore, no further actions will be taken on the program.

Stress Testing

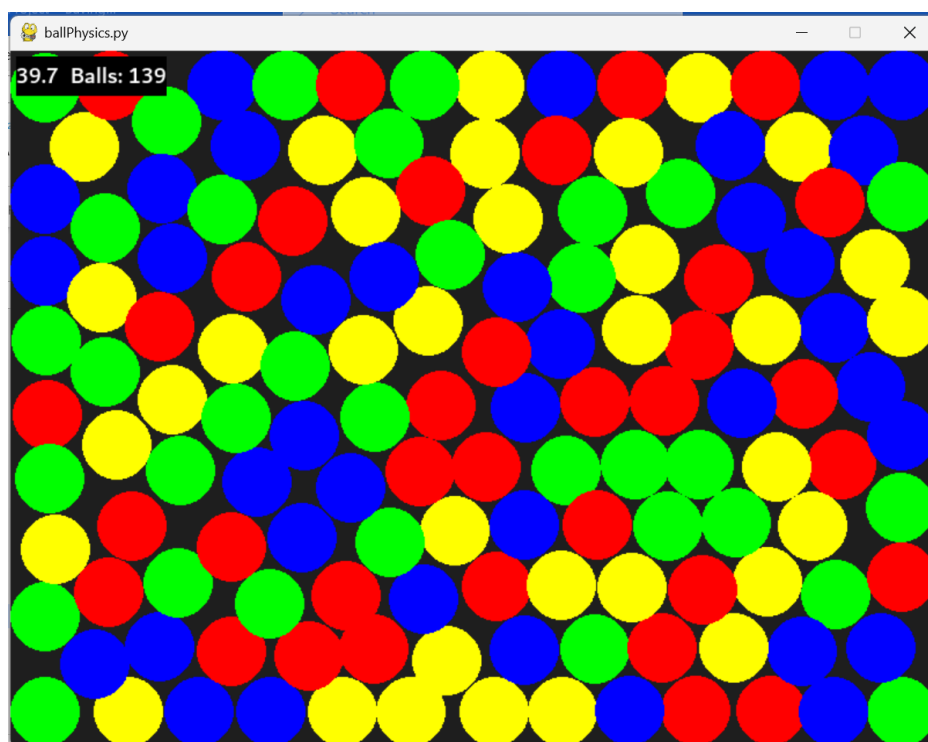
In this section, I am testing the solution so that all inputs possible erroneous data is dealt with appropriately so that the solution, holistically, performs well and is robust.

Performance Testing

For the physics of the game, I have performed performance modelling on the balls. In this scenario, I have created an environment that uses the same physics that are used to calculate the ball collisions in the game into its own environment where I can increase the number of balls in the game. This allows me to test the physics of the game to the extreme.



On my machine, the physics of the game are realistic, and the performance of the game is running well. As I add more balls to the game, the performance obviously degrades but degrades after about 130 balls are instantiated. As the game will have no more than 16 balls on the table at once, the solution is very well optimised. No further action needs to be taken.



Inputs Testing

In the game, you are required to input your controls and name. Here I will stress test these functions so that they work properly and are able to account for erroneous data.

Firstly, I will look at the text boxes:

Here I have mashed the keyboard and the result is shown.



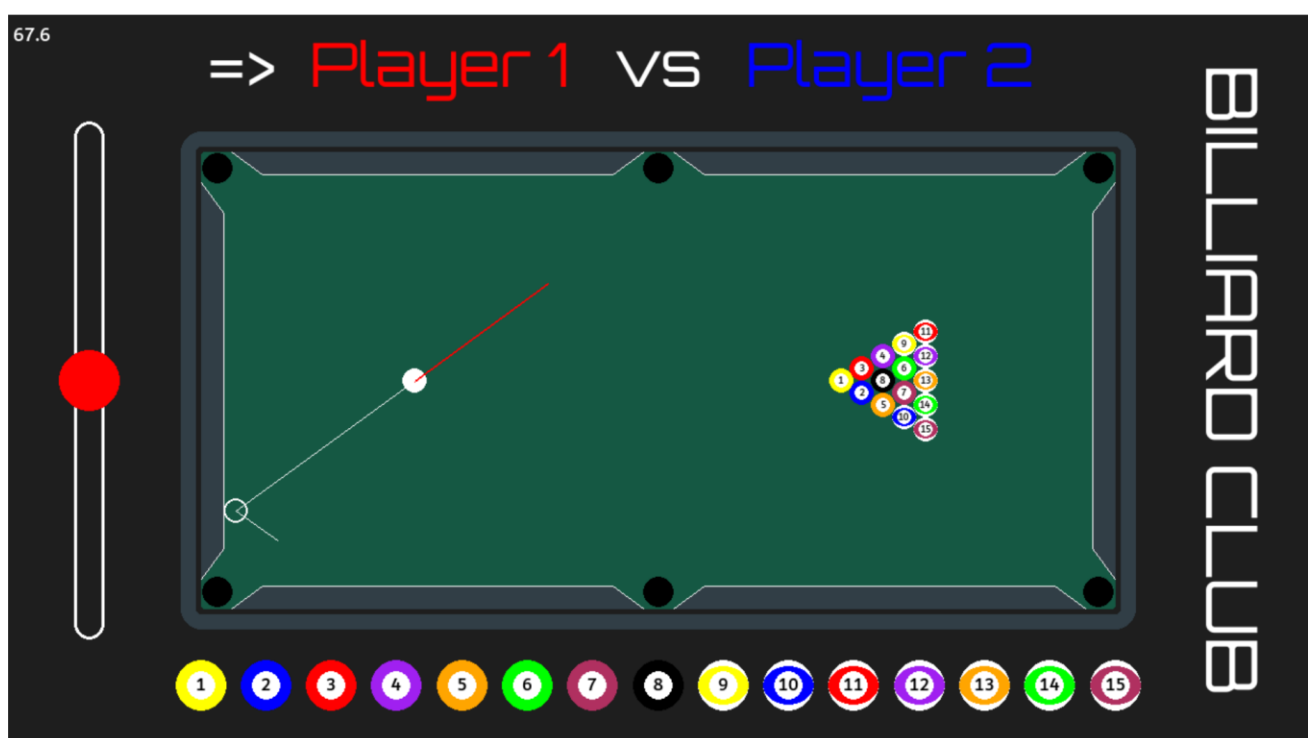
The game continued to run and produced this array of characters. There is one problem, when I pressed the tab key, the result draws a box, which is an error.

This is fixed here:

```
elif event.key == pygame.K_TAB: # Check if the tab key is pressed
    self.text += " " # Add a space
```

All is now working with textboxes.

Next, I check the game for erroneous inputs while playing the game.

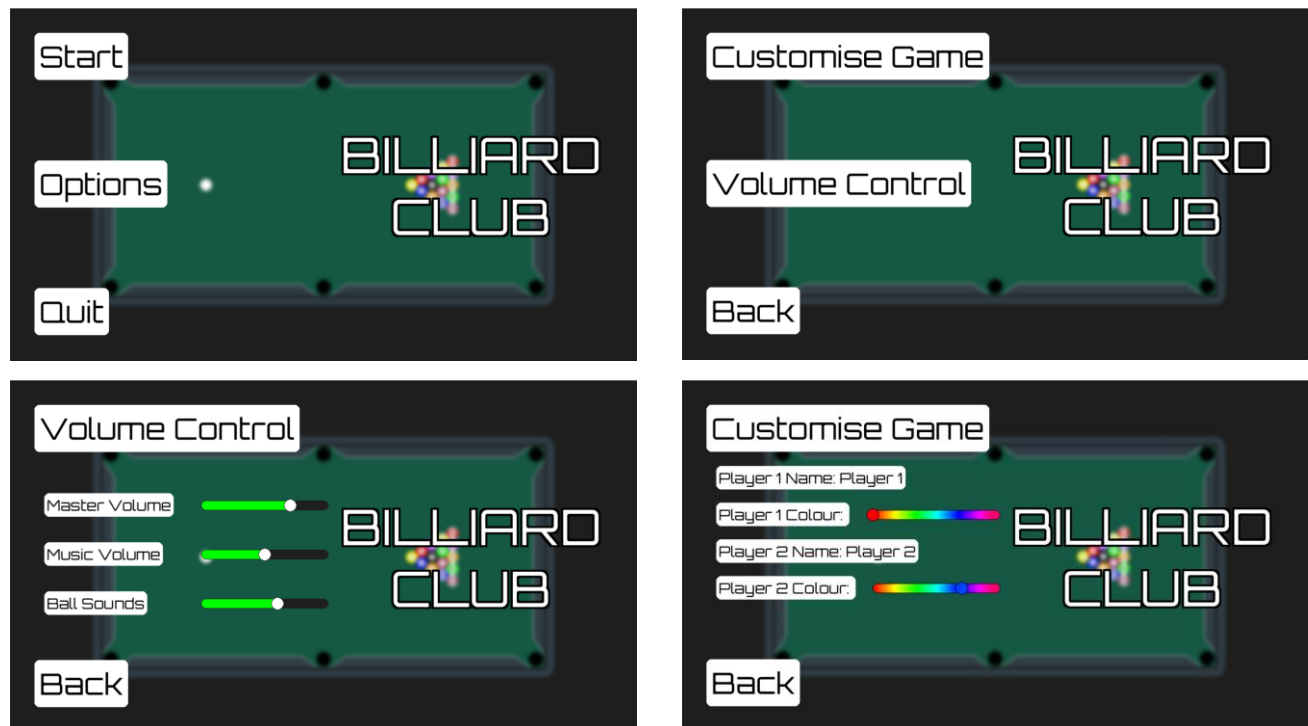


Here I mash the keyboard, hoping for an error, but no error occurs. I try to press every key and nothing happens, which is a great sign of robustness. The only input that works is the left mouse button as intended. Therefore, there is no further action required here.

Menu Testing

Here I specifically test the menu for erroneous outputs.

On each of the menu screens I mash the keyboard to check for robustness of the solution.



In addition, I have tried to break the sliders by moving the mouse extremely quickly back and forth but yielded no errors. Therefore, all the menu screens are robust and therefore no further action will be taken.

Multiple User Testing

I sent the solution to multiple users and asked for them to test if they could run and play the game properly. They received the files but had problems with the python library dependencies. To run the game, the user needs both the game files (e.g. the font, music, and sound files) and to have the python libraries: pygame, cv2, NumPy, and math. Although some come pre-installed with python, all the users required to install pygame and cv2 via PIP. Despite this, no users had any errors while playing the game. No further action will be taken.

Has the success Criteria been met?

In this section, I will iterate through each requirement in the success criteria and determine whether the success criteria has been met or not been met, providing evidence of the criterion being met or if it has not been met, an explanation to justify why it has not.

No.	Requirement	Justification
-----	-------------	---------------

As a reminder, the success criteria has been structured with column titles shown above. In the justifications, I refer to the test plan: that is to say if I refer to test 20 for example, I am referring row 20 in the test plan.

Criterion 1

1	All elements of the game Pool are drawn to the screen.	The game should display all the relevant elements of 8 ball pool.
---	--	---

As shown in test 30, all elements of the pool table have been drawn to the screen: player indicator, player names, slider, logo, cue, virtual ball projection, all non-potted balls, Ball potted UI, UI for showing which balls have been potted and all of the table elements. This means that this criterion has been met.

Criterion 2

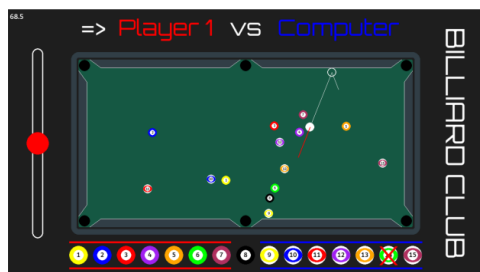
2	The player should be able to hit the balls with a cue.	This is needed to pot balls.
---	--	------------------------------

This criterion has been met given that in test 3 and 22, the balls successfully can be given a velocity, and the Cue UI is drawn to the screen.

Criterion 3

3	The game should be played from a singular camera angle from above.	This is the best camera for playing the game.
---	--	---

While playing the game, one can clearly see this criterion has been met as the game is indeed being played from a singular camera angle (top-down view). This criterion has been met.



Criterion 4

4	The game should have an intuitive user interface, where the user can navigate to all menus.	The user should be able to seamlessly navigate to play/settings/exit game.
---	---	--

As shown in tests 18-21 and 40-46, the game uses an intuitive UI which is both easy to use and effective at transporting the user to different sections of the game. Therefore, this criterion has been met.

Criterion 5

5	The balls should be deleted after they are potted.	When the balls are potted, there is no further need to simulate them.
---	--	---

As shown in tests 9, 10 and 12; when a ball is potted, it is deleted. With the only exception of the white ball as this must not be deleted. Therefore, this criterion has been successfully met.

Criterion 6

6	The system will implement visual aim assistance for the cue.	This helps users to aim the ball towards other balls.
---	--	---

As shown in tests 23 and 25, the game simulates an accurate virtual ball UI that provides a helpful and interactive visual aim assistance for the user. This therefore helps the user to aim towards the other balls and therefore I would class this criterion as successfully achieved.

Criterion 7

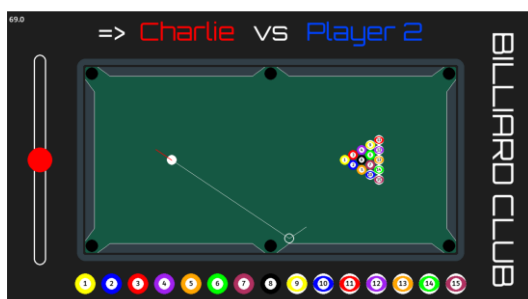
7	The visual aim assistance will turn red/white depending on the prediction of what ball it will hit.	If the system thinks that the user is going to hit a ball which is not of their type, the system will tell the user that they are hitting the wrong ball.
---	---	---

As shown in test 24, the virtual ball UI will change colour depending on the ball type of the player. This therefore shows that this criterion is met.

Criterion 8

8	When playing 8-ball, there should be 15 balls, each with different colours and markings. The game should follow the rules of 8 ball pool	The game should ensure each ball is unique and easy to recognise. Also follow the rules of the game
---	--	---

When a user starts a new game, this screen will be shown:

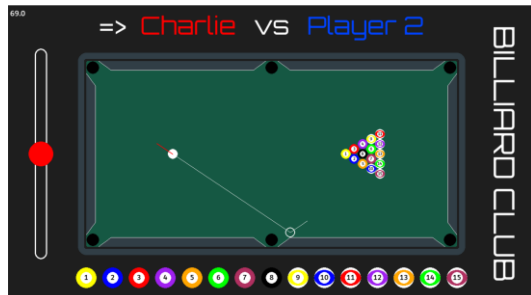


This provides the relevant evidence to show that all 15 balls are drawn to the screen. The game follows the rules of 8 ball pool shown by test 13, which tested foul detection. It passed this test. Therefore, I believe it has achieved both of these tasks and therefore meets the specification point.

Criterion 9

9	The player should be able to play the game against another player.	Two players should be able to play against each other.
---	--	--

As shown in the previous criterion check, the player is able to play against another player.



It also follows the correct rules of 8 ball pool and therefore achieves this specification point.

Criterion 10

10	After every game, the ball positions should be reset so a new game can be played after another.	There should be a starting function that starts a new game, and places all the balls in the correct positions.
----	---	--

As outlined in test 17, after a game is completed, the user has the option to start a new game and in this new game, all the ball locations are reset because they are instantiated. This therefore meets this criterion.

