technische universität
dortmund

fakultät für informatik

Fachprojekt

Erstellung eines Augmented Reality Billiardspiels

Friedemann Runte, Moritz Ludolph,
Robin Mertens, Diyar Omar

14. Juli 2018

Gutachter:

Prof. Dr. Vorname Nachname

M.Sc. Vorname Nachname

Lehrstuhl Informatik VII
Graphische Systeme
TU Dortmund

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation und Hintergrund

Die Aufgabe des Fachprojekts war eine Anwendung zu schreiben, die ein nicht-triviales Eingabegerät benutzt, genauso wie eine nicht-triviale, graphische Ausgabe. Wir haben uns dafür entschieden als Eingabe eine Kamera in Kombination mit OpenCV zu nutzen und als Ausgabe Rendering per OpenGL. Da wir bereits im Vorraus ein Projekt hatten, was ähnlich wie Billiard funktioniert hat, nämlich ein Airhockey Spiel auf einem Tablet-Tisch, haben wir uns dafür entschieden das als Basis zu benutzen, um unsere Probleme hauptsächlich auf die beiden geforderten Gebiete zu verschieben. Damit hatten wir als Grundlage ein bereits funktionierendes Kollisionssystem für eine Kugel mit statischen Objekten, sowie leichtes Rendering für Kreise. Zudem gab es auch eine Vorbereitungsaufgabe in OpenCV die uns schon ein Verständnis für Kamera-Kalibrierung gebracht hat.

1.2 Zielsetzung

Es soll ein Billiard-Computerspiel entstehen, welches durch einen Beamer auf einen Tisch projiziert wird und dann mit einem gewöhnlichen, schwarz gefärbten Stock gespielt wird. Der Stock fungiert dabei als Queue und wird mit einer Kamera erkannt.

1.3 Aufbau der Arbeit

Wir werden zunächst unsere Probleme erläutern, die uns auf dem Weg zum Endprodukt aufgekommen sind. Anschließend werden wir erklären, wie wir diese Probleme gelöst haben. Zum Schluss wird dann das Endprodukt evaluiert und unsere angewandten Lösungsstrategien diskutiert.

2 Problemstellung

Die Zielsetzung dieses Projektes war es, ein Billiard-Spiel zu entwickeln, welches die Interaktion mit dem Spiel durch einen realen Queue ermöglicht. Dabei sollte das Stoßen durch eine natürliche Bewegung mit den Queue gegen die projizierten Kugeln möglichen. Bei der Realisierung dieser Anforderungen ergeben sich mehrere Teilprobleme, die im folgenden näher erläutert werden sollen.
Ein erstes Problem bildet die Darstellung und die Physik des eigentlichen Spiels. Hierzu müssen das Spielfeld inklusive Löchern sowie die Kugeln korrekt dargestellt werden. Die Kugeln sollen dabei mit unterschiedlichen Farben und Nummern sowie die Unterscheidung zwischen vollen und halben Kugeln dargestellt werden. Weiterhin muss die Physik des Billiardspiels simuliert werden, die Kollisionen an Kugeln und Banden behandelt sowie das Einlochen von Kugeln erkennt.
Um durch den Queue mit dem Spiel zu interagieren, wird das projizierte Spielfeld durch eine Kamera aufgenommen. In dem Kamerabild müssen die für die Spielphysik relevanten Punkte des Queues erkannt werden. Weiterhin muss es möglich sein, die im Bild erkannten Punkte korrekt in Spielkoordinaten umzuwandeln. Da durch die Kameralinse jedoch das Bild verzerrt wird, muss neben der Erkennung des Spielfeldes auch eine Kalibrierung der Kamera durchgeführt werden.
Schließlich müssen die Spielregeln in das Spiel eingebunden werden sowie eine grafische Benutzeroberfläche entwickelt werden, die die einzelnen Komponenten verbindet.

3 Methoden und Lösungswege

In diesem Kapitel sollen die zuvor dargestellten Teilprobleme erneut aufgegriffen und eine Möglichkeit zur Lösung dargestellt werden.

3.1 Rendering und Physik

Das erste Problem ist das Darstellen des Spiels und die Interaktion zwischen den im Spiel existierenden Kugeln und dem Spielfeld. Zunächst schauen wir uns die Darstellung an.

3.1.1 Objekte

Die Szene lässt sich einteilen in Löcher und Kugeln. Beide Strukturen werden aufgrund der Beschränkung auf 2D durch Kreise oder Kreisteile dargestellt. Dazu wird ein Triangle-Fan verwendet, der nun kurz erklärt werden soll.
Ein Triangle-Fan zur Darstellung von Kreisen wird durch das Tripel Mittelpunkt $\mathbf{c}$, Radius r und Anzahl der Unterteilungen k parametrisiert. Die Struktur beginnt im Punkt $(0, 0)$. Das ist im Späteren Spielfeld der Punkt $\mathbf{c}$. Man gibt nun den Punkt an, bei dem das erste Dreieck beginnt. Dieser ist bei uns immer vertikal um r nach oben verschoben vom Mittelpunkt. Dann benötigt man noch einen Punkt um das Dreieck zu vollenden. Dieser Punkt ist im Uhrzeigersinn um δ auf dem Umriss verschoben. δ ist der Grad, um den in jedem Schritt der Punkt auf dem Umriss verschoben wird. Dabei muss gelten $\delta \cdot k = 360$ für einen ganzen Kreis.

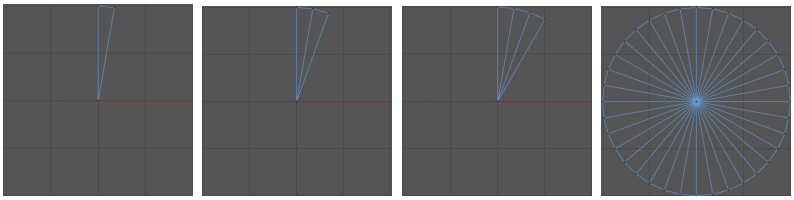
$$\delta = \frac{2 \cdot \pi}{k} \tag{3.1}$$

Sei r der Radius von der Kugel und i der aktuelle Schritt in der Schleife von $i = 1$ bis k, dann sind die Koordinaten für die Punkte auf dem Kreis:

$$\begin{aligned} x &= \cos(\delta \cdot i) \cdot r \\ y &= \sin(\delta \cdot i) \cdot r \end{aligned} \tag{3.2}$$

Für jeden Punkt der hinzugefügt wird, wird ein neues Dreieck erstellt, mit dem letzten Punkt, dem Mittelpunkt und dem neuen Punkt als Parameter. Wenn nun k mal die Koordinaten berechnet wurden, haben wir einen fertigen Kreis, bestehend aus k Dreiecken.

