INSTITUT FÜR INFORMATIK
DER CHRISTIAN–ALBRECHTS–UNIVERSITÄT KIEL


**Bachelorarbeit**

# A computer player for billiards based on artificial intelligence techniques

Jens-Uwe Bahr

Betreuer: Prof. Dr. Michael Hanus, Dr. Friedemann Simon
Abgabetermin: 28. September 2012

INSTITUT FÜR INFORMATIK
DER CHRISTIAN–ALBRECHTS–UNIVERSITÄT KIEL


**Bachelorarbeit**

# A computer player for billiards based on artificial intelligence techniques

Jens-Uwe Bahr

Betreuer: Prof. Dr. Michael Hanus, Dr. Friedemann Simon
Abgabetermin: 28. September 2012

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, den 28. September 2012

.....
*(Jens-Uwe Bahr)*

**Abstract**

This work presents JPool, a computer player for 8-ball. Billiards games present a series of interesting challenges such as a continuous action space and a stochastic element as shots may not be executed exactly as planned. JPool plays using the FastFiz framework.

Shots are identified geometrically using an event-based tree structure. JPool's focus lies on position play, attempting to move the cue ball's final resting place to a spot on the table with promising odds for future shots to be available. Therefore, a method of analyzing a table state for shape zones is presented. A physics engine has been created and is used to acquire shot parameters corresponding to the geometrically planned aiming direction. Furthermore, the cue ball's path is adjusted to aim for shape zones. Monte-Carlo sampling is used to select the best shot at hand. A rating function using several parameters defining JPool's play style is used. JPool performs well in the games that have been played; however, competitions with other computer players have not been held yet. Techniques used in JPool may be adjusted for use in other billiards games like 9-ball or 3-cushion billiards.

# Contents


**1 Introduction** **1**
- 1.1 The game of 8-ball . . . 1
- 1.2 Design challenges . . . 3
- 1.3 Computational billiards frameworks . . . 4
  - 1.3.1 The game visualizer . . . 4
  - 1.3.2 Shot parameters . . . 5
- 1.4 Competition . . . 5
  - 1.4.1 PickPocket . . . 6
  - 1.4.2 CueCard . . . 6
  - 1.4.3 PoolMaster . . . 6
  - 1.4.4 Other competitors . . . 7
- 1.5 Overview of JPool . . . 7

**2 The geometry of pocket billiards** **9**
- 2.1 Aiming . . . 9
  - 2.1.1 Aiming for the pockets . . . 9
  - 2.1.2 Aiming for a ball . . . 10
  - 2.1.3 Aiming over a cushion . . . 11
- 2.2 Shot generation . . . 12
  - 2.2.1 Modelling shots . . . 13
  - 2.2.2 Implementation . . . 14
- 2.3 Analyzing the table state for position play . . . 16
  - 2.3.1 Pixelwise rating of position zones . . . 17
  - 2.3.2 Shape zone approximation . . . 17
  - 2.3.3 The cue ball's passage through shape zones . . . 18

**3 The physics of pocket billiards** **21**
- 3.1 FastFiz: event-based simulation of shots . . . 22
  - 3.1.1 Impact of the cue-stick on the ball . . . 22
  - 3.1.2 Ball movement . . . 23
  - 3.1.3 Detecting collisions between balls . . . 25
  - 3.1.4 Handling collisions between balls . . . 25
  - 3.1.5 Collisions between a rail and a ball . . . 26
- 3.2 JPool: optimizing geometrically determined shots . . . 27
  - 3.2.1 Reaching a target point . . . 27
  - 3.2.2 Collisions between balls . . . 32
  - 3.2.3 Collisions between a rail and a ball . . . 34
  - 3.2.4 Cue Stick Impact - Shot parameters . . . 34

3.3 Applying the physics engine . . . 35
3.3.1 Adjusting aiming directions . . . 35
3.3.2 Calculating velocities in a step tree . . . 36

**4 Planning ahead in pocket billiards** **39**
4.1 Probabilistic search algorithms . . . 39
4.1.1 Expectimax . . . 39
4.1.2 Monte-Carlo sampling . . . 40
4.1.3 Comparison . . . 40
4.2 The influence of noise on a shot . . . 41
4.3 Rating shots . . . 43
4.4 Special situations . . . 44
4.4.1 The break shot . . . 44
4.4.2 Ball-in-hand . . . 45
4.4.3 Safety shots . . . 46

**5 Results** **49**
5.1 Off-break wins . . . 49
5.2 Shot success . . . 51
5.3 Domain specificity . . . 51
5.4 Further work . . . 51
5.5 Conclusion . . . 52

**Appendix** **53**
1 Physical constants used in FastFiz . . . 54
2 Commonly used variable names . . . 54

**List of Figures** **55**

**Bibliography** **57**

# 1 Introduction

Billiards is a very popular family of games being played all over the world. 8-ball pool, 9-ball pool and snooker are the most popular games of this kind. The history of billiards goes back to the 15th century. First competitions of computer billiards players have been held at the international computer olympiad [8] in 2005. While past competitions focused on the game of 8-ball, future competitions are said to feature other pool games like 9-ball as well.

All billiards games require both strategic and physical skill. In addition to pocketing a ball, the resulting position of the cue ball is planned to ensure a shot to be available in the next turn. However, a shot may not be executed exactly as it has been planned. Even professional pool players occasionally miss a shot. This is simulated in computational pool by adding random noise to each shot prior to its execution. These aspects create a very interesting and unique set of challenges for computer players: shots have to be identified in a continuous action space and the exact outcome of a shot is uncertain.

A number of computer players has been created using very different techniques to handle the challenges of billiards. Furthermore, robots capable of executing shots have been built, creating an interface for games between human players and computers. Deep Green [6] is a fully functional robotic billiards player developed at Queen's University in Canada working with the PoolFiz engine [13]. PoolFiz has also been the game engine used at past computer olympiads; FastFiz [19] is an alternative game engine with some additional features like a web interface.

This thesis presents JPool, a computer player for 8-ball. The focus on JPool lies on position play, attempting to plan the path of the cue ball such that a shot is available in subsequent turns. A new way of modelling and detecting shots on the table will be presented in chapter 2, along with a method of finding good areas for the cue ball to come to rest. Furthermore, a new method of acquiring corresponding shot parameters, heavily based on the physics simulation utilized in FastFiz, is presented in chapter 3. Chapter 4 presents the methods of planning ahead and selecting a shot as performed by JPool. This work concludes in chapter 5 with the presentation and analysis of results and ideas for further improvements.

## 1.1 The game of 8-ball

8-ball is played by two players on a special table using sixteen balls and a cue stick. While the exact size of the table may vary, its shape is rectangular with the long sides being twice as long as the short sides. The table is marked by a *headstring*, 1/4 along the length of the table, and a *footstring*, 3/4 along the length of the table. The center of the headstring is called the *head spot* and the center of the footstring is called the *foot spot*. The borders of the play area are protected with *cushions*, preventing balls from rolling off the table. There are 6 pockets: one in each corner (called the *corner pockets*) and two in the center of the

long side cushions (called the *side pockets*). The size of the pockets ranges from 1.75 to 2.25 times the size of the balls. The sixteen balls consist of one white ball called the *cue ball* and fifteen numbered balls called the *object balls*: balls 1 through 7 are called the *solid* balls, ball 8 is called the *8-ball* and balls 9 through 15 are called the *striped* balls.
During the course of the game each player is assigned a color (solid or striped). The players take turns executing *shots*, striking the cue ball with the cue stick aiming to cause a ball of their color to be *pocketed*. The first player to pocket all balls of his color and then pocket the 8-ball wins the game.

FastFiz plays 8-ball by the rules set by the Billiards Congress of America (BCA) as detailed in [3]. The most important rules are:

- A player executes a shot by striking the cue ball with the cue stick.
- The players take turns executing shots. However, a player may keep his turn and shoot again if he legally pocketed a ball or executed a legal break shot.
- The game starts with the object balls racked in a triangular formation with the 1-ball positioned on the foot spot; the triangle is extending behind the footstring. The cue ball is placed anywhere behind the headstring (see figure 1.1). While the order of the balls within the triangle is not specified by the rules, FastFiz places the balls within the triangle ordered by their number.
- One player is selected to begin the game. The first shot is called the *break shot*. A break shot is considered legal if an object ball was pocketed or at least 4 object balls contacted a rail.
- The first ball that is legally pocketed after the break shot assigns the players' colors: the player executing the shot is assigned the color of the ball being pocketed. The time prior to color assignment is called *open table* as players may target any ball other than the 8-ball.
- A player must indicate prior to each shot (except the break shot) which ball he intends to pocket. He also has to specify the pocket he aims for. If this indication is not met, the shot is deemed illegal even if another ball of the player's color is pocketed. This indication is called *calling* a shot.
- A shot is called a foul if the first ball touched by the cue ball is not of the acting player's color or a ball not of the player's color is pocketed. This includes pocketing the cue ball (which is called *scratching*).
- After executing a foul, a player loses his turn and the opposing player gets *ball in hand*: the right to place the cue ball anywhere on the table.
- Instead of calling a shot, a player may call *safety*, indicating that he does not intend to pocket a ball but to place the cue ball in an inconvenient position for the opposing player.
- If a player illegally pockets the 8-ball, he loses the game. The player that legally pockets the 8-ball wins the game.

- A player can only call an attempt to pocket the 8-ball after all balls of his color have already been pocketed. If the 8-ball is legally pocketed but the player executes a foul (i.e. by additionally pocketing the cue ball), the player loses the game.

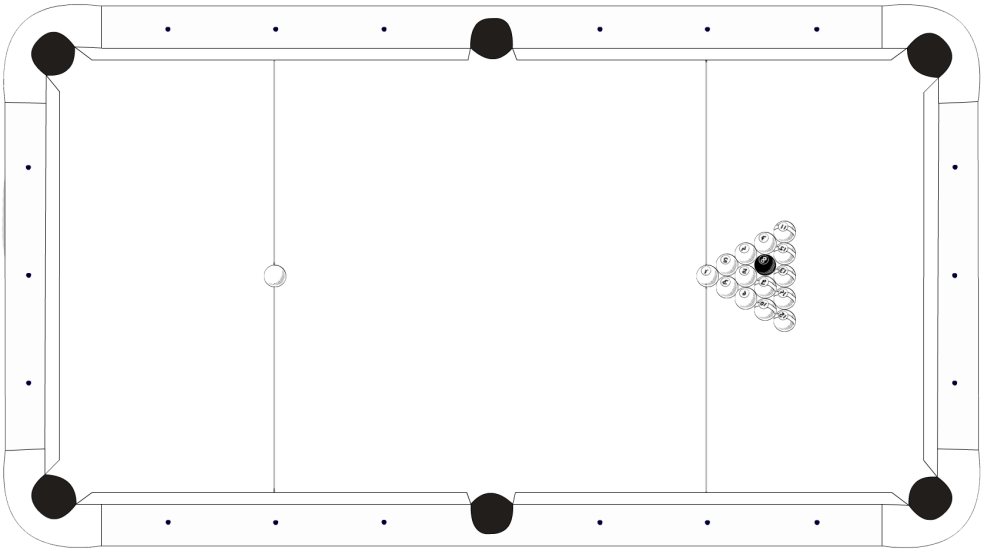


**Figure 1.1:** The initial racked table state with the headstring and footstring

## 1.2 Design challenges

Computational pool differs from games typically played by computer agents in a variety of characteristics:

- **A continuous state and action space.** There is an infinite number of possible shots to execute. Generally, a shot is identified by a set of five real numbers and a game state (also called *table state*) which consists of the positions of the 16 balls on the table. While typical games played by computer agents usually *only* have to pick the most promising out of a list of possible actions to take, one of the main challenges in billiards is identifying viable shots in the action space. This also makes it very hard to predict the actions of the opposing player.

- **A stochastic element.** Real-life pool is a game of both planning and executing the best shot at hand. Players may fail to execute a shot as it has been planned. This is simulated in both PoolFiz and FastFiz by adding small random numbers to the shot parameters a computer player calculates. This is called the *noise*. As a result, the shot calculated by the computer player is not executed exactly as planned and may not succeed or even result in unwanted events like a foul. This element of chance has to be addressed in any calculations done.

- **The turn structure.** In typical games played by computer agents, the players play in turns. In billiards, a player may never lose his turn and win the game with the opponent never executing a shot. Combined with the element of chance, every shot

may result in losing the game. As of [11], PoolMaster wins 100% of games without the enemy player ever shooting when it is the starting player and no noise is added to the shots (and about 65% when playing with regular noise).

These points disqualify billiards for most methods like MiniMax trees that are usually used when programming computer players, making billiards a very challenging and interesting family of games.

## 1.3 Computational billiards frameworks

Currently, there are two simulators for computational billiards: PoolFiz and FastFiz. PoolFiz was developed by Michael Greenspan and Will Leckie at Queen's University, Canada, and is documented in [13], [12] and [14]. FastFiz is a faster version of PoolFiz, using the same formulas to handle the physics on the pool table. It has been developed at Stanford University (USA) and is available at [19]. While all past competitions have been held in PoolFiz, the 2011 International Computational Billiards Championships were to be held in FastFiz. JPool uses the FastFiz framework.

### 1.3.1 The game visualizer

The FastFiz framework offers a variety of functions. In addition to the capability of simulating games, a web interface with a tournament manager is included. However, the game visualizer supplies by FastFiz relies on communication with a specific server. This server is offline due to a hardware failure in 2011. An alternative game visualizer has been built using Adobe Flash, see figure 1.2. Balls are rendered in 3D onto a 2D table image. Movement data is provided using a script written in C++ utilizing the FastFiz engine. This visualizer has been fully integrated into the FastFiz web interface using Perl (FastFiz is written in C++ and Perl as well).

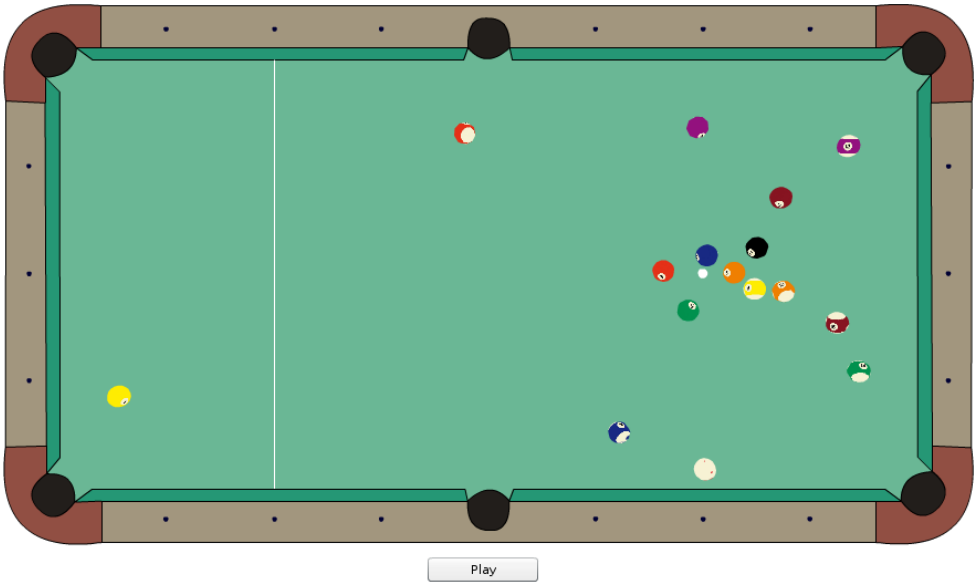



**Figure 1.2:** A screenshot of the game visualizer created for JPool

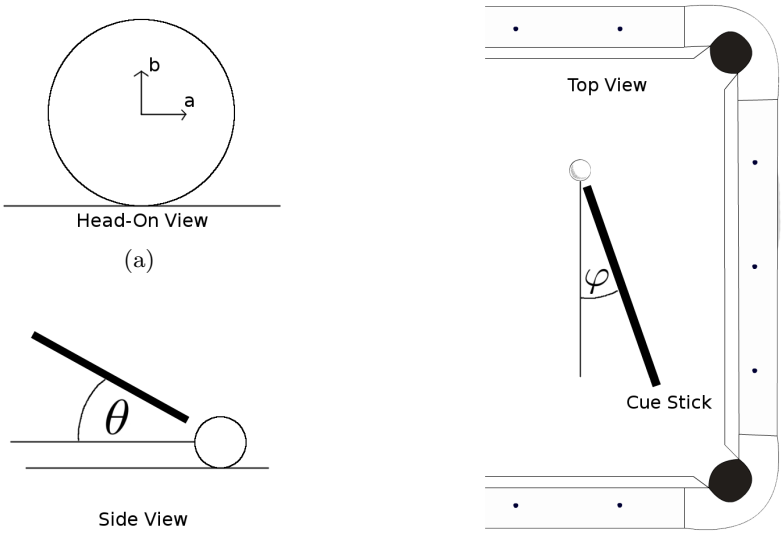



**Figure 1.3:** Shot parameters visualized (a) head-on view showing $a$ and $b$, (b) side-view showing $\theta$, (c) top view showing $\varphi$ (redrawn from [18, p. 2])

### 1.3.2 Shot parameters

Both the PoolFiz and FastFiz frameworks pass their players a table state, consisting of the positions of the 16 balls and some additional information like the states of the balls (pocketed or not), the player's color and the remaining time. The computer player returns five parameters: $a$, $b$, $\theta$, $\varphi$ and $V_0$. In detail, these define the following characteristics of the shot:

- $\varphi$ defines the aiming angle in degrees
- $\theta$ defines the elevation angle in degrees
- $a$ defines the $x$-coordinate of the cue stick impact point on the cue ball in mm
- $b$ defines the $y$-coordinate of the cue stick impact point on the cue ball in mm
- $V_0$ defines the velocity of the cue stick impact in $m/s$ (max is $4.5m/s$)

The shot parameters are illustrated in figure 1.3. The usual time limit is 10 minutes per game and player. Gaussian noise is added to each of the parameters. The noise model varies with each tournament. Generally, the noise aims to leave 75% of all shots successful [10, p. 5].

## 1.4 Competition

Starting in in Taipei (Taiwan) in 2005, computational billiards has been one of many games played at the Computer Olympiad [8] held by the International Computer Games Association (ICGA). The event has been repeated in Turin (Italy) in 2006 and in Beijing (China) in 2008.

The competition was won by PickPocket [17] in both 2005 and 2006 and by CueCard [2] in 2008. Other competitors were PoolMaster [10], SkyNet [15], Elix and Snooze. CueCard was developed in the USA, Snooze was developed in Australia and all other players were developed in Canada. This makes JPool the first european competitor.
Competitions were held using the PoolFiz simulator. In 2011, Stanford University started the 2011 International Computational Billiards Championships (ICBC-11) which was cancelled due to hardware failure. Competitions were to be held using the FastFiz framework. The following sections will present an overview of the main concepts and principles of other computational pool players.

### 1.4.1 PickPocket

PickPocket won the competitions in 2005 and 2006 and scored 2nd place in 2008. It was developed by Michael Smith at the University of Alberta (Canada) as the subject of his master thesis [17]. PickPocket consists of a move generator and a search algorithm. The move generator scans the table for common shot patterns like direct shots or kick shots (see section 2.2 for more details), leaving out more complex patterns if an easy shot has been found. Shots are identified by four parameters and rated using a precomputed table. PickPocket uses a velocity table to look up the translation of geometric shots to shot parameters, solely setting the aiming direction and strength. Position play is accounted for by altering three parameters ($V_0$, $a$ and $b$) in various random ways and evaluating them using a Monte-Carlo based algorithm searching to 2-ply depth using a sample size of 15.