Criterion 11

11	When balls collide, the system should determine what direction and speed the balls have.	Appropriate calculations are required to determine the velocities of the balls after the collisions.
----	--	--

As outlined in tests 5 and 8, the ball-to-ball collisions are highly accurate, giving accurate results for both singular collisions and multiple collisions in test 8. Therefore, I believe this criterion has been met.

Criterion 12

12	When balls collide with the walls, they are reflected off accurately.	When colliding with walls, the balls' velocities should be calculated accurately.
----	---	---

This criterion has been met with the evidence of meeting test 6 in the test plan. The ball successfully reflects off walls and rails of the pool table.

Criterion 13

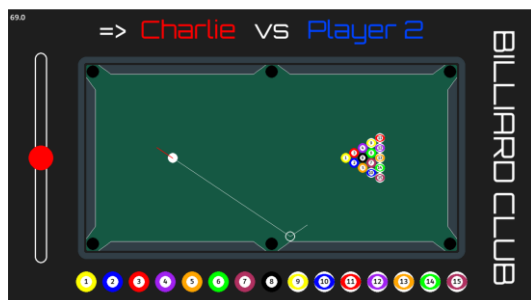
13	The game should display the winner after every game.	The game would follow the standard rules of the game and repetitively check if a user meets the win criteria.
----	--	---

As shown in test 16, the winner is displayed when a winner is determined. This means that this criterion has been met.

Criterion 14

14	The game should have a power meter to show how powerful each hit with the cue is.	This ensures that the player knows how powerful a shot is and can adjust the power to get the right type of shot.
----	---	---

As outlined in the game screen, there is an easy-to-use power slider in the game as shown in the image below on the left:



I believe therefore that this solution meets the criterion.

Criterion 15

15	The game should display which player is playing and who has the next turn.	There should be an indicator to show whose go it is.
----	--	--

As outlined in test 33, there is an arrow in the top section of the screen which points to the player who currently has their turn. Therefore, this criterion has been met.

Criterion 16

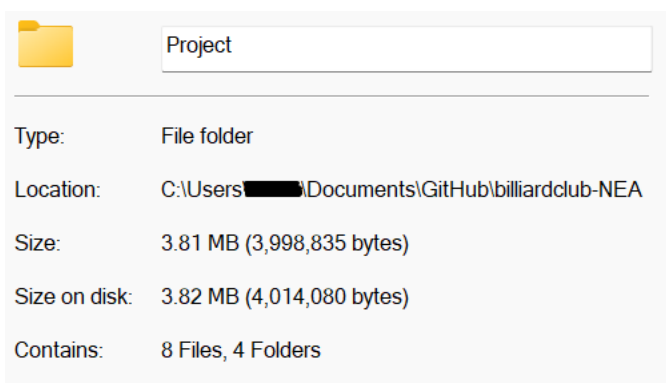
16	The game should display which balls should be potted by each player.	The game should have a graphic that shows all the available balls to pot for each player.
----	--	---

As outlined in test 11, when the first ball is potted, the game successfully displays the correct ball types of each player. Therefore, this criterion has been met.

Criterion 17

17	The size of the game should be no more than 200MB	The file size should be small in general.
----	---	---

In the image below, is the final size of the project folder (all elements of the game are contained within this folder).



As you can see, the full size of the project is well below the maximum size. Therefore, the solution meets this criterion.

Criterion 18

18	The game will have a music track to be played in the background.	Music helps players to enjoy the game, and a calming one would also help the players focus.
----	--	---

As outlined in tests 56 and 57, there is presence of a background music in the game, which successfully meets the criteria.

Criterion 19

19	A relevant sound will be played if the player pots a ball	Repetitive sounds for certain game events are helpful to tell player if they committed a foul or won the game.
----	---	--

As outlined in test 62, a sound is successfully played when a ball is potted, which means the solution successfully meets the criterion.

Criterion 20

20	A relevant sound will be played when balls collide.	A ball-to-ball collision sound will be played when a collision is calculated. Enhances user experience.
----	---	---

This criterion has evidence in tests 53 and 54. In these tests, there is clear evidence of a sound being heard when both the cue collides with the ball and two balls colliding. Therefore, meeting the criterion.

Criterion 21

21	When balls collide, the system should play a sound with a suitable volume, depending on the speed of collision.	If a ball collides with another ball at a high speed, then a loud sound should be played.
----	---	---

As shown in test 55, the speed of collision, affects the magnitude of the volume of the ball collide sound effect. This therefore meets the criterion.

Criterion 22

22	The game should play a click sound every time a menu is clicked.	This ensures user engagement and understanding of where the user is navigating to.
----	--	--

When the user clicks a button on any menu, a click sound is heard. This sound can also be adjusted in the volume section of the setting menu. Therefore, this criterion has been met.

Criterion 23

23	The game should run at 60fps or more for good performance.	The game should be able to run well when playing the game.
----	--	--

This criterion has been met due to the clear evidence in test 34. In this test, I have shown that the game runs smoothly and is well above 60fps while playing the game.

Criterion 24

24	The game is visually aesthetic/appealing.	The game should be enjoyable to play, and the elements are well designed.
----	---	---

Although this is subjective, I believe the game as a whole is very easy to play, all game elements are well designed, there is lots of colour, colours are able to be changed for the cue and the slider, ball collisions act fluidly. Overall, in my opinion the game meets this success criteria.

Criterion 25

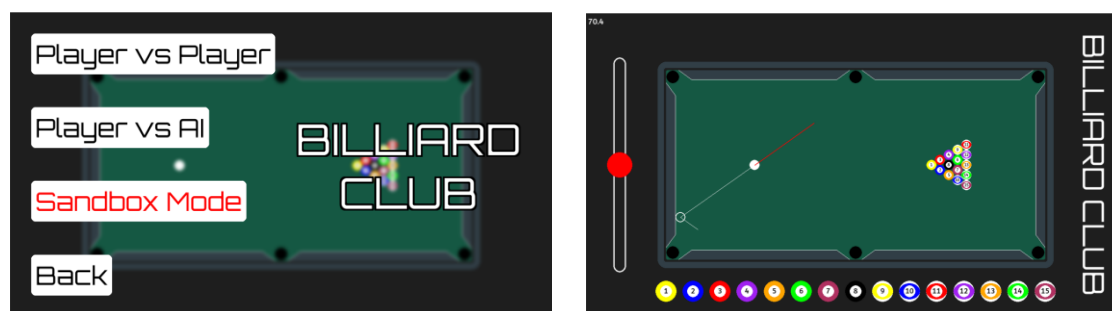
25	The player should be able to play against an AI.	As well as another real player, the game should have a suitable algorithm that finds the best shot for a given state.
----	--	---

As shown in tests 58 and 59, the computer is able to calculate a shot very quickly. This shot more often than not will result in a ball being pot. Therefore, this criterion has been met.

Criterion 26

26	The game should include a practice mode	A practice mode would be helpful for players to increase their skills and attempt challenging shots.
----	---	--

In the images below: you can see that there is the option for a practice mode:



In this mode, users can practice shots in a sandbox environment with no rules. This works well and therefore this criterion has been met.

Criterion 27

27	The player should be able to choose between different colours and patterns of cues.	The player should be able to customise their experience and decide what colours and patterns to use.
----	---	--

This criterion has been partially met with the ability to change the colour of the cue. You are not able to change the patterns of the cue. There is evidence of being able to change colour of the cue in tests 26 and 27.

Criterion 28

28	The player should be able to choose between different colours and patterns of pool tables.	The player should be able to customise their experience and decide what colours and patterns to use.
----	--	--

This criterion has not been met. You are not able to change the colour or patterns of the table. As with all the unmet criteria, time constraints are the main cause for this criterion not being met.

Criteria 29-37

29	The game could be able to simulate 9-ball pool	The game could be able to incorporate another game mode.
30	The balls could rotate when moving (i.e. not static) as if in 3D.	When a ball is moving in real life, you see the ball rotating as it moves, this 3D effect should be added to the 2D game.
31	The game could be able to handle spin shots and simulate spinning balls.	When the cue hits the ball on the edge, it induces a spin on the ball, which should be simulated in the game.
32	The solution could have an adjustable font size.	This allows people who cannot see very well to understand and play the game.
33	The game could have a customisable UI (resizing and colouring)	This adds a more individualised touch to the game, as you determine where extra details and statistics are placed on the screen.
34	The game could have a leaderboard or tracker that tracks the player's win rates etc.	When a user plays, they either win or lose, this can be turned into a percentage and the best ones will be displayed on a leaderboard.
35	The game could include a replay feature.	If a user does a particularly challenging shot, they should be able to record it will in game tools or able to replay the shot.

36	The solution could simulate an animation for chalking the cue.	When playing pool in real life, you often chalk the cue. This feature would add immersion to the game.
37	The game could have player profiles, which save progress and scores.	This feature could allow players to stop playing and exit the game, without the fear of losing their stage in the game.

With this set of criteria, these were classed as “could have” requirements. The reason for this classification was due to the inevitable time constraints of the development of the solution. Therefore, all of these criteria have not been met due to time constraints or either not required for the game to function. Given more time, these requirements were likely to be met.

Criteria 38-43

38	The game will not allow you to play other players over the internet.	This feature will not be implemented as it would require lots of work developing servers for the game.
39	The game will not have modification support	Given this feature would require a large community and constant development, this feature is not feasible.
40	The game will not have a feedback system where the players would report bugs.	There are likely to be few bugs and also development of this system would not be essential for the game to function.
41	The game will not have a chat feature that lets you chat with players in game.	Given there are no online players, there is no use for this feature
42	The game will not have a spectator feature.	Given there are no online players, there is no use for this feature
43	The game will not function with other input controls	There will be no other user inputs for example a user will not be able to play the game with a gamepad.

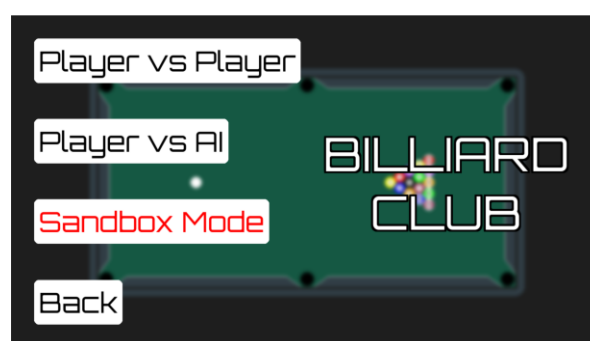
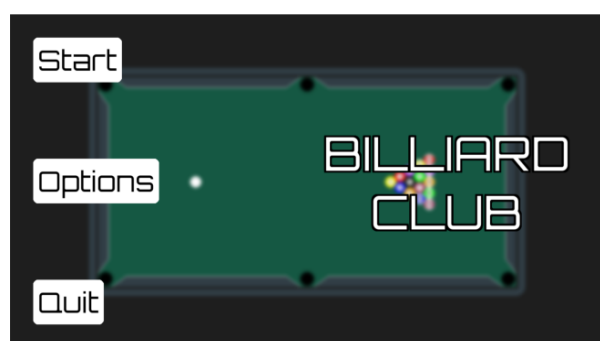
With this set of criteria, these were classed as “won’t have” requirements. This means that I was never going to intend to complete these requirements. None of the features mentioned in these requirements are in the final game solution, therefore they can be classed as all meeting the success criteria.

Evidence of implementation of Usability Features

In the design stage of the project, I outlined some usability features that will would have helped with overall user experience and those with accessibility issues. In this stage, I will assess whether each of the usability features have been completed.

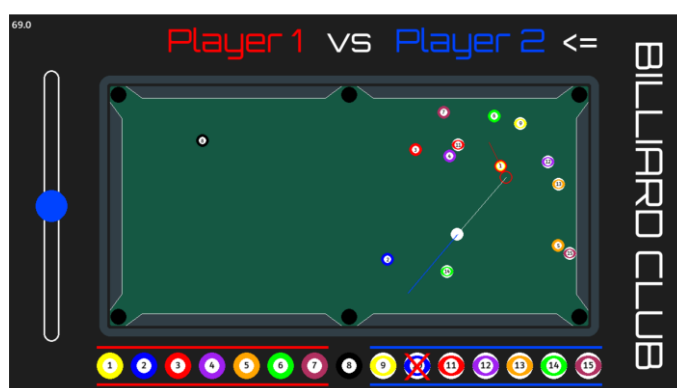
What usability features have been implemented?

In the potential usability features section of the design stage I mentioned how the menu system should be easy to navigate and use. This helps the user experience tenfold. I believe that this has been met through meeting tests 18-21 and 40-46, and therefore criterion 4 of the project has been met. The menu is shown to be easily accessible in the images below.

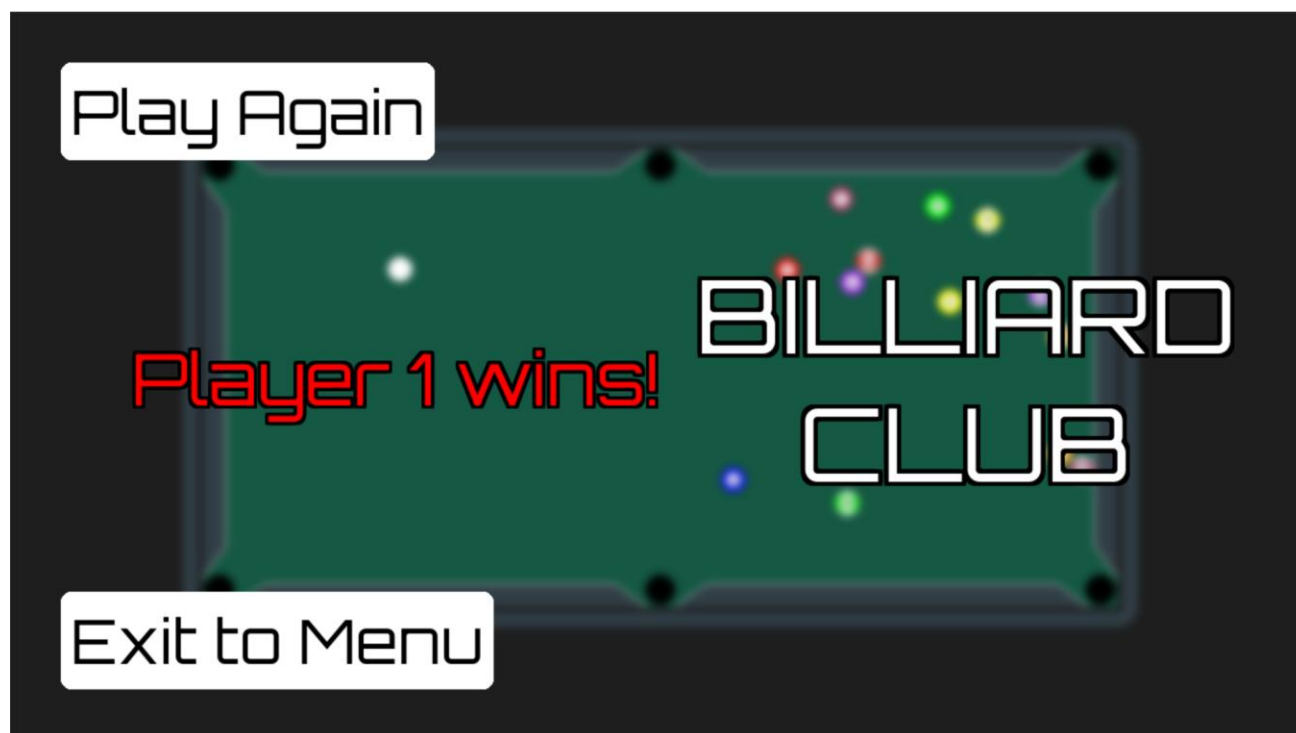


The menu system also employs both visual and audio feedback to the user when interacting with the menus. As you can see on the right, when the mouse hovers over the sandbox mode button, the text turns red. This acts as a great visual indicator that the menu is interactable. Not only this, but on click of a menu option, the solution gives an audio response when clicked. The program plays a click sound, which notifies the user that the menu has navigated successfully. This could aid those with visual impairments to faster know that their input of a click has been received.