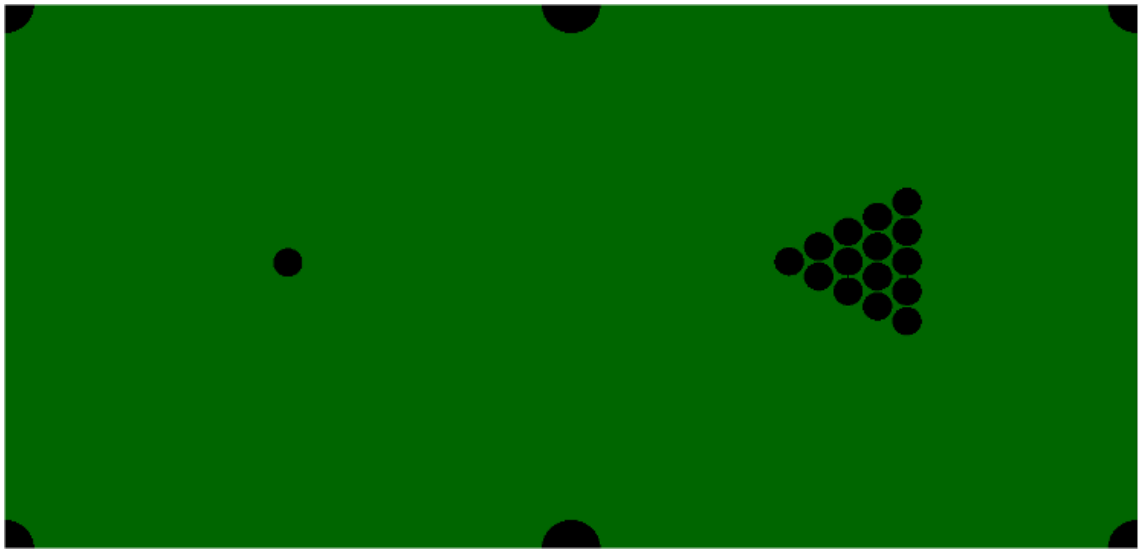
Abbildung 3.1: Bau eines Triangle-Fans mit k=36



(a) i=1 (b) i=2 (c) i=3 (d) i=k=36

Die Triangle-Fans werden nun für die Löcher und Kugeln an den Positionen auf dem Spielfeld plaziert.

Abbildung 3.2: Spielfeld ohne Texturen, mit k=64 für Triangle-Fans

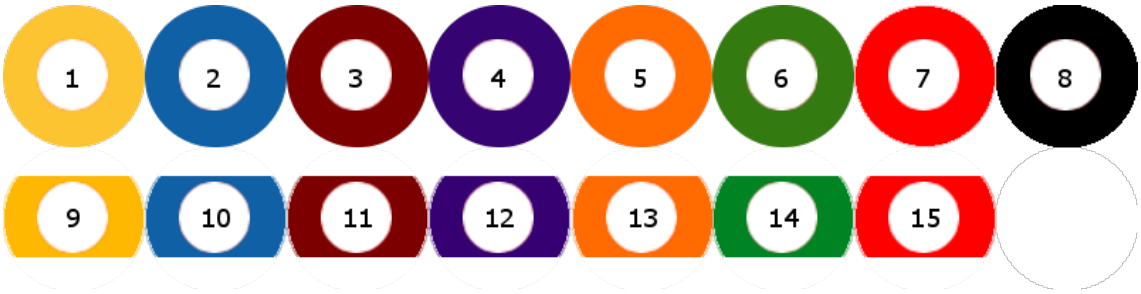


3.1.2 Texturierung

Die Farben für das Spielfeld sind trivial. Die Kugeln hingegen müssen vom Spieler unterschieden werden können. Wir müssen also Texturen auf Kreise abbilden.

Um die Textur auf unsere Kugeln abzubilden verwenden wir u/v-Koordinaten. Unsere Textur besitzt auf der u und v Achse einen Wertebereich von 0 bis 1. Dabei ist

Abbildung 3.3: Billiardkugel Textur



$(0, 0)$ oben links und $(1, 1)$ unten rechts. Jeder Kugel muss nun ein Ausschnitt der Textur zugewiesen werden. Dafür bekommt jede Farbe einen Wert c.

$$c(b) = \begin{cases} 0, & \text{falls b Gelb} \\ 1, & \text{falls b Blau} \\ 2, & \text{falls b Rot} \\ 3, & \text{falls b Lila} \\ 4, & \text{falls b Orange} \\ 5, & \text{falls b Grün} \\ 6, & \text{falls b Rot} \\ 7, & \text{falls b Schwarz oder Weiß} \end{cases} \tag{3.3}$$

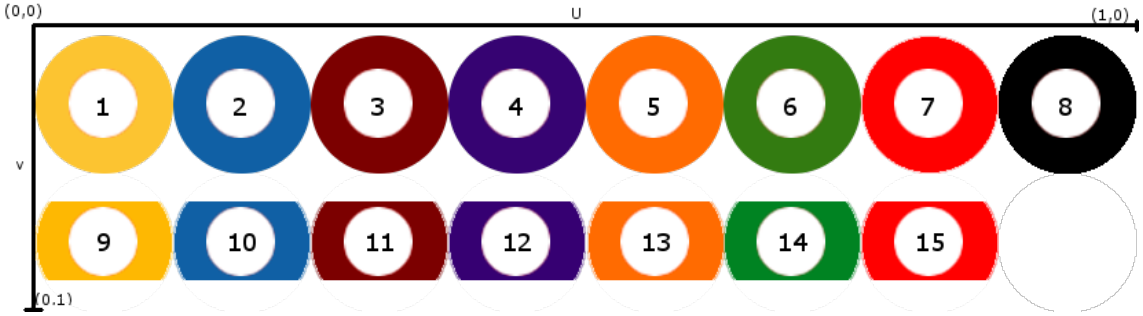
Außerdem bekommt jede Kugel einen Wert für die Fülle, wobei Voll = 0 ist und Halb = 1.

$$f(b) = \begin{cases} 0, & \text{falls Kugel voll} \\ 1, & \text{falls Kugel halb} \end{cases} \tag{3.4}$$

Nun können wir uns eine Funktion erstellen, die anhand von Farbe und Fülle die Textur ausschneidet und auf die Kugel abbildet. Dazu berechnen wir zuallererst die Anfangsposition der Textur. Diese setzt sich zusammen aus der Farbe c und der Fülle f.