### 1.4.2 CueCard

CueCard joined the Computer Olympiads in 2008 and won the competition. It was developed by Prof. Yoav Shoham, Christopher Archibald and Alon Altman at Stanford University (USA). Like PickPocket, CueCard generates shots by their aiming direction and strength and then alters some shot parameters in order to find the variant of the shot that yields the best resting position of the cue ball. CueCard rates shots by sampling them between 25 and 100 times, making use of the FastFiz engine. Search is executed in two ways: in the early game a search depth of 2 is used. In later stages the game is calculated to its end, yielding an estimate of the probability of CueCard winning the game. CueCard's authors identified the break shot and the high number of samples per shot to be the largest contributors to CueCard's success [2, p. 6]. Improvements in look-ahead search were said to have no effect on the success.

### 1.4.3 PoolMaster

PoolMaster scored 2nd place in 2005 and 4th place in 2006. It is being developed by Jean-François Landry and Jean-Pierre Dussault at the Université de Sherbrooke (Québec, Canada). Development has been thoroughly documented in [10], [5], [4] and [11]. PoolMaster differs from other contestants by the fact that its focus lies on position play. The table is analyzed for good position spots and shots are specifically aimed for the best spots on the table. Michael Smith identified PoolMaster's search algorithm to be its weakness because it does not take the probability of success of a shot into account, sometimes resulting in very risky shots [17, p. 66]. PoolMaster's development is ongoing and the authors made a lot of improvements on PoolMaster since the competition in 2006. It has not been played against

other competitors yet, but statistics indicate that the new PoolMaster performs a lot better than its contestants.

### 1.4.4 Other competitors

**SkyNet** was developed by one of the creators of PoolFiz, Will Leckie, at Queen's University, Canada [15]. It focusses on using modifications of search trees like *-Expectimax to perform look-ahead in billiards and makes heavy use of Monte-Carlo search algorithms. While it performed worst in the competitions of 2005, it scored 2nd place in 2006. It searches to a 3-ply depth.

**Elix** was developed by Marc Goddard at Queen's University, Canada. Since Mr. Goddard is a good real-life billiards player, the idea behind Elix was to translate his experience into a computer player. Shot selection is done by a set of rules. However, the complex nature of the rules would sometimes cause poor shot selections [17, p. 76]. Elix scored 3rd place in all three competitions.

**Snooze** was developed by Sven Reichard from Australia. It scored 5th place in 2006's competition. No details of its operation are available.

## 1.5 Overview of JPool

In addition to pocketing the targeted ball, the primary focus of JPool lies on optimizing the resting place of the cue ball to allow for position play. When successful, this ensures the player to have a good shot available in the subsequent turn. Another key aspect of position play is the movement of balls to easier spots - while JPool does not specifically aim to move balls other than the targeted object ball, peripheral ball movement is strongly taken into account when deciding which shot to play. Other tactical measurements are applied to minimize the odds of the enemy player.

JPool follows these steps to choose the best shot to execute:

1. For a given table state, a set of possible shots is generated using geometric means. This is done using a tree structure based on the events happening on the table arranged in steps. Usually, search is limited to three object balls involved, resulting in several hundred available shots. Each shot is rated geometrically and the shots are sorted by ascending difficulty.

2. Each geometrically determined aiming direction $\varphi$ is simulated using a series of naive shot parameters with varied random values for $a$, $b$, $\theta$ and $V_0$. Some physical effects like spin may cause the shot to fail. The simulation is analyzed step by step and ball velocities are rotated to match the geometrically planned steps. These rotations are backtracked to adjust the initial aiming direction $\varphi$. This process repairs an average of 19.7% of naively determined shots that failed without adjustments.

3. For each working shot found in step 2, the resulting table state is analyzed and position zones are identified using geometrical means. Each shot is altered using a set of equations based on the physics simulation of FastFiz, causing the cue ball to halt inside

of good position zones. These variations are simulated and enhanced in the same way as naive shots in step 2.

4. Each proposed shot is rated by several tactical values like the position rating, the enemy position rating, overall resulting table state rating, probability of success and difficulty of the pocketed ball(s). These ratings are designed to follow the ideal pool player's strategy detailed in [9]. The method of rating depends on the number of balls on the table, i.e. for the last ball to be pocketed only the probability of success is computed. Shots are compared using a Monte-Carlo search algorithm, searching with a sample size of 400.

This process can be split into three main topics: shot generation, shot parameter calculation and shot selection. Each step of this process is based on different kinds of calculations. In general, JPool's calculations can further be divided into several classes: geometry, physics, shot analysis and tactical rating using lookahead. These classes are also used to structure the implementation of JPool and are explained over the course of the following chapters:

**JGeometry:** A class for all geometrical calculations like shot generation, simple shot difficulty rating and the generation of shape zones. The techniques used in this class are detailed in chapter 2.

**JPhysics:** A collection of functions to calculate the expected outcome of an event on the table based on the equations used for simulation in FastFiz. Several functions are supplied to alter the outcome of a shot, calculating the needed velocities of a ball to achieve a certain goal. Ultimately, this class is used for shot parameter calculation. The equations used in the class are detailed in chapter 3.

**JAim:** This class coordinates the simulation and improvement of aiming directions supplied by JGeometry by determining a set of possible shot parameters. This is done by testing naive parameters which are enhanced using the functions provided by JPhysics. Essentially, JAim generates the list of shots considered by the shot selection algorithm in JLookahead. JAim is explained in section 3.3.

**JLookahead:** This class rates the shots that have been found in JAim according to several tactical considerations and performs the Monte-Carlo search, selecting the most promising shot to execute. This class is detailed in chapter 4.

The results and strengths of JPool are analyzed in chapter 5. Additionally, a list of possible improvements for further work is given.

# 2 The geometry of pocket billiards

Since pool is a game with a continuous action space, identifying possible shots on the table is the first and most important step of determining the best shot at hand. This chapter presents a dynamic and extendable way to model and generate ways to legally pocket a ball. Furthermore, the basis for planning ahead is built by showing methods for identifying and aiming for good spots on the table.

First, very basic geometric considerations will be presented, adjusting aiming angles at pockets and introducing the concept of ghostballs. Based on these aiming techniques, a method of modelling and generating possible shots on the table will be presented. These shots are rated for their difficulty. As JPool focusses on position play, this chapter concludes by presenting ways to find good spots for the cue ball on the table and for analyzing which of these spots are reachable for a given shot. JPool's shot generation works entirely geometrical and ignores physical values like spin and throw. These will be considered in later optimization of the shots found.

## 2.1 Aiming

### 2.1.1 Aiming for the pockets

In order to find the best direction to aim for, the margin of error for pocketing a ball into any pocket is calculated. Generally, the left side of the ball has to pass the left corner of the pocket and the right side of the ball has to pass the right corner of the pocket for the ball to go in (see figure 2.1, center pocket). The angle between these leftmost and rightmost directions is the margin of error for the ball to be pocketed. The bigger the margin of error, the easier the shot.

This angle is decreased if any balls are in the way (see figure 2.1, left pocket) and increased if it is possible to hit a cushion adjacent to a corner-pocket before pocketing the ball (see figure 2.1, right pocket). This is done using two vectors for the leftmost (blue) and the rightmost (red) direction which will still pocket the ball. These margins are adjusted due to the influences mentioned above. If the right margin crosses the left margin in the process

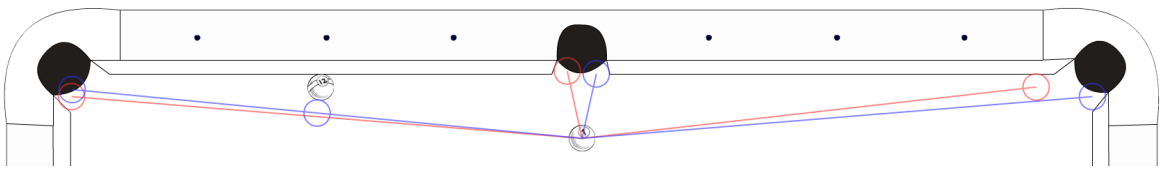
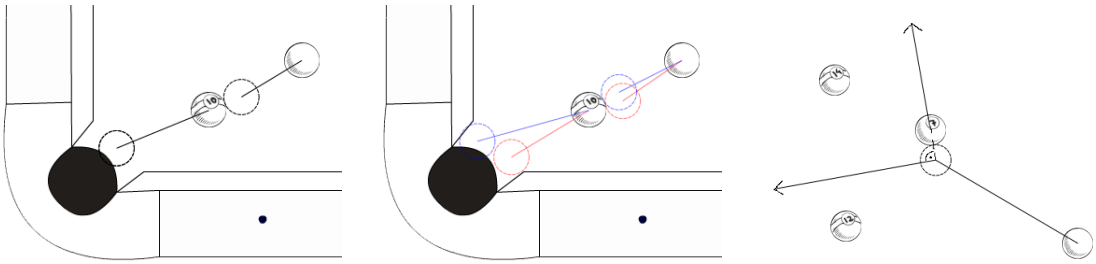


**Figure 2.1:** Aiming for a pocket with another ball in the way (left), directly (center) and with the option to strike a cushion (right)



(a) (b) (c)

**Figure 2.2:** (a) aiming for a strike using a ghostball and ideal trajectory (b) aiming using leftmost and rightmost directions (c) aiming for a kiss deflects in a perpendicular direction

(or vice versa), it is no longer possible to pocket the ball into the pocket in question. When aiming at a pocket, the center between these two vectors is the aiming direction with the highest margin of error to both sides. For each ball and each pocket, the aiming direction (as vector) and margin of error (in degrees) are saved for further processing (if the path is free).

PickPocket uses the approach of simply aiming at the center of the pocket [17, p. 17], ignoring adjustments to the aiming direction that may be necessary and not taking cushion reflections into account, resulting in a shifted area of tolerance and a reduced number of shots considered. PoolMaster chose a similar approach as JPool to target the pockets, creating a shot-window; however, PoolMaster only uses the corners of the pockets to form the window and does not take cushion-reflections or blocking balls into account [5, p. 3].

### 2.1.2 Aiming for a ball

When the cue ball and an object ball are colliding, the post-collision trajectories of both the object ball and the cue ball are calculated by JPool. A collision between a stationary and a moving ball is called a *strike* if the purpose of the collision is moving the stationary ball. A collision is called a *kiss* if the main purpose of the collision is not moving the stationary ball but deflecting the trajectory of the moving ball. JPool distinguishes between these types of collisions and calculates the result accordingly. For some collisions both factors are calculated, i.e. to determine the halting point of the cue ball for position play.

#### 2.1.2.1 Aiming for a strike

In real-life pool, players often use a *ghostball* to help aiming for the correct spot on an object ball [9]. As an approximation, the ghostball is positioned on the opposite side regarding the desired direction of movement of the object ball, see figure 2.2(a). Section 3.1.4 will show that this is not exactly the case; therefore, aiming with ghostballs is an approximation only used by JPool in the geometrical parts of shot processing. A ghostball is easily calculated using a vector of length $2R$ (where $R$ is the ball radius) in the opposite direction of the object ball's trajectory.

Using a ghostball to aim for the rightmost and leftmost direction a ball can be pocketed with yields a leftmost and rightmost ghostball position for the cue ball (see figure 2.2(b)).

The angle between these directions is the margin of error of the cue ball's trajectory, also representing the *difficulty* of the shot. When executing the shot with noise, the shot will succeed if the cue ball's trajectory stays within this angle.

Note that when aiming at a ball with a high cut angle (in order to make it roll in a direction very different than the direction from the cue ball to the object ball), the leftmost or rightmost aiming direction may not be reachable because it lies on the other side of the object ball. This may also result in a wrong margin of error. JPool addresses this problem by rotating the aiming direction to the first reachable point on the ball if this point lies between the leftmost and rightmost aiming direction (as illustrated in figure 2.3).

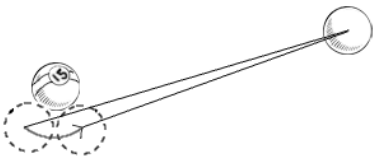


**Figure 2.3:** An aiming direction may not be reachable

This angle is the only measure JPool uses to rate the difficulty of a shot and is based on the model used in both [9] and [1]. As detailed in [5, p. 3-4], PoolMaster uses a similar method of rating the difficulty of a shot, also using the margin of error available when executing the shot. However, PoolMaster uses a complex function weighing out a series of parameters to address the problem illustrated in figure 2.3. PickPocket [17, p. 31-35] uses a precomputed table to lookup the difficulty of a shot. A shot is identified using four parameters and executed a number of times to estimate the probability of success. While this table has approximately 1.6 million entries, it is only valid for direct shots and does not take table state-specific factors like balls partly blocking the way into account and therefore is a very indirect way of rating shot difficulty. CueCard [2, p. 2] does not rate shots by any parameters but rather executes them using the FastFiz framework and analyzes the results, filtering shots that did not succeed.

#### 2.1.2.2 Aiming for a kiss

After a collision with another ball, a ball is deflected in a direction perpendicular to the line through the centers of both balls, see figure 2.2(c). This is called the *stop ball line* (see [1, p. 80]). However; the ball's actual trajectory differs from this line depending on its spin. This effect is ignored for these geometrical considerations but is considered and used in later optimization of the shot.

### 2.1.3 Aiming over a cushion

PoolMaster aims over cushions using a mirror table (see [10, p. 9]), implying that the angle of incidence equals the angle of reflection. For FastFiz, this is not the case. As further described in section 3.1.5, FastFiz reflects a ball off a rail using the following equation:

$$\vec{v}_{postcollision} = \begin{cases} (0.9\vec{v}_x, -0.6\vec{v}_y, 0)\,, & \text{if rail is horizontal} \\ (-0.6\vec{v}_x, 0.9\vec{v}_y, 0)\,, & \text{if rail is vertical} \end{cases} \tag{2.1}$$

Where $\vec{v}$ is the linear velocity of the ball. Note that this equation only rules over the linear movement of the ball and does not take spin into account, making it an approximation of the actual trajectory of the ball.

## 2.2 Shot generation

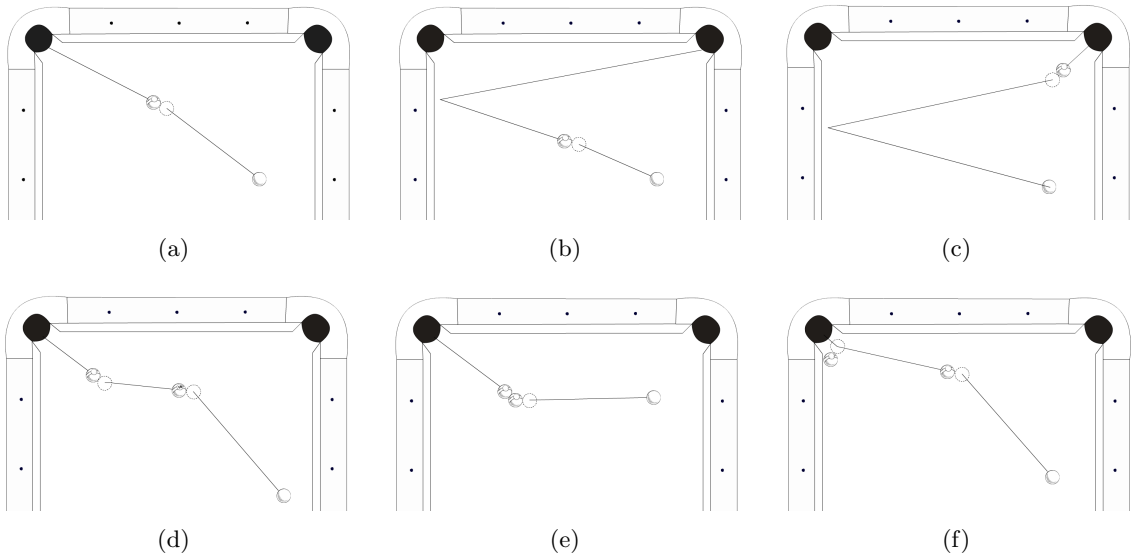



**Figure 2.4:** Different patterns to forming a shot: (a) direct shot, (b) bank shot, (c) kick shot, (d) combination shot, (e) pulk shot, (f) kiss shot

A shot can consist of a combination of the events described in previous sections (pocketing a ball, striking a ball, kissing a ball and colliding with a rail). By combining these events to common patterns, human pool players distinguish between a variety of shot types (see [9]). As JPool aims to maximize the probability of finding the best possible shot, a lot of those shot types are considered. Therefore, the table state is scanned for specific patterns, mirroring standard shots in billiards games. Figure 2.4 shows the most common shot types:

- **Direct shots** are also called straight-in shots or cut shots. The object ball is directly struck by the cue ball and moves into a pocket without cushions or other balls involved.
- **Bank shots** maneuver the object ball over a cushion into the pocket.
- **Kick shots** let the cue ball strike the object ball after hitting a cushion first.
- **Combination shots** attempt to pocket an object ball using another object ball. As of [1], the difficulty of a shot grows by the factor of 10 per additional ball involved.
- **Pulk shots** are a special case of combination shots. Two object balls are positioned very close together and aligned in a way that they are pointing towards a pocket. In this case, the difficulty of the shot is very low since the first object ball can be struck at a big range of angles to cause the second ball to be pocketed.
- **Kiss shots** are another special case of combination shots. An additional object ball is kissed to adjust the trajectory of the original object ball towards its target.

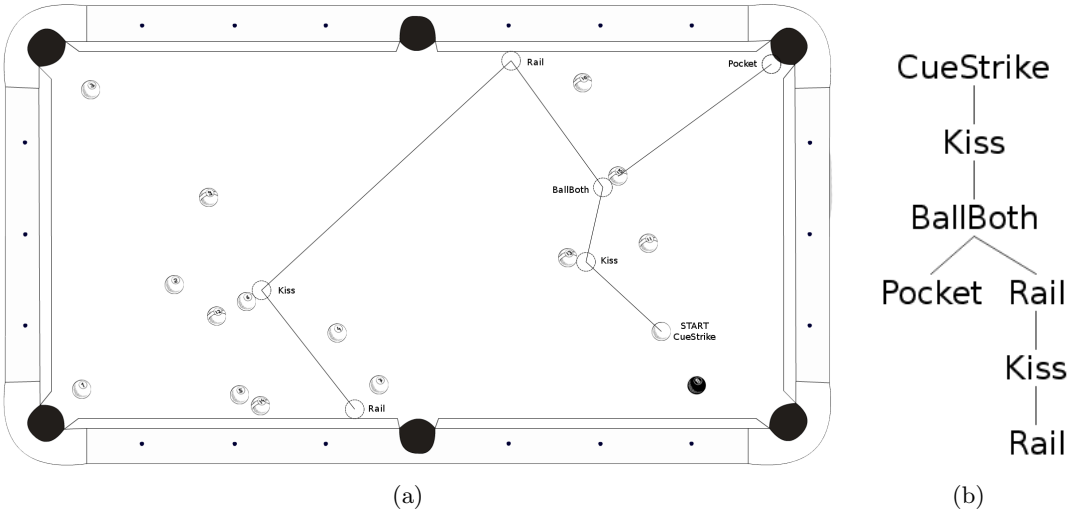
### 2.2.1 Modelling shots

PickPocket generates shots one class at a time [17, p. 14], ignoring complicated shots if a suitable direct shot has been found; however, this may result in ignoring the easiest shot at hand if it is not a direct shot (i.e. PickPocket considers combination shots last, but often a pulk shot is the easiest shot available).

Both CueCard and PoolMaster generate a variety of shots from the beginning. PoolMaster only considers direct shots, kick shots and bank shots (see [10, p. 15]) while CueCard considers "more complex shots and special shots designed to disperse clusters of balls" [2, p. 2].