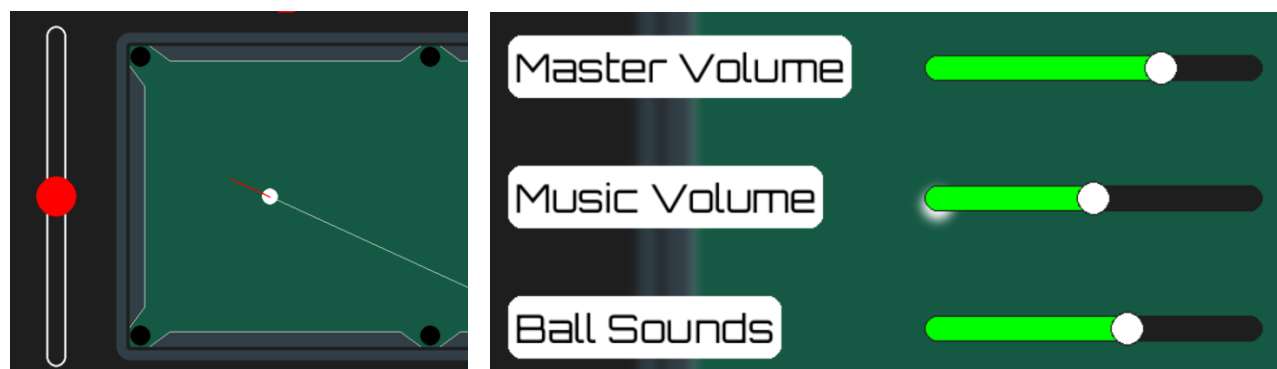
Throughout the game, a striking font with lots of colour has been utilised, as shown in the images below. This obviously helps the user use the game effectively and improves the aesthetics of the game.



The win screen also uses a striking bold font that has a black outline. This helps really emphasise the text, ensuring clarity in displaying the winner of the game. This same black outline on the font is also used for the logo of the game, which is displayed in all menus as shown in the images. The buttons also display a white box with black font which helps visuals.

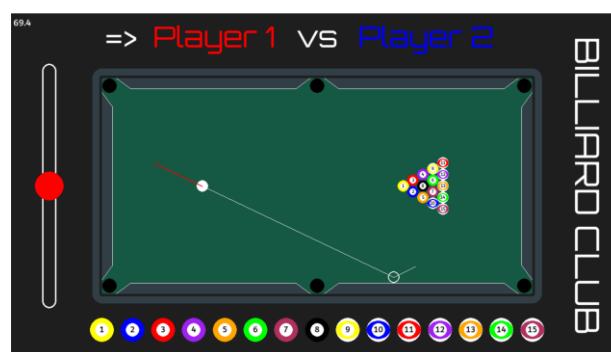


The program also uses lots of sliders. These sliders have proved to be very responsive from tests 28 and 49. This feature is therefore helpful for those who are not as apt when using computer systems. Having an easy-to-use slider is an essential usability feature for this program. The sliders are shown in the images below and the colour sliders are shown in the images above. Given the ease of use with sliders, the sliders have been added in many places to help with usability of the solution.



A sound is also played when balls collide, which helps immerse the user into the game, and aids with understanding when ball collisions occur. A sound is also played when the white ball is hit with the cue. This again helps the user understand when they give an input to the system and the system therefore provides a visual output (balls moving) and an audio response (cue sound).

What usability features have not been implemented?



The only usability features that have not be included that were previously mentioned were the ability to resize the font. This has not been included due to time constraints on the development of the solution. The idea for resizable UI is very self-explanatory, but it was hoped that the user could use a slider to change the size of all text in the game. This would allow people with visual difficulties or visual impairments to better view the game.

The main idea for the scalable font size was to account for people who may have difficulty seeing smaller objects on the screen, therefore this was designed to be scalable to cater for those who need larger text on the screen and those who wanted a variable font size.

However, on further thought and given the current state of the project, I believe that the font size being quite high anyway improves the aesthetics of the game and also accounts for those who find it more difficult to read smaller text.

I consulted George, the eldest of the three stakeholders for the project and in summary, he explained that the text was already very readable, despite himself previously saying he found it difficult to read smaller text and objects.

Despite this usability feature not being implemented, the potential problem of text being too small has been resolved with relatively large text in the across the solution.

Stakeholder Feedback

In this, section I have sent the final game to each of my stakeholders. Sam, Julia, and George. I have prepared a set of questions to ask them once they have played the game fully.

Questions Asked

As a reminder, here is a brief summary of the stakeholders:

- Sam is 17 years old and enjoys skateboarding, cricket, and casual gaming. Sam represents someone who enjoys playing pool but has not played very often due to lack of available pool tables.
- Julia, who is 31, enjoys baking, music, and is a huge fan of story-telling games. Julia represents those who enjoy gaming but has little experience with pool or pub games in general.
- George is 71 years old and loves to play pool regularly in the local pub. George is very experienced with pool yet has no experience with computer games.

Now I have prepared a set of questions to ask the stakeholders:

1. Were you able to play the game, is it easy to use?
2. Did you enjoy the game, and what do you really like about it?
3. What do you not like about the game, and what could be improved?
4. Are there any features that you think are missing that should be included?

These questions will allow me to assess the success of the solution and decide whether the solution is sufficient to the stakeholders and assess if the solution is a success.

Stakeholder Feedback 1: Sam

1. Were you able to play the game, is it easy to use?

“Yeah, I was able to play the game, and it was pretty easy to pick up. The controls felt smooth, intuitive, and aiming was very straightforward. Took me a couple of games to get the hang of adjusting shot power, but overall, it wasn’t too hard to figure out.”

2. Did you enjoy the game, and what do you really like about it?

“Yeah, I actually really liked it. I liked how realistic the ball movement felt. Each collision was easily predictable and accurate. Also, the dynamic sound of the balls colliding together was really satisfying. It kind of made me feel like I was playing on a real table, which is cool since I don’t play often in real life.”

3. What do you not like about the game, and what could be improved?

“When I played against the AI, at first the AI was really good at potting balls, but as the game went on, the AI became really bad at potting. I’m not really sure if this is intended, but because of this I won almost all the time.”

4. Are there any features that you think are missing that should be included?

“A trick shot mode would be awesome! Like, challenges where you have to hit crazy shots or make combo shots. Also, an online multiplayer mode would be cool so I could play against friends instead of just AI. Maybe some table and cue customisations too, just to add a bit of personality. Also adding spins shots too would be amazing for the trick shot mode.”

Stakeholder Feedback 2: Julia

1. Were you able to play the game, is it easy to use?

“Yes, I was able to play, though it took me a little while to get comfortable with the controls. The basic mechanics were easy to understand and aiming was really easy because of the collision projection. Once I got the hang of it, it became really enjoyable.”

2. Did you enjoy the game, and what do you really like about it?

“Surprisingly, yes! Even though I don’t usually play games like this, I liked the sense of progress as I potted each ball. The physics felt really satisfying watching the balls roll and collide made it feel very realistic, especially the with the addition of sounds. I also liked the relaxing pace of the background music; it was nice to play without feeling rushed.”

3. What do you not like about the game, and what could be improved?

“Maybe a more detailed tutorial or a practice mode that teaches techniques would help. Also, the interface felt a little technical – something more visually guided would make it easier for beginners.”

4. Are there any features that you think are missing that should be included?

“I’d love a campaign mode or some kind of larger progression system, like unlocking different locations or characters to play against. It would make the game more engaging for someone like me who enjoys a narrative! Also, a casual mode with extra aiming help would be great for those who just want to have fun without worrying too much about precision.”

Stakeholder Feedback 3: George

1. Were you able to play the game, is it easy to use?

“At first, not really. Took me a while to figure out how to control the cue properly, and using the mouse to aim is nothing like holding a real cue. But once I got used to it, I could play a decent game. It’s still much different from playing on a real table, though.”

2. Did you enjoy the game, and what do you really like about it?

“I did enjoy it once I got going. The physics are quite good, and the balls react the way they should. I liked that it felt close to real pool in terms of strategy, even if playing on a screen isn’t the same. It’s nice to be able to play without needing a table.”

3. What do you not like about the game, and what could be improved?

“As an experienced player, I enjoy performing shots that incorporate spin. However, this game does not include any spin mechanics. The balls also appear static and could be improved by adding some spin visuals to each ball.”

4. Are there any features that you think are missing that should be included?

“Other than the spin mechanic, I really liked the game and think it’s great. Maybe some more game modes like 9-ball pool could be added for a greater depth of play.”

Summary of stakeholder feedback

In summary, all of the stakeholders enjoyed playing the game. They all really liked the physics and gameplay. However, there were a number of improvements that could be made to the solution. They mentioned ideas like adding functionality for spin shots, trick shot modes, tutorial for new players, improvements to the AI and more table and cue customisation. These will be mentioned in the further development stage, where I will address how these features could be added in a later version.

Limitations of the Solution

What are the limitations of the solution?

The main limitation of the solution is the lack of features that are inside the game. The game only supports 8-ball pool, and a practice mode with the balls. This is likely due to the time constraints I was met with for the implementation stage of the solution.

For this solution, spin physics have not been implemented into the final product. This acts as a limitation as the balls are therefore not able to simulate all the types of shots in the game. For experienced players like one of my stakeholders, George, this feature is sorely missed. George enjoys more challenging types of shots when he plays 8-ball pool in person. The main reason for this not being implemented was the difficulty in coding a simulation for this as well as a tight timeframe.

Linked to spin physics is the appearance of static balls as they move across the surface. This can be regarded as a limitation of the solution as the ball movement is not accurate. Of course, in a proper game, balls roll on the surface. This is not simulated here and instead are static as mentioned before. With a greater timeframe for development and implementation, I do believe this feature could have been implemented to provide better immersion into the game.

Currently, the program relies on the user completely knowing the rules of the game. This could have been fixed with the addition of a tutorial into a user's first game. For new players, who have never played a game of 8 ball pool before, this can be problematic as they need a proper explanation. Therefore, the system would be better if it provided a one-off tutorial that teaches the player how to win. This is therefore a limitation of the program.

Another limitation of the solution is that you are only able to play against another player if they are using the same machine as you. That is to say, there is no multiplayer support. I took the decision to not develop a multiplayer environment due to the strict time constraints and the intense amount of and calibration required for multiplayer to function. In addition, I would need additional funding for hosting servers which is not possible.

How could these limitation be addressed?

For most of the points discussed above, time and funding constraints restricted the solution to have less functionality than hoped. However, the core game mechanics and functionality are all present in the solution and therefore the limitations described do not affect the core mechanics of game. However, with more time and possibly funding, these limitations could be addressed.

Maintenance of the solution

In this section, I outline the future maintenance of the solution with what has been done well to aid maintenance and what could be done to improve the solution in the future to address the limitations outlined in the previous section.

What has been done well?

Throughout development, I have used appropriate and consistent naming conventions with clear variable names. Each variable uses Camel Case, to quickly distinguish between each variable and operations and classes. This aids readability and will help with future maintaining and debugging code.

A lot of the functionality of the code has been separated into their own specific and unique functions that are called when they are needed. For instance, the functions `drawTable`, `drawBalls` and `drawCue` have been separated but stored successively in the game code improve modularity, which can greatly improve debugging as each function is independent of every other function. Therefore, this separation makes it both easier to read and will improve the future maintenance of the solution.

I have also implemented classes for key components in the game. Elements such as `Ball`, `Button`, `Menu`, `VerticalSlider` and `HorizontalSlider` encapsulate all the functionality into one section of code (in the class). This object-oriented structure helps isolate behaviour and state, making future changes and enhancements easier to debug and maintain.

What could be done in the future?

Although the program has good maintenance, that is to say the program is relatively easy to maintain, there are a few things that could be done in the future to both facilitate future development and improve the readability and modularity of the code.

Currently, the all the code is stored in one giant file. This combines the game's core logic, UI, and physics in one file. Therefore, for better readability and maintenance, a more modular design with separate files for game physics, UI, logic, sounds, would help isolating bugs and improving the testability and scalability of the solution.

In the code, there is limited error handling for exceptions. For example, the code does not include any exceptions if the sounds or music cannot load. This means that there could be potential unfixed errors in the program which could have easily been solved with adding a few edge cases. This could be done to improve the maintainability of the code in the future.

The game currently comprises many variables, many of which are declared as global. In almost all cases, global variables are not required and in almost all cases are reasons for error. Therefore, if given more time, to assist the future maintenance of the system, I would remove all instances of global variables, ensuring all variables keep to a minimal scope, reducing the chance of mistakes when performing calculations on these variables. This will facilitate maintenance and debugging massively.

Further Development

Considering all feedback and evaluation, for this stage I will explain a few features that could be added to a future version of the game, if given the time and resources to implement them.

Spin shots

This highly requested feature from the stakeholders could be implemented to the solution with surprising little difficulty. To do this, I would add spin attributes to the ball class, for instance, add an attribute of angular velocity for two axes of rotation. Then in the ball update method, I would update the ball to show the effects of the ball spinning. Then with slightly more thought and research, I the spin would affect the trajectory of the ball via an approximation of the magnus effect.

Here is a simplified example of what would be done:

```
1. ball.pos = ball.pos + ball.velocity
2. ball.velocity = ball.velocity * friction
3. ball.spin = ball.spin * spinDecayFactor
4. // Apply a simple Magnus effect using the spin:
5. perpSpin = (-ball.spin.y, ball.spin.x)    // Get a vector perpendicular to ball.spin
6. magnusEffect = perpSpin * magnitude(ball.spin) * magnusCoefficient
7. ball.velocity = ball.velocity + magnusEffect
```

Tutorial

This, too, was highly requested among the stakeholders, especially Julia. This would not be difficult, to program as there is no complex calculation required, but I could predict that this feature would require lots of testing and therefore lots of time required to ensure the tutorial functions correctly as well as teaching the user how to play the game coherently. This tutorial would ideally be togglable from the settings menu and automatically run when a new player starts a new game. The tutorial would show the player all the functionality of the game and the basic rules of the game.

Here is a detailed example of what would be done via some pseudocode:

```
1. tutorialMessages = [
2.     "Welcome to Billiard Club! Click to continue.",
3.     "This is the cue ball. Aim using your mouse.",
4.     "Click and aim to complete your shot.",
5.     "Tutorial complete! Press any key to start playing."
6. ]
7.
8. # Function to draw the tutorial overlay
9. function drawTutorialOverlay()
10.     message = tutorialMessages[currentStep - 1]
11.     drawText(message, centerScreen)    # Display the current tutorial message
12. end function
13. # Function to handle tutorial events
```

```

14. function handleTutorialEvent(event)
15.     if event.type = MOUSEBUTTONDOWN or event.type = KEYDOWN:
16.         currentStep = currentStep + 1
17.     end if
18. end function
19. # Main tutorial loop
20. function tutorialScreen()
21.     while currentStep <= length(tutorialMessages)
22.         for each event in getGameEvents()
23.             if event.type = QUIT then
24.                 quitGame()
25.             else
26.                 handleTutorialEvent(event)
27.             end if
28.         next event
29.         drawGameBackground()
30.         drawTutorialOverlay()
31.         updateDisplay()
32.     end while
33.     goToGameModeMenu()      # End tutorial and return to game mode once tutorial complete
34. end function

```

9-ball and Other Game modes

Another feature which would be great to add to the future solution is more game modes. The solution currently has two game modes, 8-ball, and practice (although the practice is simulated with 8-ball). A great addition would be incorporating 9-ball pool. This would reuse many of the physics and menu options included in the current game. It would just require the addition of a few rules, which would be very much doable given extra time.