Vorgedanke: Der Durchmesser einer Kugel auf unserer Textur ist $\frac{1}{8}$ für u, weil es 8 Kugeln pro Reihe gibt und die u Koordinate genau 1 lang ist. Der Durchmesser auf der v-Achse hingegen ist $\frac{1}{2}$, weil es nur 2 Reihen gibt und die Textur trotzdem 1

Abbildung 3.4: Textur mit u/v-Koordinaten



lang ist.
Mittelpunkt m der Textur:

$$\begin{aligned} u(b) &= \frac{1}{16} + \frac{1}{8} \cdot c(b), c(b) = \text{Farbe der Kugel (3.3)} \\ v(b) &= \frac{1}{4} + \frac{1}{2} \cdot f(b), f(b) = \text{Fülle der Kugel (3.4)} \end{aligned} \tag{3.5}$$

Nun fehlen noch die Texturkoordinaten für den Kreis um den Mittelpunkt der Textur herum. Die Berechnung läuft dabei analog zum Triangle-Fan mit dem Unterschied in der Skalierung und Startposition. So benutzen wir nicht den Radius der Kugeln des Spiels, sondern die Größe der Kugel auf der Textur und als Startpunkt unser vorher berechnetes u und v in (3.5).

$$\begin{aligned} xTex(b) &= u(b) + \cos(\delta \cdot i) \cdot \frac{1}{16}, \\ yTex(b) &= v(b) + \sin(\delta \cdot i) \cdot \frac{1}{4}, \\ \text{mit } \delta &\text{ nach (3.1) und } i \text{ nach (3.2)} \end{aligned} \tag{3.6}$$

Die Textur wird dann beim erstellen des Triangle-Fans auf die Dreiecke gezeichnet. Dabei bekommen die Dreiecke für den Mittelpunkt (x, y) als Texturkoordinaten (3.5) und der Punkt auf dem Umriss bekommt dann $(xTex, yTex)$ (3.6).

3.1.3 Kollisionsberechnung

Die Kollisionsberechnung lässt sich aufteilen in 3 Bereiche:

1. Kollision von Kugeln mit Wand und Löchern
2. Kollision von Kugeln mit anderen Kugeln
3. Kollision der Weißen Kugel mit dem Queue (Kapitel 3.3.3)

Kollision von Kugeln mit Wand und Löchern

Das Kollidieren mit den Wänden ist sehr simpel: Wir geben unsere Breite und Höhe des Spielfeldes als Grenzen an. Wenn die Kugel zu nah an eine der Wände kommt wird ihre Geschwindigkeit umgekehrt. Dementsprechend machen wir eine Fallunterscheidung mit 4 Fällen für die 4 verschiedenen Wände:

1. Die Linke Wand (x=0)
2. Die Rechte Wand (x=w)
3. Die Obere Wand (y=0)
4. Die untere Wand (y=h)

Dabei gilt h = Höhe des Spielfeldes, w = Breite des Spielfeldes und l = Radius der Löcher. Die Änderung der Geschwindigkeit ist dann:

$$vx = \begin{cases} -vx, & \text{falls } y \leq (h-l) \wedge y \geq l \\ vx, & sonst \end{cases} \tag{3.7}$$

Das gilt für Fall 1 und 2. Für Fall 3 und 4 benötigen wir noch eine Bedingung mehr:

$$vy = \begin{cases} -vy, & \text{falls } (x \leq (w/2-l) \wedge x \geq l) \vee (x \geq (w/2+l) \wedge (x \leq (w-l) \\ vy, & sonst \end{cases} \tag{3.8}$$

Wir überprüfen also, ob die Kugel eine der Beiden Strecken zwischen den Löchern trifft, wenn sie vorher den Rand des Spielfeldes erreicht hat.

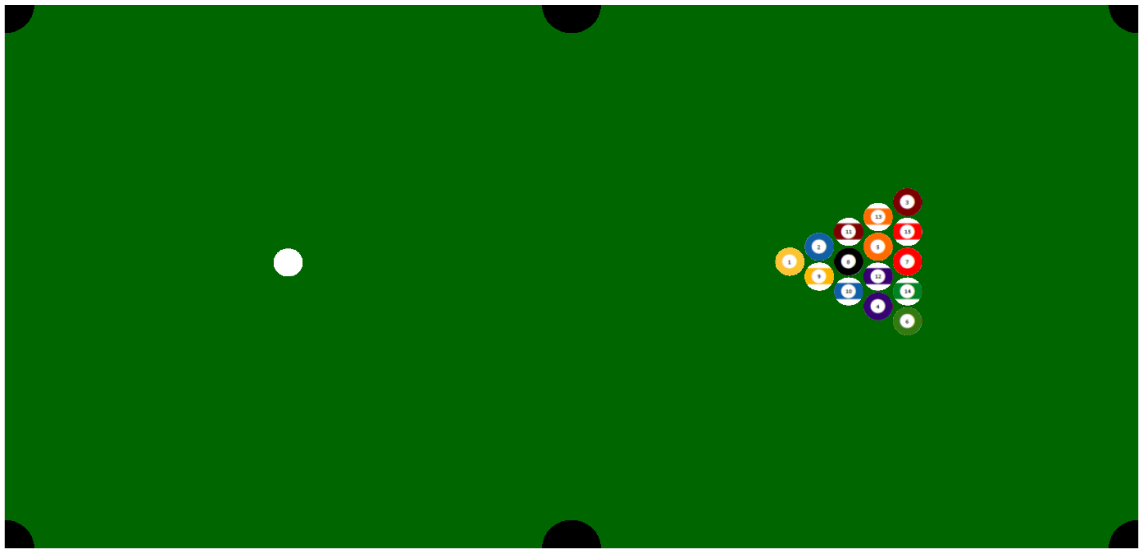
Kollision mit anderen Kugeln

Die Kollisionsberechnung für 2 Kugeln war uns bereits vorgegeben aus einem Beispiel für ein AirHockey Spiel. Der Unterschied liegt darin, dass im Airhockey nur eine Kugel durch die Kollision ihre Geschwindigkeit ändert. Die beiden Schläger werden durch die Kollision nicht verändert.
Wir nehmen nun eine Kugel und berechnen mit der Methode alle Kollisionen mit anderen Kugeln, indem wir statt dem Schläger die jeweils andere Kugel einsetzen. Das muss für jede Kugel einmal ausgeführt werden.

Zusammenfassung

Nun haben wir ein Spielfeld mit Texturierten Kugeln die miteinander kollidieren.