JPool takes a similar approach and generates as many available shots for a given table state as possible; however, JPool is not limited to predefined shot types but freely combines possible events on the table to shots. To do this, JPool models a shot as a series of steps, each step having its own type. The types of steps are:

- **Cue strike** - this step is always the first step in a shot, representing the cue ball being struck be the cue stick causing its desired movement.
- **Pocketing a ball**
- **Rail collision**
- **Ball strike** - a ball collides with another ball. This step only considers the striking of the subsequent ball
- **Ball kiss** - a ball kisses another ball, altering its own trajectory
- **Ball both** - in this step two balls collide and the post-collision trajectories of both balls (strike and kiss) are calculated further




(a) (b)

**Figure 2.5:** Steps of a kiss shot (a) on the pool table (b) as step tree as used by JPool

These step types mirror the geometrical considerations done in previous sections. Note that for steps of the type **ball both**, a series of steps parallel to the main set of steps is initiated, called the branch. This results in shots having a tree-like structure with a single branching point, called the **step tree**. JPool generates these steps backwards from a ball being pocketed towards the cue ball being struck and calculates the events in the branch only if the shot is chosen for further investigation. Since the geometric calculations done don't take the ball's velocity into consideration, there is no way of knowing where the cue ball will halt; therefore, JPool simply simulates a shot using very strong shot parameters to determine the cue ball's potential path.

Figure 2.5(a) shows a kiss shot: after the cue strike event, the cue kisses ball 13 and then strikes ball 15, which is pocketed. However, after striking ball 15, the cue ball's trajectory is not over: it first collides with a rail, then kisses ball 6 and then collides with another rail. Figure 2.5(b) shows the corresponding step tree of this shot.

### 2.2.2 Implementation

As mentioned above, JPool calculates a series of possible combinations of events on the table resulting in a ball being pocketed using a tree structure. Each node in the step tree has a type indicating which kind of event on the table it represents, saves data about the shot itself like starting- and ghostball-positions and has pointers to the next step in the shot and to a branching step, if there is one for this event (see listing 2.1).

The construction of a shot is done using two lists: one list for completed shots called *doneShots* (each starting with a cue strike event) and one list for uncompleted shots called *undoneShots*. As JPool generates the shots backwards, *undoneShots* contains paths for balls going into the pocket but no information on how the ball needs to be hit by the cue ball to achieve that. In the first step, for each pocket a step of the type **pocketing** is added to *undoneShots*. JPool iterates over *undoneShots*, trying to reach the target points with balls that are not yet involved in the shot and thereby generating a new list of uncompleted shots. This process is repeated recursively until no more shot steps are found. The search is additionally limited by two parameters: a search depth and a *direct* flag. The *direct* flag decides whether rail and kiss collisions should be considered. The depth parameter determines how many ball collisions may happen before the targeted object ball is pocketed. The depth is set to 2 by default, allowing regular combination shots. See listing 2.2 for an overview of the shot generation progress. Note that this procedure of shot generation requires careful deletion of unused shot steps.

Note that for a single entry in *undoneShots* multiple ways to reach the target point in question may be found. Since shot generation is implemented using pointer logic this means that different shots may share steps.

```
// enumeration for all the possible types a JShotstep can have
enum JShotstepType {  CUESTRIKE, POCKET, RAIL, STRIKE, KISS, BALLBOTH  };

// a class for storing a shot, broken down to steps
class JShotstep {
  protected:
    JShotstepType type;       // type of this step
    JShotstep* next;          // next step of this shot, if any
    JShotstep* branch;        // branching step of this shot, if any

    Pool::Point posB1;        // starting point for b1
    Pool::Point ghostball;    // target point for b1
    Pool::Point leftmost;     // tolerances to both sides
    Pool::Point rightmost;
    int b1;                   // number of the main ball acting in this step
    int b2;                   // number of a secondary ball
};
```

**Listing 2.1:** Excerpt of the class 'JShotstep' for a single node in the step tree

```
void generateShotTree(vector<JShotstep*> &doneShots, int depth, bool direct) {
  // initialize the shot tree by adding all 6 pockets
  vector<JShotstep*> pocketways = generatePocketingWays();

  // for the specified depth, check if a shot can be formed
  generateShotTreeHelper(doneShots, pocketways, depth, direct);
}

void generateShotTreeHelper(vector<JShotstep*> doneShots, vector<JShotstep*>
    myTargets, int depth, bool direct) {
  vector<JShotstep*> undoneShots; // temporarily saves unfinished shots

  Pool::Point targetPoint;

  // Attempt to calculate steps for every way to pocketing a ball
  for(int i = 0; i < (int) myTargets.size(); i++) {
    if(myTargets[i] == NULL) continue;

    // Calculate where to hit the next ball
    targetPoint = generateGhostball(myTargets[i]);

    // Check for every ball if this point is reachable
    for(int b = 0; b < 16; b++) {
      // only check for valid balls
      if(!myBall(b) || !inPlay(b) || !(*myTargets[i]).ballNotUsed(b) ||
        (depth == 0 && b != 0))     continue;

      // generates ways to the  target point directly, per rail or by kiss
      // if the ball used is the cue ball, the shot is saved to doneShots
      ballToTargetWays(b, ballpos, targetPoint,
          (b == 0 ? doneShots : undoneShots), direct);
    }
  }

  if(depth != 0)
    generateShotTreeHelper(doneShots, undoneShots, depth-1, direct);
}
```

**Listing 2.2:** Excerpt of the code generating the shots

## 2.3 Analyzing the table state for position play

The reason for calculating the trajectory of the cue ball after the targeted ball has been struck is that the position at which the cue ball comes to rest greatly influences the possibility to form a shot in the following turn. Ideally, the cue ball comes to rest at a point from where a very easy shot is possible. In this case it is said that the cue ball 'has position' or 'has shape' [9]. JPool analyzes the path formed in the branch of the step tree and determines good halting positions along this path. Ultimately, as most shots can be played at a range of velocities, the fitting velocity for both pocketing the ball and causing the cue ball to rest in position is determined.

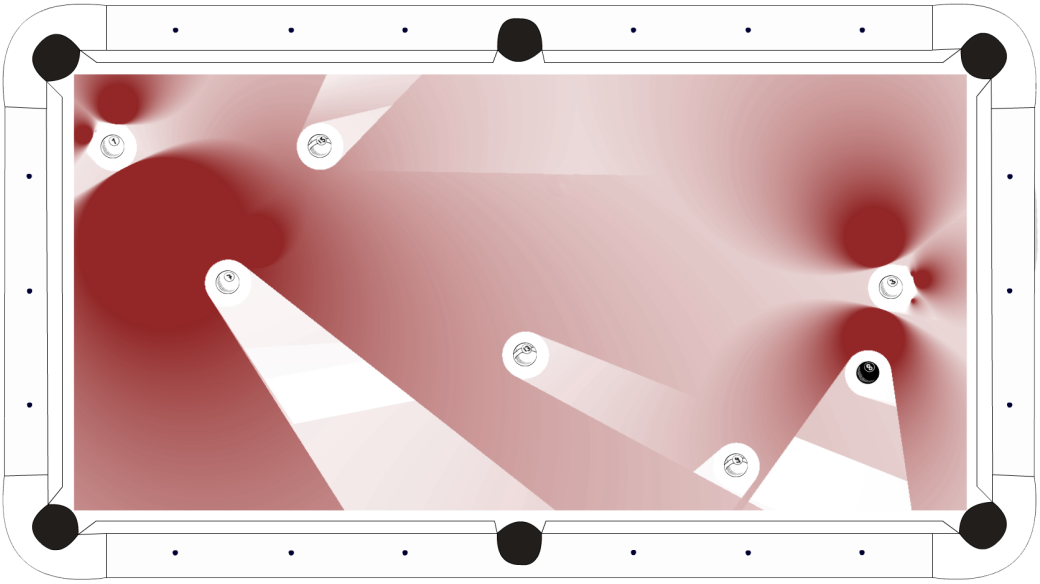


**Figure 2.6:** Position zones determined iteratively for a player playing the solid balls

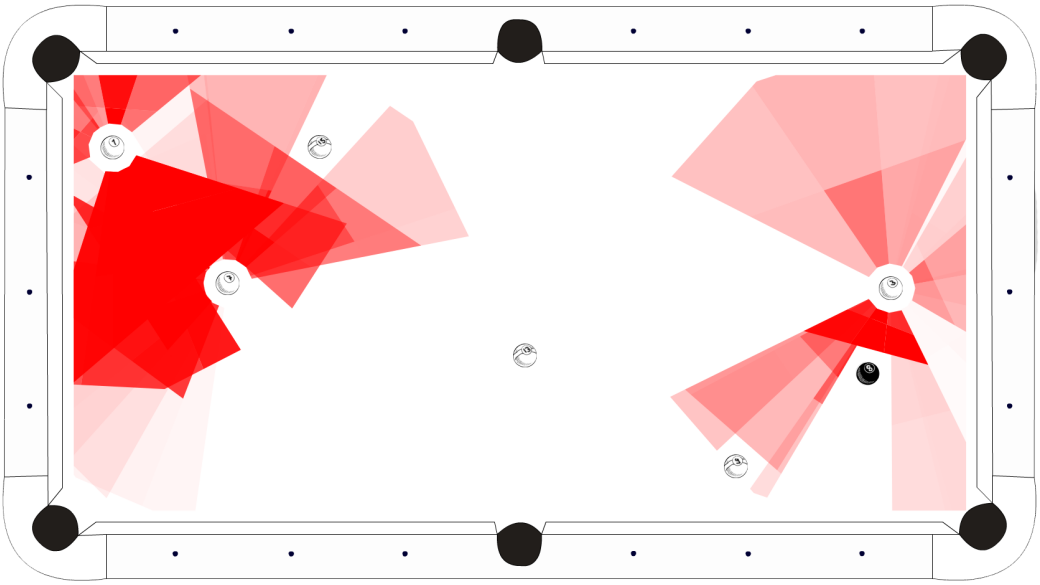


**Figure 2.7:** Approximation of the position zones shown in figure 2.6

### 2.3.1 Pixelwise rating of position zones

For a given table state, the position rating at a point $A$ on the table is calculated using the difficulty of the easiest shot available when pretending the cue ball were positioned at $A$. Figure 2.6 shows a random table state that has been rated pixel-wise in the described manner. Darker spots mean a better position, i.e. the big dark spot in the top left corner of the table indicates a big zone from which ball 1 can easily be pocketed in the top left corner pocket. Lighter spots indicate positions from where the player would most likely not have an easy shot at his disposal. These spots are used to minimize the position of the enemy player in the event of a safety shot.

JPool uses only direct shots to rate a given position. This is done for three reasons: first, when rating many positions on the table, speed is an important factor and direct shots can be formed faster than more complicated shots. Second, direct shots are generally easier to execute – more complicated shots are not likely to have a great influence on the position rating. And third, as positioning is targeted towards finding good spots for the cue ball to rest at for the next shot, changes in the positioning of balls involved can nullify potential position zones that have been found. Therefore, the less balls are involved in the rating of a point, the smaller the chance that one of the balls is moved during the shot that is to be executed. JPool filters which zones are considered in later steps to ignore zones targeting balls that are to be moved in the planned shot.

PoolMaster calculates a table rating in a similar way. It also aims to optimize the position of the cue ball after a shot and does so by identifying the best (and worst) spots on the table, using an optimization function to find a shot both pocketing the targeted ball and positioning the cue ball on one of the found best spots. In [5, p. 6 - 8], the creators of PoolMaster investigated whether it is best to take the sum, the average or the maximum of the difficulties of found shots for a position on a table as base for the position rating and came to the conclusion that both the average and the maximum are suitable since their value is independent of the number of balls involved. Additionally, they identify the maximum function to result in a more aggressive play. JPool uses the maximum function to rate a position on the table – not because JPool aims to be an aggressive player, but because only a single shot can be executed from the attained position before another position is reached and a multi-shot rating is attained using the search algorithm presented in section 4.1.2. Both CueCard and PickPocket do not rate the table for position play.

### 2.3.2 Shape zone approximation

While the pixel wise rating for position play is a useful technique both for visualization and rating single positions on the table, it is not suited for JPool's position play calculations. As JPool aims for zones on the table, these would have to be identified on the pixel wise rating, i.e. using pattern recognition algorithms. Additionally, pixel wise rating is a rather slow process taking several seconds on a current machine. For these reasons, JPool uses a polygon-based approximation of the table ratings, making use of the fact that position zones are usually very close to the balls in question. The zones generated are called shape zones. See figure 2.7 for an example approximating the zones shown in figure 2.6.

For each of the current players' balls and each pocket it is checked if the path for the ball into the pocket is clear. If this is the case, a polygon grid is created on the side of the ball facing away from the pocket (much like a ghostball), spanning over 60 cm in 45 degrees to the left and to the right. Figure 2.8 shows an example of the polygon grid created on the side of a ball. The polygon grid is altered in 2 ways. First, it is clipped to the playable areas of the table surface. Second, for every ball placed inside of the grid, 3 cuts through the grid are made: one in front, one to the left and one to the right of the ball. These cuts cut out the area of the shape zone that is blocked by the ball.

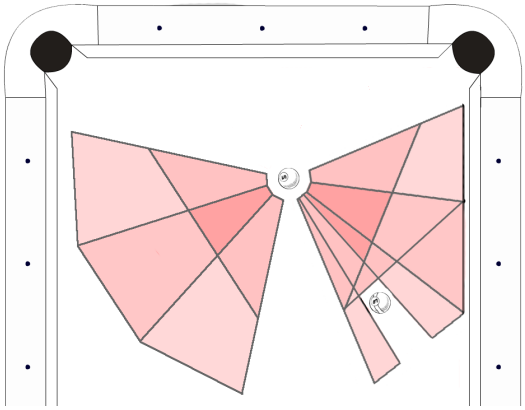


**Figure 2.8:** Polygons created for shape zone approximation

Each zone is rated by the average position rating of all its corner points and the its center point. Zones are filtered by both their rating and their area, removing small or bad areas from the approximation. For each shape zone, the number of the ball and pocket it is aiming for is stored. This way the shape zones considered by a shot may be filtered, preventing aiming for shape zones of balls that are to be moved or pocketed in the current shot. Note that multiple shape zones may overlap due to relatively close ball placement – this does not interfere with JPool's aiming process because multiple crossed shape zones are grouped together. Neither PoolMaster nor PickPocket or CueCard use any kind of shape zone approximation.

### 2.3.3 The cue ball's passage through shape zones

As the intention behind calculating shape zones is taking aim for them with the cue ball after its intended strike collision, the zones that are crossed by the path of the cue ball need to be identified. This is done using a simple line-polygon overlap detection algorithm. As a shot is executed with noise, the halting point of the cue ball cannot be determined with certainty and does not necessarily lie on the calculated path. Therefore, the halting position of the cue ball within a shape zone is approximated using a crossing zone.

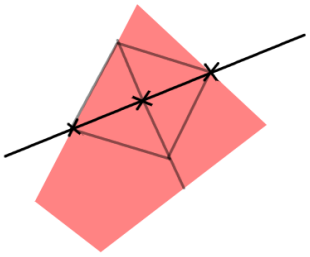


**Figure 2.9:** Rhombus formed on shape zone crossing

For each shape zone crossed by the cue ball's path the points of entry and exit are calculated. Furthermore, the center point of the crossing and the tolerance to the sides at the center point are calculated. As the noise window of the cue ball is equally distributed to both sides, a rhombus is formed as illustrated in figure 2.9. This rhombus approximates the halting area for a single shape zone.
The simplification of the shape of the halting area to a rhombus is done for three reasons: first, calculations with rhombuses are a lot simpler (and therefore faster) than calculations with the actual halting areas. These would need to be acquired using repeated simulation of the shot in question. Second, as multiple rhombuses are merged to polygons in the next

step, the actual shape of the original crossing zones is greatly altered. And third, aiming will focus on points on the precalculated path of the cue ball and the shape of the generated polygons is solely used for rating the probability of success of a shot.

Crossing zones acquired by this method may overlap or be very small. This makes aiming for the zones both hard and time consuming since JPool would attempt to aim at multiple zones representing virtually the same position on the table. Therefore, rhombuses are combined to polygons if they are overlapping and their rating differs by no more than 30%. Figure 2.10 shows an example shot pocketing ball 3 in the bottom right pocket. After the collision with ball 3, the path of the cue ball extends over the table, ultimately reaching shape zones for balls 1 and 7. The crossing of these zones is marked by crossing zones. Note that the shape zones for ball 3 are ignored since they will no longer be present in the following shot. Each crossing zone is rated by its area and the average rating of the shape zones it crosses. JPool will aim for those areas with a rating above a certain threshold (currently, a rating above 0.4 is required).

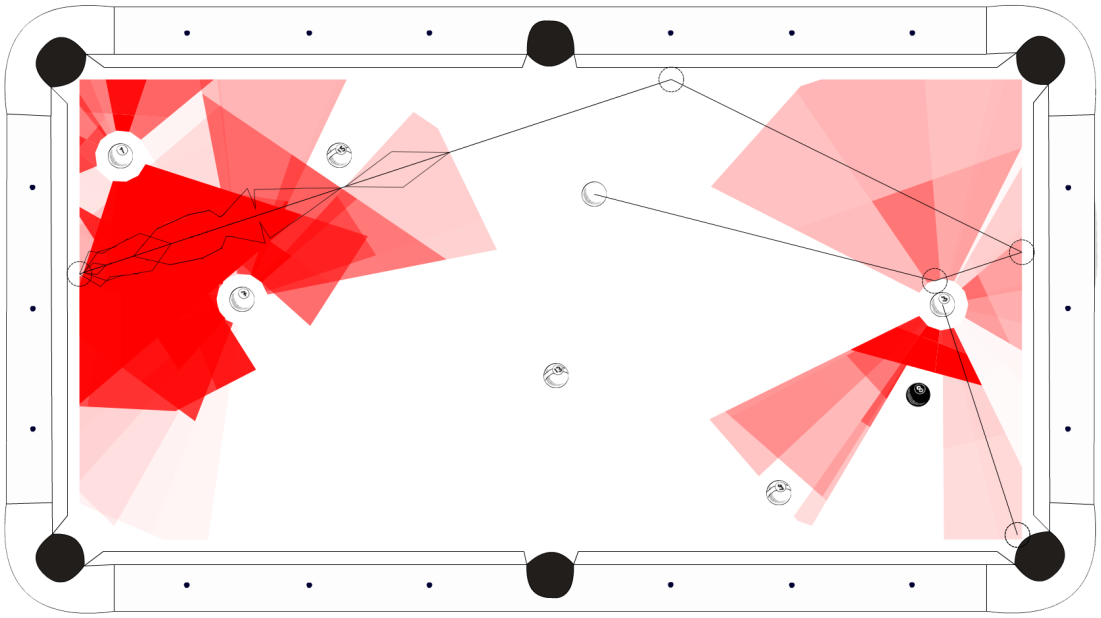


**Figure 2.10:** Crossing zones created for shape zone approximation on a sample shot

# 3 The physics of pocket billiards

After generating a shot geometrically, it has to be translated to shot parameters. As JPool aims to compete in the FastFiz-framework, it uses the same physics model that is used in FastFiz.[1] This chapter gives a brief overview of the principles and equations used by FastFiz and describes how JPool uses these equations to predict and enhance the outcome of possible shots, ultimately calculating a set of shot parameters causing the shot in question.

Both PickPocket [17, p. 27] and CueCard [2, p. 2] solely calculate the aiming direction $\varphi$ and determine a minimal $V_0$ for the ball to be pocketed (PickPocket uses a lookup table), other parameters are altered randomly. Early versions of PoolMaster use an optimization algorithm to determine the initial velocities of the cue ball; however, PoolMaster worked under the assumption that the cue stick is horizontal, limiting $\theta$ to 0 [4, p. 8].
JPool uses another approach: first, random variations of shot parameters for a geometrically determined aiming direction are simulated. This is similar to PickPocket's and CueCard's technique; however, these parameters are acquired and enhanced using physical calculations, both optimizing the probability of success of a shot and causing the cue ball to halt at a desired point on the table, i.e. a crossing zone.

The equations and physical considerations presented in section 3.1 have either been presented in [13] (which is based on [16]), or have directly been extracted from the FastFiz source code which is available at [19]. Section 3.2 presents equations used by JPool that are based on and derived from those used by FastFiz. All physical constants are listed in appendix 1, common variable-names used are listed in appendix 2.