Here is a detailed example of what would be done via some pseudocode:

```

1. # Game Mode Initialisation
2. if gameMode == "9-ball" then
3.     createCueBall(startPosition)
4.     # Create 9 balls in a diamond formation with numbers 1 to 9
5.     for number from 1 to 9
6.         createBall(colorFor(number), positionFromDiamond(number), number)
7.     next number
8.     currentTarget = 1
9. end if
10.

```

```
11. # Main Game Loop
12. while gameRunning
13.     processPlayerInput()
14.     updateBallPhysics()
15.     resolveCollisions()
16.
17.     if gameMode == "9-ball" then
18.         # Check for fouls (e.g. cue ball fails to hit the current target first)
19.         if foulOccurred() or not cueBallFirstHit(currentTarget) then
20.             handleFoul()
21.             switchTurn()
22.         else
23.             if ballPotted(currentTarget) then
24.                 removeBall(currentTarget)
25.                 currentTarget = lowestNumberRemaining()
26.                 if currentTarget == 9 then
27.                     declareWinner(currentPlayer)
28.                 end if
29.             end if
30.             # Allow continued turn if a legal ball is potted
31.         end if
32.     else
33.         # Use existing 8-ball or practice mode logic
34.         run8BallLogic()
35.     end if
36.     renderGameElements()
37.     updateDisplay()
38.
```

Final Code

A full link to the solution can be found on this GitHub page:

GitHub – <https://github.com/robertscharlie/billiardclub-NEA>

Here the full code has been pasted below:

```

1. import pygame
2. import math
3. import numpy as np
4. import cv2
5.
6. pygame.init() # Initialise all imported pygame modules
7.
8. # Variables
9. FPS = 70
10. resolution = (1280, 720) # Screen dimensions
11. borderName = "Billiard Club" # Displays at the top of the program
12. VEC = pygame.math.Vector2
13. root3 = math.sqrt(3)
14. friction = 0.991
15. restitutionCoeff = 0.95
16.
17. # Pygame Functions and Setup
18. pygame.display.set_caption(borderName)
19. screen = pygame.display.set_mode(resolution) # Sets up the display
20. clock = pygame.time.Clock()
21.
22. # Table Variables and Setup
23. primaryTableColour = "#155843"
24. secondaryTableColour = "#313e46"
25. outerTableColour = "#313e46"
26. tableDimensions = (900, 450)
27. centre = (resolution[0]//2, resolution[1]//2)
28. holeRadius = 15
29. ballRadius = 12
30.
31. # Coordinates for the Table Corners.
32. tableCorners = (
33.     (centre[0] - tableDimensions[0] / 2, centre[1] - tableDimensions[1] / 2), # Top Left
34.     (centre[0] + tableDimensions[0] / 2, centre[1] - tableDimensions[1] / 2), # Top Right
35.     (centre[0] - tableDimensions[0] / 2, centre[1] + tableDimensions[1] / 2), # Bottom Left
36.     (centre[0] + tableDimensions[0] / 2, centre[1] + tableDimensions[1] / 2), # Bottom Right
37. )
38.
39. # Coordinates for Hole Positions
40. holePos = {
41.     "TL": (tableCorners[0][0]+1.1*holeRadius, tableCorners[0][1]+1.1*holeRadius), # Top Left
42.     "TR": (tableCorners[1][0]-1.1*holeRadius, tableCorners[0][1]+1.1*holeRadius), # Top Right
43.     "BL": (tableCorners[0][0]+1.1*holeRadius, tableCorners[2][1]-1.1*holeRadius), # Bottom Left
44.     "BR": (tableCorners[1][0]-1.1*holeRadius, tableCorners[2][1]-1.1*holeRadius), # Bottom Right
45.     "TM": (centre[0], tableCorners[0][1]+1.1*holeRadius), # Top Middle
46.     "BM": (centre[0], tableCorners[2][1]-1.1*holeRadius), # Bottom Middle
47. }
48.
49. # Coordinates for Table Rails
50. tableRails = {
51.     "TL": (
52.         (centre[0] - 1*holeRadius, tableCorners[0][1]),
53.         (centre[0] - 3*holeRadius, tableCorners[0][1]+1.5*holeRadius),
54.         (tableCorners[0][0] + 4*holeRadius, tableCorners[0][1]+1.5*holeRadius),

```

```

55.         (tableCorners[0][0] + 2*holeRadius, tableCorners[0][1]),
56.     ),
57.
58.     "TR": (
59.         (centre[0] + 1*holeRadius, tableCorners[0][1]),
60.         (centre[0] + 3*holeRadius, tableCorners[0][1]+1.5*holeRadius),
61.         (tableCorners[1][0] - 4*holeRadius, tableCorners[0][1]+1.5*holeRadius),
62.         (tableCorners[1][0] - 2*holeRadius, tableCorners[0][1]),
63.     ),
64.
65.     "BL": (
66.         (centre[0] - 1*holeRadius, tableCorners[2][1]-1),
67.         (centre[0] - 3*holeRadius, tableCorners[2][1]-1.5*holeRadius),
68.         (tableCorners[0][0] + 4*holeRadius, tableCorners[2][1]-1.5*holeRadius),
69.         (tableCorners[0][0] + 2*holeRadius, tableCorners[2][1]-1),
70.     ),
71.
72.     "BR": (
73.         (centre[0] + 1*holeRadius, tableCorners[2][1]-1),
74.         (centre[0] + 3*holeRadius, tableCorners[2][1]-1.5*holeRadius),
75.         (tableCorners[1][0] - 4*holeRadius, tableCorners[2][1]-1.5*holeRadius),
76.         (tableCorners[1][0] - 2*holeRadius, tableCorners[2][1]-1),
77.     ),
78. ),
79.
80.     "ML": (
81.         (tableCorners[0][0], tableCorners[3][1] - 2*holeRadius),
82.         (tableCorners[0][0]+1.5*holeRadius, tableCorners[3][1] - 4*holeRadius),
83.         (tableCorners[0][0]+1.5*holeRadius, tableCorners[1][1] + 4*holeRadius),
84.         (tableCorners[0][0], tableCorners[1][1] + 2*holeRadius),
85.     ),
86.
87. ),
88.
89.     "MR": (
90.         (tableCorners[1][0]-1, tableCorners[3][1] - 2*holeRadius),
91.         (tableCorners[1][0]-1.5*holeRadius, tableCorners[3][1] - 4*holeRadius),
92.         (tableCorners[1][0]-1.5*holeRadius, tableCorners[1][1] + 4*holeRadius),
93.         (tableCorners[1][0]-1, tableCorners[1][1] + 2*holeRadius),
94.     ),
95.
96. ),
97. }
98.
99. # Ball info
100. ballInfo = ( # COLOUR, NUMBER, STRIPED
101.     ("white", 0, None),
102.     ("yellow", 1, False),
103.     ("blue", 2, False),
104.     ("red", 3, False),
105.     ("purple", 4, False),
106.     ("orange", 5, False),
107.     ("green", 6, False),
108.     ("maroon", 7, False),
109.     ("black", 8, None),
110.     ("yellow", 9, True),
111.     ("blue", 10, True),
112.     ("red", 11, True),
113.     ("purple", 12, True),
114.     ("orange", 13, True),
115.     ("green", 14, True),
116.     ("maroon", 15, True),
117. )

```

```

118.
119. # Define where each ball should be relative to the start position
120. ballStartPos = {
121.     "white0": (centre[0] - 20*ballRadius, centre[1]),
122.     "yellow1": (centre[0] + 15*ballRadius, centre[1]),
123.     "blue2": (centre[0] + 15*ballRadius + 1 * root3 * ballRadius, centre[1] + 1 * ballRadius),
124.     "red3": (centre[0] + 15*ballRadius + 1 * root3 * ballRadius, centre[1] - 1 * ballRadius),
125.     "purple4": (centre[0] + 15*ballRadius + 2 * root3 * ballRadius, centre[1] - 2 * ballRadius),
126.     "orange5": (centre[0] + 15*ballRadius + 2 * root3 * ballRadius, centre[1] + 2 * ballRadius),
127.     "green6": (centre[0] + 15*ballRadius + 3 * root3 * ballRadius, centre[1] - 1 * ballRadius),
128.     "maroon7": (centre[0] + 15*ballRadius + 3 * root3 * ballRadius, centre[1] + 1 * ballRadius),
129.     "black8": (centre[0] + 15*ballRadius + 2 * root3 * ballRadius, centre[1]),
130.     "yellow9": (centre[0] + 15*ballRadius + 3 * root3 * ballRadius, centre[1] - 3 * ballRadius),
131.     "blue10": (centre[0] + 15*ballRadius + 3 * root3 * ballRadius, centre[1] + 3 * ballRadius),
132.     "red11": (centre[0] + 15*ballRadius + 4 * root3 * ballRadius, centre[1] - 4 * ballRadius),
133.     "purple12": (centre[0] + 15*ballRadius + 4 * root3 * ballRadius, centre[1] - 2 * ballRadius),
134.     "orange13": (centre[0] + 15*ballRadius + 4 * root3 * ballRadius, centre[1]),
135.     "green14": (centre[0] + 15*ballRadius + 4 * root3 * ballRadius, centre[1] + 2 * ballRadius),
136.     "maroon15": (centre[0] + 15*ballRadius + 4 * root3 * ballRadius, centre[1] + 4 * ballRadius),
137. }
138.
139. # Game Variables
140. computerPlaying = True # Set to True if you want the computer to play
141. player1Turn = True # initially player 1's turn
142. player1BallType = None # Stores the type of balls player 1 is potting
143. player2BallType = None # Stores the type of balls player 2 is potting
144. winner = None # Stores the winner of the game
145. player1Name = "Player 1"
146. player2Name = "Player 2"
147. player1Colour = "red"
148. player2Colour = "blue"
149.
150. # Fonts
151. ballFont = pygame.font.SysFont("dubaimedium", int(ballRadius*0.9))
152. ballUIFont = pygame.font.SysFont("dubaimedium", 18)
153. fpsFont = pygame.font.SysFont("dubaimedium", 20)
154. logoFont = pygame.font.Font(".\Project\other\Orbitron-VariableFont_wght.ttf", 70)
155. ballPottedFont = pygame.font.Font(".\Project\other\Orbitron-VariableFont_wght.ttf", 70)
156. playerIndicatorFont = pygame.font.Font(".\Project\other\Orbitron-VariableFont_wght.ttf", 60)
157. MenuLogoFont = pygame.font.Font(".\Project\other\Orbitron-VariableFont_wght.ttf", 100)
158.
159. # Classes
160. class Ball(pygame.sprite.Sprite): # Defines a class for a ball
161.     def __init__(self, colour, radius, pos, velocity, mass, number, isStriped): # Constructor
Method
162.         super().__init__()
163.         self.colour = colour
164.         self.radius = radius
165.         self.pos = VEC(pos) # Position Vector
166.         self.velocity = VEC(velocity) # Velocity vector
167.         self.mass = mass
168.         self.number = number # stores the ball number, e.g. the 8 ball
169.         self.isStriped = isStriped # boolean, stores whether the ball is dotted or striped
170.
171.     def update(self): # Updates the position of the ball
172.         self.pos += self.velocity # moves ball to next position
173.         self.velocity *= friction # reduces the velocity of the ball
174.
175.         if self.velocity.magnitude_squared() < 0.005:
176.             self.velocity = VEC(0,0)
177.
178.         # Create exception for white ball:
179.         if self.pos == VEC(0,0):

```



```

180.         return None
181.
182.     # Checking collisions with the outer walls: (ensures all balls stay in region)
183.     if self.pos[0] - self.radius <= tableCorners[0][0]: # Left Wall
184.         self.pos[0] = self.radius + tableCorners[0][0] # Put ball on edge of wall
185.         self.velocity[0] *= -1 # Reverses X component
186.
187.     if self.pos[0] + self.radius >= tableCorners[1][0]: # Right Wall
188.         self.pos[0] = tableCorners[1][0] - self.radius # Put ball on edge of wall
189.         self.velocity[0] *= -1 # Reverses X component
190.
191.     if self.pos[1] - self.radius <= tableCorners[0][1]: # Top Wall
192.         self.pos[1] = self.radius + tableCorners[0][1] # Put ball on edge of wall
193.         self.velocity[1] *= -1 # Reverses Y component
194.
195.     if self.pos[1] + self.radius >= tableCorners[2][1]: # Bottom Wall
196.         self.pos[1] = tableCorners[2][1] - self.radius # Put ball on edge of wall
197.         self.velocity[1] *= -1 # Reverses Y component
198.
199.     def checkResolveCollision(self, other): # Check if balls are colliding, then resolves
collision
200.         # Check if balls colliding
201.         def checkColliding(self, other):
202.             posDifference = self.pos - other.pos # find difference in positions
203.             distance = self.pos.distance_to(other.pos) # distance squared between points
204.             if distance == 0: # prevents normalisation error
205.                 return False, posDifference
206.             elif distance < (self.radius + other.radius): # If colliding
207.                 return True, posDifference
208.             return False, posDifference
209.
210.         colliding, posDifference = checkColliding(self, other) # checks if colliding
211.         calculateCollision = False # set initial case to False
212.
213.         if colliding:
214.             calculateCollision = True # calculate collision at the end
215.             self.pos -= self.velocity # step back position of both balls
216.             other.pos -= other.velocity # step back position of both balls
217.             colliding, posDifference = checkColliding(self, other) # checks if colliding
218.             while not colliding: # while colliding
219.                 # incrementally move the balls forward until they are tangential
220.                 self.pos += 0.05 * self.velocity
221.                 other.pos += 0.05 * other.velocity
222.                 colliding, posDifference = checkColliding(self, other) # checks if colliding
223.
224.         if calculateCollision: # if colliding, calculate velocities after collision
225.             n = VEC(posDifference).normalize() # creates a normalised vector as an origin for
calculations
226.             relVel = self.velocity - other.velocity # calculates the relative velocity of the
balls
227.
228.             # Calculate volume of ball collision sound
229.             collisionSpeed = VEC(relVel).magnitude() # obtains rough value of speed
230.             volume =
masterVolumeSlider.returnValue()*ballVolumeSlider.returnValue()*(collisionSpeed/2)
231.             ballCollideSound.set_volume(max(0.1,min(0.8,volume))) # updates volume of sound
232.             ballCollideSound.play() # plays sound when balls collide
233.
234.             # Calculate Vectors of balls after collision
235.             impulse = 2 * (n * relVel) / (self.mass + other.mass) # calculates the impulse each
ball will get
236.             self.velocity -= impulse * other.mass * n # applies impulse to each ball
237.             other.velocity += impulse * self.mass * n