Abbildung 3.5: Fertiges Spielfeld



3.2 Kamerakalibrierung

3.2.1 Darstellung des Erkennungsmusters

Um die Kamera kalibrieren und das Kamerabild bzw. die -punkte entzerren zu können, muss die Kamera ein bekanntes Erkennungsmuster finden und in diesem bestimmte Eckpunkte erfassen und auswerten können. In unserem Projekt wird ein Schachbrettmuster genutzt, von welchem die inneren Eckpunkte erkannt werden. Hierfür muss die Anzahl der Kacheln in der Horizontalen $= Hor$, sowie in der Vertikalen $= Vert$, bekannt bzw. festgelegt sein, um für jede Kachel die selbe Höhe und Breite zu haben und trotzdem die gesamte Höhe und Breite des Spielfeldes abzudecken. Die Höhe und Breite wird wie folgt berechnet:
$Hoehe_{Kachel} = \frac{Hoehe_{Game}}{Vert}$, $Breite_{Kachel} = \frac{Breite_{Game}}{Hor}$
Nun wird für jedes $i \in [1, Hor]$ alle $j \in [1, Vert]$ eine Kachel gerendert mit den Koordinaten oben links $(i \cdot Hoehe_{Kachel}, j \cdot Breite_{Kachel})$ und den Koordinaten unten rechts $((i+1) \cdot Hoehe_{Kachel}, (j+1) \cdot Breite_{Kachel})$. Zudem wird bei jeder neuen Kachel in der Horizontalen die invertierte Farbe der vorherigen Kachel als Grundfarbe gewählt und in der Vertikalen für die erste Kachel die invertiere Farbe der darüber liegenden Kachel gewählt, damit das klassische Schwarz-Weiß-Muster eines Schachbretts visualisiert wird.

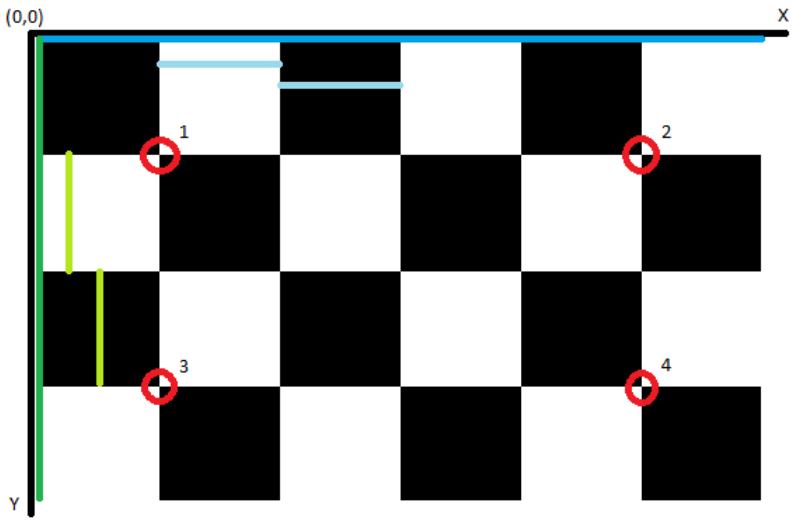


Abbildung 3.6: Schachbrettmuster rendern

3.2.2 Entzerrung des Kamerabildes/Punktes

Für die Kalibrierung des Kamerabildes mittels eines Patternmusters, muss zwischen zwei Ansichten unterschieden werden. Zum einen sind das die Weltkoordinaten, welche in einem 3D-Koordinatensystem liegen, zum anderen die Bildkoordinaten, welche in einem 2D-Koordinatensystem liegen. In unserem Fall wird in den Weltkoordinaten ein bekanntes Muster bzw. die Erkennungspunkte des Schachbretts festgehalten, um im Anschluss das erkannte Muster auf dem Foto mit diesem abzugleichen und somit die Entzerrungsparameter herauszufinden. Da dies eine festgelegte Anzahl an Erkennungspunkten aufweist, werden genau diese Erkennungspunkte, mit einer festen Kantenlänge für jede Kachel, im Weltkoordinatensystem modelliert. Da die übergebenen Punkte nachher in ein 2D-Koordinatensystem, das vom Spiel, dargestellt werden sollen, werden die Erkennungspunkte des Schachbrettmusters als 2D-Punkte gespeichert:
$WeltCoords = (x \cdot a, y \cdot a, 0), a =$ Kantenlänge, $x \in [0, Vert - 1], y \in [0, Hort - 1]$
Nun werden die aufgenommenen Bilder der Kamera von dem, im vorherigen Abschnitt gerenderten, Schachbrettmuster analysiert, ob auf diesen das bekannte Pattern erkennbar ist. Dies wird mit der OpenCV-Methode $findChessboardCorners$ gemacht, welche neben dem aufgenommenem Bild auch die Anzahl der vertikalen und horizontalen inneren Musterpunkte als Parameter übergeben bekommt und, falls alle gefunden wurden, einen Wert ungleich 0 zurück gibt und diese in einem Bildkoordinatensystem festgehalten, ansonsten 0. Falls die Rückgabe ergibt, dass das Muster erkannt wurde, existiert nun für jedes Bild, auf dem das Patternmuster erkannt wurde, ein Bild- und ein Weltkoordinatensystem. Im Anschluss wird die eigentliche Kamerakalibrierung gestartet, welche durch folgende Formel repräsentiert wird:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Als Eingabe werden die Bildkoordinaten als 2D Matrix, sowie die Weltkoordinaten als 3D Matrix der gefundenen Pattern und der Anzahl vertikaler und horizontaler Fixpunkte im Schachbrett übergeben. Aus diesen Parametern wird eine 3x3 Kamera- bzw. intrinsische Matrix, welche den Standpunkt der Kamera in der Welt darstellt, sowie ein Vektor mit den Verzerrungskoeffizienten und die Rotations- und Translationsvektoren, welche die extrinsischen Parameter darstellen. Kurz gefasst

werden die Weltkoordinaten zu Kamerakoordinaten unter Verwendung der extrinsischen Parameter und die Kamerakoordinaten zu Bildkoordinaten unter Verwendung der intrinsischen Parameter. Nun sind alle nötigen Eingaben gegeben um einen be-

