North Creek High School

Advanced Programming Topics Final Project
Design Document

# 8-Ball Pool

Mohamed Morsy
Gautham Manigantan

| Date | Revision | Release Notes |
|---|---|---|
| 3/21/2025 | Rev 01 | Initial Release |
| 3/24/2025 | Rev 02 | Sketches added |
| 3/31/2025 | Rev 03 | Adjustments made (feedback given) |
| 4/4/2025 | Rev 04 | High-Level Architecture & Testing |

# Contents

**1 Project Description** **2**

**2 What is Pool/Billiards?** **2**

**3 Minimum Viable Product** **3**

**4 Purpose & Motivation** **4**

**5 Core Features** **4**

**6 Additional or "Stretch" Features[1]** **5**

**7 Sweeping Algorithm** **5**
7.1 Algorithm Pseudocode . . . . . 11
7.2 Algorithm Alternatives . . . . . 11

**8 Storyboard Sketches** **12**
8.1 Schematics Design . . . . . 15

**9 Learning Targets & Challenge Goals** **15**

**10 Testing Strategy** **16**

**11 Class Roles & Responsibilities** **16**

**12 Class Diagrams** **16**
12.1 Class Diagram . . . . . 17
12.2 Sequence Diagram . . . . . 17
12.3 Flow Chart Diagram / AI Algorithm Flowchart . . . . . 17

**A Full Algorithm Pseudocode** **19**

[1]Features are ranked by difficulty, top being the most difficult, and bottom being the least.

# 1 Project Description

Our project is an 8-ball pool game similar to the one on Game Pigeon. This game requires a lot of strategical thinking and calculations to win. This project is made purely using Java and uses Swing for the visuals. The user will be able to interact with the game using mouse clicks and drags, which will result in the cue stick hitting the cue ball. Swing will be used to animate the movement of the balls and the intensity of the shot meter. For more detail on how the game of billiards works, see **Section 2**.

# 2 What is Pool/Billiards?

Billiards, or "pool", as many know it, is a cue sport where there is a table with six pockets. There are 15 balls numbered 1 through 15, and 1 cue ball that's completely white. The 8-ball, a completely black ball, is in the center of the rack, which is a triangle that holds the balls at the start. The yellow 1-ball is put in the front of the rack, a random solid in one corner, and a random stripe in the other. The rest of the object balls' (non-cue balls) order doesn't matter. The rack is lined up at one end of the table, where the 1 ball (the front of the rack) is lined up with the foot spot. The foot spot is the spot where the 3 lines shown in **Figure 1** intersect. The rack is lifted, and the first player then performs the break shot, which is where they hit the cue ball from behind the headstring, a line that's similar to the one going through the footspot, but on the other side. For the break shot to be legal, at least four balls must hit the sides of the table, or a ball must be pocketed. If the 8-ball is sunk on the break, the player can re-rack (restart) or is spotted, where the 8-ball is placed on the footspot. If the first player misses, the second player attempts the shot to sink a ball, and the first to legally pocket a ball chooses to be solids (1-7) or stripes (9-15). A player will continue to keep shooting as long as they legally pocket their assigned ball type, but if they miss or do a foul, the opponent gets a turn. Fouls that are common and will be detected by our game include the following:

- Failing to hit a ball
- Hitting the cue ball off of the table [2]
- Pocketing the cue ball [3]
- Hitting the opponents' group of balls first

[2] A "scratch" means the cue ball goes into a pocket or jumps off the table

[3] This is an example of a "scratch"

- The cue ball not hitting a table side or pocketing a ball after contact

If a foul is detected, then the opponent receives "ball-in-hand", where they can place the cue ball anywhere on the table that is behind the head string (outside of the table is prevented). In order to win the game, a player must sink all of their assigned balls and then legally pocket the 8-ball at the end to win. However, if a scratch[2] occurs, then the opponent would receive ball-in-hand or the opponent wins if the scratch occurs while shooting at the 8-ball.

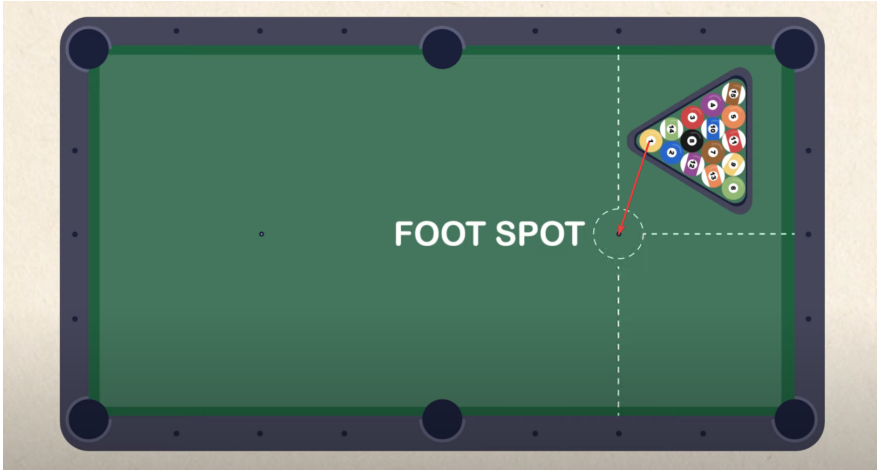



Figure 1: The "foot spot" on a billiards table. Source: wikiHow on YouTube.