```

```

238.         self.velocity *= restitutionCoeff # decreases velocity of both balls
239.         other.velocity *= restitutionCoeff
240.
241.         while colliding: # if still colliding
242.             # push balls apart to prevent sticking
243.             self.pos += n
244.             other.pos -= n
245.             colliding, posDifference = checkColliding(self, other) # checks if still colliding
246.
247.         def checkResolveWallCollision(self): # Check if balls are colliding with the walls, then
resolves collision
248.             if self.velocity == VEC(0,0): # Prevents Normalisation Error
249.                 return None
250.
251.         def checkColliding():
252.             # Loops through each of the Rails (where collisions should be done)
253.             for i in tableRails:
254.                 for j in range(3):
255.                     lineStart = tableRails[i][j] # Coordinates of the Line Start and End
256.                     lineEnd = tableRails[i][j+1] # e.g. [ [x1,y1] , [x2,y2] ]
257.                     lineVec = VEC(lineEnd[0] - lineStart[0], lineEnd[1] - lineStart[1]) # Vector
of the Line
258.                     circleVec = VEC(self.pos[0] - lineStart[0], self.pos[1] - lineStart[1]) #
Vector of the circle to the start of the line
259.                     t = max(0, min(1, (circleVec[0] * lineVec[0] + circleVec[1] * lineVec[1]) /
lineVec.magnitude_squared()))
260.                     # This creates a parameter that will calculate the s.f of the normal
projection that maps the circle onto the line segment
261.                     linePoint = (lineStart[0] + t * lineVec[0], lineStart[1] + t * lineVec[1]) #
Calculate closest point on the line to the circle
262.                     circlePointVec = VEC(linePoint[0] - self.pos[0], linePoint[1] - self.pos[1]) #
Vector of the circle to the closest point
263.                     distance = circlePointVec.magnitude() # calculates distance between the circle
and closest point
264.                     # Check if colliding
265.                     if distance <= self.radius:
266.                         return True, circlePointVec
267.                     return False, None # if not colliding, return False
268.
269.         calculateCollision = False
270.         isColliding, circlePointVec = checkColliding() # checks if colliding
271.
272.         if isColliding: # if colliding
273.             calculateCollision = True # calculate collision at the end
274.             n = circlePointVec.normalize() # creates a normal vector as an origin for calculations
275.
276.             while isColliding: # while colliding
277.                 self.pos -= self.velocity.normalize() # incrementally move the ball back until it is
no longer colliding
278.                 isColliding, circlePointVec = checkColliding() # checks if colliding
279.
280.             # Check if colliding
281.             if calculateCollision:
282.                 impulse = 2*(n*self.velocity)/(self.mass) # calculates the impulse the ball will get
283.                 self.velocity -= impulse * n # applies impulse to each ball
284.                 self.velocity *= restitutionCoeff # decreases velocity
285.
286.         def checkHoleCollision(self):
287.             for i in holePos: # for each hole on the table
288.                 distance2 = self.pos.distance_squared_to(holePos[i]) # compute the distance
289.                 if distance2 <= (holeRadius*0.8)**2: # if colliding
290.                     return True
291.             return False

```

```

292.
293.     def draw(self, screen): # Draws the ball to the screen
294.         pygame.draw.circle(screen, self.colour, self.pos, self.radius) # draws outer circle
295.
296.         if self.isStriped:
297.             pygame.draw.circle(screen, "white", self.pos, self.radius) # draws outer circle to
overlap stripe
298.             # Draw the stripe
299.             stripe_height = ballRadius / 0.65 # height of the stripe
300.             pygame.draw.ellipse(
301.                 screen,
302.                 self.colour,
303.                 (
304.                     self.pos[0] - ballRadius,
305.                     self.pos[1] - stripe_height // 2,
306.                     ballRadius * 2,
307.                     stripe_height
308.                 )
309.             )
310.
311.         if self.number != 0: # draws a number on every ball except white
312.             text = ballFont.render(str(self.number), True, "black") # renders font
313.             pygame.draw.circle(screen, "white", self.pos, text.get_height()/2.2) # draws inner
circle
314.
315.             fontPos = [self.pos[0] - text.get_width()/2, self.pos[1] - text.get_height()/2] #
offsets the font to be central with ball
316.
317.             screen.blit(text, fontPos) # adds font to screen
318.
319.     def delete(self):
320.         self.kill()
321.
322. class VerticalSlider(pygame.sprite.Sprite): # Vertical Slider Class
323.     def __init__(self, outerColour, sliderColour, position, width, height, value):
324.         super().__init__()
325.         self.outerColour = outerColour # colour of the outer section
326.         self.sliderColour = sliderColour # colour of the slider
327.         self.position = position # (x,y)
328.         self.width = width # width of the slider
329.         self.height = height # height of the slider
330.         self.value = value # starting value of the slider
331.         self.mouseDown = False # for checking if user is interacting with slider
332.
333.     def draw(self, screen): # draw method that draws slider to screen
334.         # Outer Box
335.         pygame.draw.rect(
336.             screen,
337.             self.outerColour,
338.             pygame.Rect(
339.                 self.position[0],
340.                 self.position[1],
341.                 self.width,
342.                 self.height
343.             ),
344.             3,
345.             self.width
346.         )
347.         # Slider
348.         pygame.draw.circle(
349.             screen,
350.             self.sliderColour,
351.             (self.position[0]+self.width/2, self.position[1] + self.value * self.height),

```

```

352.         self.width,
353.     )
354.
355.     def update(self, event): # updates slider position
356.         if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # when left mouse button
clicked
357.             mousePos = pygame.mouse.get_pos() # get mouse position
358.             if 0 <= mousePos[0] - self.position[0] <= self.width: # if click is within x bounds
359.                 if 0 <= mousePos[1] - self.position[1] <= self.height: # if click is within y
bounds
360.                     self.mouseDown = True
361.         if event.type == pygame.MOUSEBUTTONUP: # when mouse button is released
362.             self.mouseDown = False
363.         if self.mouseDown:
364.             mousePos = pygame.mouse.get_pos() # get mouse position
365.             distance = mousePos[1]-self.position[1] # finds distance between bottom and mousePos
366.             self.value = distance/self.height # sets the value of slider to the ratio of
distances.
367.                 self.value = max(0, min(1, self.value))
368.
369.     def returnValue(self): # returns slider position value
370.         return round(1-self.value, 3)
371.
372.     def updateColour(self, newColour): # updates the colour of the slider
373.         self.sliderColour = newColour
374.
375. class HorizontalSlider(pygame.sprite.Sprite): # Horizontal Slider Class
376.     def __init__(self, x, y, width, height, startValue, sliderColour, outerColour, circleColour,
sliderType):
377.         self.sliderType = sliderType
378.         self.outerColour = outerColour # Colour of the outer section
379.         self.sliderColour = sliderColour # Colour of the slider
380.         self.x = x # X position of the slider
381.         self.y = y # Y position of the slider
382.         self.width = width # Width of the slider
383.         self.height = height # Height of the slider
384.         self.value = startValue # Starting value of the slider [0,1]
385.         self.rect = pygame.Rect(x, y, width, height) # Rectangle representing the slider's
position and size
386.         self.circlePos = [self.x + self.value * self.width, self.y + self.height / 2] # Calculate
the position of the slider circle
387.         self.clicked = False
388.         self.circleColour = circleColour
389.
390.     def draw(self):
391.         pygame.draw.rect(screen, self.outerColour, self.rect, 0, 10) # Draw the slider rectangle
392.         if self.sliderType == "colourPicker": # Check if the slider type is a colour picker
393.             for i in range(self.width-18): # Iterate through the width of the slider
394.                 colour = pygame.Color(0) # Create a new colour object
395.                 colour.hsla = (int(360 * i / self.width), 100, 50, 100) # Set the colour based on
the position
396.                 pygame.draw.rect(screen, colour, (self.x+i, self.y, self.width - i, self.height),
0, 10) # Draw the colour rectangle
397.             else:
398.                 pygame.draw.rect(screen, self.sliderColour, (self.x, self.y, self.width*self.value,
self.height), 0, 10) # Draw the colour rectangle
399.                 pygame.draw.rect(screen, self.outerColour, self.rect, 1, 10) # Draw the outer slider
rectangle
400.                 pygame.draw.circle(screen, self.outerColour, self.circlePos, self.height * 0.6 + 1) #
Draw outer circle
401.                 if self.sliderType == "colourPicker":
402.                     pygame.draw.circle(screen, self.getColour(), self.circlePos, self.height * 0.6) #
Draw the slider circle

```

```

403.         else:
404.             pygame.draw.circle(screen, self.circleColour, self.circlePos, self.height * 0.6) #
Draw the slider circle
405.
406.     def getColour(self):
407.         # calculate selected Colour
408.         selectedColour = pygame.Color(0)
409.         selectedColour.hsla = (int(self.value * (self.width+60)), 100, 50, 100)
410.         return selectedColour
411.
412.     def returnValue(self):
413.         self.value = (self.circlePos[0]-self.x)/self.width # update value (between 0 and 1)
414.         return self.value
415.
416.
417.     def handleEvent(self, event):
418.         if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # Check if the left mouse
button is clicked
419.             if self.rect.collidepoint(event.pos): # Check if the text box was clicked
420.                 self.clicked = True
421.             else:
422.                 self.clicked = False
423.             elif event.type == pygame.MOUSEBUTTONUP and event.button == 1: # When mouse button is
released
424.                 self.clicked = False
425.
426.     def update(self):
427.         if self.clicked: # when clicked
428.             mousePos = pygame.mouse.get_pos() # get mouse position
429.             self.circlePos[0] = min(max(self.x, mousePos[0]), self.x + self.width) # update
position
430.             self.value = (self.circlePos[0]-self.x)/self.width # update value (between 0 and 1)
431.
432. class Button: # Defines Button class
433.     def __init__(self, text, fontSize, x, y, padding, callback):
434.         self.text = text # Text to display on the button
435.         self.font = pygame.font.Font("C:/Users/crobe/Documents/GitHub/billiardclub-
NEA/Project/other/Orbitron-VariableFont_wght.ttf", fontSize) # Font for the button text
436.         self.callback = callback # Function to call when the button is clicked
437.
438.         self.hovering = False # Set to true if the mouse is hovering over.
439.         self.wasClicked = False
440.
441.         self.boxColour = "white" # Button colour
442.         self.boxColourClicked = "grey" # Button colour when clicked
443.         self.textColour = "black" # Text colour when the mouse is not hovering over
444.         self.textColourHovering = "red" # Text colour when the mouse is hovering over
445.
446.         self.textSurface = self.font.render(self.text, True, self.textColour) # Render the button
text
447.         # Rectangle representing the button's position and size
448.         self.rect = pygame.Rect(x, y, self.textSurface.get_width() + padding,
self.textSurface.get_height() + padding,)
449.
450.     def draw(self):
451.         textColour = ""
452.         boxColour = ""
453.         if self.hovering: # Check if the button is being hovered over
454.             textColour = self.textColourHovering # Set textColour to hovering colour
455.         else:
456.             textColour = self.textColour # Set textColour to default colour
457.         if self.wasClicked: # Check if the button is clicked
458.             boxColour = self.boxColourClicked # Set boxColour to clicked colour

```

```

459.         else:
460.             boxColour = self.boxColour # Set boxColour to default colour
461.             self.textSurface = self.font.render(self.text, True, textColour) # Render the button text
462.             pygame.draw.rect(screen, boxColour, self.rect, 0, 10) # Draw the button rectangle
463.             textRect = self.textSurface.get_rect(center=self.rect.center) # Get the rectangle for the
text surface
464.             screen.blit(self.textSurface, textRect) # Draw the text on the button
465.
466.         def checkHover(self, mousePos): # checks if the mouse is over the button.
467.             if self.rect.collidepoint(mousePos): # Check if the mouse position is within the button
rectangle
468.                 self.hovering = True # Sets to True if hovering
469.                 return True # returns true to say the mouse is hovering over
470.             else:
471.                 self.hovering = False # Sets to False if hovering
472.                 return False # returns False to say the mouse is hovering over
473.
474. class Menu: # Defines Menu class
475.     def __init__(self, fontSize, pos, options, functions): # Initialises the Menu class
476.         self.fontSize = fontSize # Font size for the menu options
477.         self.pos = pos # Position of the menu
478.         self.options = options # List of menu options
479.         self.functions = functions # List of functions corresponding to each menu option
480.         self.buttons = [] # List to store buttons
481.         self.create_buttons() # Calls the method to create buttons
482.
483.     def create_buttons(self): # Method to create buttons
484.
485.         increment = (resolution[1]-200)/(len(self.options)-1) # Vertical spacing between buttons
486.
487.         for i in range(len(self.options)): # Iterate through the options
488.             # Create a button for each option and append it to the buttons list along with its
function when clicked
489.             self.buttons.append(Button(self.options[i], self.fontSize, self.pos[0], self.pos[1] +
i*increment, 20, self.functions[i]))
490.
491.     def draw(self): # Method to draw buttons
492.         for button in self.buttons: # Iterate through each button
493.             button.draw() # Draw the button
494.
495.     def update(self): # Method to update button states
496.         for button in self.buttons: # Iterate through each button
497.             button.checkHover(pygame.mouse.get_pos()) # Check if the mouse is hovering over the
button
498.
499.     def handleEvent(self, event): # Method to handle events
500.         if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # Check if the left mouse
button is clicked
501.             for button in self.buttons: # Iterate through each button
502.                 if button.checkHover(event.pos): # Check if click occurred and if hovering over
button
503.                     clickSound.play() # plays click sound
504.                     button.wasClicked = True # Set button as clicked
505.                 else:
506.                     button.wasClicked = False # Set button as not clicked
507.             elif event.type == pygame.MOUSEBUTTONUP and event.button == 1: # When mouse button is
released
508.                 for button in self.buttons: # Iterate through each button
509.                     if button.wasClicked:
510.                         button.wasClicked = False
511.                         eval(button.callback) # Execute the button's callback function
512.                         button.wasClicked = False # Set button as not clicked
513.