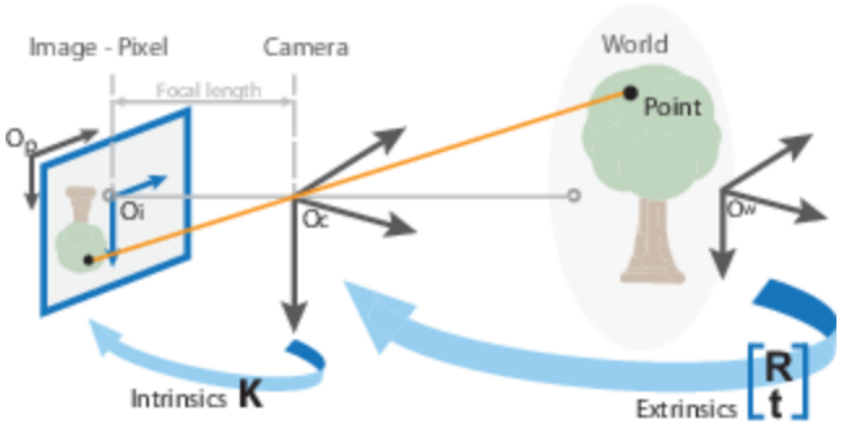


Abbildung 3.7: Extrinsisch und Intrinsisch

liebigen Punkt aus dem verzerrten Bild zu entzerren. Hierfür wird zunächst, mit Hilfe von *undistortPoints*, der Eingabe des 2D Punkts, der Kameramatrix und den Verzerrungskoeffizienten, der eingegebene Punkt entzerrt. Anschließend muss dieser entzerrte Punkt noch korrekt rotiert werden. Hierfür wird eine 2x3 Rotationsmatrix benötigt. Da bisher nur Rotationsvektoren ermittelt wurden, wird zunächst mit *Rodrigues* der (erste) Rotationsvektor in eine 3x3 Rotationsmatrix konvertiert. Daraufhin werden die ersten beiden Reihen dieser Matrix mit *vconcat* zu einer 2x3 Matrix umgeformt. Als letzten Schritt wird auf den entzerrten Punkt diese Rotationsmatrix mit $warpAffine$ angewendet um den endgültig entzerrten und rotierten Punkt zu erhalten. Somit besteht nun das vollständige Verfahren einen beliebigen Punkt aus den Bildkoordinaten korrekt in die Weltkoordinaten zu transformieren.

3.2.3 Punktübertragung von Kamera ins Spiel

Der letzte Abschnitt der Kamerakalibrierung stellt das Übertragen eines von der Queue-Erkennung gegebenen Punktes, welcher sich im Kamerakoordinatensystem befindet, in das Spielfeldkoordinatensystem um diesen dann in den Spielmechanismus als Queueposition einzubinden. Um dies einwandfrei zu ermöglichen, müssen die äußersten Eckpunkte des, von der Kamera erfassten, Schachbrettmusters im Kamerakoordinatensystem bekannt sein. Da allerdings nur die Eckpunkte auf der Innenseite des äußersten Ringes des Schachbretts erkannt werden, müssen die ganz

äußersten Eckpunkte wie folgt berechnet werden:
$Lila_{min} = Gruen_{min} - Rot_{min}$, $Lila_{max} = Rot_{max} - Gruen_{max}$

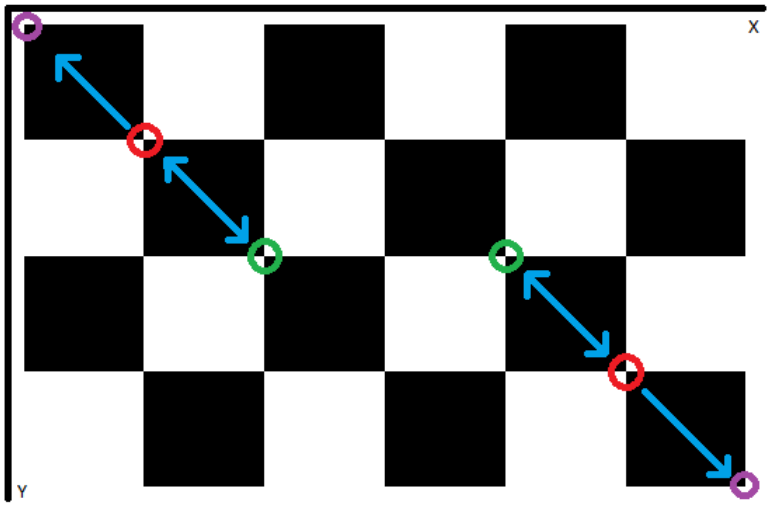


Abbildung 3.8: Schachbrettmuster Eckpunkte

Nun sind die äußersten Eckpunkte des Spielfeldes im Kamerakoordinatensystem bekannt und es ist möglich einen beliebigen 2D Punkt in das Spiel zu projizieren. Hier wird unter zwei Fällen unterschieden. Der übergebene Punkt der Kamera liegt im Spielbereich gdw. $X \in [x_{min}, x_{max}]$ und $Y \in [y_{min}, y_{max}]$ gilt. Sofern dies erfüllt ist, wird der gegebene Punkt mit folgenden Gleichungen jeweils mit der X- und Y-Koordinate umgerechnet :

$$X = \frac{(P.x - MIN.x) \cdot UR}{(MAX.x - MIN.x)}, Y = \frac{(MAX.y - P.y) \cdot OL}{(MAX.y - MIN.y)}$$

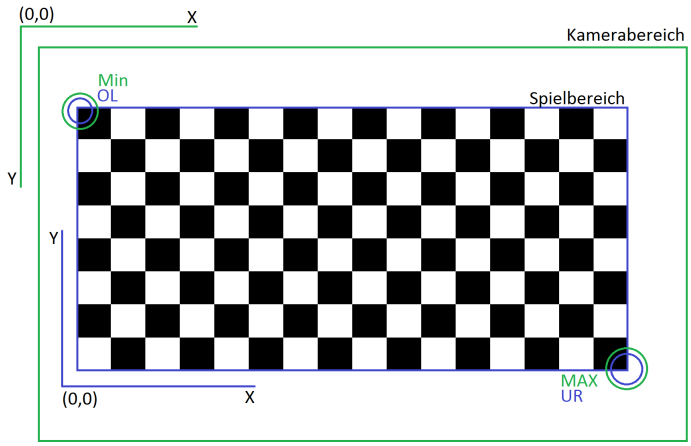


Abbildung 3.9: Kamera- und Spielkoordinatensystem

3.2.4 Zusammenfassung

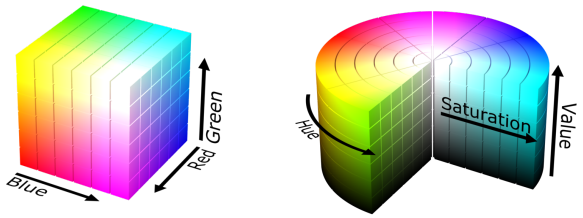
Zusammenfassend wurde ein, von der Kamera erfasster, Punkt in das zweidimensionale Spielfeld projiziert. Dafür wurde zunächst ein Muster generiert und angezeigt, in welchem die Kamera bekannte Fixpunkte erkennen und mit diesen das aufgenommene Bild entzerren kann. Mit den Fixpunkten wurden die äußersten Kanten des Musters berechnet und mit diesen der Anfangs gegebene Punkt ins Spiel übertragen.

3.3 Queue-Detektion

Wie schon zuvor erläutert, soll es dem Spieler möglich sein, mit dem realen Queue die simulierten Kugeln zu stoßen. Dazu wird das Spielfeld von einer Kamera erfasst. Um die Interaktion des Queues mit den Kugeln zu ermöglichen, müssen in dem Eingabebild der Kamera die für die Kollisionsberechnung relevanten Punkte des Queues extrahiert werden.
In den folgenden Abschnitten wird dazu eine Möglichkeit beschrieben, welche zunächst mittels eines Schwellwertverfahrens den Queue vom Hintergrund segmentiert. Basierend auf dem segmentierten Bild werden dann unter Verwendung der Hauptkomponentenanalyse zwei Kollisionspunkte ermittelt. Schließlich wird die Einbindung der Kollisionspunkte in die Kollisionsberechnung erläutert, welche die Detektion des Queues abschließt.