# 3 Minimum Viable Product

Our M.V.P. follows these requirements:

- A basic top-down two-dimensional SwingUI-based Java game
- User input handling (keyboard shortcuts [See **Table 1**] and mouse click processing)
- A menu screen and a game screen, game over screen, and help/instructions screen.
- The user can hold down and drag back a cue stick and let go to hit the ball
- The ball goes in the direction the cue stick is facing
- Functioning buttons to allow traversal between screens
- Background music
- Path of ball is shown with a basic graphic (shown in Section 8)

# 4 Purpose & Motivation

The purpose of the game is to help people enjoy time together playing a game they like. Many people can't afford to buy something like a pool table, and many online versions of pool aren't available to easily play with friends other than the one on **Game Pigeon ??**[4]. The problem with Game Pigeon is that it only works on iOS so this means that anyone who has an Android phone can't enjoy the game online with their friends. We want to solve this problem by making a 2 player 8-ball pool game that can be played anywhere offline with just 2 people. The motivation for this project was that we both really enjoy playing pool in person and online in Game Pigeon. We are also pretty interested in physics, so this allows us to combine our interests and put it into one game. Pool is also a very strategical game that is fun and simple at the same time so it is one of the most doable games in the short time frame.


# 5 Core Features

- Interactive & animated buttons that respond to clicks and make sound
- Accurate physical responses to ball collisions and hitting the ball with a cue
- Sound effects and music
- High quality graphics and user interface that is easily navigable and not distracting

[4]Game Pigeon is an iOS mobile app that allows users to play mini games with each other through Message

- Dynamic and high quality animations for the objects in the game

One important thing to mention is how keystrokes will be mapped. They will work under the following table:

| Key | Action |
|---|---|
| F11 | Screenshot |
| P | Pause Game |
| Escape | Pause Game |
| Q | Quit Game |
| R | Restart Game |
| D | Activate Debug Mode |

Table 1: Table of Various Key Mappings

# 6 Additional or "Stretch" Features[5]

1. Custom algorithm-based AI mode that responds to user moves
2. Different AI "bots" to play against that play with different difficulties?
3. Game replays (moves are saved and replayed in game upon user response)
4. Tutorial mode that teaches the user how to play step-by-step (rather than an instructions page)
5. Statistics screen at the end with detailed information

# 7 Sweeping Algorithm

If we have time, we plan to implement a sweeping algorithm for the game AI. The reason why we will use this algorithm is alternative methods, such as just always playing random moves, or brute force strategies are inefficient or are not actually challenging/playable for the user. The idea is that it's similar to a Bounding Volume Hierarchy, wherein the playing area is split into separate areas and each section is evaluated based on four categories of

[5]Features are ranked by difficulty, top being the most difficult, and bottom being the least.

balls. A diagram is shown below, displaying the split up sections: A-F, and the four types of balls are in the top right corner. **S** is for striped, **N** is for non-striped, **C** is for cue ball, and **8** is for 8-ball. There are six circles indicating pocket locations on the rectangle that represents the billiard table. There are two line intersections that represent either footspot.

The steps for how the algorithm functions are as follows:

1. Each separate area labeled A-F will be checked for the cue ball first. We set this found section as our Area of Interest (AoI). This will be checked by seeing if the coordinates of the cue ball are within the specified region. A visualization of this separation of regions can be shown in **Figure 2**.

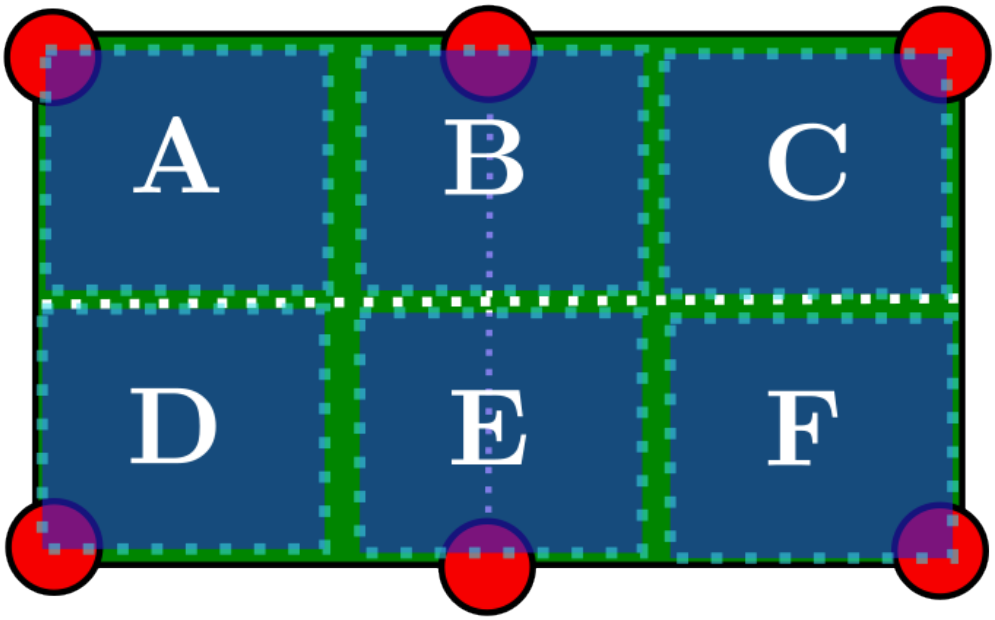



Figure 2: A visualization of the billiards table, with separate labeled regions A-F, and 6 pockets.

2. Then we "sweep" the area with a ray as shown in **Figure 3**, by about $\sim 1°$ at a time, checking to see if we've hit the cue ball[6]. The ray originates from the center of the

[6]To address potential issues with whether or not the sweep will work, if the cue ball is at the center, then the sweep will circle around the ball, as when a path is first chosen, skipping the first step.