```

```

514. class TextBox: # Defines TextBox class
515.     def __init__(self, x, y, permanentText, initialText, fontSize): # Initialises the TextBox
class
516.         self.permanentText = permanentText # permanent Text that can't be deleted by user in the
text field
517.         self.text = initialText # Initial text to display in the text box
518.         self.font = pygame.font.Font("Project/other/Orbitron-VariableFont_wght.ttf", fontSize) #
Font for the text box text
519.         self.textColour = "black" # Text colour
520.         self.boxColour = "white" # Box colour
521.         self.active = False # Indicates if the text box is active (clicked)
522.         # Colours for the text box when active and passive
523.         self.colourActive = "red"
524.         self.colourPassive = "black"
525.         self.textSurface = self.font.render(permanentText + self.text, True, self.textColour) #
Render the text
526.         # Rectangle representing the text box's position and size
527.         self.rect = pygame.Rect(x, y, self.textSurface.get_width() + 10,
self.textSurface.get_height() + 10)
528.
529.     def draw(self): # Method to draw the text box
530.         if self.active: # Check if the text box is active
531.             self.textColour = self.colourActive # Set text colour to active colour
532.         else:
533.             self.textColour = self.colourPassive # Set text colour to passive colour
534.
535.         self.rect.w = self.textSurface.get_width() + 10 # Update the width of the text box
rectangle based on the text width
536.
537.         self.textSurface = self.font.render(self.permanentText + self.text, True, self.textColour)
# Render the text
538.         pygame.draw.rect(screen, self.boxColour, self.rect, 0, 10) # Draw the text box rectangle
539.         textRect = self.textSurface.get_rect(center=self.rect.center) # Get the rectangle for the
text surface
540.         screen.blit(self.textSurface, textRect) # Draw the text on the text box
541.
542.     def handle_event(self, event): # Method to handle events
543.         if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1: # Check if the left mouse
button is clicked
544.             if self.rect.collidepoint(event.pos): # Check if the text box was clicked
545.                 self.active = True # Set text box as active
546.             else:
547.                 self.active = False # Set text box as inactive
548.         if event.type == pygame.KEYDOWN: # Check if a key is pressed
549.             if self.active: # Check if the text box is active
550.                 if event.key == pygame.K_RETURN: # Check if the Enter key is pressed
551.                     self.active = False # Set text box as inactive
552.                 elif event.key == pygame.K_BACKSPACE: # Check if the Backspace key is pressed
553.                     self.text = self.text[:-1] # Remove the last character
554.                 else:
555.                     self.text += event.unicode # Add the character to the text
556.                     self.textSurface = self.font.render(self.permanentText + self.text, True,
self.textColour) # Re-render the text
557.
558. # Initialise Menus
559. mainMenu = Menu(60, [50,50], ["Start", "Options", "Quit"], ["gameModeScreen()", "optionsScreen()",
"quitGame()"]) # create a menu
560. gameModeMenu = Menu(60, [50,50], ["Player vs Player", "Player vs AI", "Sandbox Mode", "Back"],
["runGame(False, False)", "runGame(True, False)", "runGame(False, True)", "mainMenuScreen()"]) # create
a menu
561. optionsMenu = Menu(60, [50,50], ["Customise Game", "Volume Control", "Back"],
["customisationScreen()", "volumeControlScreen()", "mainMenuScreen()"]) # create a menu

```



```

562. customisationMenu = Menu(60, [50,50], ["Customise Game", "Back"], ["customisationScreen()",
"optionsScreen()"]) # create a menu
563. pauseMenu = Menu(60, [50,50], ["Game Paused", "Exit Game"], ["print()", "mainMenuScreen()"]) #
create a menu
564. volumeMenu = Menu(60, [50,50], ["Volume Control", "Back"], ["volumeControlScreen()",
"optionsScreen()"]) # create a menu
565. winMenu = Menu(60, [50,50], ["Play Again", "Exit to Menu"], ["gameModeScreen()",
"mainMenuScreen()"]) # create a menu
566.
567. # Initialise Slider sprite
568. ballPowerSlider = VerticalSlider("white", "red", (tableCorners[0][0]/2 - 30,tableCorners[0][1]-
30), 30, tableDimensions[1] + 60, 0.5)
569.
570. # Initialise Sliders
571. masterVolumeSlider = HorizontalSlider(390, 245, 260, 20, 0.7, "green", (30, 30, 30), "white",
"normal")
572. musicVolumeSlider = HorizontalSlider(390, 345, 260, 20, 0.5, "green", (30, 30, 30), "white",
"normal")
573. ballVolumeSlider = HorizontalSlider(390, 445, 260, 20, 0.6, "green", (30, 30, 30), "white",
"normal")
574.
575. # Initialise Background Music
576. pygame.mixer.music.load("Project/other/chill-house-sunset-groove-251697.mp3")
577. pygame.mixer.music.play(-1,0.0) # Play Game Music in background indefinitely
578.
579. # Initialise Other Music/Sounds
580. cclickSound = pygame.mixer.Sound("Project/other/click-151673.mp3")
581. ballCollideSound = pygame.mixer.Sound("Project/other/pool_break-105353.mp3")
582. ballPottedSound = pygame.mixer.Sound("Project/other/billiard-hole-288419.mp3")
583.
584. # Initially sets the volumes of the Sound effects and music:
585. pygame.mixer.music.set_volume(masterVolumeSlider.returnValue()*musicVolumeSlider.returnValue())
586. cclickSound.set_volume(masterVolumeSlider.returnValue()*ballVolumeSlider.returnValue())
587. ballCollideSound.set_volume(masterVolumeSlider.returnValue()*ballVolumeSlider.returnValue())
588. ballPottedSound.set_volume(masterVolumeSlider.returnValue()*ballVolumeSlider.returnValue())
589.
590. ballGroup = pygame.sprite.Group() # Groups all balls for easier maintenance
591.
592. balls = []
593.
594. def initialiseBalls():
595.     global balls
596.     # Initialise each ball
597.     balls = [
598.         Ball("white", ballRadius, ballStartPos["white0"], (0, 0), 1, 0, None), # Whiteball
599.         Ball("yellow", ballRadius, ballStartPos["yellow1"], (0, 0), 1, 1, False), # Yellow 1
600.         Ball("blue", ballRadius, ballStartPos["blue2"], (0, 0), 1, 2, False), # Blue 2
601.         Ball("red", ballRadius, ballStartPos["red3"], (0, 0), 1, 3, False), # Red 3
602.         Ball("purple", ballRadius, ballStartPos["purple4"], (0, 0), 1, 4, False), # Purple 4
603.         Ball("orange", ballRadius, ballStartPos["orange5"], (0, 0), 1, 5, False), # Orange 5
604.         Ball("green", ballRadius, ballStartPos["green6"], (0, 0), 1, 6, False), # Green 6
605.         Ball("maroon", ballRadius, ballStartPos["maroon7"], (0, 0), 1, 7, False), # Maroon 7
606.         Ball("black", ballRadius, ballStartPos["black8"], (0, 0), 1, 8, None), # Black 8
607.         Ball("yellow", ballRadius, ballStartPos["yellow9"], (0, 0), 1, 9, True), # Yellow 9
608.         Ball("blue", ballRadius, ballStartPos["blue10"], (0, 0), 1, 10, True), # Blue 10
609.         Ball("red", ballRadius, ballStartPos["red11"], (0, 0), 1, 11, True), # Red 11
610.         Ball("purple", ballRadius, ballStartPos["purple12"], (0, 0), 1, 12, True), # Purple 12
611.         Ball("orange", ballRadius, ballStartPos["orange13"], (0, 0), 1, 13, True), # Orange 13
612.         Ball("green", ballRadius, ballStartPos["green14"], (0, 0), 1, 14, True), # Green 14
613.         Ball("maroon", ballRadius, ballStartPos["maroon15"], (0, 0), 1, 15, True), # Maroon 15
614.     ]
615.
616.     for i in balls: # adds each ball to the group

```



```

617.         ballGroup.add(i)
618.
619. def deleteAllBalls():
620.     for ball in ballGroup.sprites(): # loops through all balls
621.         ball.delete() # deletes each ball
622.
623. initialiseBalls() # call function straight away to initialise balls on table
624.
625. # Game Functions
626. def checkIfBallMoving(): # Checks if any ball is moving (returns True if any ball is moving)
627.     ballMoving = False # sets initial case to False
628.     for ball in ballGroup.sprites():
629.         if ball.velocity != (0,0): # checks if the ball is completely still
630.             ballMoving = True # if even one ball is still moving, it is set to True.
631.     return ballMoving
632.
633. def distanceToClosestBall(intialPos, type): # returns distance from closest ball
634.     closestDistance = float("inf") # sets initial value to infinity
635.     otherBall = None # sets initial value to none
636.     for ball in ballGroup.sprites(): # loops through every ball (except whiteball)
637.         if (type == "striped" and ball.isStriped and not ball.number == 8) or \
638.            (type == "dotted" and not ball.isStriped and not ball.number == 8) or \
639.            (type == None): # if the ball is the specified type or not classified
640.             if ball.number != 0: # skip if the ball is itself (white)
641.                 distance = VEC(intialPos).distance_to(ball.pos) # finds the distance between the
ball and the whiteball
642.                 if distance < closestDistance: # if the new distance is smaller, set the new
distance to the actual distance
643.                     closestDistance = distance
644.                     otherBall = ball
645.     return closestDistance, otherBall # return the distance and then the instance of the other
ball
646.
647. def distanceToClosestRail(intialPos): # returns distance from closest rail
648.     closestDistance = float("inf") # sets initial value to infinity
649.     railVec = None
650.     # Loops through each of the Rails (where collisions should be done)
651.     for i in tableRails:
652.         for j in range(3):
653.             lineStart = tableRails[i][j] # Coordinates of the Line Start and End
654.             lineEnd = tableRails[i][j+1] # e.g. [ [x1,y1] , [x2,y2] ]
655.             lineVec = VEC(lineEnd[0] - lineStart[0], lineEnd[1] - lineStart[1]) # Vector of the
Line
656.             pointVec = VEC(intialPos[0] - lineStart[0], intialPos[1] - lineStart[1]) # Vector of
the circle to the start of the line
657.             dot = VEC.dot(pointVec,lineVec) # finds the dot product of the vectors
658.             # This creates a parameter that will calculate the s.f of the normal projection that
maps the circle onto the line segment
659.             t = max(0, min(1, dot / lineVec.magnitude_squared()))
660.             linePoint = (lineStart[0] + t * lineVec[0], lineStart[1] + t * lineVec[1]) # Calculate
closest point on the line to the circle
661.             circlePointVec = VEC(linePoint[0] - intialPos[0], linePoint[1] - intialPos[1]) #
Vector of the circle to the closest point
662.             distance = circlePointVec.magnitude() # calculates distance between the circle and
closest point
663.
664.             # if the new distance is smaller, set the new distance to the actual distance
665.             if distance < closestDistance:
666.                 closestDistance = distance
667.                 railVec = lineVec
668.
669.     return closestDistance, railVec # returns the distance to the rail and the vector of the rail
670.

```

```

671. def resolveAllCollisions(): # Checks and resolves every collision between every pair of balls
(only if colliding)
672.     for i in range(len(ballGroup) - 1):
673.         for j in range(i, len(ballGroup)):
674.             ballGroup.sprites()[i].checkResolveCollision(ballGroup.sprites()[j])
675.
676. def resolveWallCollisions(): # Checks and resolves every collision between every ball and the
walls
677.     for ball in ballGroup.sprites():
678.         ball.checkResolveWallCollision() # checks wall collisions
679.
680. def resolveBallPotted(): # Checks if a ball is potted
681.     toDelete = [] # deletes balls after looped through list
682.     for ball in ballGroup.sprites():
683.         if ball.checkHoleCollision(): # checks if each ball is in a hole
684.             if ball.number == 0: # if the white ball is potted
685.                 ballPottedSound.play() # plays sound when ball potted
686.                 gameQueue.append(resetWhiteBall) # tells the game to reset the white ball on the
start of the round
687.                 ball.velocity = VEC(0,0) # resets the velocity of the white ball
688.                 ball.pos = VEC(0,0) # resets the position of the white ball
689.             else:
690.                 ballPottedSound.play() # plays sound when ball potted
691.                 # On first run, set the ball type for each player
692.                 global player1BallType, player2BallType
693.                 if player1BallType == None and player2BallType == None:
694.                     if ball.isStriped and player1Turn or not ball.isStriped and not player1Turn:
695.                         player1BallType = "striped"
696.                         player2BallType = "dotted"
697.                     elif ball.isStriped and not player1Turn or not ball.isStriped and player1Turn:
698.                         player1BallType = "dotted"
699.                         player2BallType = "striped"
700.                 else:
701.                     print("Error: Ball type not set")
702.
703.                 pottedBallInfo = [ball.number, ball.isStriped] # collects information about the
potted ball
704.
705.                 pottedBalls.append(pottedBallInfo) # adds the ball number to the list of potted
balls
706.                 roundPottedBalls.append(pottedBallInfo) # adds the ball number to the list of the
round's potted balls
707.
708.                 toDelete.append(ball) # removes the ball from the game
709.
710.                 for i in toDelete:
711.                     i.delete()
712.
713.                 ball.draw(screen) # draws each ball
714.
715. def resetWhiteBall(): # resets the white ball
716.     balls[0].pos = ballStartPos["white0"] # resets the position of the white ball
717.
718. def gameLogic(): # determines state of next round on all factors
719.     global player1Turn
720.     global player1BallType
721.     global player2BallType
722.     global pottedBalls
723.     global roundPottedBalls
724.     global gameQueue
725.     global winner
726.
727.     changeTurn = False # set initial case to False

```

```

728.     eightBallPotted = False # set initial case to False
729.
730.     if len(roundPottedBalls) == 0: # if no balls are potted
731.         changeTurn = True # switch turns
732.     else:
733.         for i in roundPottedBalls: # loop through each ball potted
734.             number = i[0]
735.             isStriped = i[1]
736.
737.             if number == 8: # if the 8 ball is potted
738.                 eightBallPotted = True # set case to True
739.
740.             if number == 0: # if the white ball is potted
741.                 changeTurn = True
742.
743.             if player1Turn: # if it is player 1's turn
744.                 if player1BallType == "dotted": # if player 1 type is dotted
745.                     if isStriped: # if the ball is striped
746.                         changeTurn = True
747.                 elif player1BallType == "striped": # if player 1 type is striped
748.                     if not isStriped: # if the ball is dotted
749.                         changeTurn = True
750.             if not player1Turn: # if it is player 2's turn
751.                 if player2BallType == "dotted": # if player 2 type is dotted
752.                     if isStriped: # if the ball is striped
753.                         changeTurn = True
754.                 elif player2BallType == "striped": # if player 2 type is striped
755.                     if not isStriped: # if the ball is dotted
756.                         changeTurn = True
757.
758.     if eightBallPotted: # if the 8 ball is potted
759.         # check if all other balls are potted.
760.         dottedPotted = 0
761.         stripedPotted = 0
762.         # count the number of potted balls from each type
763.         for i in pottedBalls:
764.             if i[1]:
765.                 stripedPotted += 1
766.             else:
767.                 dottedPotted += 1
768.
769.         if dottedPotted == 7 or stripedPotted == 7:
770.             # whoever potted the 8 ball wins
771.             if player1Turn:
772.                 winner = player1Name
773.             else:
774.                 winner = player2Name
775.         else:
776.             # whoever potted the 8 ball loses
777.             if player1Turn:
778.                 winner = player2Name
779.             else:
780.                 winner = player1Name
781.
782.     if changeTurn: # if the turn is to be changed
783.         player1Turn = not player1Turn # switch turns
784.         if player1Turn:
785.             ballPowerSlider.updateColour(player1Colour) # update the colour of the power slider
786.         else:
787.             ballPowerSlider.updateColour(player2Colour) # update the colour of the power slider
788.
789.     ballPowerSlider.value = 0.5 # reset the power slider
790.