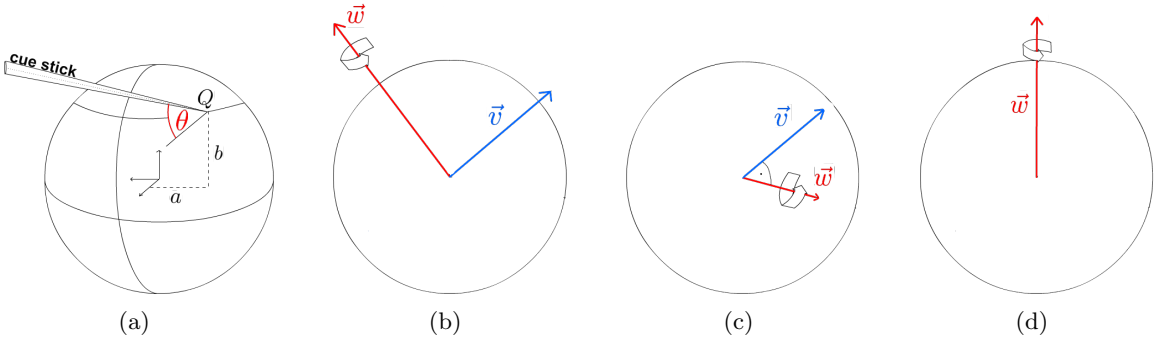



**Figure 3.1:** Impact of the cue stick on the ball at the point Q (a) and ball velocity dynamics in the sliding state (b) in the rolling state (c) and in the spinning state (d)

[1] FastFiz makes some simplifying assumptions to handle the complexity of the topic. These assumptions apply for JPool as well.

## 3.1 FastFiz: event-based simulation of shots

As described in [13], FastFiz uses an event-based simulation to compute the outcome of a shot. In this type of simulations all future events for a given table state are calculated, ordered by their time of occurrence. The balls are moved to the time of the earliest event, the event is handled and the process starts again until all balls stop moving due to friction. This type of simulation is faster and more precise than a classical frame-based simulation. In general, the events on the table can be described by the following ruleset:

1. For a table state, a shot consisting of five parameters is given and the resulting motion of the cue ball is calculated as described in section 3.1.1
2. All balls with any velocity move under the influence of friction (as described in section 3.1.2). In the process, the following events can trigger:
   a) A ball changes its movement state as described in section 3.1.2
   b) If two balls collide, they react as described in section 3.1.4
   c) If a ball collides with a cushion, it reacts as described in section 3.1.5
   d) If a ball is pocketed it is taken off the table and is no longer considered
   e) If all balls stop moving the shot execution is complete

Note that most events considered by FastFiz resemble the types of shot steps used in geometrical shot generation (see section 2.2).

### 3.1.1 Impact of the cue-stick on the ball

As described in section 1.3.2, shots in FastFiz consist of 5 parameters: $a$, $b$, $V_0$, $\theta$ and $\varphi$. In the first step of simulating a shot, these parameters are translated to the initial velocities of the cue ball. As suggested by [16], FastFiz models ball movement using two types of velocities: the linear velocity $\vec{v}$ and the angular velocity $\vec{w}$. The linear velocity defines the linear displacement of the center of the ball whereas the angular velocity defines the angular displacement (or spin) of the ball in counter-clockwise direction, not necessarily causing movement on the center of the ball. Let $A$ be the point at which the ball starts its movement and let $\vec{v}_A$ and $\vec{w}_A$ be its initial velocities.

FastFiz makes the simplifying assumption that the cue stick hits the ball at a single point, ignoring the shape of the tip of the cue stick. The point of contact between the cue-stick and the ball $Q = (a, c, b)$ can easily be calculated as it must be on the ball's surface (see figure 3.1(a)), therefore the distance between $Q$ and the center of the ball equals $R$. This results in:

$$c = \sqrt{R^2 - a^2 - b^2} \tag{3.1}$$

FastFiz calculates the initial linear velocity $\vec{v}_A$ using Newton's Second Law of Motion. The force $\vec{F}$ exerted on the ball is simplified to $\vec{F} = m\vec{v}_A$ (where $m$ is the ball mass) by assuming that the duration of the impact is close to zero. To derive this equation to $\vec{v}_A$, the magnitude of the force $F$ in terms of the impact parameters is given as:

$$F = \frac{2mV_0}{1 + \frac{m}{M} + \frac{5}{2R^2}(a^2 + b^2\cos^2\theta + c^2\sin^2\theta - 2bc\cos\theta\sin\theta)} \tag{3.2}$$

Where $M$ is the mass of the cue stick. This results in the initial linear velocity of the cue ball:

$$\vec{v}_A = \left(0, \frac{-F}{m}\cos\theta, \frac{-F}{m}\sin\theta\right) \tag{3.3}$$

The $\hat{z}$-component of $\vec{v}_A$ describes a force going *into* the table, sometimes resulting in jumping balls. For simplicity, FastFiz ignores this component, limiting ball movement to the surface of the table. The initial angular velocity of the cue ball is calculated using the following equation:

$$\vec{w}_A = I\left(-cF\sin\theta + bF\cos\theta, aF\sin\theta, -aF\cos\theta\right) \tag{3.4}$$

In a final step, these vectors are rotated for $\vec{v}_A$ to point along the direction of $\varphi$.

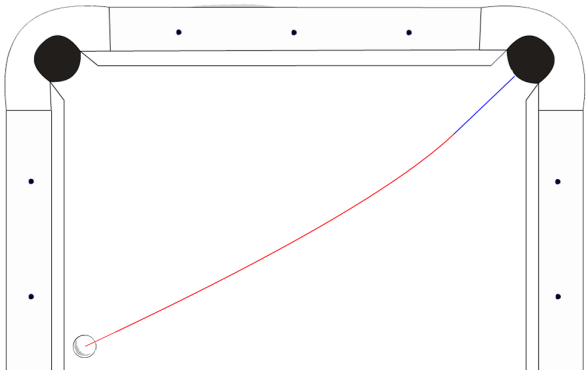
### 3.1.2 Ball movement

The movement of the ball can be divided in 3 phases. When first hit, a ball starts moving in the direction it got struck but it does not necessarily spin in the corresponding direction. When the ball spins in a direction other than the direction of movement, the ball is considered to be sliding across the table surface. In this state, curvilinear motion is possible. Due to friction with the table cloth the ball eventually starts rolling. Only if the ball is spinning in the direction of movement and travels exactly its perimeter per full revolution it is rolling across the table surface. A ball is spinning if it is only rotating around its center with no linear velocity. Figure 3.2 gives an example of sliding and rolling motion of a ball.

To determine the movement phase of a ball at the time $t$, the relative velocity $\vec{u}$ of a ball is calculated:

$$\vec{u}(t) = \vec{v}(t) + R\hat{z} \times \vec{w}(t) \tag{3.5}$$

The ball is in the sliding state if $\vec{u}(t) \neq 0$ and it is rolling if $\vec{u}(t) = 0$. FastFiz limits all movement to the surface of the table; jump-shots are not possible. Note that this results in $\vec{u}(t)_z = 0$ since $\vec{v}(t)$ lies in the $\hat{x}\hat{y}$-plane and $\hat{z} \times \hat{z} = 0$.



| | $x$ | $y$ | $z$ |
|---|---|---|---|
| $\vec{v}_A$ | 0.390659 m | 0.182167 m | 0 m |
| $\vec{w}_A$ | 2.745495 m | -20.311257 m | -10.557965 m |
| $\vec{u}_A$ | 0.965819 m | 0.259215 m | 0 m |
| $\vec{v}_B$ | 0.113215 m | 0.107704 m | 0 m |
| $\vec{w}_B$ | -3.769191 m | 3.962051 | 0 m |
| $A$ | 0 m | 0 m | 0 m |
| $B$ | 0.368869 m | 0.212205 m | 0 m |
| $C$ | 0.428983 m | 0.269392 m | 0 m |
| $\tau_s$ | 1.464131 s | | |
| $\tau_r$ | 1.061928 s | | |

**Figure 3.2:** Sliding (red) and rolling (blue) motion of a ball struck with parameters ($V_0 = 0.5, \varphi = 25, \theta = 30, a = 0.008, b = 0.001$) and a list of all the relevant velocities, coordinates and times of the ball motion (table image is out of scale and was only added for illustrational purpose)

#### 3.1.2.1 The sliding state

In the sliding state, the ball moves along the direction of its linear velocity and spins counter-clockwise around the direction of its angular velocity (see figure 3.1(b)). Friction is the only force considered by FastFiz to be influencing the motion of the balls. Based on Newton's Laws of motion, FastFiz uses the following equations to calculate a sliding ball's linear and angular velocities at the time $t$:

$$\vec{v_s}(t) = \vec{v}_A - tg\mu_s\hat{u}_A \tag{3.6}$$

$$\vec{w_s}(t) = \vec{w}_A - t\frac{5g\mu_s}{2R}\left(\hat{z} \times \hat{u}_A\right) \tag{3.7}$$

Note that equation 3.7 does not change $\vec{w_s}(t)_z$ since $(\hat{z} \times \hat{u}_A)_z = 0$. To compensate for that, FastFiz adjusts the $z$-component as if the ball were in the spinning state:

$$\vec{w_s}(t)_z = \vec{w}_{A_z} - t\frac{5g\mu_{sp}}{2R} \tag{3.8}$$

FastFiz checks this value and stops its evolvement once it passes zero from either side. Using equations 3.5, 3.6 and 3.7, the time $\tau_s$ at which the sliding state ends can be determined using $\vec{u}(\tau_s) = 0$:

$$\tau_s = \frac{2|\vec{u}_A|}{7g\mu_s} \tag{3.9}$$

The position of the ball at a time $t$ is determined using the following equation:

$$\vec{r_s}(t) = A + \vec{v}_A t - \frac{1}{2}\mu_s g t^2 \hat{u}_A \tag{3.10}$$

#### 3.1.2.2 The rolling state

A ball enters the rolling state at the position $B = \vec{r}_s(\tau_s)$ and with the velocities $\vec{v}_B = \vec{v}_s(\tau_s)$ and $\vec{w}_B = \vec{w}_s(\tau_s)$. Per definition, the relative velocity $\vec{u}_r$ is zero and the angular velocity in the rolling state $\vec{w}_r$ lies parallel to the surface of the table and perpendicular to the linear velocity $\vec{v}_r$ (see figure 3.1(c)). The linear velocity and the position of the ball at the time $t$ are calculated using:

$$\vec{v}_r(t) = \vec{v}_B - g\mu_r t\hat{v}_B \tag{3.11}$$

$$\vec{r}_r(t) = B + \vec{v}_B t - \frac{1}{2}\mu_r g t^2 \hat{v}_B \tag{3.12}$$

Note that the time $t$ in these equations starts at 0 and not at $\tau_s$. The rolling state ends when the linear velocity equals zero. As done in the sliding state, the time $\tau_r$ of the end of movement can be determined using:

$$\tau_r = \frac{|\vec{v}_B|}{g\mu_r} \tag{3.13}$$

This results in the final resting position of the ball $C = \vec{r}_r(\tau_r)$. In the rolling state, the angular velocity is constrained. It does not affect future movements of the ball; however, it does play a role in the calculation of collisions. Its magnitude is calculated using the following equation:

$$|\vec{w}_r(t)| = \frac{|\vec{v}_r(t)|}{R} \tag{3.14}$$

### 3.1.3 Detecting collisions between balls

FastFiz detects ball collisions using a simple distance-metric applied to the positions of two balls $b_1$ and $b_2$ at the time $t$ (see [13]):

$$\vec{d}(t) = \vec{r}_{b_2}(t) - \vec{r}_{b_1}(t) \tag{3.15}$$

A collision between two balls occurs when the distance between their positions equals $2R$. The time of collision $\tau_C$ can be determined using equation 3.16, resulting in the following quartic polynomial:

$$|\vec{d}(t)| = \sqrt{\vec{d}(t)_x^2 + \vec{d}(t)_y^2} = 2R \tag{3.16}$$

$$(a_x^2+a_y^2)t^4+(2a_xb_x+2a_yb_y)t^3+(b_x^2+2a_xc_x+2a_yc_y+b_y^2)t^2+(2b_xc_x+2b_yc_y)t+c_x^2+c_y^2-4R^2 = 0 \tag{3.17}$$

Where $a$, $b$ and $c$ are determined by using equations 3.10 or 3.12, depending on which state of movement the colliding balls are in (sliding or rolling). The quartic polynomial is solved for $t$ using a function provided by the *GSL*-library. Since a quartic polynomial has four possible roots, FastFiz chooses the smallest positive real root of equation 3.17 – if there is no positive real root, no collision occurs between the balls in question.

### 3.1.4 Handling collisions between balls

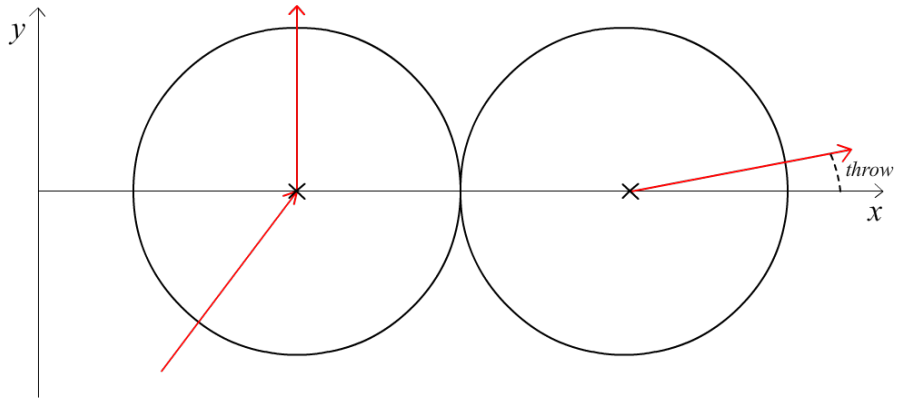



**Figure 3.3:** Example of the dynamics of linear velocities during a collision between two balls in top-down-view

Let two balls $b_1$ and $b_2$ collide at the positions $\vec{r}_{b_1}$ and $\vec{r}_{b_2}$ with the linear and angular velocities $\vec{v}_{b_1}$, $\vec{v}_{b_2}$, $\vec{w}_{b_1}$ and $\vec{w}_{b_2}$. Since at least one ball has to be moving for a collision-event to occur, let $\vec{v}_{b_1} \neq 0$. For simplicity, FastFiz aligns the two balls along the $x$-axis. The relative velocity of the balls at the point of collision is calculated using:[2]

$$\vec{u}_c = (\vec{v}_{b_1} - \vec{v}_{b_2}) - (R\hat{x} \times (\vec{w}_{b_1} + \vec{w}_{b_2})) \tag{3.18}$$

The new linear velocities of the balls are calculated using the following equations:

$$\vec{v'}_{b_1} = \left(\vec{v}_{b_{2_x}}, \vec{v}_{b_{1_y}} - \mu_c \vec{v}_{b_{1_x}} \hat{u}_{c_y}, 0\right) \tag{3.19}$$

[2]These equations were directly extracted from the FastFiz-source code.

$$\vec{v'}_{b_2} = \left(\vec{v}_{b_{1_x}}, \vec{v}_{b_{2_y}} - \mu_c \vec{v}_{b_{1_x}} \hat{u}_{c_y}, 0\right) \tag{3.20}$$

Consider $\vec{v}_{b_2} = 0$. This is the case for all collisions considered by JPool. Note that in this case $\vec{v'}_{b_1}$ has no $x$-component, constraining the cue ball's linear velocity after its first contact to a direction perpendicular to the axis between the two colliding balls. The new linear velocity of $b_2$ has a strong $x$-component and a small $y$-component, throwing it slightly off the course used for the geometric considerations done in section 2.1.2. This effect is called *throw* and is being considered by professional human pool players, rendering previously considered *ghostballs* imprecise [9, p. 57]. Figure 3.3 illustrates the dynamics of the linear velocities during ball collisions.

The angular velocity transferred from one ball to the other depends highly on the movement state of $b_2$. If $b_2$ is moving, FastFiz simply makes both balls roll, setting their angular velocities perpendicular to their new linear velocities. If $b_2$ is stationary, the post-collision angular velocities are determined using the following equations:

$$\vec{w}_{add} = \frac{-2.5\mu_c \vec{v}_{b_{1_x}}}{1.000575R}(\hat{x} \times \hat{u}_c) \tag{3.21}$$

$$\vec{w'}_{b_1} = \vec{w}_{b_1} - \vec{w}_{add} \tag{3.22}$$

$$\vec{w'}_{b_2} = \vec{w}_{b_2} + \vec{w}_{add} \tag{3.23}$$

### 3.1.5 Collisions between a rail and a ball

Let $(\vec{v}, \vec{w})$ be the initial and $(\vec{v'}, \vec{w'})$ be the post-collision velocities of the ball colliding with a rail. FastFiz simplifies the calculation of rail collisions. While in actual pool the spin of a ball greatly influences its trajectory after hitting a rail (see [9, p. 107]), FastFiz divides the angular velocity by 10 and uses a simple reflection for the linear velocity:[3]

$$\vec{v'} = \begin{cases} (0.9\vec{v}_x, -0.6\vec{v}_y, 0)\,, & \text{if rail is horizontal} \\ (-0.6\vec{v}_x, 0.9\vec{v}_y, 0)\,, & \text{if rail is vertical} \end{cases} \tag{3.24}$$

$$\vec{w'} = 0.1\vec{w} \tag{3.25}$$

[3] These equations were directly extracted from the FastFiz-source code.

## 3.2 JPool: optimizing geometrically determined shots

To successfully execute a shot, a set of goals (i.e. reaching ghostball position or pocketing a ball) have to be achieved. JPool determines these goals geometrically and saves them as a sequence of steps (see section 2.2) to enhance them using physical considerations like *throw* or curvilinear motion in the sliding state. This section shows calculations done by JPool for each possible type of shot steps:

1. Reaching a position on the table (includes pocketing a ball)
2. Ball-Ball-Collisions: causing the cue ball to have specific post-collision velocities (strike collision)
3. Ball-Ball-Collisions: causing the object ball to have specific post-collision velocities (kiss collision)
4. Ball-Ball-Collisions: both balls are aimed for
5. Rail-Ball-Collisions
6. Cue stick-Impact

Ultimately, JPool calculates the velocities of acting balls for each step of a shot using the calculations shown in this section. The initial velocities of the cue ball are translated to shot parameters.

### 3.2.1 Reaching a target point

The most basic task for the physics-system in JPool is directly reaching a target point on the table. Since a point can be reached in an unlimited number of ways, JPool always defines the velocities the ball should have when it reaches the desired position. This can not always be done with certainty - for example, the velocity a ball has when being pocketed is free of constraints and has to be guessed. This is done by using the velocities that occurred during the simulation of a naive version of the shot in question.

Let $D$ be the targeted *ghostball*-point on the table and $(\vec{v}_D, \vec{w}_D)$ the desired linear and angular velocities the ball should have at the point $D$. For simplicity, the center of the coordinate system is moved to the starting point of the ball in question, leaving $A = (0, 0, 0)^T$. Depending on the movement state the ball should have at $D$, the initial velocities $(\vec{v}_A, \vec{w}_A)$ are calculated according to section 3.2.1.1 or 3.2.1.2.

#### 3.2.1.1 Reaching a point in the sliding state

If the desired position is in the sliding state, there is only one way to reach the desired combination of position and velocities. JPool uses a set of equations to express the distance of a ball to its target point relative to the time $t$. Newton's Method is used to determine the initial velocities needed.