AoI, and points outwards up until the edge of the square (unless it hits a ball). The magnitude of the ray increases slowly in order to account for a closer/farther cue ball

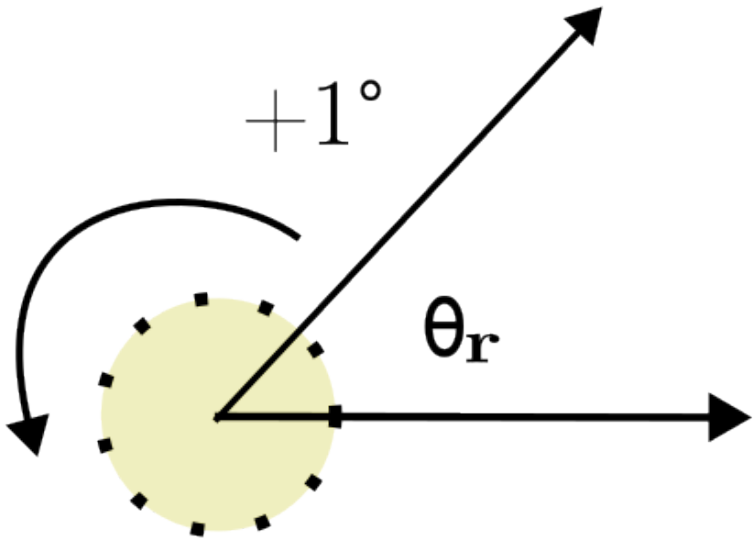



Figure 3: The sweeping motion

3. Once that is complete, a second rotation is performed around the ball, beginning the process of choosing a path. To find a path, we continue in the same direction of the ray until we find a ball of the AI's type (let's say Stripes), and if not, until a pocket is found. If still not, alternate pathways are considered, which is explained further down the line. This process repeats with each subsequent ball, where the ray continues in the same direction as the reflection in a recursive process, where the exit condition is the force dampening factor has hit below 0.05 (our significance level). This is all for *one* path, mind you.

4. The reference angle from a line going right from the ball is saved for both the cue ball and next ball(s). The first angle is the shooting angle, denoted by $\theta_s$, and the rest are reference angles by a 0-th order, which we'll call $\theta_n$, where $n \in \mathbb{Z}, n \geq 0$ and $n$ represents any positive integer value that is equal to or greater than 0. The reference angles will be saved in a Stack.

5. Two other factors are considered: a force dampening factor, and alternate paths:

    - The force dampening is a constant that represents how much more weakened the force will become as more collisions, friction, and distance traveled come into play (air resistance is negligible for our purposes). We define our net force equation

on the ball using this dampening factor as shown:

$$\Sigma F = \frac{(F_s - F_f)}{d_f} \tag{1}$$

$$F_f = F_N \cdot C_{rr} \tag{2}$$

$$F_N = m \cdot g \tag{3}$$

$$D = \sqrt{(x_{b2} - x_{b1})^2 + (y_{b2} - y_{b1})^2} \tag{4}$$

$$d_f = \frac{D}{(C_{rr} \cdot r_{b1})} \tag{5}$$

$$\tag{6}$$

- $\Sigma F$ represents the magnitude of the net force on the ball, $F_s$ is the "shooting" force applied by the user at some intensity, $F_f$ is the friction force, $C_{rr}$ is the coefficient of rolling friction of the billiard table, $F_N$ is the normal force, $d_f$ is the dampening factor, $m$ is the mass of the ball, $D$ represents the distance between the two balls using the distance formula, where $b2$ represents the second ball and $b1$ is the first, and $r_{b1}$ is the radius of the first ball. $F_D$ the drag force on the cue ball, where $C_d$ is the drag coefficient, which is about 0.47 for a sphere, $\rho$ is the air density at sea level, $A$ is the cross-sectional area of the ball, and $v$ is the velocity of the ball. (Note: $d_f$ must be between 0.0 and 1.0, and cannot be negative). To reference all of these values, here's a Reference Table:

| **Quantity** | **Symbol** | **Value (Units)** |
|---|---|---|
| Gravitational Acceleration | $g$ | 9.81 m/s$^2$ |
| Rolling Friction Coefficient | $C_{rr}$ | 0.01 - 0.03 (dimensionless) |
| Radius of Cue Ball | $r_{b1}$ | 0.05715 m |
| Mass of Cue Ball | $m$ | 0.17 kg |

Table 2: Reference Table of Physical Values Used in Equations Above

- A key point to mention is this is just the magnitude, and doesn't consider direction. Direction is determined using the Law of Reflection, wherein:

$$\theta_i = \theta_r \tag{7}$$

- $\theta_i$ represents the incident angle, or angle shot towards the ball, and $\theta_r$ is the reflected angle. They are congruent if measured from the normal, so therefore, for our purposes:

$$\theta_r = 360^\circ - \theta_i \tag{8}$$

Where $\theta_i$ is measured as a reference angle just like $\theta_r$.

- As for the alternate pathways, lets say we recursed and could not find a ball to bounce off of or a pocket. Alternatives are then considered, such as hitting off of the wall and reflecting back, and then shooting a ray until we hit a ball of our type or a pocket. Maybe double reflections are considered. If there are no possible moves, the AI will choose a random shooting angle between 0 and 360 degrees.

6. A check is then performed, where now that the move is a possibility, we count the amount of pockets that will result from this. This will require, when evaluating a collision, checking if the collided ball will follow a pathway that leads to a pocket. The goal is to find a move with maximum pockets. If all balls have been pocketed, then the first step is changed, where now the AI is trying to find a section with the 8-ball and cue-ball, and attempts to make a final pocket (this is an exception to the typical algorithm).

7. Finally, we repeat step 3, where we rotate once again around the ball, avoiding any other paths that have been evaluated, and count pockets for each. Each possible pathway is considered a branch in an unbalanced tree (See **Figure 4**). The optimal path is chosen by the branch with the most pockets (saved as the value in each node). The leaf nodes represent the final move in the total moveset, and if one were to traverse the tree, it would essentially be a replay of all the AI's move (may be used as a replay feature).