3.3.1 Segmentierung

Sei $\mathbf{I} \in \{0, \ldots, 255\}^{b \times h \times 3}$ das von der Kamera aufgenommene $b \times h$ große RGB-Farbbild. Ziel der Segmentierung ist es, ein Binärbild $\mathbf{B} \in \{0, 1\}^{b \times h}$ zu erzeugen, welches die Pixel im Eingabebild charakterisiert, die dem Queue zugeordnet werden. Da der Queue mit einer schwarzen Farbe lackiert wurde, wird dies mithilfe eines Schwellwertverfahrens basierend auf dem Farbwert der Pixel bewerktstelligt. Das im RGB-Farbraum vorliegende Eingabebild $\mathbf{I}$ wird dazu zunächst in den HSV-Farbraum überführt. Im HSV-Farbraum wird ein Farbwert durch seinem Farbton $H \in [0, 360]$, der Sättigung $S \in [0, 1]$ und dem Helligkeitswert $V \in [0, 1]$ dargestellt.



(a) (b)

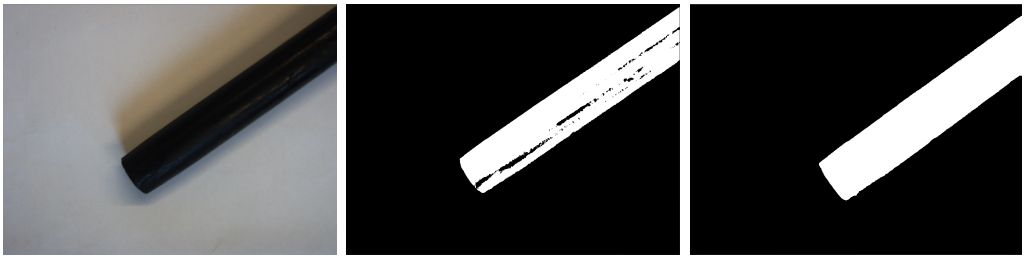
Abbildung 3.10: Darstellung von Farbtönen im (a) RGB-Farbraum [2] durch Rot-, Grün- und Blauanteil, und im (b) HSV-Farbraum [1] durch Farbton (Hue), Sättigung (Saturation) und Helilgkeitswert (Value). Dunkle Farben im HSV-Farbraum haben einen kleinen Helligkeitswert.

Somit kann allein durch einen Schwellwert θ für den Helligkeitswert der schwarze Queue vom Rest des Bildes segmentiert werden. Für das transformierte Bild $\mathbf{I}_{HSV}$

ergibt sich die Berechnung des Wertes von $\mathbf{B}$ an der Stelle i, j für einen Schwellwert θ somit durch folgende Formel:

$$\mathbf{B}[i,j] = \begin{cases} 1 & \text{, falls } \mathbf{I}_{HSV}[i,j,3] \leq \theta \\ 0 & \text{sonst} \end{cases}$$

Durch die unterschiedliche Beleuchtung des Queues ergeben sich in dem entstehenden Binärbild größere Lücken, die die weitere Erkennung des Queues beeinflussen würden. Um dies zu verhindern werden diese durch die Anwendung eines morphologischen Closings gefüllt. Bei einem morphologischen Closing wird zunächst eine Dilatation und daraufhin eine Erosion auf dem Bild mit einem $k \times k$ großen, ellipsoiden Fenster durchgeführt. Für die Dilatation wird mit dem Fenster über das Bild gelaufen und dabei der zentrale Pixel des Fensters auf den maximalen Wert im Fenster gesetzt. Analog dazu wird bei der Erosion der Wert des zentralen Pixels auf den minimalen Wert im Fenster gesetzt. Insgesamt werden durch diese beiden Operationen die Lücken im Binärbild geschlossen ohne die Kontur des Queues zu vergrößern.



(a) (b) (c)

Abbildung 3.11: Ergebnis der Segmentierung des Eingabebildes. (a) Eingebild mit 1024×768 Pixeln, (b) Binärbild aus Schwellwertverfahren ($\theta = 30$), (c) Ergebnis des morphologischen Closings ($k = 9$)

3.3.2 Erkennung der Queue-Enden

Um nicht mit jedem im Binärbild als Queue klassifizierten Punkt kollidieren zu müssen wird nur mit den Enden des Queues kollidiert. Zur Berechnung der Position der beiden Enden wird zunächst mittels der Hauptkomponentenanalyse [3] die Hauptachse des Queues im Bild bestimmt. Dazu werden die Koordinaten i, j der Punkte aus $\mathbf{B}$ mit $\mathbf{B}[i,j] = 1$ als Zeilenvektoren in einer $n \times 2$ Matrix $\mathbf{X}$ zusammengeführt. Daraufhin wird der Mittelpunkt $\mathbf{c}$ der n Punkte sowie die Abweichung $\mathbf{A}$ zu dem

Mittelpunkt berechnet:

$$\mathbf{c}[j] = \frac{1}{n}\sum_{i=1}^{n}\mathbf{X}[i,j] \text{ mit } j = 1,2$$

$$\mathbf{A} = \mathbf{X} - \mathbf{h}\mathbf{c}^T \text{ mit } \mathbf{h}[i] = 1 \text{ für } i = 1,\ldots,n$$

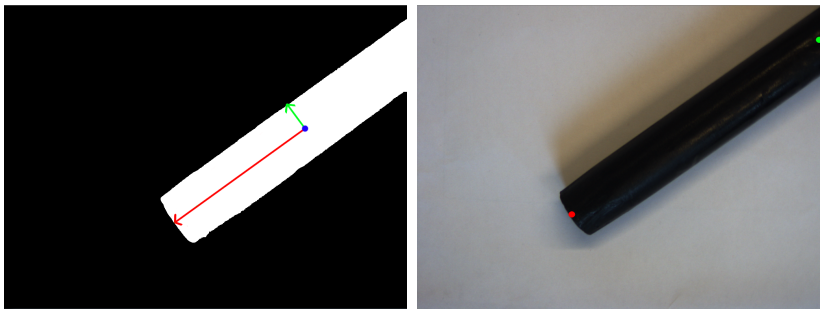
Aus der Matrix $\mathbf{A}$ lässt sich nun die 2×2 empirische Kovarianzmatrix $\mathbf{C}$ berechnen:

$$\mathbf{C} = \frac{1}{n-1}\mathbf{A}^T\mathbf{A}$$

Schließlich werden die zwei Eigenvektoren und Eigenwerte von $\mathbf{V}$ bestimmt. Da der Eigenvektor $\mathbf{e}$, dem der größere Eigenwert λ_e zugeordnet ist, in die Richtung der größten Varianz der Punkte zeigt, ist die gesuchte Hauptachse gegeben durch $\mathbf{x}(t) = t \cdot \mathbf{e} + \mathbf{c}$.
Um nun die Endpunkte des Queues zu bestimmen, müssen alle Punkte auf die zuvor bestimmente Hauptachse projiziert werden. Für einen Punkt $\mathbf{x} = (i,j)$ mit $\mathbf{B}[i,j] = 1$ ergeben sich die neuen Koordinaten $\mathbf{x}'$ durch

$$\mathbf{x}' = s \cdot \mathbf{e} + \mathbf{c} \text{ mit } s = \frac{\mathbf{e} * (\mathbf{x} - \mathbf{c})}{||\mathbf{e}||}$$