Note that in the sliding state, $\vec{v}_D$ and $\vec{w}_D$ must be such that $\vec{u}_D \neq 0$. Equations 3.6 and 3.10

can be rearranged to $\hat{u}_A$. This yields an equation for $\vec{v}_A$, the initial linear velocity needed, with only $t$ unknown:

$$\hat{u}_A = \frac{D - A - \vec{v}_A t}{-\frac{1}{2}\mu_s g t^2} = \frac{\vec{v}_D - \vec{v}_A}{-\mu_s g t} \tag{3.26}$$

$$\vec{v}_A = \frac{2D}{t} - \vec{v}_D \tag{3.27}$$

Substituting $\vec{v}_A$ from equation 3.27 in equation 3.26, $\hat{u}_A$ may also be expressed as:

$$\hat{u}_A = \frac{D - \vec{v}_A t}{-\frac{1}{2}\mu_s g t^2} = \frac{D - \left(\frac{2D}{t} - \vec{v}_D\right) t}{-\frac{1}{2}\mu_s g t^2} = \frac{D - \vec{v}_D t}{\frac{1}{2}\mu_s g t^2} \tag{3.28}$$

Using equations 3.28 and 3.7, the initial angular velocity $\vec{w}_A$ can be expressed as

$$\vec{w}_A = \vec{w}_D + \frac{5\mu_s g t}{2R}\left(\hat{z} \times \hat{u}_A\right) = \vec{w}_D + \frac{5}{Rt}\begin{pmatrix} \vec{v}_{D_y} t - D_y \\ -\vec{v}_{D_x} t + D_x \\ 0 \end{pmatrix} \tag{3.29}$$

Note that the use of all three equations 3.6, 3.7 and 3.10 ensures that the linear and angular velocities and the position of the ball at the time $t$ match the desired values $D$, $\vec{v}_D$ and $\vec{w}_D$. Using $\vec{r}_s(t) = D$, the vector $\vec{d}(t)$ describes the distance of the ball to its destination point $D$ at the time $t$ using equations 3.10, 3.29 and 3.27:

$$p = \sqrt{\left|\left(\vec{w}_{D_x} + \frac{5\left(t\vec{v}_{D_y} - D_y\right)}{Rt}\right)R - \vec{v}_{D_y} + \frac{2\,D_y}{t}\right|^2 + \left|-\left(\vec{w}_{D_y} - \frac{5\left(t\vec{v}_{D_x} - D_x\right)}{Rt}\right)R - \vec{v}_{D_x} + \frac{2\,D_x}{t}\right|^2} \tag{3.30}$$

$$\begin{aligned} \vec{d}(t) &= \vec{v}_A t - \tfrac{1}{2}\mu_s g t^2 \tfrac{\vec{v}_A + R\hat{z}\times\vec{w}_A}{|\vec{v}_A + R\hat{z}\times\vec{w}_A|} - D \\ &= \begin{pmatrix} \left(\left(\vec{w}_{D_y} - \frac{5\left(t\vec{v}_{D_x} - D_x\right)}{Rt}\right)R + \vec{v}_{D_x} - \frac{2\,D_x}{t}\right) \\ -\left(\left(\vec{w}_{D_x} + \frac{5\left(t\vec{v}_{D_y} - D_y\right)}{Rt}\right)R - \vec{v}_{D_y} + \frac{2\,D_y}{t}\right) \end{pmatrix} \frac{g\mu_s t^2}{2\,p} - \left(\vec{v}_D - \tfrac{2\,D}{t}\right)t - D \\ &= (R\vec{w}_{D_\perp} - 4\vec{v}_D + \tfrac{3D}{t})\tfrac{g\mu_s t^2}{2p} - \vec{v}_D t + D \\ &= (R\vec{w}_{D_\perp} - 4\vec{v}_D)\tfrac{g\mu_s}{2p}t^2 + (3D\tfrac{g\mu_s}{2p} - \vec{v}_D)t + D \end{aligned} \tag{3.31}$$

Where $\vec{w}_{D_\perp} = (\vec{w}_{D_y}, -\vec{w}_{D_x}, 0)^T$. Since $\vec{d}(t)$ is the distance between the ball and its destination, the time $t$ at which the ball traverses its target point is calculated by solving $\vec{d}(t) = 0$, implicating both $\vec{d}(t)_x = 0$ and $\vec{d}(t)_y = 0$:

$$\begin{aligned} 0 &= \vec{d}(t)_x + \vec{d}(t)_y \\ &= (R(\vec{w}_{D_y} - \vec{w}_{D_x}) - 4(\vec{v}_{D_x} + \vec{v}_{D_y}))\tfrac{g\mu_s}{2p}t^2 + (3\tfrac{g\mu_s}{2p}(D_x + D_y) - (\vec{v}_{D_x} + \vec{v}_{D_y}))t + D_x + D_y \end{aligned} \tag{3.32}$$

Note that equation 3.32 is not quadratic since $p$ depends on $t$. Therefore, $t$ is approximated using equation 3.32 and Newton's Method as described in [7]. Newton's Method uses the first derivative of an equation to efficiently determine a possible root with a complexity of $O(h^2)$ where $h$ is the desired approximation level. The derivative of equation 3.32 has been determined machinally. After $t$ has been calculated, equations 3.27 and 3.29 are used to determine the needed velocities to reach the target point. The precision of this method is only limited by floating point accuracy and the depth chosen for the approximation algorithm.

**Example** Consider the shot illustrated in figure 3.2. At $t = \frac{\tau_s}{2} \approx 0.732065$, the ball has the position $D$ and velocities $(\vec{v}_D, \vec{w}_D)$ as listed in figure 3.4. Figure 3.4 also shows a plot of equation 3.32 solely dependant on $D$, $\vec{v}_D$ and $\vec{w}_D$. The Newton's Method algorithm returns $t' = 0.732066$ after 20 iterations. Inserting $D$, $\vec{v}_D$, $\vec{w}_D$ and $t'$ in equations 3.27 and 3.29 returns the initial velocities deciphered in figure 3.4, which mirror the velocities in figure 3.2. Note that $\vec{w'}_{A_z} = 0$ does not influence the trajectory of the ball.

| | $x$ | $y$ | $z$ |
|---|---|---|---|
| $D$ | 0.235211 m | 0.119730 m | 0 m |
| $\vec{v}_D$ | 0.251937 m | 0.144936 m | 0 m |
| $\vec{w}_D$ | 6.002839 m | -32.447911 m | 0 m |
| $\vec{v}'_A$ | 0.390659 m | 0.182167 m | 0 m |
| $\vec{w}'_A$ | 2.745501 m | -20.311268 m | 0 m |

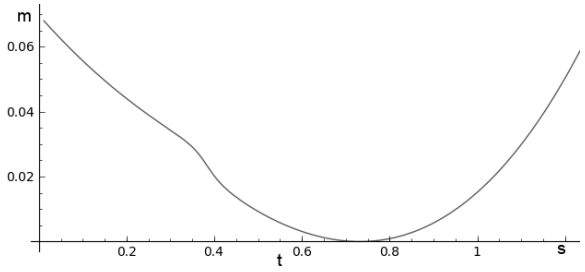



**Figure 3.4:** The desired velocities and position $D$ for the shot illustrated in figure 3.2 for $t = 0.732065$ and a plot of equation 3.32 with these values

#### 3.2.1.2 Reaching a point in the rolling state

Since a ball is always first sliding and then rolling, the initial velocities needed to reach a destination in the rolling state are determined by calculating the point at which the ball leaves the sliding state. Ultimately, its initial velocities are determined using the equations presented in section 3.2.1.1. Let $B$ be the point at which the ball enters the rolling state, $(\vec{v}_B, \vec{w}_B)$ be the velocities the ball has at the point $B$ and $t_D$ be the time at which the ball reaches $D$. Since movement in the rolling state is linear, $B$ can not be determined with certainty. JPool calculates one possible shot that is passing $D$ with $\vec{v}_D$ using a 3-dimensional modification of the Newtons's Method algorithm used in section 3.2.1.1.

Since there is no curvilinear motion possible in the sliding state, $\vec{v}_B$ and $\vec{v}_D$ point in the same direction, resulting in $\hat{v}_B = \hat{v}_D$. This can be used in equations 3.11 and 3.12 to determine $\vec{v}_B$ and $B$ with only $t_D$ unknown:

$$\vec{v}_B = \vec{v}_D + \mu_r g t_D \hat{v}_D \tag{3.33}$$

$$\begin{aligned} D &= B + (\vec{v}_D + \mu_r g t_D \hat{v}_D)\, t_D - \tfrac{1}{2}\mu_r g t_D^2 \hat{v}_D \\ B &= D - \vec{v}_D t_D - \tfrac{1}{2}\mu_r g t_D^2 \hat{v}_D \end{aligned} \tag{3.34}$$

Note that $\vec{w}_B = (\frac{-\vec{v}_{B_y}}{R}, \frac{\vec{v}_{B_x}}{R}, 0)^T = -\frac{1}{R}\vec{v}_{B_\perp}$ since it has to be perpendicular to the linear velocity. $B$ marks the point at which the ball switches from the sliding to the rolling state and is a valid target point for the calculations done in section 3.2.1.1. Inserting $B$ from

equation 3.34 as $D$ in equation 3.31 yields a function $f : (\mathbb{R}, \mathbb{R}) \to \mathbb{R}$ for $t_D$ and $t$:

$$\begin{aligned} f(t_D,t) &= (R\vec{w}_{B_\perp} - 4\vec{v}_B)\tfrac{g\mu_s}{2p}t^2 + (3B\tfrac{g\mu_s}{2p} - \vec{v}_B)t + B \\ &= ((\vec{v}_D + \mu_r g t_D \hat{v}_D)_\perp - 4\vec{v}_D - 4\mu_r g t_D \hat{v}_D)\tfrac{g\mu_s}{2p}t^2 \\ &\quad +(3D - 3\vec{v}_D t_D - \tfrac{3}{2}\mu_r g t_D^2 \hat{v}_D \tfrac{g\mu_s}{2p} - \vec{v}_D - \mu_r g t_D \hat{v}_D)t \\ &\quad +D - \vec{v}_D t_D - \tfrac{1}{2}\mu_r g t_D^2 \hat{v}_D \\ &= -\tfrac{1}{2}t_D^2 g\mu_r \hat{v}_D - \tfrac{1}{2}(4t_D g\mu_r \hat{v}_D - t_D g\mu_r \hat{v}_{D_\perp} + 4\vec{v}_D - \vec{v}_{D_\perp})\tfrac{g\mu_s}{p}t^2 \\ &\quad -\tfrac{1}{4}(3g\mu_s t_D^2 g\mu_r \hat{v}_D \tfrac{1}{p} + 4t_D g\mu_r \hat{v}_D + 12t_D\vec{v}_D - 12D + 4\vec{v}_D)t - t_D\vec{v}_D + D \end{aligned} \tag{3.35}$$

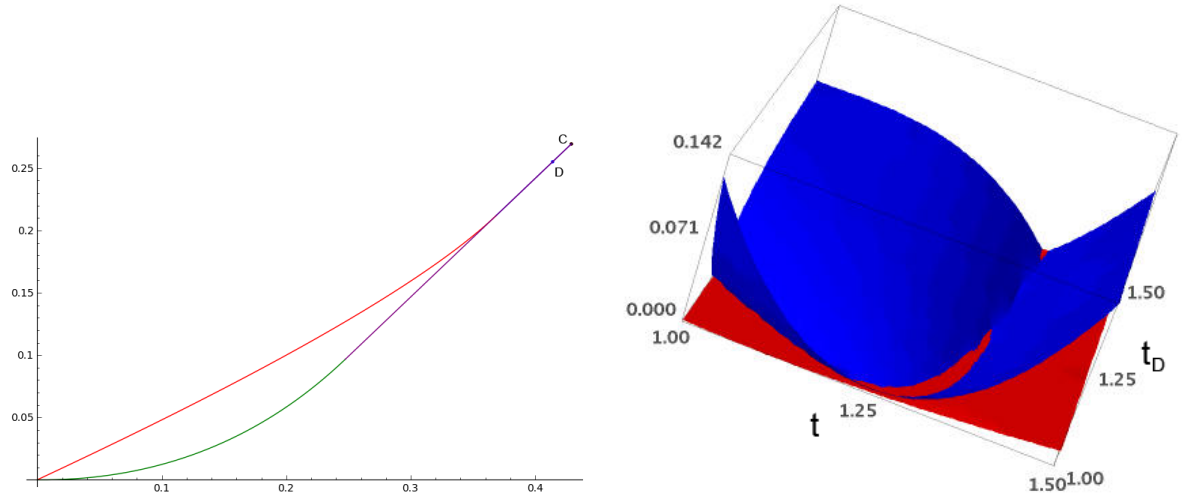
A valid solution of equation 3.35, consisting of two positive real values for $t$ and $t_D$ so that $f(t_D, t) = 0$, can be used to calculate initial shot velocities as it has been done in section 3.2.1.1; however, this equation only ensures that the ball is rolling at $B$ and not that it *starts* rolling at $B$. Since all equations assume that the ball is sliding prior to $B$ (equations for sliding motion are used), the actual simulation of the determined shot would greatly differ from the calculated outcome if the ball started rolling before it reaches $B$. To make sure that $B$ is the exact point at which the ball starts rolling, recall from equation 3.9 that given an initial relative velocity $\vec{u}_A$, the time $\tau_s$ at which the ball switches from the sliding to the rolling state is given as $\frac{2|\vec{u}_A|}{7g\mu_s}$. For $B$ to be the point where the sliding state ends, $\tau_s$ must equal t, resulting in the function $g : (\mathbb{R}, \mathbb{R}) \to \mathbb{R}$:

$$\begin{aligned} g(t_D,t) &= \tfrac{2|\vec{u}_A|}{7g\mu_s} - t \\ &\overset{\text{eq. 3.5}}{=} \tfrac{2}{7g\mu_s}\left|\vec{v}_A + R\hat{z}\times\vec{w}_A\right| - t \\ &\overset{\text{eq. 3.27, eq. 3.29}}{=} \tfrac{2}{7g\mu_s}\left|\tfrac{2B}{t} - \vec{v}_B + R\begin{pmatrix} -\vec{w}_{B_y} - \frac{5}{Rt}(-\vec{v}_{B_x}t + B_x) - t \\ \vec{w}_{B_x} + \frac{5}{Rt}(\vec{v}_{B_y}t - B_y) \\ 0 \end{pmatrix}\right| - t \\ &= \tfrac{2}{7g\mu_s}\left|\tfrac{2B}{t} - \vec{v}_B + R\left(-\vec{w}_{B_\perp} + \tfrac{5}{Rt}(\vec{v}_B t - B)\right)\right| - t \\ &= \tfrac{2}{7g\mu_s}\left|\tfrac{2B}{t} - \vec{v}_B + \vec{v}_{B_{\perp\perp}} + \tfrac{5}{t}(\vec{v}_B t - B)\right| - t \\ &= \tfrac{2}{7g\mu_s}\left|\tfrac{2B}{t} - \tfrac{5B}{t} + 5\vec{v}_B\right| - t \\ &= \tfrac{2}{7g\mu_s}\left|-\tfrac{3B}{t} + 5\vec{v}_B\right| - t \\ &\overset{\text{eq. 3.33, eq. 3.34}}{=} \tfrac{2}{7g\mu_s}\left|-\tfrac{3(\vec{v}_D + \mu_r g t_D \hat{v}_D)}{t} + 5(\vec{v}_D + \mu_r g t_D \hat{v}_D)\right| - t \\ &= \tfrac{2}{7g\mu_s}\left|\left(5 - \tfrac{3}{t}\right)(\vec{v}_D + t_D \breve{v}_D)\right| - t \end{aligned} \tag{3.36}$$

Note that this equation could not have been used in section 3.2.1.1 since the target point there does not necessarily mark the point at which the sliding state ends. To acquire a solution of $t$ and $t_D$ that satisfies both $f(t_D, t) = 0$ and $g(t_D, t) = 0$, a multidimensional version of Newton's Method is used as described in [7], approximating a solution for 2 functions with 2 unknowns using their Jacobian Matrix in $O(h^2)$.

**Example** Consider the shot illustrated in figure 3.2. At $t = \frac{\tau_r}{2} \approx 0.530964$ in the rolling state, the ball has the position $D$ and velocities $\vec{v}_D$ as listed in table 3.1 (note that $\vec{w}_D$ is not needed). Figure 3.5(b) shows a 3-dimensional plot of $f(t_D, t)$ and $g(t_D, t)$ for $t_D, t \in [1, 1.5]$

which is solely dependant on $D$ and $\vec{v}_D$. Note that there are multiple possible solutions satisfying $f(t_D, t) = g(t_D, t) = 0$.
The Newton's Method algorithm returns $(t'_D = 1.314162, t' = 1.046995)$ after 20 iterations with the starting values $(t_{D_0} = 0.5, t_0 = 0.5)$. Inserting $D$, $\vec{v}_D$, $t'_D$ and $t'$ in equations 3.33, 3.34, 3.27 and 3.29 returns the initial velocities deciphered in table 3.1 . These do not equal the original shot velocities; however, it is a valid solution since the ball does travel through $D$ with the desired velocities and stops at $C$ as illustrated in figure 3.5(a) and listed in table 3.1.




(a) (b)

**Figure 3.5:** (a) The original shot as illustrated in figure 3.2 (red / blue) and the calculated alternative shot (green / purple) (b) A 3D-plot of $f(t_D, t)$ in red and $g(t_D, t)$ in blue for $t_D, t \in [1, 1.5]$

| | $x$ | $y$ | $z$ |
|---|---|---|---|
| $D$ | 0.413954 m | 0.255096 m | 0 m |
| $\vec{v}_D$ | 0.056607 m | 0.053852 m | 0 m |
| $\vec{v}'_A$ | 0.276069 m | -0.002334 m | 0 m |
| $\vec{w}'_A$ | -23.125976 m | -0.058475 m | 0 m |
| Simulating this shot results the following positions: | | | |
| $B'$ | 0.247501 m | 0.096744 m | 0 m |
| $C'$ | 0.428983 m | 0.269392 m | 0 m |

**Table 3.1:** The desired velocities and position $D$ for the shot illustrated in figure 3.2 for $t = 0.530964$ in the rolling state

#### 3.2.1.3 Halting point of a ball

For a given set of initial velocities $(\vec{v}_A, \vec{w}_A)$ and a starting point $A$, the halting position $C$ of a ball can be calculated using equations 3.9, 3.10, 3.12 and 3.13:

$$\begin{aligned}
C \overset{\text{eq. 3.12}}{=}\ & B + \vec{v}_B t - \tfrac{1}{2}\mu_r g t^2 \hat{v}_B \\
\overset{\text{eq. 3.10}}{=}\ & A + \vec{v}_A \tau_s - \tfrac{1}{2}\mu_s g \tau_s^2 \hat{u}_A + \vec{v}_B \tau_r - \tfrac{1}{2}\mu_r g \tau_r^2 \hat{v}_B \\
=\ & A + \vec{v}_A \tau_s - \tfrac{1}{2}\tau_s \vec{u}_A + \vec{v}_B \tau_r - \tfrac{1}{2}\tau_r \vec{v}_B \\
=\ & A + \tau_s \left(\vec{v}_A - \tfrac{1}{7}\vec{v}_A - \tfrac{1}{7}R\hat{z} \times \vec{w}_A\right) + \tfrac{1}{2}\tau_r \left(\vec{v}_A - \tau_s g \mu_s \hat{u}_A\right) \\
=\ & A + \tau_s \left(\tfrac{6}{7}\vec{v}_A - \tfrac{1}{7}R\vec{w}_{A_\perp}\right) + \tfrac{1}{2}\tau_r \left(\vec{v}_A - \tfrac{2}{7}\vec{v}_A - \tfrac{2}{7}R\vec{w}_{A_\perp}\right) \\
=\ & A + \tau_s \left(\tfrac{6}{7}\vec{v}_A - \tfrac{1}{7}R\vec{w}_{A_\perp}\right) + \tfrac{1}{2}\tau_r \left(\tfrac{5}{7}\vec{v}_A - \tfrac{2}{7}R\vec{w}_{A_\perp}\right)
\end{aligned} \tag{3.37}$$

$$\begin{aligned}
\overset{\text{eq. 3.9, eq. 3.13}}{=}\ & A + \frac{2\sqrt{(\vec{v}_{A_x} - R\vec{w}_{A_y})^2 + (\vec{v}_{A_y} + R\vec{w}_{A_x})^2}}{7g\mu_s}\left(\tfrac{6}{7}\vec{v}_A - \tfrac{1}{7}R\vec{w}_{A_\perp}\right) \\
& + \frac{\sqrt{(\frac{5}{7}\vec{v}_{A_x} + \frac{2}{7}R\vec{w}_{A_y})^2 + (\frac{5}{7}\vec{v}_{A_y} - \frac{2}{7}R\vec{w}_{A_x})^2}}{2g\mu_r}\left(\tfrac{5}{7}\vec{v}_A - \tfrac{2}{7}R\vec{w}_{A_\perp}\right)
\end{aligned}$$

### 3.2.2 Collisions between balls

When collisions between balls are calculated, the intentions for both balls have to be met. JPool calculates targets for both the object ball and the cue ball to reach after the collision, requiring the collision to cause two sets of velocities. This section first shows techniques to calculate velocities needed pre-collision to cause a desired outcome for the object ball (see section 3.2.2.1) and the cue ball (see section 3.2.2.2) and then shows a way of achieving both targets as good as possible (see section 3.2.2.3).