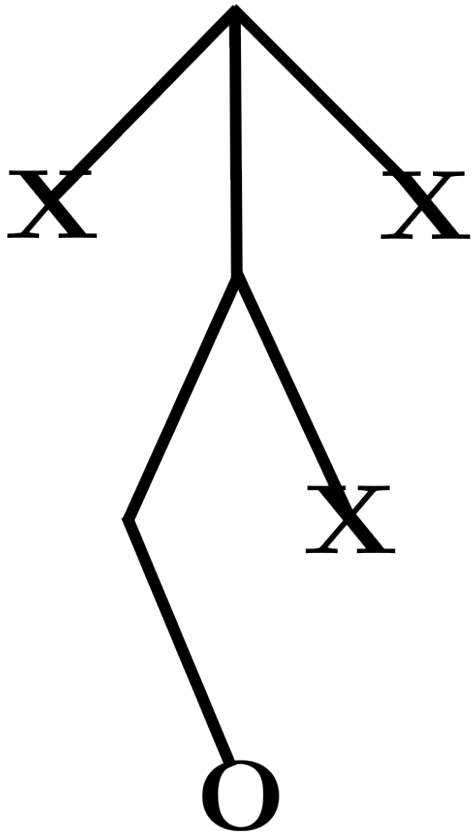



Figure 4: An unbalanced tree that is scaled down to represent the various pathways the AI could take. The X's are cut off points, and the O represents the final leaf node where we have chosen a path.

## 7.1 Algorithm Pseudocode

Overall, the way the algorithm works is a few simple steps:

1. Find the area of interest in the table that will give the AI the most amount of its ball type
2. Find the cue ball in the AoI
3. Continously draw rays at different angles until we find a ball of the AI's type, and use that as a potential pathway
4. We consider alternate solutions like reflections if that doesn't work
5. We create more than one pathway, and the one with most successful pockets is chosen as the best move
6. We then draw the moves out, and hand it over to the player at the end

For a more detailed explanation, see **Section A**. You can also see **Section 12.3** for a flowchart diagram representation of the algorithm.

## 7.2 Algorithm Alternatives

There are some key components to mention when evaluating the effectiveness of our algorithm, such as efficiency, potential bottlenecks, or alternative approaches. We already mentioned that alternative approaches like brute force are inefficient and unplayable, but in terms of efficiency, our algorithm approximately follows a cubic time complexity, or $O(n^3)$.

The calculations for this are as follows:

$$O(p, s, s_o, \theta, \theta_i) = p(s + \theta + s_o(\theta_i)) + p \tag{9}$$

$$O(p, s_o, \theta_i) = ps_o\theta_i \tag{10}$$

$$O(n^3) = O(p) \cdot O(s_o) \cdot O(\theta_i) \tag{11}$$

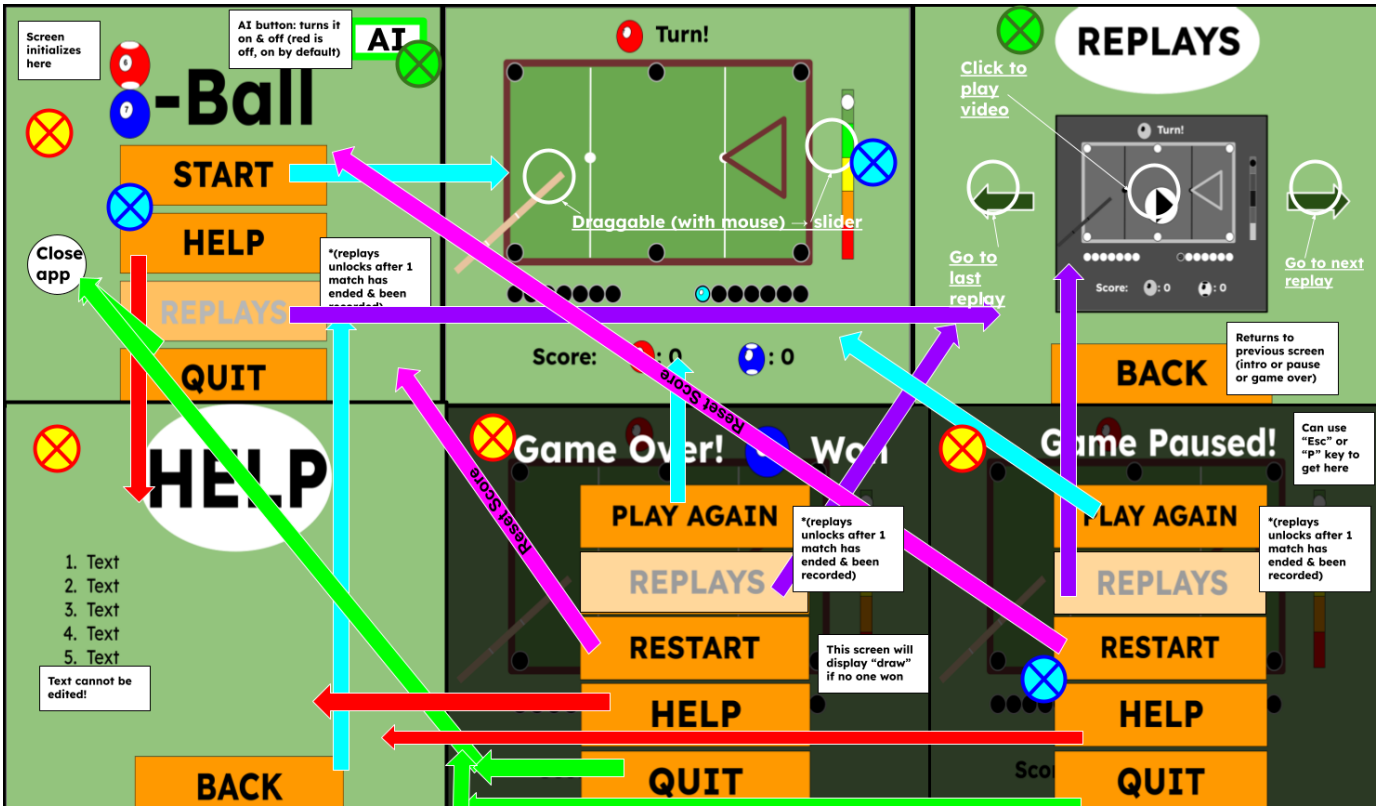
$$\tag{12}$$

Where $p$ represents the number of pathways, $s$ is the number of sections the AI had to choose between to find the AoI, $s_o$ is the number of successing shots, $\theta$ is the total number of reference angles chosen in the end, and $\theta_i$ is the total iterated number of individual hit angles that had to be processed to choose the pathway. As $s$ is constant, $\theta$ is non-dominant (being far less than the other terms), and the combined multiplication of the pathways, individual hit angles, and number of successing shots has the biggest influence on the number of operations needed, an $O(n^3)$ time complexity is the outcome of efficiency. A cubic time complexity is slow, but for a complex process such as this, other methods would be far worse and more inefficient (such as the brute force approach of just going through every possible move).