```

```

791.     roundPottedBalls = [] # resets the list of potted balls
792.
793. def checkWin(winner): # checks if there is a winner
794.     if winner != None: # if there is a winner
795.         if winner == player1Name: # if player 1 wins
796.             winScreen(winner, player1Colour) # display the win screen
797.         else:
798.             winScreen(winner, player2Colour) # display the win screen
799.
800. def computerShoot(): # computer finds shot and takes shot
801.     global player2BallType
802.
803.     if player2BallType == None: # if the ball type is not set
804.         # find closest ball and hit it
805.         distance, closestBall = distanceToClosestBall(balls[0].pos, type=None) # find closest ball
806.         ballVec = (closestBall.pos - balls[0].pos).normalize() # vector of the ball to the white
ball
807.         balls[0].velocity = ballVec*20 # give ball a velocity
808.
809.     else:
810.         # find closest ball of that ball type and hit it
811.
812.         hitEightBall = True # set initial case to True
813.         for i in ballGroup.sprites(): # loop through each ball
814.             if i.number == 8 or i.number == 0: # if the ball is the 8 ball or the white ball
815.                 continue # skip the ball
816.             elif (player2BallType == "striped" and i.isStriped) or (player2BallType == "dotted"
and not i.isStriped): # if there is a ball of correct type, hit that ball
817.                 hitEightBall = False # set case to False
818.                 # else:
819.                 #     print("no ball of correct type to hit") # if there is no ball of correct type,
return error
820.
821.         if hitEightBall: # if there is no ball of correct type, hit the 8 ball
822.             ballVec = (balls[8].pos - balls[0].pos).normalize() # vector of the ball to the white
ball
823.             balls[0].velocity = ballVec*20 # give ball a velocity
824.         else:
825.             distance, closestBall = distanceToClosestBall(balls[0].pos, type=player2BallType) #
find closest ball of correct type
826.             ballVec = (closestBall.pos - balls[0].pos).normalize() # vector of the ball to the
white ball
827.             balls[0].velocity = ballVec*20 # give ball a velocity
828.
829. def drawMenuLogo():
830.     # Render the menu logo text
831.     menuLogo1 = MenuLogoFont.render(("BILLIARD"), True, "white")
832.     menuLogo2 = MenuLogoFont.render(("CLUB"), True, "white")
833.     menuLogo1OUTLINE = MenuLogoFont.render(("BILLIARD"), True, "black")
834.     menuLogo2OUTLINE = MenuLogoFont.render(("CLUB"), True, "black")
835.
836.     # Calculate the position for the logos
837.     logoPos = (centre[0] + 300, centre[1])
838.     logo1Pos = (logoPos[0] - menuLogo1.get_width() // 2, logoPos[1] - menuLogo1.get_height())
839.     logo2Pos = (logoPos[0] - menuLogo2.get_width() // 2, logoPos[1])
840.
841.     # Draw the outline of the logo
842.     for i in range(-4,5):
843.         for j in range(-4,5):
844.             screen.blit(menuLogo1OUTLINE, (logo1Pos[0]+i,logo1Pos[1]+j))
845.             screen.blit(menuLogo2OUTLINE, (logo2Pos[0]+i,logo2Pos[1]+j))
846.
847.     # Draw the logos on the screen

```

```

848.     screen.blit(menuLogo1, logo1Pos)
849.     screen.blit(menuLogo2, logo2Pos)
850.
851. def quitGame(): # Function to quit the game
852.     pygame.quit()
853.     exit()
854.
855. # Draw Elements to Screen
856. def drawTable(): # Draws the table
857.
858.     # Outer Ring
859.     pygame.draw.rect(
860.         screen,
861.         outerTableColour,
862.         pygame.Rect(
863.             tableCorners[0][0]-20, tableCorners[0][1]-20,
864.             tableDimensions[0]+40, tableDimensions[1]+40
865.         ),
866.         15, # Width
867.         20 # Border Radius
868.     )
869.
870.     # Draw Table
871.     pygame.draw.rect(
872.         screen,
873.         primaryTableColour, # Colour
874.         pygame.Rect(
875.             tableCorners[0][0], # Top Left x
876.             tableCorners[0][1], # Top Left y
877.             tableDimensions[0], # Width
878.             tableDimensions[1] # Height
879.         ),
880.         border_radius=3 # Border Radius
881.     )
882.
883.     # Draw Table Rails
884.     pygame.draw.polygon(screen, secondaryTableColour, tableRails["TL"]) # Top Left Side
885.     pygame.draw.polygon(screen, secondaryTableColour, tableRails["TR"]) # Top Right Side
886.     pygame.draw.polygon(screen, secondaryTableColour, tableRails["BL"]) # Bottom Left Side
887.     pygame.draw.polygon(screen, secondaryTableColour, tableRails["BR"]) # Bottom Right Side
888.     pygame.draw.polygon(screen, secondaryTableColour, tableRails["ML"]) # Left Middle Side
889.     pygame.draw.polygon(screen, secondaryTableColour, tableRails["MR"]) # Right Middle Side
890.
891.     # Draw Holes
892.     for i in holePos:
893.         pygame.draw.circle(screen, "black", (holePos[i][0],holePos[i][1]), holeRadius)
894.
895.     # Draws a white line on each of the Rails (where collisions should be done)
896.     for i in tableRails:
897.         for j in range(3):
898.             pygame.draw.line(screen, "white", tableRails[i][j],tableRails[i][j+1])
899.
900. def drawBalls(): # Draws all the balls
901.     for ball in ballGroup.sprites():
902.         ball.draw(screen)
903.
904. def drawBallUI(isSandbox): # Draw 15 balls on screen for UI (NOT ON TABLE)
905.     increment = 0 # adds a value on each time so the balls are not drawn on top of each other
906.     distanceBetweenBalls = tableDimensions[0]/(len(ballInfo)-2)
907.     largerBallRadius = 25
908.
909.     for i in range(len(ballInfo)-1): # loops through each ball except white ball

```

```

910.         circleCentre = (tableCorners[0][0] + increment, tableCorners[2][1]+75) # coordinates of
each circle
911.
912.         pygame.draw.circle(screen, ballInfo[i+1][0], circleCentre, largerBallRadius) # draws outer
circle
913.
914.         if ballInfo[i+1][2]:
915.             pygame.draw.circle(screen, "white", circleCentre, largerBallRadius) # draws outer
circle to overlap stripe
916.             # Draw the stripe
917.             stripe_height = largerBallRadius / 0.65 # height of the stripe
918.             pygame.draw.ellipse(
919.                 screen,
920.                 ballInfo[i+1][0],
921.                 (
922.                     circleCentre[0] - largerBallRadius,
923.                     circleCentre[1] - stripe_height // 2,
924.                     largerBallRadius * 2,
925.                     stripe_height
926.                 )
927.             )
928.
929.             numberFont = ballUIFont.render(str(ballInfo[i+1][1]),True,"black") # renders font
930.             pygame.draw.circle(screen, "white", circleCentre, numberFont.get_height()/2.2) # draws
inner circle
931.             numberFontPos = [circleCentre[0] - numberFont.get_width()/2,circleCentre[1] -
numberFont.get_height()/2]# offsets the font to be central with ball
932.             screen.blit(numberFont,numberFontPos) # adds font to screen
933.
934.
935.
936.             potted = True # set initial case to True
937.             for ball in ballGroup:
938.                 if ball.number == ballInfo[i+1][1]: # If the ball still exists
939.                     potted = False # do not draw X
940.             if potted: # if ball does not exist on table
941.                 xFont = ballPottedFont.render("X",True,"red") # render font
942.                 xFontPos = [circleCentre[0] - xFont.get_width()/2,circleCentre[1] -
xFont.get_height()/2] # centre font
943.                 screen.blit(xFont, xFontPos) # draw ball
944.
945.             increment += distanceBetweenBalls
946.
947.             if not isSandbox: # don't draw this if in sandbox mode
948.                 line1start = (tableCorners[0][0] - largerBallRadius, circleCentre[1] - largerBallRadius -
10)
949.                 line1end = (centre[0] - distanceBetweenBalls + largerBallRadius, circleCentre[1] -
largerBallRadius - 10)
950.
951.                 line2start = (tableCorners[3][0] + largerBallRadius, circleCentre[1] - largerBallRadius -
10)
952.                 line2end = (centre[0] + distanceBetweenBalls - largerBallRadius, circleCentre[1] -
largerBallRadius - 10)
953.
954.                 line3start = (tableCorners[0][0] - largerBallRadius, circleCentre[1] + largerBallRadius +
10)
955.                 line3end = (centre[0] - distanceBetweenBalls + largerBallRadius, circleCentre[1] +
largerBallRadius + 10)
956.
957.                 line4start = (tableCorners[3][0] + largerBallRadius, circleCentre[1] + largerBallRadius +
10)
958.                 line4end = (centre[0] + distanceBetweenBalls - largerBallRadius, circleCentre[1] +
largerBallRadius + 10)

```

```

959.
960.     if player1BallType == "dotted":
961.         pygame.draw.line(screen, player1Colour, line1start, line1end, 5) # draw lines of
player 1
962.         pygame.draw.line(screen, player1Colour, line3start, line3end, 5) # draw lines of
player 1
963.         pygame.draw.line(screen, player2Colour, line2start, line2end, 5) # draw line of player
2
964.         pygame.draw.line(screen, player2Colour, line4start, line4end, 5) # draw line of player
2
965.     elif player1BallType == "striped":
966.         pygame.draw.line(screen, player2Colour, line1start, line1end, 5) # draw lines of
player 2
967.         pygame.draw.line(screen, player2Colour, line3start, line3end, 5) # draw lines of
player 2
968.         pygame.draw.line(screen, player1Colour, line2start, line2end, 5) # draw line of player
1
969.         pygame.draw.line(screen, player1Colour, line4start, line4end, 5) # draw line of player
1
970.
971. def drawLogo(): # Draws the logo
972.     logo = logoFont.render(("BILLIARD CLUB"), True, "white")
973.     logo = pygame.transform.rotate(logo, 270)
974.     screen.blit(logo, (resolution[0] - logo.get_width() - 30, centre[1]-logo.get_height()/2))
975.
976. def drawFPS(): # FPS Display
977.     currentFPS = str(round(clock.get_fps(), 1))
978.     text = fpsFont.render((str(currentFPS)), True, "white")
979.     screen.blit(text, (5, 3))
980.
981. def drawCue(): # draws the cue
982.     mousePos = pygame.mouse.get_pos() # get mouse position
983.     cueVec = (VEC(balls[0].pos) - VEC(mousePos)).normalize() # vector of line
984.     virtualBallPos = VEC(balls[0].pos) # sets initial position of virtual ball
985.
986.     railDistance, closestRail = distanceToClosestRail(virtualBallPos) # updates distance to
closest rail
987.     ballDistance, closestBall = distanceToClosestBall(virtualBallPos, type=None) # updates
distance to closest ball
988.
989.     # rough check for virtual ball collisions
990.     maxIterations = 1000 # limits the number of iterations the initial while loop will do
991.     while railDistance > ballRadius and ballDistance > 2*ballRadius and maxIterations > 0 and (
992.         tableCorners[0][0] + ballRadius <= virtualBallPos[0] <= tableCorners[1][0] -
ballRadius\
993.         and tableCorners[0][1] + ballRadius <= virtualBallPos[1] <= tableCorners[2][1] -
ballRadius
994.     ): # while the ball is not close to a rail (and within the table bounds)
995.         virtualBallPos += ballRadius*cueVec # get closer to rail
996.         railDistance, closestRail = distanceToClosestRail(virtualBallPos) # find new distance
997.         ballDistance, closestBall = distanceToClosestBall(virtualBallPos, type=None)
998.         maxIterations -= 1 # decrement counter
999.
1000.     # if ball collides with other ball, set this to true
1001.     collidingWithBall = False
1002.     if ballDistance < 2 * ballRadius:
1003.         collidingWithBall = True
1004.
1005.     # if ball collides with outer wall first or other ball, don't draw projection.
1006.     drawReflectedProjection = True
1007.     if collidingWithBall or not (tableCorners[0][0] + ballRadius <= virtualBallPos[0] <=
tableCorners[1][0] - ballRadius\

```



```

1008.         and tableCorners[0][1] + ballRadius <= virtualBallPos[1] <= tableCorners[2][1] -
ballRadius):
1009.         drawReflectedProjection = False
1010.
1011.         # finer adjustment for virtual ball collisions
1012.         while railDistance < ballRadius or ballDistance < 2*ballRadius or not (
1013.             tableCorners[0][0] + ballRadius <= virtualBallPos[0] <= tableCorners[1][0] -
ballRadius\
1014.             and tableCorners[0][1] + ballRadius <= virtualBallPos[1] <= tableCorners[2][1] -
ballRadius
1015.         ): # while the ball is colliding with a rail or out of the table bounds
1016.             virtualBallPos -= cueVec # move the ball back by a very small amount
1017.             railDistance, closestRail = distanceToClosestRail(virtualBallPos) # find new distance
1018.             ballDistance, closestBall = distanceToClosestBall(virtualBallPos, type=None)
1019.
1020.         cueColour = "" # set colour of cue
1021.         if player1Turn:
1022.             cueColour = player1Colour
1023.         else:
1024.             cueColour = player2Colour
1025.
1026.         virtualBallColour = "white" # set colour of virtual ball
1027.         if collidingWithBall:
1028.
1029.             if closestBall.number == 8: # if the ball is the 8 ball
1030.                 virtualBallColour = "red" # set colour to black
1031.
1032.             if player1Turn: # if it is player 1's turn
1033.                 if player1BallType == "dotted": # if player 1 type is dotted
1034.                     if closestBall.isStriped: # if the ball is striped
1035.                         virtualBallColour = "red" # set colour to red
1036.                     elif player1BallType == "striped": # if player 1 type is striped
1037.                         if not closestBall.isStriped: # if the ball is dotted
1038.                             virtualBallColour = "red" # set colour to red
1039.             if not player1Turn: # if it is player 2's turn
1040.                 if player2BallType == "dotted": # if player 2 type is dotted
1041.                     if closestBall.isStriped: # if the ball is striped
1042.                         virtualBallColour = "red" # set colour to red
1043.                     elif player2BallType == "striped": # if player 2 type is striped
1044.                         if not closestBall.isStriped: # if the ball is dotted
1045.                             virtualBallColour = "red" # set colour to red
1046.
1047.         pygame.draw.line(screen, cueColour, balls[0].pos, pygame.mouse.get_pos(), 2) # draw cue
1048.         pygame.draw.circle(screen, virtualBallColour, virtualBallPos, ballRadius, 2) # draw virtual
ball
1049.         pygame.draw.line(screen, "white", balls[0].pos, virtualBallPos) # draw projection
1050.
1051.         if collidingWithBall:
1052.             virtualBallVec = (virtualBallPos - closestBall.pos).normalize() # vector of virtual ball
to other ball
1053.             pygame.draw.circle(screen, virtualBallColour, closestBall.pos, ballRadius, 3) # draw
virtual ball
1054.             pygame.draw.line(screen, virtualBallColour, closestBall.pos, (closestBall.pos-
50*virtualBallVec)) # draw reflected projection
1055.
1056.             if drawReflectedProjection: # only draw if ball did not collide with outer wall
1057.                 reflectedCueVec = VEC(cueVec - 2*(VEC.dot(cueVec, closestRail)/VEC.dot(
1058.                     closestRail, closestRail))*closestRail).normalize() # calculated reflected vector
projection
1059.                 pygame.draw.line(screen, virtualBallColour, virtualBallPos, (virtualBallPos-
50*reflectedCueVec)) # draw reflected projection
1060.
1061.             if mouseDown: # when clicked