#### 3.2.2.1 Aiming for an object ball

When determining shot possibilities geometrically, JPool does not consider effects such as *throw* and *spin* which could set the object ball off its course. This section explores the velocities required for the cue ball to cause the object ball to reach its destination after a collision with the cue ball, i.e. venture into a pocket.

Let the object ball $b_2$ be positioned at $A_2$, $D_1$ be the ghostball position as determined geometrically and let $(\vec{v'}_2, \vec{w'}_2)$ be the velocities $b_2$ needs to have post-collision in order to reach its destination. To define the required pre-collision velocities of the cue ball $(\vec{v}_1, \vec{w}_1)$, recall from equations 3.18, 3.20, 3.21 and 3.23 that the relative velocity at the point of the collision and the post-collision linear and angular velocities of $b_2$, rotated so that both $A_2$ and $D_1$ are on the $x$-axis, are given by:

$$\vec{u}_c = \vec{v}_1 - (R\hat{x} \times \vec{w}_1) = \begin{pmatrix} \vec{v}_{1_x} \\ \vec{v}_{1_y} + R\vec{w}_{1_z} \\ -R\vec{w}_{1_y} \end{pmatrix} \tag{3.38}$$

$$\vec{v'}_2 = \left(\vec{v}_{1_x}, -\mu_c \vec{v}_{1_x} \hat{u}_{c_y}, 0\right)^T \tag{3.39}$$

$$\vec{w'}_2 = \frac{-2.5\mu_c \vec{v}_{1_x}}{1.000575R}(\hat{x} \times \hat{u}_c) \tag{3.40}$$

These equations work under the assumption that $b_2$ is stationary. Since equations 3.39 and 3.40 do not directly contain $\vec{v}_1$ and $\vec{w}_1$, determining $\vec{u}_c$ is the key to determining the pre-collision velocities needed. Note that equation 3.39 yields $\vec{v}_{1_x} = \vec{v'}_{2_x}$ and as a result equation 3.38 yields $\vec{u}_{c_x} = \vec{v'}_{2_x}$. Rearranging equations 3.39 and 3.40 to components of $\hat{u}_c$ yields the following equations:

$$\hat{u}_{c_y} = \frac{\vec{v'}_{2_y}}{-\mu_c \vec{v'}_{2_x}} \tag{3.41}$$

$$\hat{u}_{c_z} = \frac{-1.000575R\vec{w'}_{2_y}}{-2.5\mu_c \vec{v'}_{2_x}} \tag{3.42}$$

The length of $\vec{u}_c$ can be determined using $|\hat{u}_c| = 1$ and $\vec{u}_{c_x}$:

$$\begin{aligned} \hat{u}_{c_x} &= \sqrt{1 - \hat{u}_{c_y}^2 - \hat{u}_{c_z}^2} \\ |\vec{u}_c| &= \frac{\vec{u}_{cx}}{\hat{u}_{c_x}} = \frac{\vec{v'}_{2_x}}{\sqrt{1-\hat{u}_{c_y}^2-\hat{u}_{c_z}^2}} \end{aligned} \tag{3.43}$$

With $|\vec{u}_c|$, $\hat{u}_{c_y}$ and $\hat{u}_{c_z}$ known and using equation 3.38, $\vec{w}_{1_y}$ is determined:

$$\vec{w}_{1_y} = -\frac{\vec{u}_{c_z}}{R} \tag{3.44}$$

$\vec{w}_{1_z}$ and $\vec{v}_{1_y}$ cannot be determined with certainty but can be set in relation to each other:

$$\vec{u}_{c_y} = \vec{v}_{1_y} + R\vec{w}_{1_z} \tag{3.45}$$

These are set according to the simulation result of a naive shot. Additionally, these values may be altered to increase the robustness of the shot by finding a combination that requires minimal spin applied to the cue ball.

#### 3.2.2.2 Aiming for the cue ball

When performing a kiss shot, the velocities caused on the secondary ball do not matter - JPool only calculates the required velocities for the kissing ball. However, no direct technique for acquiring the pre-collision velocities has been found. JPool uses the velocities found during simulation as a basis for an approximation algorithm, acquiring the pre-collision velocities $\vec{v}$ and $\vec{w}$ using Newton's Method. Let $\vec{v'}$ and $\vec{w'}$ be the desired post-collision velocities. The following equations, based on equations 3.18 - 3.23, define the difference between the caused velocities and the desired velocities:

$$f_1 = \frac{-(R\vec{w}_z + \vec{v}_y)\mu_c \vec{v}_x}{\sqrt{(R\vec{w}_z + \vec{v}_y)^2 + (-R\vec{w}_y)^2 + (\vec{v}_x)^2}} - \vec{v'}_y + \vec{v}_y \tag{3.46}$$

$$f_2 = \frac{2.5\mu_c \vec{v}_x \vec{w}_y}{1.000575\sqrt{(R\vec{w}_z + \vec{v}_y)^2 + (-R\vec{w}_y)^2 + (\vec{v}_x)^2}} - \vec{w'}_y + \vec{w}_y \tag{3.47}$$

$$f_3 = \frac{2.5\mu_c \vec{v}_x (R\vec{w}_z + \vec{v}_y)}{1.000575\sqrt{(R\vec{w}_z + \vec{v}_y)^2 + (-R\vec{w}_y)^2 + (\vec{v}_x)^2}} - \vec{w'}_z + \vec{w}_z \tag{3.48}$$

$f_1$, $f_2$ and $f_3$ are approximated to 0 using Newton's Method, yielding a set of pre-collision velocities of the cue ball matching the desired post-collision velocities. The derivatives of equations 3.46, 3.47 and 3.48 have been determined machinally.

#### 3.2.2.3 Ball ball collisions - handling both

Let $(\vec{v'}_1, \vec{w'}_1)$ be the desired post-collision velocities of $b_1$ and $(\vec{v'}_2, \vec{w'}_2)$ be the desired post-collision velocities of $b_2$. As it has been done in section 3.2.2.2, Newton's Method is used to acquire a set of pre-collision velocities $(\vec{v}_1, \vec{w}_1)$ of $b_1$ to cause all desired post-collision velocities if possible. Note that equation 3.20 yields $\vec{v}_{1_x} = \vec{v'}_{2_x}$ and equation 3.22 yields $\vec{w}_{1_x} = \vec{w'}_{1_x}$, leaving $\vec{v}_{1_y}$, $\vec{w}_{1_y}$ and $\vec{w}_{1_z}$ to be determined:

$$f_1 = \frac{(R\vec{w}_{1_z} + \vec{v}_{1_y})\mu_c\vec{v}_{1_x}}{\sqrt{(R\vec{w}_{1_z} + \vec{v}_{1_y})^2 + (-R\vec{w}_{1_y})^2 + (\vec{v}_{1_x})^2}} + \vec{v'}_{2_y} \tag{3.49}$$

$$f_2 = \frac{5\mu_c\vec{v}_{1_x}\vec{w}_{1_y}}{1.000575\sqrt{(R\vec{w}_{1_z} + \vec{v}_{1_y})^2 + (-R\vec{w}_{1_y})^2 + (\vec{v}_{1_x})^2}} + \vec{w'}_{2_y} + \vec{w'}_{1_y} - \vec{w}_{1_y} \tag{3.50}$$

$$f_3 = \frac{(R\vec{w}_{1_z} + \vec{v}_{1_y})\mu_c\vec{v}_{1_x}}{\sqrt{(R\vec{w}_{1_z} + \vec{v}_{1_y})^2 + (-R\vec{w}_{1_y})^2 + (\vec{v}_{1_x})^2}} + \vec{v'}_{1_y} - \vec{v}_{1_y} \tag{3.51}$$

These equations are based on equations 3.18 - 3.23. Note that the velocities acquired using Newton's Method and equations 3.49, 3.50 and 3.51 may not be valid because the combination of desired velocities for both balls may not be causeable by a single ball-ball-collision. This results in extreme pre-collision velocities.
This is being handled by executing the physical enhancement process (as detailed in section 3.3.1) several times using the velocities found, evening out irregularities in subsequent iterations.

### 3.2.3 Collisions between a rail and a ball

Let $(\vec{v},\ \vec{w})$ be the initial and $(\vec{v}_D,\ \vec{w}_D)$ be the desired post-collision velocities of the ball colliding with a rail. Using equations 3.24 and 3.25, the needed pre-collision velocities are determined:

$$\vec{v} = \begin{cases} \left(\frac{1}{0.9}\vec{v}_{D_x}, \frac{-1}{0.6}\vec{v}_{D_y}, 0\right), & \text{if rail is horizontal} \\ \left(\frac{-1}{0.6}\vec{v}_{D_x}, \frac{1}{0.9}\vec{v}_{D_y}, 0\right), & \text{if rail is vertical} \end{cases} \tag{3.52}$$

$$\vec{w} = 10\vec{w}_D \tag{3.53}$$

### 3.2.4 Cue Stick Impact - Shot parameters

Since all physical calculations are based on the linear and angular velocities of the balls and not on shot parameters, JPool needs to calculate shot parameters that cause the desired initial velocities of the cue ball. Let $\vec{v}_A$ and $\vec{w}_A$ be the desired initial velocities of the cue ball. $\varphi$ is easily calculated by geometric means since it is the angle between $\vec{v}_A$ and the $x$-axis. Let $\vec{v}$ and $\vec{w}$ be $\vec{v}_A$ and $\vec{w}_A$ rotated so that $\vec{v}$ points exactly along the $x$-axis. Equation 3.4 yields $w_y = aIF\sin\theta$ and $w_z = -aIF\cos\theta$. These can be simplified:

$$I = \frac{w_y}{aF\sin\theta} = \frac{w_z}{-aF\cos\theta} \tag{3.54}$$

$$\frac{w_y}{w_z} = \frac{aF\sin\theta}{-aF\cos\theta} = -\frac{\sin\theta}{\cos\theta} = -\tan\theta \tag{3.55}$$

$$\theta = \arctan\frac{-w_y}{w_z} \tag{3.56}$$

In the next step, $F$ can be determined using $v_y = \frac{-F}{w_y}\cos\theta$ from equation 3.3. Additionally, using $\cos\theta = \frac{-Iw_z}{Fa} = \frac{-mv_y}{F}$ from equations 3.3 and 3.4 , $a$ can be determined:

$$F = -\frac{w_y v_y}{\cos\theta} \tag{3.57}$$

$$a = \frac{Iw_z}{mv_y} \tag{3.58}$$

FastFiz accepts $a$ and $b$ if a value $c$ can be found with the resulting point $Q$ lying on the surface of the ball. $b$ is determined using $w_x$ and $w_y$ in the same manner as in equation 3.54, substituting $|\sqrt{R^2 - a^2 - b^2}|$ for $c$ (using equation 3.1):

$$b = -\frac{aw_x w_z \pm \sqrt{-a^2 w_x^2 - a^2 w_y^2 - a^2 w_z^2 + (w_y^2 + w_z^2)R^2}w_y}{w_y^2 + w_z^2} \tag{3.59}$$

Note that there are two possible solutions for $b$, resulting in two possible values for $V_0$. Both shots are saved for further evaluation. $V_0$ is calculated using equation 3.2:

$$V_0 = \frac{F(1 + (\frac{m}{M} + \frac{5}{2R^2}(a^2 + b^2\cos^2\theta + c^2\sin^2\theta - 2bc\cos\theta\sin\theta)))}{2m} \tag{3.60}$$

This results in a shot $(V_0, \varphi, \theta, a, b)$ for the desired initial velocities $\vec{v}$ and $\vec{w}$.

## 3.3 Applying the physics engine

The shot generation performed in chapter 2 only yields aiming directions for potential shots. Since FastFiz expects five shot parameters defining a shot, the missing parameters need to be determined. JPool uses two methods for acquiring those parameters: random variation of parameters other than the aiming direction $\varphi$ and physical calculation up the step tree. Additionally, a technique for adjusting aiming directions is used if a naive execution of a shot in the direction in question fails.

### 3.3.1 Adjusting aiming directions

Each geometrically determined aiming direction $\varphi$ is simulated using the following naive shot parameters:

$$(a = 0, b = 0, \theta = 25, \varphi, V_0 = 4.5) \tag{3.61}$$

The ball is struck with maximum speed in the direction in question. If the targeted ball is not pocketed, the shot is not disbanded but thoroughly analyzed. The shot step that fails to be triggered is identified and the velocities determined in the simulation are rotated for the ball to run through the desired ghostball position. This is done by calculating the closest point of the ball's trajectory to the ghostball position. The angle between this point and the ghostball position is used to rotate the ghostball position of the previous shot step. In this manner, velocities are calculated back up the step tree to the cue ball, resulting in an adjusted aiming direction $\varphi$. The shot is disbanded if it fails in another round of simulation. Of a thousand failing shots, 19.7% could be repaired using this technique.

### 3.3.2 Calculating velocities in a step tree

Section 2.2.1 shows JPool's technique for geometrical shot generation which is based on events that can happen on the table. These events are further investigated in sections 3.1 and 3.2, detailing the equations FastFiz uses to simulate each event and presenting equations and techniques that can be used to find velocities that cause the events that have been planned geometrically. This section presents a technique to combine both aspects of events, calculating the velocities of balls in a geometrically determined step tree.

In the first step, the shot is executed and adjusted as explained in section 3.3.1. A number of variations of the shot are generated, setting $a$, $b$, $\theta$ and $V_0$ to a series of characteristic combinations. This results in multiple possible shots for the same step tree. These shots are simulated noiseless and adjusted as done with the naive shot in section 3.3.1. The resulting table state is analyzed for position zones and the path of the cue ball is analyzed as detailed in section 2.3.3.

For each zone within a specific distance to the cue ball's path, it is attempted to adjust the shot using the physics engine. The velocities of the cue ball are calculated up the shot tree, starting at the event at which the cue ball is halting in the shape zone. The shot tree is traversed recursively, calculating velocities for each step: first, the velocities needed to reach the next event and the velocities needed there are calculated. Equations corresponding to the current event are applied. Listing 3.1 shows the algorithm used in pseudocode.

When the cue ball's ball strike event is reached, the process is aborted and started for the pocketing event of the shot tree. This way all velocities are known when the cue ball's strike event is calculated using the methods for ball collisions where both velocities need to be matched. Note that some values cannot be determined with certainty. For example, a ball's velocities when it is pocketed cannot be determined but need to be guessed. This is done using the simulation result of the underlying naively determined shot. When the cue ball is reached, the determined velocities are translated to shot parameters.

```
1  applyPhysicsBackwards(JShotstep cur, Vector mainV, Vector mainW) {
2    // calculate how to reach ghostball position with the required velocities
3    if(cur.next != NULL)
4      physics.reachPointWithVelocity(targetPoint, mainV, mainW);
5
6    // apply current event handler
7    switch(cur.getType()) {
8      case CUE_STRIKE:
9        return physics.shotParameters(mainV, mainW);
10
11     case RAIL_COLLISION:
12       physics.railCollision();
13       break;
14
15     case BALL_HALT:
16       mainV = 0; mainW = 0; break;
17
18     case KISS_COLLISION:
19       physics.ballCollisionKiss();
20       break;
21
22     case STRIKE_COLLISION:
23       if(ball_acting == CUE)
24         calculateCueBallPositionPlayVelocities();
25         physics.ballCollisionBoth();
26       else
27         physics.ballCollisionStrike();
28       break;
29   }
30
31   // go back through the steps recursively
32   applyPhysicsBackwards(cur->prevShot, mainV, mainW);
33 }
```

**Listing 3.1:** Algorithm applying the physics calculations to a shot tree

# 4 Planning ahead in pocket billiards

A list of shots is computed for the given table state using the methods shown in chapters 2 and 3. In the next step, the best shot in this list of shots needs to be identified and returned to the FastFiz simulator. To do this, each shot is simulated a number of times and rated for its value to the player. This chapter details the shot selection process performed by JPool. Two methods for planning ahead in stochastic games are detailed and the value of planning ahead in computational pool is analyzed in the process. Furthermore, techniques used in JPool to handle special situations on the table are given.

## 4.1 Probabilistic search algorithms

There is a range of search algorithms available for traditional games like Nine Men's Morris, Connect Four or Chess - MiniMax and $\alpha\beta$-pruning being the most common ones ($\alpha\beta$-pruning is a modification of MiniMax). However, these are not applicable to the game of billiards because there is an element of chance involved. Furthermore, the turn structure in billiards is not static which is a base requirement for MiniMax-search. This section presents two algorithms for search in games with an element of chance like billiards: Expectimax and Monte-Carlo sampling.

### 4.1.1 Expectimax

Expectimax is a modification of Minimax. Chance nodes are added to the search tree, representing situations with a non-deterministic outcome. For each possible outcome of a node, the probability of occurrence is calculated. The overall rating of a chance node is the sum of the ratings of its possible outcomes, each weighted by its probability of occurrence.

Consider the following example of a dice game: the player has to decide by which rules he wants to throw a dice. Using ruleset A he wins if the dice shows an odd number, using ruleset B he wins if the dice shows a number $\leq 2$. Figure 4.1(a) shows the complete Expectimax-tree of this game. A winning node is rated with 1, a loosing node is rated with -1. Each chance node is rated by the sum of its child nodes, weighted by their probability of occurrence. Since the chance of rolling an odd number with a dice is $\frac{1}{2}$, ruleset A is rated using $\frac{1}{2} - \frac{1}{2} = 0$. The chance for rolling a number $\leq 2$ is $\frac{1}{3}$, rating ruleset B by $\frac{1}{3} - \frac{2}{3} = -\frac{1}{3}$. The results in a higher rating for ruleset A, advising the player to choose this set of rules.

When using Expectimax in the game of billiards, a number of adjustments have to be done. Since each shot can result in an infinite number of table states, it is impossible to rate each table state and the corresponding probabilities of occurrence (which would be extremely small for each specific state). Furthermore, the actions of the enemy player cannot be guessed with certainty in the game of billiards. This is being accounted for by not expanding nodes that result in the loss of the turn, practically ignoring the shots done by the opponent.