Dabei bezeichnet $*$ das Skalarprodukt zweier Vektoren sowie $||\cdot||$ die euklidische Norm eines Vektors. Die beiden Punkte, die am weitesten von dem Mittelpunkt $\mathbf{c}$ entfernt sind, sind die beiden Enden des Queues. Für diese Punkte $\mathbf{x}'_1$, $\mathbf{x}'_2$ ist s entweder minimal oder maximal.



(a) (b)

Abbildung 3.12: Erkannte Endpunkte des Queues. (a) Binärbild mit Zentrum $\mathbf{c}$ in blau sowie die berechneten Eigenvektoren in rot und grün, (b) Originalbild mit erkannten Endpunkten des Queues

Da das Vorzeichen für s sich in aufeinanderfolgenden Bildern verändern kann, müs-

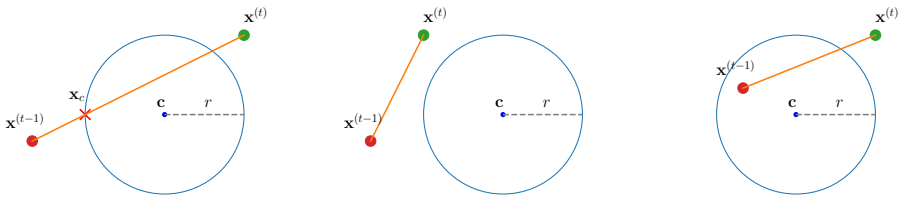
sen die Punkte für die Kollisionsberechnung korrekt ihrem Vorgänger zugeordnet werden. Dazu wird ein Punkt immer dem näheren der beiden Vorgänger-Punkte zugeordnet. Weiterhin kann keines der beiden Enden mit Sicherheit als Spitze des Queues klassifiziert werden. Daher wird die Kollisionsbehandlung mit beiden Enden ausgeführt, was auf den Spielfluss jedoch keine Auswirkungen hat da eines der Enden meist am Rand des Spielfeldes ist. Die erkannten Endpunkte werden schließlich wie in Abschnitt 3.2.3 beschrieben, in das Spielkoordinatensystem überführt.

3.3.3 Kollisionserkennung

Bei Stoßbewegungen mit dem Billard-Queue kann es zu hohen Geschwindigkeiten der Queue-Enden kommen. Da die Kamera mit einer Wiederholfrequenz von 25 Bildern pro Sekunde Bildaufnahmen macht, kann es passieren, dass ein Ende sich zu schnell bewegt und über eine Kugel hinwegspringt ohne ein Kollision auszulösen. Daher kann die Kollisionsüberprüfung nicht nur auf der aktuellen Position des Queue-Endes basieren. Stattdessen wird der Kollisionspunkt durch das Schneiden der Kugel und einem Liniensegment berechnet. Das Liniensegment ergibt sich dabei durch die aktuelle Position und der voherigen Position der Queue-Enden.
Die Schnittpunkte einer Kugel mit Mittelpunkt $\mathbf{c}$ und Radius r und einem Liniensegment mit Startpunkt $\mathbf{x}^{(t-1)}$ und Endpunkt $\mathbf{x}^{(t)}$ lassen sich wie folgt berechnen:

$$\mathbf{d} = \mathbf{x}^{(t)} - \mathbf{x}^{(t-1)} \qquad \mathbf{f} = \mathbf{x}^{(t-1)} - \mathbf{c} \qquad s_{1,2} = \frac{-2 \cdot (\mathbf{d} * \mathbf{f}) \mp \sqrt{||\mathbf{f}||^2 - r^2}}{2 \cdot ||\mathbf{d}||^2}$$

Falls $s_1 \in [0, 1] \subset \mathbb{R}$ ist, so ist der Schnittpunkt durch $\mathbf{x}_c = s_1 \cdot d + \mathbf{x}^{(t-1)}$ gegeben. Mit diesem wird dann die übliche Kollisionsberechnung durchgeführt. In den anderen Fällen darf keine Kollision ausgeführt werden, da entweder kein Schnittpunkt exisiert oder der Schnittpunkt kein gültiger Kollisionspunkt wäre:



(a) (b) (c)

Abbildung 3.13: Unterschiedliche Fälle bei der Kollisionserkennung. (a) Kollision in $\mathbf{x}_c$, (b) Keine Kollision, (c) Kollision mit Schnittpunkt wäre fehlerhaft

3.4 Spielregeln und Benutzerinteraktion

Damit die Benutzerinteraktion funktioniert, müssen zunächst die Spielregeln ins Spiel eingebunden werden. Das Spiel das 8er- Ball-Billiard:
8er Ball wird mit einem Spielball (weiss) und 15 nummerierten farbigen Kugeln gespielt. 14 der farbigen Kugeln werden in zwei Gruppen eingeteilt. Nr. 1 bis 7 sind vollfarbige (volle) Kugeln. Nr.9 bis 15 sind gestreiftfarbige (halbe) Kugeln. Zu Beginn werden alle 15 farbigen Bälle mit dem Dreieck so aufgestellt, dass die vorderste Kugel des Dreiecks auf dem Fusspunkt zu liegen kommt. Ziel des Spieles ist es, zuerst eine Serie von vollen oder halben Bällen und zuletzt den 8er Ball in die Löchern zu versenken. Jeweils der Gewinner eines Spieles eröffnet das nächste Spiel(Alternativ kann auch abwechslungsweise angespielt werden).