```



```

1062.         if tableCorners[0][0] <= mousePos[0] <= tableCorners[1][0] \
1063.             and tableCorners[0][1] <= mousePos[1] <= tableCorners[2][1]: # if mouse is on the pool
table
1064.             balls[0].velocity = cueVec*20*ballPowerSlider.returnValue() # give ball a velocity
1065.
ballCollideSound.set_volume(masterVolumeSlider.returnValue()*ballVolumeSlider.returnValue())
1066.             ballCollideSound.play()
1067.
1068. def drawPlayerIndicator(): # Draws the player indicator
1069.     # render font
1070.     player1Text = playerIndicatorFont.render((player1Name), True, player1Colour)
1071.     vsText = playerIndicatorFont.render(("vs"), True, "white")
1072.     player2Text = playerIndicatorFont.render((player2Name), True, player2Colour)
1073.
1074.     player1arrowText = playerIndicatorFont.render(("=>"), True, "white")
1075.     player2arrowText = playerIndicatorFont.render(("<="), True, "white")
1076.
1077.     # draw font to screen
1078.     screen.blit(player1Text, (centre[0] - player1Text.get_width() - vsText.get_width(), 10))
1079.     screen.blit(vsText, (centre[0] - vsText.get_width()/2, 10))
1080.     screen.blit(player2Text, (centre[0] + vsText.get_width(), 10))
1081.
1082.     if player1Turn:
1083.         screen.blit(player1arrowText, (centre[0] - player1Text.get_width() - vsText.get_width() -
player1arrowText.get_width()*1.5, 10))
1084.     else:
1085.         screen.blit(player2arrowText, (centre[0] + player2Text.get_width() + vsText.get_width() +
player2arrowText.get_width()/2, 10))
1086.
1087. def gaussianBlur(screen, radius):
1088.     # uses cv2 to apply a gaussian blur to the screen
1089.     screenArray = np.transpose(pygame.surfarray.array3d(screen), (1, 0, 2)) # Convert screen to
array and transpose axes for cv2
1090.     blurredSurface = np.transpose(cv2.GaussianBlur(screenArray, (radius, radius), 0), (1, 0, 2))
# Apply Gaussian blur and transpose back to original format
1091.     blurredSurface = pygame.surfarray.make_surface(blurredSurface) # Convert blurred array back
into a pygame surface
1092.     screen.blit(blurredSurface, [0, 0]) # Draw the blurred surface onto the screen
1093.
1094.
1095. # Screens
1096. def mainMenuScreen(): # Main loop for the main menu
1097.     running = True
1098.     while running:
1099.         for menuEvent in pygame.event.get(): # Iterate through the events
1100.             if menuEvent.type == pygame.QUIT: # Check if the quit event is triggered
1101.                 running = False # Exit the loop
1102.                 mainMenu.handleEvent(menuEvent) # Check if any button is clicked
1103.
1104.             screen.fill((30, 30, 30)) # Fills the screen with a grey colour
1105.             drawTable() # Draws the table on the screen
1106.             drawBalls() # Draws the balls on the screen
1107.             gaussianBlur(screen, 25) # Apply a gaussian blur to the screen
1108.             drawMenuLogo() # Draws the menu logo on the screen
1109.             mainMenu.draw() # Draw the menu
1110.             mainMenu.update() # Update the menu
1111.             pygame.display.flip() # Update the display
1112.
1113. def gameModeScreen(): # Main loop for the game mode selection screen
1114.     running = True
1115.     while running:
1116.         for menuEvent in pygame.event.get(): # Iterate through the events
1117.             if menuEvent.type == pygame.QUIT: # Check if the quit event is triggered

```

```

1118.         running = False # Exit the loop
1119.         gameModeMenu.handleEvent(menuEvent) # Check if any button is clicked
1120.
1121.         screen.fill((30, 30, 30)) # Fills the screen with a grey colour
1122.         drawTable()
1123.         drawBalls()
1124.         gaussianBlur(screen, 25) # Apply a gaussian blur to the screen
1125.         drawMenuLogo()
1126.         gameModeMenu.draw() # Draw the menu
1127.         gameModeMenu.update() # Update the menu
1128.         pygame.display.flip() # Update the display
1129.
1130. def optionsScreen(): # Main loop for the options screen
1131.     running = True
1132.     while running:
1133.         for menuEvent in pygame.event.get(): # Iterate through the events
1134.             if menuEvent.type == pygame.QUIT: # Check if the quit event is triggered
1135.                 running = False # Exit the loop
1136.                 optionsMenu.handleEvent(menuEvent) # Check if any button is clicked
1137.                 screen.fill((30, 30, 30)) # Fills the screen with a grey colour
1138.                 drawTable()
1139.                 drawBalls()
1140.                 gaussianBlur(screen, 25) # Apply a gaussian blur to the screen
1141.                 drawMenuLogo()
1142.                 optionsMenu.draw() # Draw the menu
1143.                 optionsMenu.update() # Update the menu
1144.                 pygame.display.flip() # Update the display
1145.
1146. def customisationScreen(): # Main loop for the customisation screen
1147.     global player1Name
1148.     global player2Name
1149.     global player1Colour
1150.     global player2Colour
1151.
1152.     if player2Name == "Computer":
1153.         player2Name == "Player 2"
1154.
1155.     TextBox1 = TextBox(70, 175, "Player 1 Name: ", player1Name, 30)
1156.     TextBox3 = TextBox(70, 250, "Player 1 Colour: ", "", 30)
1157.     TextBox2 = TextBox(70, 325, "Player 2 Name: ", player2Name, 30)
1158.     TextBox4 = TextBox(70, 400, "Player 2 Colour: ", "", 30)
1159.     colourSlider1 = HorizontalSlider(390, 265, 260, 20, 0, "orange", "black", "white",
"colourPicker")
1160.     colourSlider2 = HorizontalSlider(390, 415, 260, 20, 0.7, "orange", "black", "white",
"colourPicker")
1161.
1162.     running = True
1163.     while running:
1164.         for menuEvent in pygame.event.get(): # Iterate through the events
1165.             if menuEvent.type == pygame.QUIT: # Check if the quit event is triggered
1166.                 running = False # Exit the loop
1167.                 customisationMenu.handleEvent(menuEvent) # Check if any button is clicked
1168.                 TextBox1.handle_event(menuEvent)
1169.                 TextBox2.handle_event(menuEvent)
1170.                 colourSlider1.handleEvent(menuEvent)
1171.                 colourSlider2.handleEvent(menuEvent)
1172.
1173.         screen.fill((30, 30, 30)) # Fills the screen with a grey colour
1174.         drawTable()
1175.         drawBalls()
1176.         gaussianBlur(screen, 25) # Apply a gaussian blur to the screen
1177.         drawMenuLogo()
1178.

```

```

1179.     customisationMenu.draw() # Draw the menu
1180.     customisationMenu.update() # Update the menu
1181.
1182.     TextBox1.draw() # Draw the first text box
1183.     TextBox2.draw() # Draw the second text box
1184.
1185.     # Not interactive Text boxes
1186.     TextBox3.draw() # Draw the first text box
1187.     TextBox4.draw() # Draw the second text box
1188.
1189.     player1Name = TextBox1.text # Update player 1 name with the text from the first text box
1190.     player2Name = TextBox2.text # Update player 2 name with the text from the second text box
1191.
1192.     colourSlider1.update()
1193.     colourSlider1.draw()
1194.
1195.     colourSlider2.update()
1196.     colourSlider2.draw()
1197.
1198.     player1Colour = colourSlider1.getColour()
1199.     player2Colour = colourSlider2.getColour()
1200.
1201.
1202.
1203.     pygame.display.flip() # Update the display
1204.
1205. def pauseScreen(): # Main loop for the pause screen
1206.     running = True
1207.     while running:
1208.         for event in pygame.event.get(): # Iterate through the events
1209.             if event.type == pygame.QUIT: # Check if the quit event is triggered
1210.                 running = False # Exit the loop
1211.             if event.type == pygame.KEYDOWN: # Check if a key is released
1212.                 if event.key == pygame.K_ESCAPE: # Check if the key is ESCAPE
1213.                     running = False # Exit the loop
1214.                 pauseMenu.handleEvent(event)
1215.
1216.             screen.fill((30, 30, 30)) # Fills the screen with a grey colour
1217.             drawTable() # Draw the table
1218.             drawBalls() # Draw the balls
1219.             gaussianBlur(screen, 25) # Apply a gaussian blur to the screen
1220.             drawMenuLogo() # Draw the Logo for the menu screen
1221.
1222.             pauseMenu.draw() # draw the menu
1223.             pauseMenu.update() # Update the menu
1224.             pygame.display.flip() # Update the display
1225.
1226. def volumeControlScreen(): # Main loop for the volume control screen
1227.     # Initialise Text Boxes
1228.     TextBox1 = TextBox(70, 230, "Master Volume", "", 30)
1229.     TextBox2 = TextBox(70, 330, "Music Volume", "", 30)
1230.     TextBox3 = TextBox(70, 430, "Ball Sounds", "", 30)
1231.
1232.     running = True
1233.     while running:
1234.         for event in pygame.event.get(): # Iterate through the events
1235.             if event.type == pygame.QUIT: # Check if the quit event is triggered
1236.                 running = False # Exit the loop
1237.
1238.             volumeMenu.handleEvent(event)
1239.             masterVolumeSlider.handleEvent(event)
1240.             musicVolumeSlider.handleEvent(event)
1241.             ballVolumeSlider.handleEvent(event)

```

```

1242.
1243.
1244.     screen.fill((30, 30, 30)) # Fills the screen with a grey colour
1245.     drawTable() # Draw the table
1246.     drawBalls() # Draw the balls
1247.     gaussianBlur(screen, 25) # Apply a gaussian blur to the screen
1248.     drawMenuLogo() # Draw the Logo for the menu screen
1249.
1250.     masterVolumeSlider.update() # Update master volume slider
1251.     masterVolumeSlider.draw() # Draw master volume slider
1252.
1253.     musicVolumeSlider.update() # Update music volume slider
1254.     musicVolumeSlider.draw() # Draw music volume slider
1255.
1256.     ballVolumeSlider.update() # Update ball volume slider
1257.     ballVolumeSlider.draw() # Draw ball volume slider
1258.
1259.     TextBox1.draw() # Draw the Master Volume text box
1260.     TextBox2.draw() # Draw the Music Volume text box
1261.     TextBox3.draw() # Draw the Ball sounds text box
1262.
1263.     volumeMenu.draw() # draw the menu
1264.     volumeMenu.update() # Update the menu
1265.
1266.     # Updates the volumes of the Sound
1267.     clickSound.set_volume(masterVolumeSlider.returnValue()*ballVolumeSlider.returnValue())
1268.     ballCollideSound.set_volume(masterVolumeSlider.returnValue()*ballVolumeSlider.returnValue())
1269.     ballPottedSound.set_volume(masterVolumeSlider.returnValue()*ballVolumeSlider.returnValue())
1270.
1271.     # Updates Background Music Volume
1272.
1273.     pygame.mixer.music.set_volume(masterVolumeSlider.returnValue()*musicVolumeSlider.returnValue())
1274.     pygame.display.flip() # Update the display
1275.
1276. def winScreen(winner, colour): # Main loop for the win screen
1277.     running = True
1278.     while running:
1279.         for event in pygame.event.get(): # Iterate through the events
1280.             if event.type == pygame.QUIT: # Check if the quit event is triggered
1281.                 running = False # Exit the loop
1282.             if event.type == pygame.KEYDOWN: # Check if a key is released
1283.                 if event.key == pygame.K_ESCAPE: # Check if the key is ESCAPE
1284.                     running = False # Exit the loop
1285.                 winMenu.handleEvent(event)
1286.
1287.             screen.fill((30, 30, 30)) # Fills the screen with a grey colour
1288.             drawTable() # Draw the table
1289.             drawBalls() # Draw the balls
1290.             gaussianBlur(screen, 25) # Apply a gaussian blur to the screen
1291.             drawMenuLogo() # Draw the Logo for the menu screen
1292.
1293.             winText = logoFont.render(winner + " wins!", True, colour) # renders the font
1294.             winOutlineText = logoFont.render(winner + " wins!", True, "black") # renders the font
1295.
1296.             winTextPos = [centre[0] - winText.get_width(), centre[1] - winText.get_height()/2] #
offsets the font to be central
1297.             # Draw the outline of the logo
1298.             for i in range(-3,4):
1299.                 for j in range(-3,4):
1300.                     screen.blit(winOutlineText, (winTextPos[0]+i,winTextPos[1]+j))

```

```

1301.
1302.         screen.blit(winText,winTextPos) # adds font to screen
1303.
1304.         winMenu.draw() # draw the menu
1305.         winMenu.update() # Update the menu
1306.         pygame.display.flip() # Update the display
1307.
1308. # Run game
1309. def runGame(computerPlaying, sandbox): # Function to start a new game
1310.     newGame(computerPlaying, sandbox, player1Name, player2Name, player1Colour, player2Colour)
1311.
1312. def newGame(computerPlaying, sandbox, player1Nametemp, player2Nametemp, player1Colourtemp,
player2Colourtemp):
1313.     global gameQueue
1314.     global pottedBalls
1315.     global roundPottedBalls
1316.     global mouseDown
1317.
1318.     global player1Name
1319.     global player2Name
1320.     global player1Colour
1321.     global player2Colour
1322.
1323.     global winner
1324.     global player1BallType
1325.     global player2BallType
1326.
1327.     player1Name = player1Nametemp
1328.     player2Name = player2Nametemp
1329.     player1Colour = player1Colourtemp
1330.     player2Colour = player2Colourtemp
1331.
1332.     ballPowerSlider.updateColour(player1Colour) # update the colour of the power slider
1333.
1334.     if computerPlaying:
1335.         player2Name = "Computer" # Set player2Name to 'Computer' if computer is playing
1336.     else:
1337.         if player2Name == "Computer":
1338.             player2Name = "Player 2" # Change name to 'Player 2' if it was previously 'Computer'
1339.
1340.     gameQueue = [] # Queue/List for storing events in between each turn.
1341.     pottedBalls = [] # List for storing potted balls in whole game
1342.     roundPottedBalls = [] # List for storing potted balls in each round
1343.
1344.     winner = None # so there is no winner
1345.     player1BallType = None # Reset player 1 ball type
1346.     player2BallType = None # Reset player 2 ball type
1347.
1348.     deleteAllBalls() # Deletes all balls
1349.     initialiseBalls() # Then Initialise all balls to ensure all balls on table
1350.
1351. # Game Loop
1352. while True:
1353.     mouseDown = False
1354.     for event in pygame.event.get():
1355.         if event.type == pygame.QUIT: # Checks if user closes program
1356.             pygame.quit()
1357.             exit()
1358.
1359.         if event.type == pygame.MOUSEBUTTONDOWN: # on left click
1360.             mouseDown = True
1361.
1362.         if event.type == pygame.KEYDOWN:# Check if any key is pressed

```

```

1363.         if event.key == pygame.K_ESCAPE: # Check if the key is ESCAPE
1364.             pauseScreen() # Call the pause screen function
1365.
1366.         ballPowerSlider.update(event) # Update Slider
1367.         ballGroup.update() # update each ball
1368.
1369.         resolveAllCollisions() # Checks and resolves every collision between every pair of balls
1370.         resolveWallCollisions() # Checks and resolves every collision between every ball and the
1371.         walls
1372.         resolveBallPotted() # checks if a ball is potted and deletes it if it is.
1373.
1374.         # Draw Everything to Screen:
1375.
1376.         screen.fill((30, 30, 30)) # Fills the screen with a grey colour
1377.
1378.         drawTable() # Draws the table
1379.         drawBalls() # draws the balls
1380.
1381.         if not checkIfBallMoving(): # when all the balls are still
1382.             # execute all functions/procedures in the queue
1383.             for function in gameQueue:
1384.                 function() # execute the event
1385.             gameQueue.clear() # clear the queue
1386.
1387.             if computerPlaying and not player1Turn: # if computer is playing
1388.                 computerShoot() # computer finds shot and takes shot
1389.             else: # if computer is not playing
1390.                 drawCue() # draw the cue and give velocity to the white ball if clicked
1391.
1392.             # if the ball has been hit
1393.             if checkIfBallMoving() and not sandbox:
1394.                 gameQueue.append(gameLogic) # when the balls are next still, apply game logic to
1395.                 determine the next round.
1396.
1397.         ballPowerSlider.draw(screen) # Draws slider
1398.
1399.         drawBallUI(sandbox) # draws the ball UI (15 balls that tells the user which ball to pot
1400.         next)
1401.         drawLogo() # draws the logo
1402.         drawFPS() # draws the FPS
1403.
1404.         if not sandbox:
1405.             drawPlayerIndicator() # draws the player indicator
1406.             checkWin(winner) # checks if there is a winner and draws it to the screen
1407.
1408.         pygame.display.flip() # Updates the display
1409.         clock.tick(FPS) # Caps the frame rate
1410.
1411.     mainMenuScreen()

```