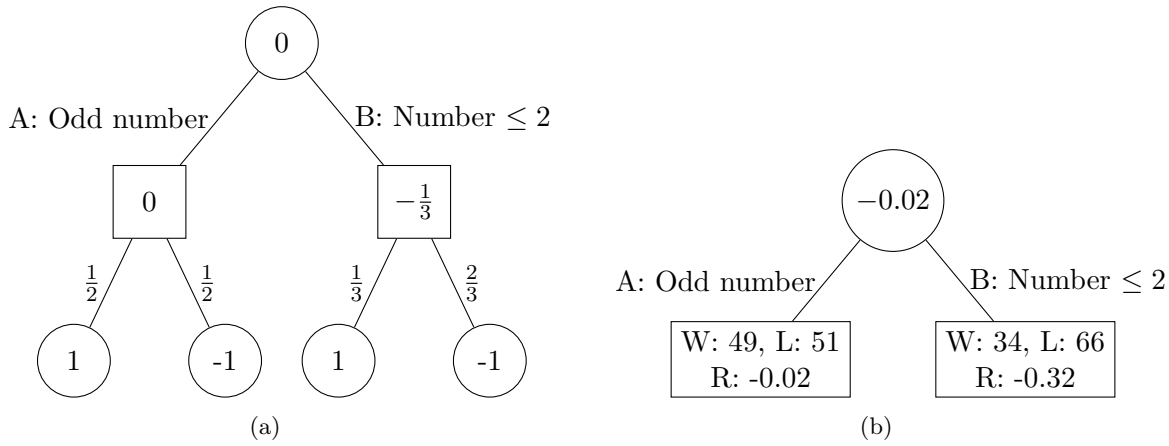



**Figure 4.1:** (a) Expectimax and (b) Monte-Carlo sampling trees applied to a simple dice game

PickPocket implements Expectimax by grouping shots to successful and unsuccessful shots [18, p. 4]. Successful shots are expanded using the resulting table state when played without noise. A similar approach is followed in SkyNet [15, p. 8]. Both players estimate the probability of success of a shot by simulating it a number of times, forming a hybrid of Monte-Carlo sampling and Expectimax.

### 4.1.2 Monte-Carlo sampling

In Monte-Carlo sampling, each possible move is tried a number of times and the average rating is computed. Following the example of the dice game from section 4.1.1, a Monte-Carlo sampling method would try each ruleset offered to the player $n$ times and calculate the average rating that has occurred. Figure 4.1(b) shows the tree built by Monte-Carlo sampling for a sample size of 100. Ruleset A is won 49 times and lost 51 times, resulting in an overall rating of $\frac{49-51}{100} = -0.02$. Ruleset B is rated using $\frac{32-66}{100} = -0.32$. While the ratings are not as precise as the ones formed by Expectimax, the chosen move is the same – the player is advised to choose ruleset A.

Note that a higher number of samples may result in more precise ratings. Low sample rates are prune to errors. Monte-Carlo sampling can directly be applied to the game of billiards. Each shot is executed a number of times and each executed shot is expanded and investigated to a maximum depth, based on its simulation result. Monte-Carlo sampling is implemented in PickPocket, using a sample size of 15 and search with a depth of 2 [18, p. 5].

### 4.1.3 Comparison

The differences between Monte-Carlo sampling and Expectimax search are big in general, but are small when applied to the game of billiards since Expectimax cannot be utilized directly and needs to be adjusted. The probabilities of success used in Expectimax cannot be calculated but need to be acquired by testing a shot a number of times. This adds elements of Monte-Carlo sampling to Expectimax. Most importantly, the probabilities used for weighting states are prune to errors, depending on the number of samples used.

Since the number of possible states is infinite, not all resulting states can be analyzed. Past implementations of Expectimax only expand upon table states that have been acquired by noiseless simulation of shots. Section 4.2 will show that this is an insufficient approximation. Monte-Carlo sampling works with the actual table states resulting of noiseful simulation, taking a number of shot outcomes into account.

These are two strong arguments for using Monte-Carlo sampling instead of Expectimax in the game of billiards. This corresponds with the results of [17, p. 72], where agents utilizing both search methods have been played against each other in various setups using a variety of noise models. The agent using Monte-Carlo sampling won 67% of games using a strong noise model and 62% of games using a weak noise model. For these reasons, JPool uses Monte-Carlo sampling for shot selection. Listing 4.1 shows the Monte-Carlo sampling algorithm as it is being used by JPool in pseudocode.

```
1  monteCarloSearch(tablestate, depth, num_samples) {
2    shots[] = moveGenerator(tablestate)
3
4    bestRating = -1000
5
6    for each shot in shots[] {
7      curRating = 0
8
9      for j = 0 .. num_samples {
10       if(testShotWithNoise() == good) {
11         if(depth == 1)
12           curRating += rateResultingTableState(resultState)
13         else
14           curRating += monteCarloSearch(resultState, depth-1, num_samples)
15       }
16     }
17
18     curRating = curRating / num_samples
19     if(curRating > bestRating)
20       bestRating = curRating
21   }
22
23   return bestRating
24 }
```

**Listing 4.1:** Monte-Carlo sampling algorithm as used by JPool

## 4.2 The influence of noise on a shot

Section 4.1 detailed that JPool uses Monte-Carlo sampling for shot selection. Search is limited by two parameters: the sample size $n$ and the search depth $m$. For traditional games played by search algorithms, a higher search depth usually improves the strength of the computer player. This is different in the game of billiards: since a shot is executed with noise, the following table state is uncertain. A generated subsequent shot may not be possible or have a different value to the player. While Monte-Carlo sampling attempts to compensate for this effect by computing the average rating of $n$ table states, the rating still loses quality the higher the search depth $m$. This section investigates the effect of noise and the value of planning ahead in computational pool.

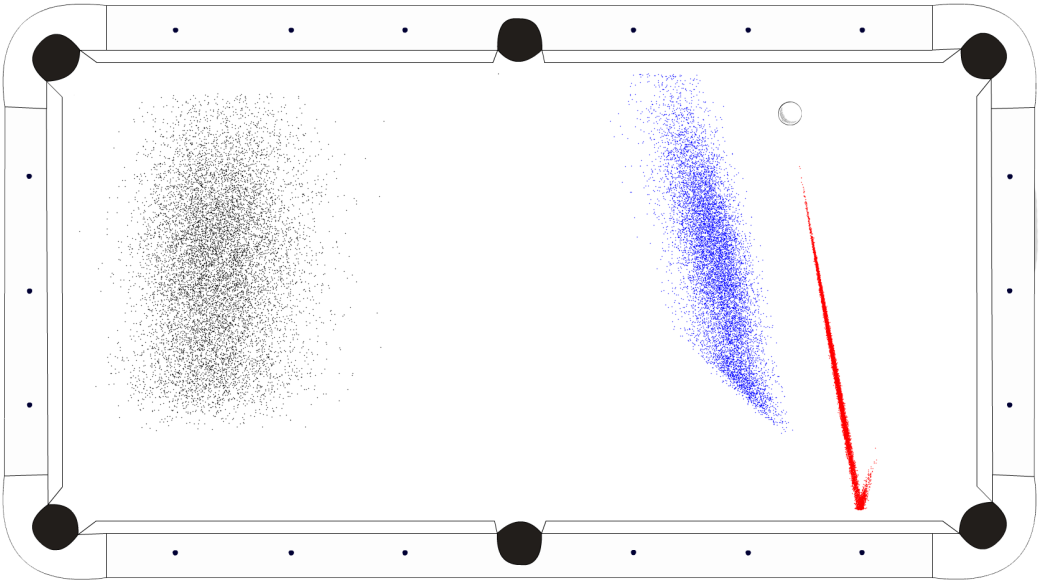


**Figure 4.2:** Halting points in a sequence of three shots after executing them a thousand times. Red - after 1 shot, blue - after 2 shots, black - after 3 shots

FastFiz applies zero-mean Gaussian noise on shots prior to their execution. This aims to simulate the difficulty of executing shots as they have been planned for humans or robots. The standard deviations of the noise model used by FastFiz are:

$$E_{FastFiz} = \{\sigma_a = 0.5mm, \sigma_b = 0.5mm, \sigma_\theta = 0.1°, \sigma_\phi = 0.125°, \sigma_{V_0} = 0.075m/s\} \quad (4.1)$$

Other noise models have been used each year at the computer olympiad. This chapter analyzes the effect of noise on shots using the standard noise model $E_{FastFiz}$; however, JPool has been played using different noise models to allow for comparison to other players.

When a shot is executed with noise, the path of the cue ball is altered. This results in a different halting point of the cue ball and may cause the shot to fail. Since the strength of the noise is limited, the difference to the original shot is limited as well - in general, the probability of success for each shot is lower when a stronger noise model is used. Figure 4.2 shows a range of possible cue ball halting points in a sequence of three shots that has been simulated a thousand times. After planning these three shots ahead, the cue ball can come to rest anywhere in the area marked by black dots. In addition to the cue ball's position, the positions of other balls may or may not be influenced as well. The resulting table state is highly uncertain. A shot that is planned based on this table state is very unlikely to be valid after the three shots it is based on are executed with noise, rendering attempts to plan ahead to this depth useless.

Other works have investigated the influence of search depth on the success of computational billiards players. PickPocket has been tested against a version with a higher search depth in [17, p. 64]. The agent with a search depth of 1 won 66% of games against an agent searching to a depth of 2 and 63% of games against an agent searching to depth 3. The authors of CueCard identified the search depth to have no effect on the success of their agent [2, p. 6].

Search in JPool is being limited to a depth of 1 for these reasons. However, an indirect way of planning ahead is performed by aiming for position zones on the table, ensuring a good available shot in the subsequent table state. A lot of computation time is saved by not computing a second level of search, allowing JPool to utilize a sample size of 400 samples per shot (search is performed concurrently for 4 shots at a time). A high sample size is identified to have a major positive effect on a computer players' strength in [2, p. 6]. PickPocket searches with a sample size of 15; the sample size used in CueCard is varied between 25 and 100, depending on the available time.

## 4.3 Rating shots

Since search depth is very limited in computational billiards, the method of rating a shot is an integral factor to the strength of a computer player. In addition to the focus on position play, JPool closely follows the strategic advice for human pool players given by Jack Koehler in [9]. In summary, the following points are considered to form the rating strategy of JPool:

- Problem balls are balls that are generally hard to play because they are in inconvenient positions on the table or close to other balls. According to [9, p. 162], these should be pocketed whenever there is a chance and easier balls should be left for later stages of the game where less shots are available in general. If an easy ball is played in an early stage of the game, it should be used to get position on a problem ball.
- Clusters are collections of two or more balls touching or in proximity to each other. Koehler advises to open clusters as soon as possible because position play accuracy is diminished when attempting to open a cluster and it is more likely to still attain a shot in earlier stages of the game.
- While position play is important, the most important aspect of any shot is pocketing a ball and keeping the turn. However, the weights of each goal are different for each individual player.
- Defensive considerations [9, p. 161] are minimizing the position rating of the enemy, eventually leaving the enemy with no available shot in the event of the planned shot failing. Furthermore, Koehler advises to pocket balls that are blocking some of the enemy player's balls later in the game.

These strategic considerations are not directly aimed for because the primary target of a shot remains pocketing a ball and these considerations concern the outcome of a shot other than being pocketed. To take them into account, they are translated to the computational domain for JPool's shot rating function. Each ball is rated for its difficulty to be pocketed (as detailed in section 2.1.1) and its distance to other balls. The rating is formed so that higher ratings represent problem balls and lower ratings represent easy balls. JPool computes the following values for each shot and its resulting table state:

**A:** The position rating for the current player

**B:** The position rating for the enemy player (inverted)

**C:** The sum of ball ratings of pocketed balls

**D:** Average ball rating of the current player's balls

**E:** Average ball rating of the enemy player's balls (inverted)

**F:** Moved ball rating. For each ball that has been moved but not pocketed, the change of its rating compared to the previous table state is computed

The total rating is formed by the weighted sum of these values:

$$rating = \lambda_1 A + \lambda_2 B + \lambda_3 C + \lambda_4 D + \lambda_5 E + \lambda_6 F \tag{4.2}$$

The values of $\lambda_i$ for $i \in [1, 6]$ define the play style of the agent. High values for $\lambda_2$ and $\lambda_5$ let the player focus on minimizing the odds of the enemy player, resulting in a defensive play style. High values for $\lambda_1$, $\lambda_4$ and $\lambda_6$ lay a focus on the player's position play. Identifying optimal values for these parameters is an important task for defining the play style and strategy of JPool. While a machine learning algorithm may be applicable, these parameters have been estimated using a trial-and-error method: a proposed set of parameters has been played over the course of 5 games and the resulting play style has been analyzed intuitively. Adjustments to the values of each parameter have been made based on this analysis. In its current version, JPool uses the following values:

$$\{\lambda_1 = 16, \lambda_2 = 2, \lambda_3 = 4, \lambda_4 = 4, \lambda_5 = 2, \lambda_6 = 3\} \tag{4.3}$$

These values may be categorized to create a rather offensive play style as the enemy player's ratings have little influence on shot selection.

## 4.4 Special situations

There are 3 situations in the game of pool that cannot be handled by the regular shot detection algorithm but have to be handled separately: ball-in-hand, safety shots and the break shot. This section briefly presents the techniques used in JPool to handle each of these situations.

### 4.4.1 The break shot

The break shot is the first shot in each game of billiards. It is always performed on the racked table state as illustrated in figure 1.1, qualifying the break shot to be pre-computed. A break shot is considered legal if a ball has been pocketed or four balls touched a rail. Prior to the break shot, the cue ball may be placed anywhere behind the headstring. The quality of the break shot is an integral influence to the success of a player since it lays the foundation for all subsequent shots. The authors of CueCard identified the break shot to be the most important factor to the success of their agent [2, p. 6].

JPool's break shot has been determined using a particle system algorithm [20]. A shot has been rated by probability of success (using $E_{FastFiz}$) and the resulting table state, rated by the sum of the rating of each ball. 64 particle swarms, each consisting of 64 particles, have been applied concurrently to a 7-dimensional search space (5 shot parameters and the cue ball's initial coordinates). After an extended period of time, shots have been hand-picked and compared by the tactical value of their resulting table states. The 5 best break shots

that have been found are listed in table 4.1.

The use of an algorithm results in rather unconventional yet very successful break shots. Note that these break shots are very different in their nature. Each shot is aiming at a different aiming direction $\varphi$. None of these shots strikes the balls directly (as usually done by human pool players) but strikes the racked balls over a rail. Figure 4.3 shows the paths of the five break shots. Shot (a) has been picked to be the break shot used by JPool. Note that despite the nature of particle swarm algorithms swarms may get stuck in local maxima. There may be better break shots found when the same algorithm is started once again (starting values were randomized).

| # | $a$ | $b$ | $\theta$ | $\varphi$ | $V_0$ | $x$ | $y$ | % |
|---|---|---|---|---|---|---|---|---|
| (a) | 2.122 | 4.795 | 21.027 | 307.421 | 4.354 | 0.756 | 1.808 | 98.7% |
| (b) | −0.054 | 15.789 | 32.569 | 119.248 | 4.317 | 0.742 | 1.877 | 99.6% |
| (c) | 0.083 | 16.018 | 39.481 | 136.646 | 4.469 | 0.671 | 1.905 | 99.5% |
| (d) | 8.974 | 12.460 | 31.537 | 210.755 | 3.799 | 0.215 | 1.914 | 96.2% |
| (e) | 6.169 | 3.472 | 7.216 | 237.691 | 4.017 | 0.294 | 1.731 | 98.8% |

**Table 4.1:** Break shots acquired using a particle swarm algorithm in no particular order

PickPocket used a sampling algorithms to acquire its break shot. Search was limited to $x$, $y$, $\varphi$ and $V_0$ and a sampling size of 200 was used to find the shot with the highest probability of sinking a ball at the break shot (the rule that break shots are legal if four balls touched a rail has been added does not apply in PoolFiz). CueCard's break shot has been determined using 'extensive offline search' [2, p. 3] and has a success rate of 92%. No details of PoolMaster's break shot are known.

### 4.4.2 Ball-in-hand

A player is awarded the right to place the cue ball anywhere on the table after the enemy player performed a foul. This is a very advantageous situation for the player. JPool makes use of the shape zones (their generation is explained in section 2.3.2). Each shape zone is considered as a possible placement for the cue ball, regardless of its rating or size. For each placement, one shot is generated and executed and the resulting table state is rated as detailed in section 4.3.

This is done to add some strategic value to the placement of the ball, attempting to choose a place for the cue ball that will not only yield a single good shot (as a decision solely based on the shape zone rating would produce). Tendentially, this will result in the ball being placed to pocket a problem ball. In the very unlikely event of no valid position being found by this technique, random positions on the table are rated for the player and the first advantageous position is chosen.

PickPocket acquires a ball-in-hand shot by discretizing the table into a grid of cells [17, p. 49]. For each cell, available shots are calculated and the cell with the shot most probable to be successful is chosen to place the cue ball. PoolMaster's strategy for placing the cue ball is very similar to the technique used in JPool; however, the table is rated using a grid-based

heuristic to identify local maxima in the table position rating function (opposed to JPool's approximation using polygon grids) [5, p. 8]. Both PickPocket and PoolMaster focus on enabling easy shots, adding little tactical value. No details of CueCard's ball-in-hand method are known.

### 4.4.3 Safety shots

Whenever no valid shot can be found for a given table situation, the computer player is forced to make a last-resort shot. A player may call 'safety' instead of a ball to be pocketed, acknowledging that he cannot legally pocket a ball. The player will lose his turn after he performed his shot. This is done with the intention of placing the ball in a difficult position for the enemy player. To avoid leaving the enemy player in a ball-in-hand situation, the cue ball must touch one of the player's balls to prevent a foul.

JPool handles these situations by generating random shots aimed in the direction of each of the player's balls on the table. These shots are tested a number of times and the shot with the lowest average resulting position rating for the enemy is chosen. PickPocket utilizes sampling to find a safety shot - a wide range of $\varphi$ and $V_0$ values are sampled [17, p. 36]. No details of CueCard's or PoolMaster's safety strategies are known.

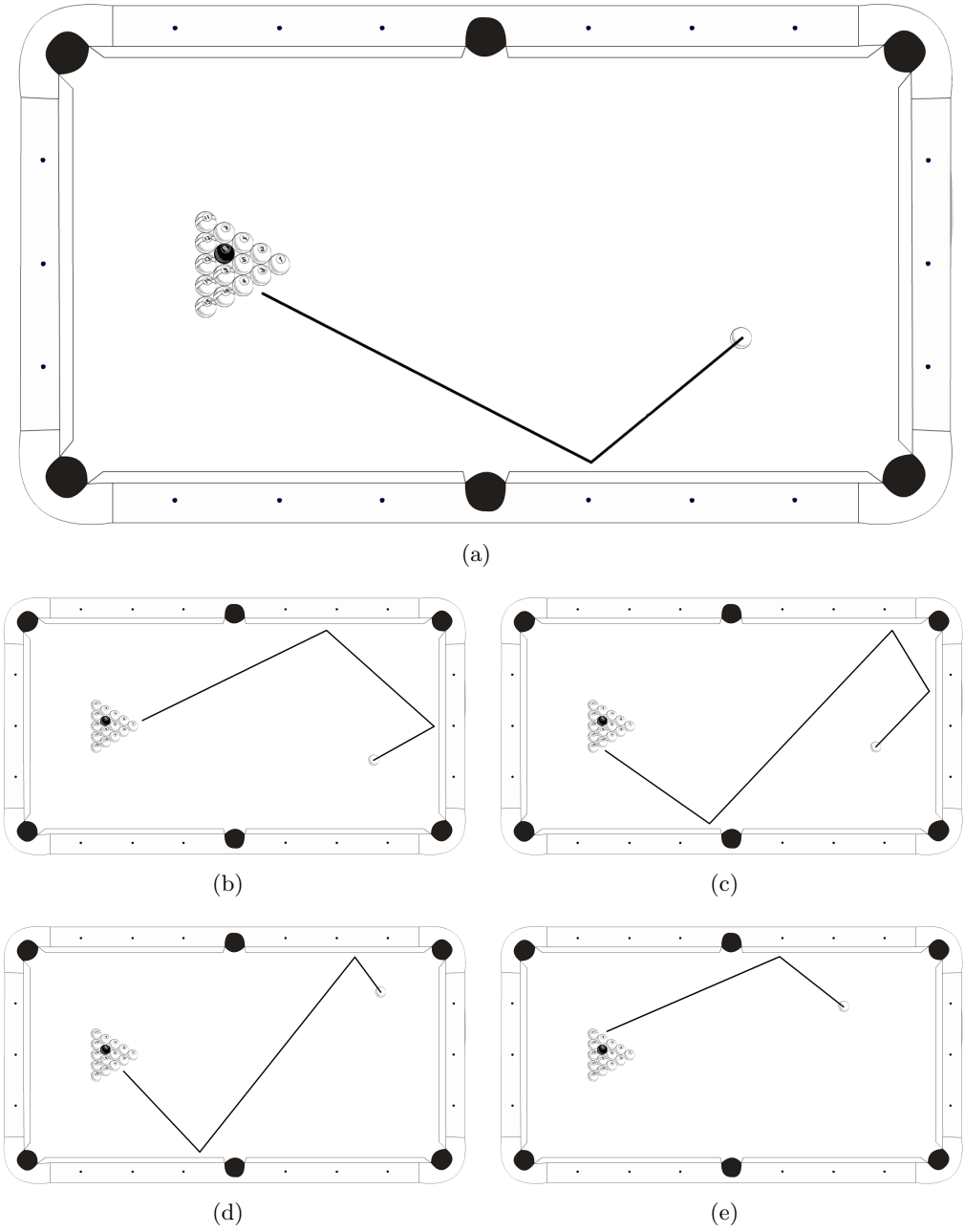



**Figure 4.3:** The top 5 break shots acquired using a particle swarm algorithm. JPool uses shot (a)

# 5 Results

In order to analyze its success, JPool has been played in various setups.[1] This chapter presents and analyzes the results of simulation and concludes with ideas for further work. Note that some results have a low stochastic significance because of a relatively low sample size. This is due to games taking a long time to be calculated (each player has a time limit of 10 minutes) and the limited time available for the creation of this thesis. Over the course of all games simulated, the average time spent for acquiring a single shot has been 44.1 seconds.