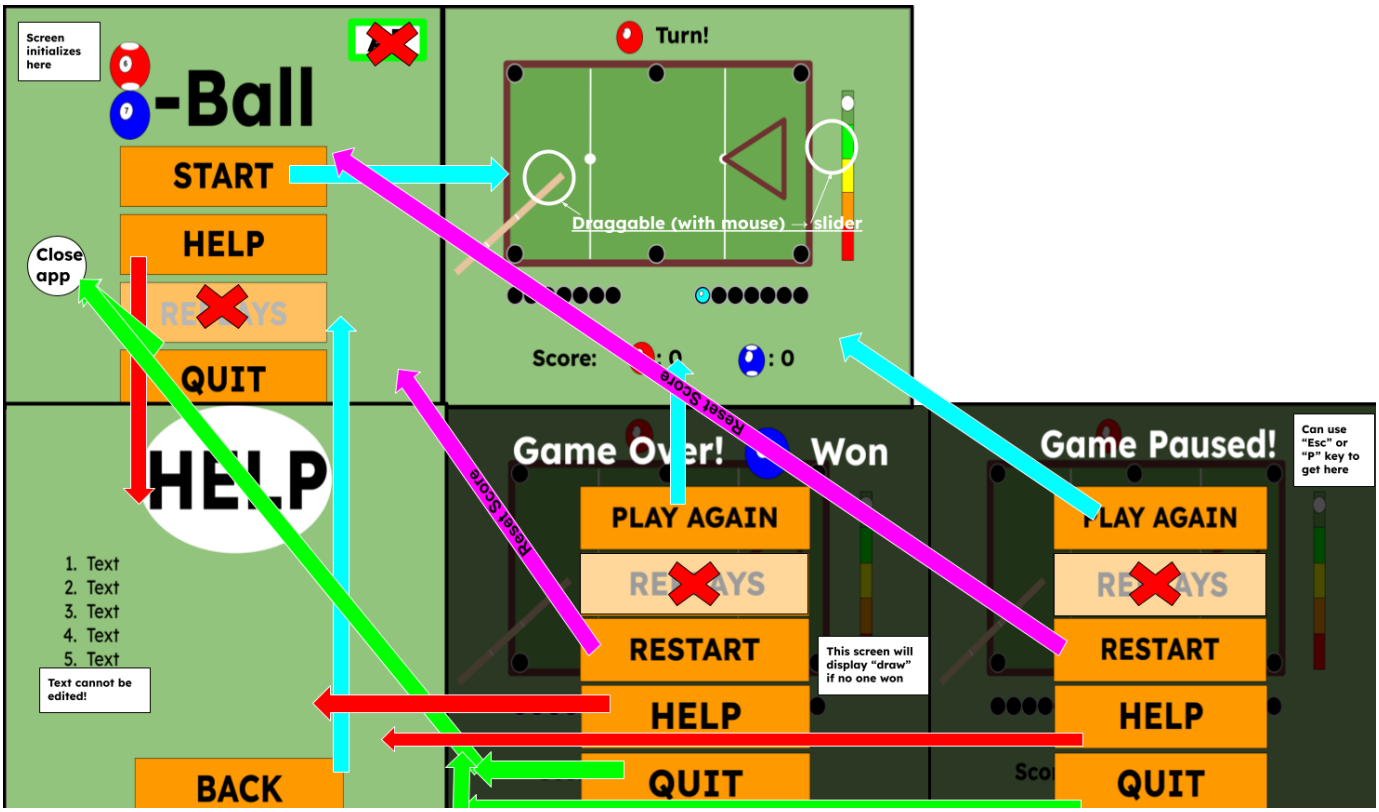
As for potential bottlenecks, we should mention the worst and best case scenario for the algorithm. The best case scenario would be getting minimal number of pathways $p$ and minimal iterated hit angles, meaning that the AI found a shot essentially at or near the first ray cast. The worst case would be a high number of pathways $p$, and shots being found at very high angles (meaning a high iteration count). We ideally want a game with minimal possible pathways with balls of the AI's type being relatively close.

# 8 Storyboard Sketches

Here is a basic game screen flowchart indicating a rough idea of how the GUI (Graphical User Interface) for our game will look like from screen to screen, with a legend beneath (mouse interactions are shown with arrows, but for keystroke processing, see **Table 1**):




We also have a diagram of just the M.V.P. & Core screens:




## LEGEND

= game/intro screen

= replays

= score reset

= help screen

= quit

= helpful note

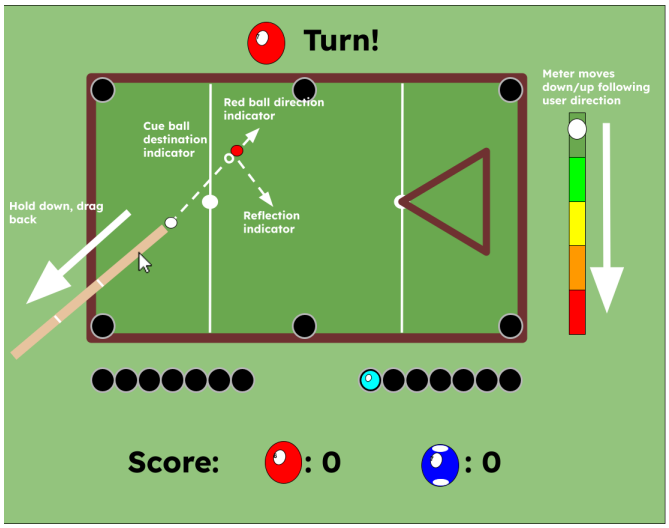
= special graphic/interactable

= MVP

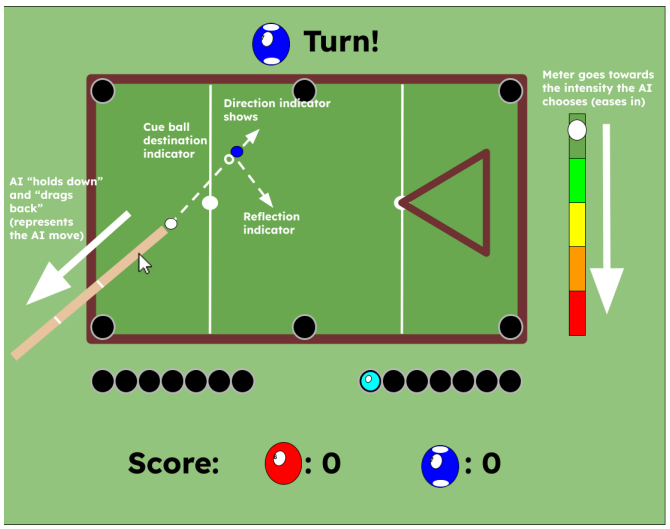
= Core

= Stretch

And here is a diagram of how the player will actually be able to interact with the game (mouse clicks, dragging, etc): (for more information on key presses, see **Table 1**)



Additionally, we provide a diagram for how the AI's response will look like in-game from the user's perspective (with debug lines included). **Debug Mode** is a feature that can be activated by running the game and pressing the D key. The diagram is below:




## 8.1 Schematics Design

We used Google Draw to plan out how our schematics will look like. We have a link to the Google Drive link containing all of our schematics here. This will provide a more detailed view that's easier to read (images of the schematics needed to be dramatically scaled down to fit in the pdf format).

# 9 Learning Targets & Challenge Goals

Data Structures that will be used are a Stack for the reference angles, a Tree for the AI pathway algorithm, an ArrayList to save all balls on the screen currently, all balls for both the AI and Human that are pocketed or not, etc. We intend to use Java SwingUI, the Java Sound API, Canva, Visual Studio Code, Overleaf, JummBox, GitHub, and other technologies to design and create our project. We hope to successfully implement a working menu screen, options/settings, AI debugging & ray-tracing tools, as well as using the Java Preferences API to save user and game data, and JavaFX Media with the Robot API for audio and video recording (for replays), and to screenshot in the game.

The main things we hope to learn from this experience include:

1. How to successfully plan and design a high-level project from start to finish
2. Learn how to effectively coordinate and work collaboratively on a project, in an attempt to improve from our previous team project
3. Learn how to not only build a product, but build useful developer tools (as in, we make a game, and the tools to debug the game)

# 10 Testing Strategy

We will be using Java's built-in JUnit testing architecture to write tests for each major aspect of our code, ensuring each portion works independently before the final product is added together. We have three separate testing classes that we will use to test individual components of our application.

# 11 Class Roles & Responsibilities

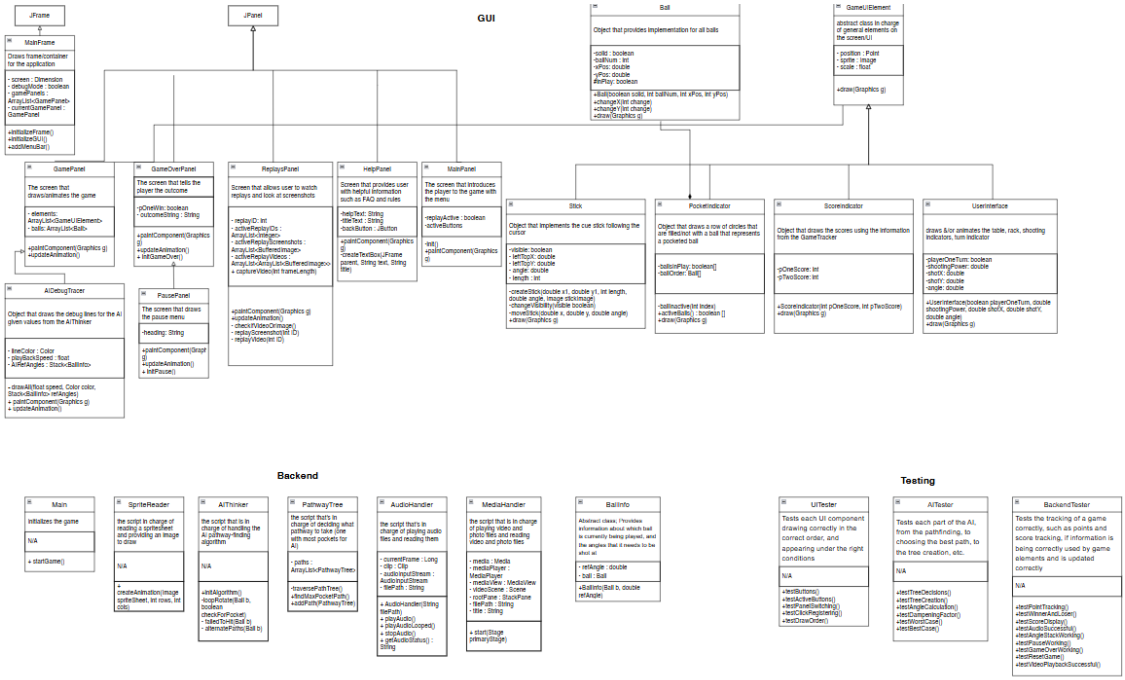
Table 3: Class Roles and Responsibilities