3.4.1 Spielregeln

Sobald der erste Spieler seinen Zug gemacht hat, started das Spiel. Sollte der Spieler einen Ball eingelocht haben, überprüft die Spiellogik, ob es sich hierbei um eine vollen Ball handelt oder halben Ball. Entsprechend wird dabei der Balltyp des Spielers auf 'True' für einen vollen Ball oder auf 'False' für einen halben Ball gesetzt. Anschließend darf der derzeitig aktuelle Spieler einen weiteren Zug machen.

$$BallTyp = \begin{cases} True, & \text{Für Volle Kugel} \\ False, & \text{Für Halbe Kugel} \end{cases} \tag{3.9}$$

$$currentPlayer = \begin{cases} 0, & \text{Für Ersten Spieler} \\ 1, & \text{Für Zweiten Spieler} \end{cases} \tag{3.10}$$

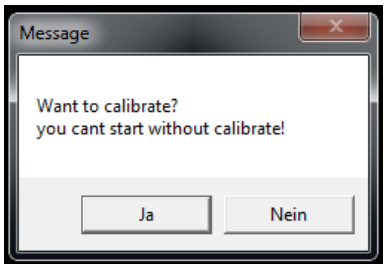
Hat der Spieler den Ball nicht eingelocht, dann wird der Spieler gewechselt.

dabei wird der BallType beim ersten spielzug nicht gesetzt, da niemand eine Farbe zugeordnet bekommen hat. Um das Spiel zu Gewinnen muss man alle Kugeln eingelocht haben, die für den Spieler zugewiesen wurden. Die Letzte Kugel ist die Schwarze Kugel. Sollte diese versenkt worden sein, bevor alle Kugeln des Spielers versenkt sind, dann hat der Aktuelle Spieler entsprechen Verloren und ein Pop-up Fenster erscheint. Andernfalls hat Er Gewonnen.

3.4.2 Benutzerinteraktion

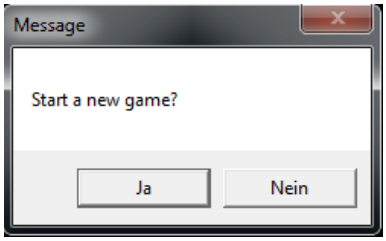
Bevor ein Benutzer das Programm gestartet hat, müssen vorher einige voraussetzungen erfüllt werden. Das Programm benötigt zu ausführung eine Kamera. Diese wird zum Kalibrieren des Spielfeldes benötigt. Sollte die Kamera verbunden worden sein, so muss sie nun auf das Programm-Fenster zeigen. Zu Begin bei der Ausführung des Programms, wird ein Pop-up Fenster angezeigt, welche fragt ob die Kalibrierung gestartet werden kann(siehe Abbildung 3.6: Kalibrierung-Pop-up).

Abbildung 3.14: Kalibrierung-Pop-up



Mit dem Drücken des 'Ja'-Knopfes wird ein Signal richtung der Kamerakalibrierung geschickt und sie wird gestartet. Sollte 'Nein' gedrückt worden sein, so wird das Programm geschlosse, da das Programm die kalibrierung benötigt. Nach der Kalibrierung wird ein weiteres Pop-up Fenster angezeigt, ob man das Spiel nun starten möchte. Wenn 'Ja' gedrückt wird, dann wird das ganze Spielfeld angezeigt.

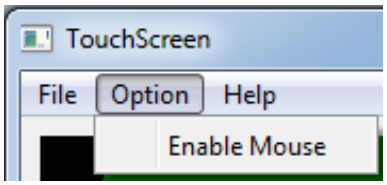
Abbildung 3.15: StartGame-Pop-up



Nebenbei werden auch die Labels für den CurrentPlayer sowie des Balltypen angezeigt. Das Spiel kann nun gespielt werden mit einem Queue. Hierbei muss man mit dem Queue versuchen die weiße Kugel anzustoßen. Wichtig dabei ist, das die Kamera eingeschaltet bleiben muss.

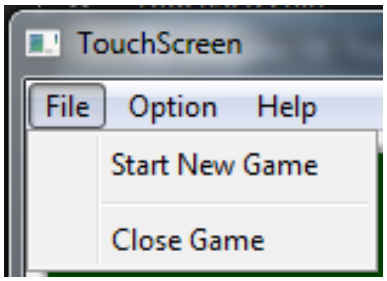
Als zusätliche funktion, hat der Spieler die möglichkeit mit der maus zu spielen(siehe Abb. 3.7: Maus Funktion), indem er sie entprechend zu der weißen kugel schlägt.

Abbildung 3.16: Maus Funktion



Unter File(siehe Abbildung 3.8: Start New Game) hat der Benutzer die möglichkeit das Spiel zurück zu setzten bzw. ein neues Spiel zu starten.

Abbildung 3.17: Start New Game



4 Ergebnisse

4.1 Aufbau der Umfrage

4.2 Darstellung der Ergebnisse

4.2.1 Benutzerfreundlichkeit

4.2.2 Genauigkeit der Spielsteuerung

5 Diskussion

In diesem Kapitel soll die entwickelte Lösung kritisch betrachtet werden. Dabei soll gerade auf die Schwächen der Lösung eingangen und Verbesserungsvorschläge für diese dargestellt werden.

5.1 Verbesserungsvorschläge

Rendering und Physik

Kamerakalibrierung

Queue-Detektion

Die vorgestellte Erkennung des Queues basiert stark auf der Farbe des Queues. Da andere Gegenstände im Kamerabild eventuell ebenfalls einen dunklen Farbton können, wäre die Erkennung in diesem Fall sehr fehleranfällig. Um die Erkennung robuster gegenüber diesen Störfaktoren zu machen, könnte die Erkennung des Queues auf weiteren Merkmalen, wie zum Beispiel Markern, basieren.
Weiterhin limitiert die Wiederholfrequenz der Kamera von 25 Bildern die Sekunde stark die Präzision der Steuerung. Durch die hohe Belichtungszeit ergibt sich eine starke Bewegungsunschärfe, gerade bei schnellen Bewegungen. Diese wirkt sich negativ auf die genaue Erkennung der Queue Spitze aus. Zusätzlich sorgt die Wiederholfrequenz dafür, dass das Spiel selbst auch nur mit 25 Bildern pro Sekunde dargestellt wird. Anders als das erste Problem könnte dies jedoch z.B. durch ein Double-Buffering der Eingabebilder behoben werden.
Schließlich lässt sich die Modellierung des Queues an sich verbesseren. Die Kollision basiert in der vorgestellten Lösung lediglich auf einem Punkt, der Queue-Spitze. In der Realität hat der Queue jedoch noch eine Ausdehnung in die Breite, die bei dieser Modellierung nicht berücksichtigt wird.

Literaturverzeichnis

[1] *HSV Color Space.* – URL https://commons.wikimedia.org/wiki/File:HSV_color_solid_cylinder.png. – Eingesehen am 14.07.2018

[2] *RGB Color Space.* – URL https://commons.wikimedia.org/wiki/File:RGB_color_solid_cube.png. – Eingesehen am 14.07.2018

[3] *OpenCV Tutorials - Introduction to Principal Component Analysis (PCA).* https://docs.opencv.org/3.1.0/d1/dee/tutorial_introduction_to_pca.html. – URL https://docs.opencv.org/3.1.0/d1/dee/tutorial_introduction_to_pca.html. – Eingesehen am 14.07.2018