## 5.1 Off-break wins

JPool has been played against an agent called 'Conceeder'. Conceeder gives up the game whenever it gets the chance, resulting in games played against Conceeder being tests if the game can be won without ever losing turn. These wins are called *off-break wins*.

Of 100 games played with regular noise, JPool won 44 games without ever losing its turn, 54 games were ended by Conceeder and 2 games were lost because the 8-ball has been pocketed accidentally by JPool. This results in 44% of games being won off-break. Note that a number of 100 games played is very low concerning the statistical significance of this data. 35 games have been played with a noise level scaled by 0.5. 26 of these games were won, 9 lost - resulting in a win rate of 74.2%. 100% of the 35 games played without noise have been won; however, none of these games have been exactly the same. FastFiz randomizes the spaces between balls in the racked state, resulting in different outcomes of break shots even when played without noise.

PickPocket and PoolMaster have been tested in a similar way in [11]. PoolMaster has been refined since the competitions in 2008 and appears to be a very strong player – it wins $\sim 64\%$ of games played with regular noise, $\sim 82\%$ of games played with a noise factor of 0.5 and 100% of games played without noise off-break. PickPocket won an average of $\sim 51\%$ of games played with regular noise, $\sim 67\%$ of games with a noise factor of 0.5 and $\sim 85\%$ of games simulated without noise off-break. No results for CueCard are known.

These results show that JPool is more vulnerable to the effects of noise than its competitors: while it performs very well when played with none or weak noise applied, the rate of off-break wins of JPool falls behind the rates of competitors when playing with full noise. Figure 5.1 shows a direct comparison of JPool's results.

[1] Sadly, no games against PickPocket, PoolMaster or CueCard have been played yet as these agents have not been available for testing.

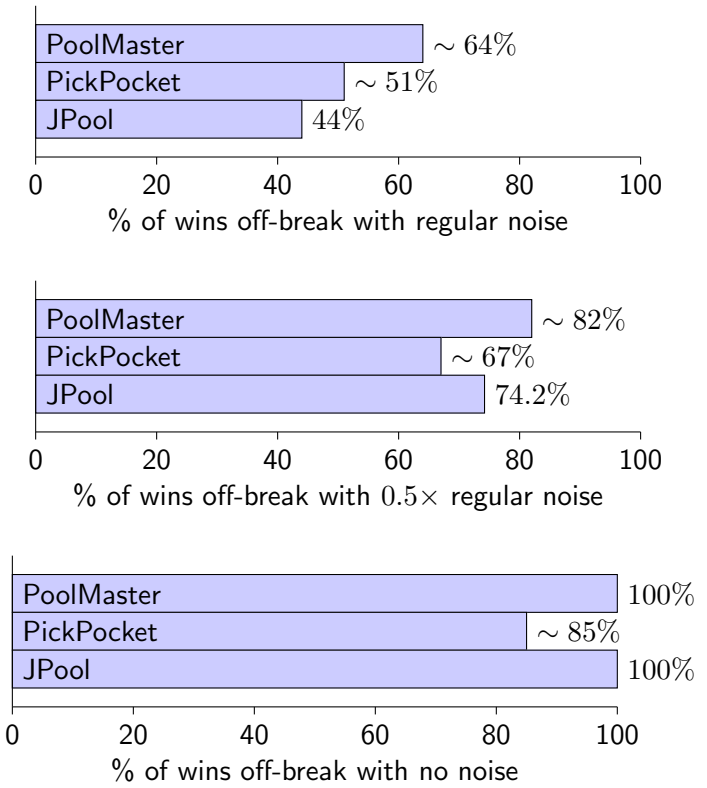
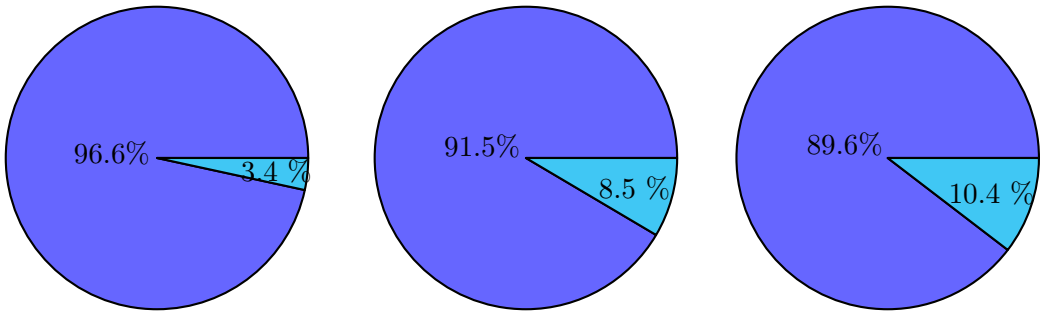



**Figure 5.1:** Off-break-win percentage comparison of various agents




(a) $0.5 \times E_{FastFiz}$ (b) $1 \times E_{FastFiz}$ (c) $E_{Taiwan}$

**Figure 5.2:** Average success rate of shots calculated by JPool when executed using various noise models. Darker areas mark successful shots.

## 5.2 Shot success

According to [17, p. 57], $E_{Taiwan}$ (the noise model used at the 2005 computer olympiad) has been tested on 10,000 random shots with a success rate of 69.07%. However, PickPocket achieves a shot success rate in the $77\% - 81\%$ range by choosing shots with high probabilities using Monte-Carlo sampling [17, p. 62].

JPool has been adjusted to play using the $E_{Taiwan}$ noise model. Of 250 shots that occurred during regular play, 26 shots have failed (one shot pocketed the 8-ball accidentally) resulting in a shot success rate of 89.6%. Over the course of 100 games played using $E_{FastFiz}$, 663 shots have been executed. 56 shots resulted in the loss of turn (two of which because the 8-ball has been pocketed), resulting in an average shot success rate of 91.5% and an average streak length of 6.63 shots. Using $0.5\times$ regular noise, 265 shots have been executed over the course of 35 games. 9 shots have failed, resulting in a shot success rate of 96.6% and an average streak length of 7.57 shots.

Since noise has been designed to simulate a success rate of $\sim 75\%$ [10, p. 5], these values are an indicator that the Monte-Carlo sampling algorithm and the high sample size used in JPool work well at selecting which shot will have a high probability to be successful. Figure 5.2 shows an overview of the success rates of shots.

## 5.3 Domain specificity

While most techniques presented in this thesis are specific to the domain of 8-ball, some techniques are applicable to other areas:

- The technique used for shot generation using step trees is specific to games of billiards; however, it can easily be adjusted to be used in games like 9-ball (by limiting search to a single ball) or 3-cushion-billiards.
- Shape zone generation is applicable to the games of 8-ball and 9-ball.
- The physics engine may be used for any ball game played on a table surface as it is based on the actual physics of ball movement; however, some simplifications done by FastFiz (like limiting movement to the table surface or the simple reaction to rail collisions) need to be accounted for.
- The methods used for rating table states and shots are adjustable to fit most other games like 8-ball (i.e. for 9-ball, all ratings may be limited to focus on the ball in question and the ratings of pocketed balls should be ignored).

## 5.4 Further work

**Looking ahead** The high shot success rate of JPool is a sign that the techniques used to acquire shots and their corresponding parameters are successful. Furthermore, the high sampling rate used in the Monte-Carlo sampling algorithms further ensures a high probability of success for each individual shot. While JPool can compete with other players when little noise is involved (concerning off-break wins), JPool's performance decreased when noise was

increased while the shot success rate stayed relatively the same. This is an indicator that JPool's method of looking ahead is more vulnerable to noise than the methods used by competitors.
Further work is required in this area. CueCard uses a technique of clustering table states in order to reduce the number of states considered when calculating several steps ahead [2, p. 3]. This technique may be adapted to allow for a strategically valuable way of clustering and looking ahead. This technique may even produce arguments to use Expectimax search in billiards because the probabilities of each cluster may not be extremely small (as they are when treating each table state separately).

**Play style parameters** Section 4.3 introduced 6 parameters defining the play style of JPool. While the current parameters used have been acquired using extensive trial-and-error search, there still is room for improvement. Multiple sets of parameters may be identified, choosing a specific set for the current game situation (i.e. a more defensive set of parameters may be used for game situations in which the enemy player is very close to winning the game).

## 5.5 Conclusion

This thesis described JPool, a computer player for billiards. Original methods of modeling and geometrically detecting shot candidates have been presented, creating a dynamic way of conquering the continuous action space. As the focus of JPool lies on position play, a method of detecting and rating shape zones on the table was shown. After acquiring aiming directions geometrically, a corresponding set of shot parameters had to be acquired. This is done using an extensive physics engine created for JPool, allowing the computation of shot parameters rather than setting them to random or default values. The well-known technique of Monte-Carlo sampling has been used for shot selection, making use of a rating function that may be altered simply by adjusting six parameters defining JPool's play style.

While no games against other agents have been played yet, simulations showed that JPool's rate of success lies close to the success rates of its competitors. Further work on JPool should focus on improving the techniques for looking ahead and the utilization of play style parameters. The domain of computational billiards continues to offer many interesting challenges that yet have to be conquered using new techniques of artificial intelligence. Future competitions at the computer olympiad are said to feature games like 9-ball, expanding areas covered by computer players for billiards. Robots like Deep Green appear to be ready for competitions between computer players and human players.

# Appendix

# 1 Physical constants used in FastFiz

| Symbol | Value | Description |
|---|---|---|
| $M$ | $528.695652g$ | The mass of the cue stick |
| $m$ | $163.01g$ | The mass of a ball |
| $R$ | $0.028575m$ | The radius of a ball |
| $I$ | $0.0532410629gm^2$ | The inertia of a ball (from $I = \frac{2}{5}mR^2$ for solid spheres) |
| $\mu_c$ | 0.01 | coefficient of ball-ball friction as determined by [16] |
| $\mu_{cb}$ | 0.7 | coefficient of cue tip-ball friction as determined by [16] |
| $\mu_s$ | 0.2 | coefficient of sliding friction as determined by [16] |
| $\mu_r$ | 0.015 | coefficient of rolling friction as determined by [16] |
| $\mu_{sp}$ | 0.044 | coefficient of spinning friction as determined by [16] |
| $g$ | 9.81 $m/s^2$ | gravitational constant |

**Table 1:** Physical constants used in FastFiz

# 2 Commonly used variable names

| Symbol | Description |
|---|---|
| $\varphi$ | defines the direction of the shot |
| $a$ | defines the horizontal displacement of the impact point on the ball |
| $b$ | defines the vertical displacement of the impact point on the ball |
| $\theta$ | defines the angle between the pool table surface and the cue stick |
| $V_0$ | defines velocity of the cue stick at the time of impact |
| $t$ | the time the ball has been in its current movement state |
| $\tau$ | the time of an event, e.g. a collision |
| $\vec{v}$ | a linear velocity |
| $\vec{w}$ | an angular velocity |
| $\vec{u}$ | a relative velocity |
| $\vec{r}$ | a position |
| $\hat{v}$ | a normalized vector (brought to length 1); here $\vec{v}$ |
| $_s$ | (subscript s) generally a formula or variable in the sliding state of a ball |
| $_r$ | (subscript r) generally a formula or variable in the rolling state of a ball |
| $_c$ | (subscript c) generally a formula or variable relating to a collision |
| $_\perp$ | (subscript $\perp$) a vector rotated by 90 degree to the right |
| $A$ | generally the point at which a ball starts moving |
| $B$ | generally the point at which a ball starts rolling |
| $C$ | generally the point at which a ball halts |
| $D$ | generally used to mark a desired position of the ball |

**Table 2:** Variables used in FastFiz and JPool

# List of Figures


1.1 The initial racked table state with the headstring and footstring . . . . . 3
1.2 A screenshot of the game visualizer created for JPool . . . . . 4
1.3 Shot parameters visualized (a) head-on view showing $a$ and $b$, (b) side-view showing $\theta$, (c) top view showing $\varphi$ (redrawn from [18, p. 2]) . . . . . 5

2.1 Aiming for a pocket with another ball in the way (left), directly (center) and with the option to strike a cushion (right) . . . . . 9
2.2 (a) aiming for a strike using a ghostball and ideal trajectory (b) aiming using leftmost and rightmost directions (c) aiming for a kiss deflects in a perpendicular direction . . . . . 10
2.3 An aiming direction may not be reachable . . . . . 11
2.4 Different patterns to forming a shot: (a) direct shot, (b) bank shot, (c) kick shot, (d) combination shot, (e) pulk shot, (f) kiss shot . . . . . 12
2.5 Steps of a kiss shot (a) on the pool table (b) as step tree as used by JPool . . 13
2.6 Position zones determined iteratively for a player playing the solid balls . . . 16
2.7 Approximation of the position zones shown in figure 2.6 . . . . . 16
2.8 Polygons created for shape zone approximation . . . . . 18
2.9 Rhombus formed on shape zone crossing . . . . . 18
2.10 Crossing zones created for shape zone approximation on a sample shot . . . . 19

3.1 Impact of the cue stick on the ball at the point Q (a) and ball velocity dynamics in the sliding state (b) in the rolling state (c) and in the spinning state (d) . 21
3.2 Sliding (red) and rolling (blue) motion of a ball struck with parameters ($V_0 = 0.5, \varphi = 25, \theta = 30, a = 0.008, b = 0.001$) and a list of all the relevant velocities, coordinates and times of the ball motion (table image is out of scale and was only added for illustrational purpose) . . . . . 23
3.3 Example of the dynamics of linear velocities during a collision between two balls in top-down-view . . . . . 25
3.4 The desired velocities and position $D$ for the shot illustrated in figure 3.2 for $t = 0.732065$ and a plot of equation 3.32 with these values . . . . . 29
3.5 (a) The original shot as illustrated in figure 3.2 (red / blue) and the calculated alternative shot (green / purple) (b) A 3D-plot of $f(t_D, t)$ in red and $g(t_D, t)$ in blue for $t_D, t \in [1, 1.5]$ . . . . . 31

4.1 (a) Expectimax and (b) Monte-Carlo sampling trees applied to a simple dice game . . . . . 40
4.2 Halting points in a sequence of three shots after executing them a thousand times. Red - after 1 shot, blue - after 2 shots, black - after 3 shots . . . . . 42
4.3 The top 5 break shots acquired using a particle swarm algorithm. JPool uses shot (a) . . . . . 47

5.1 Off-break-win percentage comparison of various agents . . . . . 50
5.2 Average success rate of shots calculated by JPool when executed using various noise models. Darker areas mark successful shots. . . . . 50

# Bibliography


[1] Alfieri, D., and Sander, U. *Positionsspiel im Poolbillard 2: Einstieg in den Poolbillard-Sport nach den Lehrmethoden der Pool School Germany.* No. v. 2. Litho-Verlag, 2003.

[2] Archibald, C., Altman, A., and Shoham, Y. Analysis of a winning computational billiards player. In *Proceedings of the 21st international jont conference on Artifical intelligence* (San Francisco, CA, USA, 2009), IJCAI'09, Morgan Kaufmann Publishers Inc., pp. 1377–1382.

[3] BCA. *Billiards: the official rules and records book.* Billiards : The Official Rules and Records Book. Billiard Congress of America, 2004.

[4] Dussault, J.-P., and Landry, J.-F. Optimization of a billiard player: position play. In *Proceedings of the 11th international conference on Advances in Computer Games* (Berlin, Heidelberg, 2006), ACG'05, Springer-Verlag, pp. 263–272.

[5] Dussault, J.-P., and Landry, J.-F. Optimization of a billiard player: tactical play. In *Proceedings of the 5th international conference on Computers and games* (Berlin, Heidelberg, 2007), CG'06, Springer-Verlag, pp. 256–270.

[6] Greenspan, M., Lam, J., Godard, M., Zaidi, I., Jordan, S., Leckie, W., Anderson, K., and Dupuis, D. Toward a competitive pool-playing robot. *Computer 41* (2008), 46–53.

[7] Harder, D. W. Newton's method in n dimensions (last accessed (07/05/2012). https://ece.uwaterloo.ca/ dwharder/NumericalAnalysis/10RootFinding/newtonND/.

[8] ICGA. Computer olypmpiad: Pool (last accessed 08/21/2012). http://www.grappa.univ-lille3.fr/icga/game.php?id=8.

[9] Koehler, J. *The Science of Pocket Billiards.* Sportology Publications, 1995.

[10] Landry, J.-F., Dussault, J.-P., and Mahey, P. Optimization of a billiard player. *J. Intell. Robotics Syst. 50*, 4 (Dec. 2007), 399–417.

[11] Landry, J.-F., Dussault, J.-P., and Mahey, P. A robust controller for a two-layered approach applied to the game of billiards. *Entertainment Computing 3*, 3 (2012), 59 – 70.

[12] Leckie, W., and Greenspan, M. A. Pool physics simulation by event prediction 1: Motion states and events. *ICGA Journal* (2005).

[13] Leckie, W., and Greenspan, M. A. An event-based pool physics simulator. In *Proceedings of the 11th international conference on Advances in Computer Games* (Berlin, Heidelberg, 2006), ACG'05, Springer-Verlag, pp. 247–262.

[14] Leckie, W., and Greenspan, M. A. Pool physics simulation by event prediction 2: Collisions. *ICGA Journal 29*, 1 (2006), 24–31.

[15] Leckie, W., and Greenspan, M. A. Monte-carlo methods in pool strategy game trees. In *Proceedings of the 5th international conference on Computers and games* (Berlin, Heidelberg, 2007), CG'06, Springer-Verlag, pp. 244–255.

[16] Marlow, W. *The Physics of Pocket Billiards*. MAST, 1995.

[17] Smith, M. *PickPocket: An Artificial Intelligence for Computer Billiards*. Canadian theses. University of Alberta (Canada), 2006.

[18] Smith, M. Running the table: an ai for computer billiards. In *Proceedings of the 21st national conference on Artificial intelligence - Volume 1* (2006), AAAI'06, AAAI Press, pp. 994–999.

[19] Stanford. 2011 international computational billiards championships (last accessed 07/06/2012). http://www.stanford.edu/group/billiards/software.html.

[20] Wikipedia. Particle swarm optimization (last accessed 06/28/2012). http://en.wikipedia.org/wiki/Particle_swarm_optimization.