| Class | Roles | Responsibilities | Description |
|---|---|---|---|
| MainFrame | IS-A JFrame | Check the class diagram | |
| GamePanel | IS-A JPanel | Check the class diagram | |
| GameOverPanel | IS-A JPanel | Check the class diagram | |
| PausePanel | IS-A GameOverPanel | Check the class diagram | |
| ReplaysPanel | IS-A JPanel | Check the class diagram | |
| HelpPanel | IS-A JPanel | Check the class diagram | |
| MainPanel | IS-A JPanel | Check the class diagram | |
| Ball | Standalone class | Check the class diagram | |
| GameElement | Standalone class | Check the class diagram | |
| Stick | IS-A GameElement | Check the class diagram | |
| PocketIndicator | IS-A GameElement | Check the class diagram | |
| ScoreIndicator | IS-A GameElement | Check the class diagram | |
| UserInterface | IS-A GameElement | Check the class diagram | |
| AIDebugTracer | IS-A GamePanel | Check the class diagram | |
| Stick | IS-A GameElement | Check the class diagram | |
| BallInfo | Standalone class | Check the class diagram | |
| Main | Standalone class | Check the class diagram | |
| Sprite | Standalone class | Check the class diagram | |
| AIThinker | Standalone class | Check the class diagram | |
| PathwayTree | Standalone class | Check the class diagram | |
| AudioHandler | Standalone class | Check the class diagram | |
| MediaHandler | Standalone class | Check the class diagram | |
| UITester | Standalone class | Check the class diagram | |
| AITester | Standalone class | Check the class diagram | |
| BackendTester | Standalone class | Check the class diagram | |

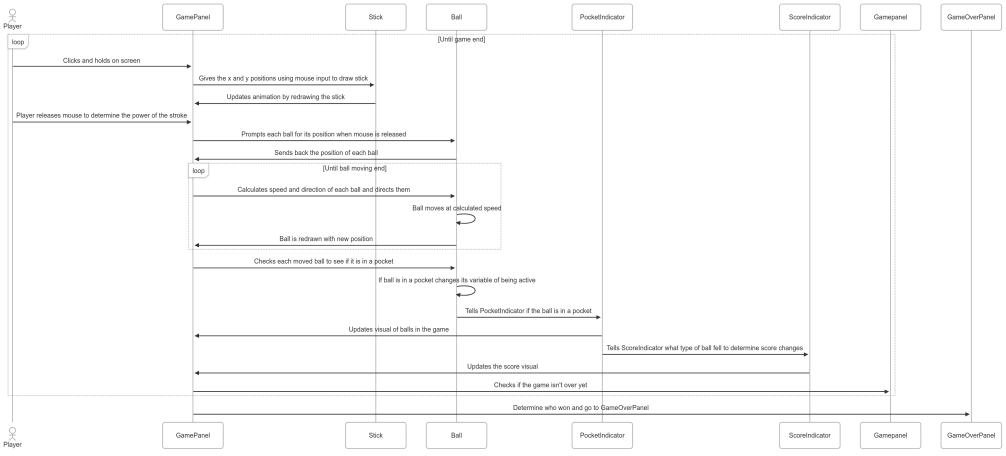
# 12 Class Diagrams

Here are the diagrams for our class structure, object interactions as a sequence, and a flow chart to represent AI decision-making visually (link here for readability):

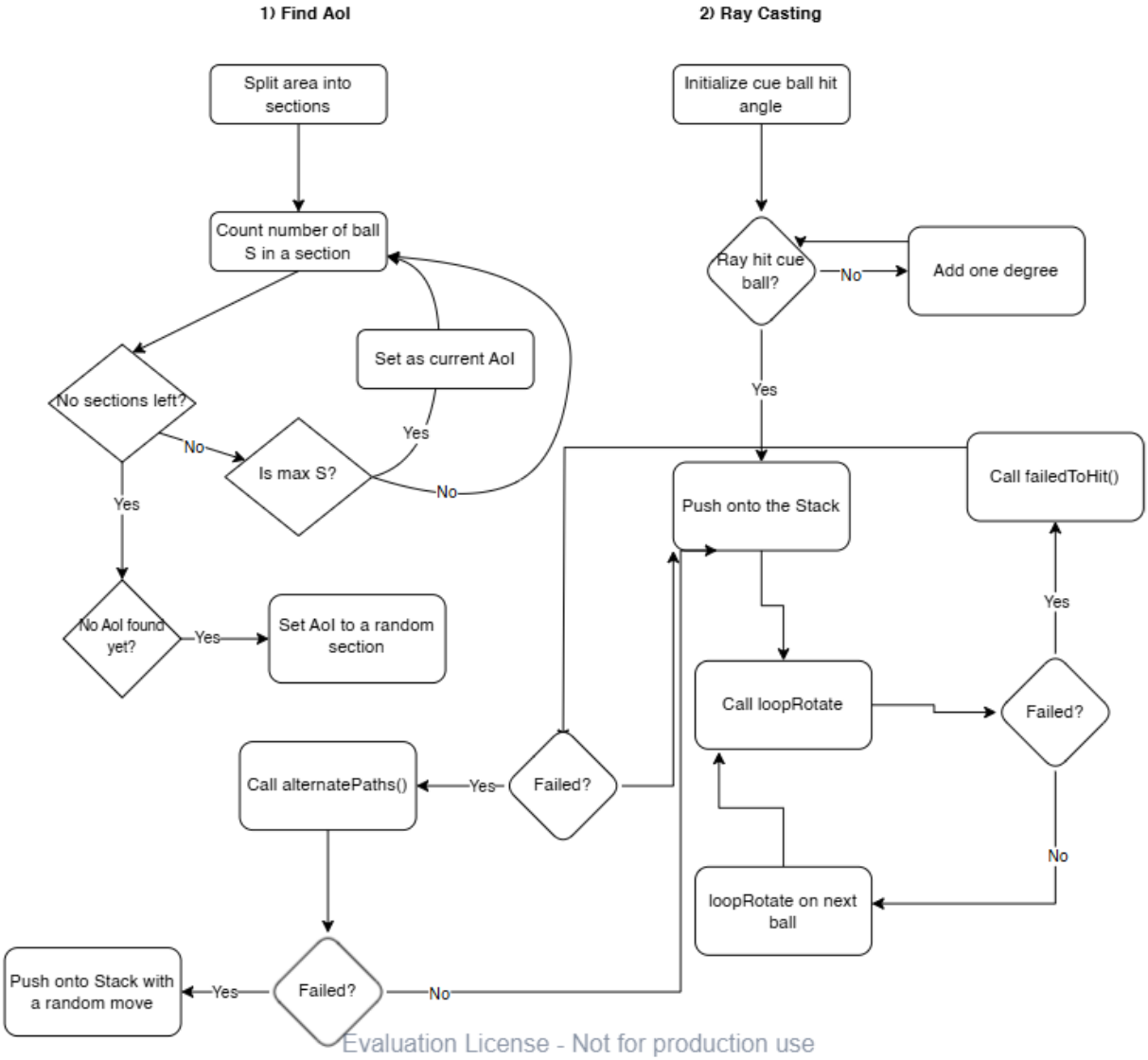
## 12.1 Class Diagram




## 12.2 Sequence Diagram




## 12.3 Flow Chart Diagram / AI Algorithm Flowchart


1) Find AoI
2) Ray Casting
Split area into sections
Count number of ball S in a section
No sections left?
No
Yes
Is max S?
Yes
No
Set as current AoI
No AoI found yet?
Yes
Set AoI to a random section
Initialize cue ball hit angle
Ray hit cue ball?
No
Add one degree
Yes
Push onto the Stack
Call failedToHit()
Call loopRotate
Failed?
Yes
No
loopRotate on next ball
Failed?
Yes
Call alternatePaths()
Failed?
Push onto Stack with a random move
Yes
Failed?
No
Evaluation License - Not for production use

# A Full Algorithm Pseudocode

Sweeping Algorithm Part 1

**Function Sweeping():**

- balltype ← S for Striped
- areaOfInterest ← Dimension from $(x_1, y_1)$ to $(x_2, y_2)$
- refAngle ← 0°
- outsideSweepAngle ← Angle from refAngle counterclockwise
- dimensionX ← Screen x dimension
- dimensionY ← Screen y dimension
- maxCount ← $-1$
- refAngles ← new Stack
- dampeningFactor ← 1000000
- numberOfSuccessfulPockets ← new ArrayList

Find area of Interest by splitting dimensionX / 3 and dimensionY / 2
**While** there are still dimensions:

- Count number of ball S in each dimension square
- **If** num > maxCount:
  - maxCount ← num
  - AoI ← currentDimension
  - **break**

centerRotate ← new Point at center of currentDimension
cueFindAngle ← 0°
**While** ray hasn't hit cue ball:

- cueFindAngle++
- Make new ray
- **break**

Push a BallInfo containing the cue & cueFindAngle into refAngles
hitAngle ← 0
increaseAmount ← 2
Draw ray from cue stick to end of AoI at hitAngle
Call loopRotate(cueBall, false)
Start new path: reset dampeningFactor, update successfulPockets, new cueFindAngle
Start new path (reset dampeningFactor, add new element to the numberOfSuccessFulPockets, use a new cueFindAngle), whilst adding pathway with an identifier Repeat adding paths Find max number of successful pockets out of the list, and that is our chosen path, and we set refAngles to that pathway using its identifier
**For each** angle in refAngles:

- Draw given ball at given angle

Give turn to player

**Function loopRotate(Ball b, boolean checkForPocket):**

- **If** checkForPocket:
  - **While** ray hasn't hit a pocket:
    - **If** checkForPocket:
    - b.hitAngle + = increaseAmount
  - **If** b.hitAngle > 360 and increaseAmount > 1:
    - increaseAmount−−
    - b.hitAngle = 0
- **Else**:
  - **If** b.hitAngle > 360:
  - Call failedToHit(b)
- Push BallInfo into refAngles

  **While** ray hasn't hit a ball of type S:
  - **If** dampeningFactor < 0.05:
  - Call loopRotate(b, true)
  - b.hitAngle + = increaseAmount
  - **If** b.hitAngle > 360 and increaseAmount > 1:
    - increaseAmount−−
    - b.hitAngle = 0
  - **Else if** b.hitAngle > 360:

        * Call failedToHit(b)

    $b_2 \leftarrow$ ball that got hit

    Push BallInfo into refAngles

    Recalculate dampeningFactor

    Call loopRotate($b_2$, false)

**Function failedToHit(Ball b):**

- **While** ray hasn't hit a pocket:
  - b.hitAngle += increaseAmount
  - **If** b.hitAngle > 360 and increaseAmount > 1:
    * increaseAmount−−
    * b.hitAngle = 0
  - **Else if** b.hitAngle > 360:
    * Call alternatePaths(b)
- Push BallInfo into refAngles
- **Return**

**Function alternatePaths(Ball b):**

- $Reflected_1 \leftarrow -1$, $Reflected_2 \leftarrow -1$
  **While** not reflected into pocket or twice:
  - b.hitAngle + = increaseAmount
  - Calculate $Reflected_1$, $Reflected_2$
  - **If** b.hitAngle > 360 and increaseAmount > 1:
    * increaseAmount−−
    * b.hitAngle = 0
    * $Reflected_1 \leftarrow -1$, $Reflected_2 \leftarrow -1$
  - **Else if** b.hitAngle > 360:
    * b.hitAngle ← random angle between 0 and 360
    * Push BallInfo to refAngles
    * Quit turn
- Push b.hitAngle, $Reflected_1$, and $Reflected_2$ to refAngles
- **Return**