technische universität
dortmund

fakultät für informatik

Fachprojekt

Erstellung eines Augmented Reality Billiardspiels mit Hilfe von Computer Vision

Friedemann Runte, Moritz Ludolph,
Robin Mertens, Diyar Omar

11. Juli 2018

Gutachter:

Prof. Dr. Vorname Nachname

M.Sc. Vorname Nachname

Lehrstuhl Informatik VII
Graphische Systeme
TU Dortmund

Inhaltsverzeichnis

Mathematische Notation

Notation	Bedeutung
$\mathbb{N}$	Menge der natürlichen Zahlen $1, 2, 3, \ldots$
$\mathbb{R}$	Menge der reellen Zahlen
$\mathbb{R}^d$	d-dimensionaler Raum
$\mathcal{M} = \{m_1, \ldots, m_N\}$	ungeordnete Menge $\mathcal{M}$ von N Elementen m_i
$\mathcal{M} = \langle m_1, \ldots, m_N \rangle$	geordnete Menge $\mathcal{M}$ von N Elementen m_i
$\mathbf{v}$	Vektor $\mathbf{v} = (v_1, \ldots, v_n)^T$ mit N Elementen v_i
$v_i^{(j)}$	i-tes Element des j-ten Vektors
$\mathbf{A}$	Matrix $\mathbf{A}$ mit Einträgen $a_{i,j}$
$G = (V, E)$	Graph G mit Knotenmenge V und Kantenmenge E

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 projeziert 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

Rendering und Phsyik
Das Erstellen des Spielfeldes an sich, stell dabei die erste Herausforderung dar: Man benötigt ein Spielfeld bzw. einen virtuellen Billardpool, bestehend aus Platte und Löchern. Außerdem braucht man Kugeln, die entweder Halb oder voll sind, verschiedene Nummern tragen, als auch verschieden Farben haben. Die Kugeln müssen sich zudem auch noch bewegen können und miteinander, als auch mit der Wand und den Löchern kollidieren können. Kamera Kalibrierung
Kö-Detektion
GUI-Zusammenbau

3 Methoden und Lösungswege

3.1 Rendering und Physik

3.1.1 Objekte

Die Szene lässt sich einteilen in Billiard-Pool und Billiard-Kugeln. Der Pool besteht aus einer grünen Fläche und 6 Löchern. Er besitzt zudem ein 2:1 Verhältnis von Breite zu Höhe.
Die Grüne Farbe ist lediglich die Clear-Color unseres OpenGL Kontextes. Dies können wir machen, weil es sich um eine 2D Anwendung handelt und niemals etwas über den Boden hinaus gezeichnet werden muss. Die Löcher werden simuliert, indem ein sogenannter Triangle-Fan aufgespannt wird. Dieser ist ein Konstrukt aus Dreiecken, die einen Knoten in der Mitte des Objekts haben und dann 2 weiteren Knoten auf dem Kreis.
Im Folgenden gehen wir zur Vereinfachung davon aus, dass wir stets nur 2 Koordinaten haben, da die z-Koordinate aufgrund der Zweidimensionalität der Anwendung immer 0 ist.
Die Struktur beginnt in der Mitte beim zum Objekt Relativen Punkt $(0, 0)$. Von dort aus werden Dreiecke aufgespannt. Wir benötigen immer nur die Koordinaten von dem Randpunkt, der im Uhrzeigersinn weiter verschoben ist, da der GL-Triangle-Fan immer den neuen Punkt mit dem letzten Randpunkt und dem Mittelpunkt verbindet. Der nachfolgende Randpunkt wird berechnet durch die Anzahl der Punkte die wir haben wollen, hier genannt k und der Berechnung von Kreis-Koordinaten:
δ ist der Grad, um den in jedem Schritt der Punkt auf dem Kreis verschoben wird. Dabei muss gelten $\delta \cdot k = 360$ damit wir einen ganzen Kreis bekommen.

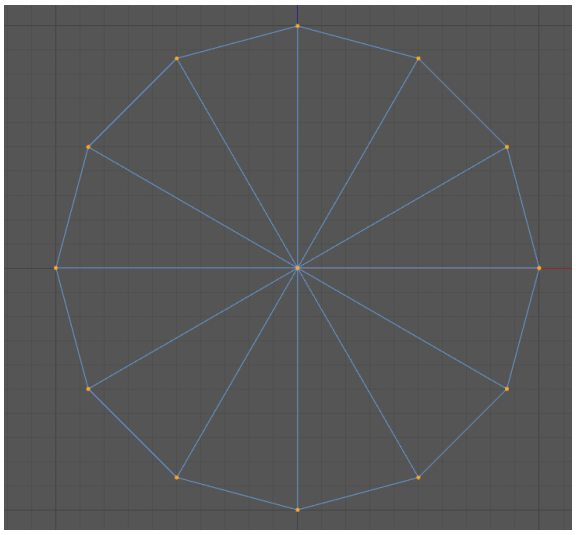
$$\delta = \frac{2 \cdot \pi}{k} \tag{3.1}$$

Sei n der gewünschte 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:

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

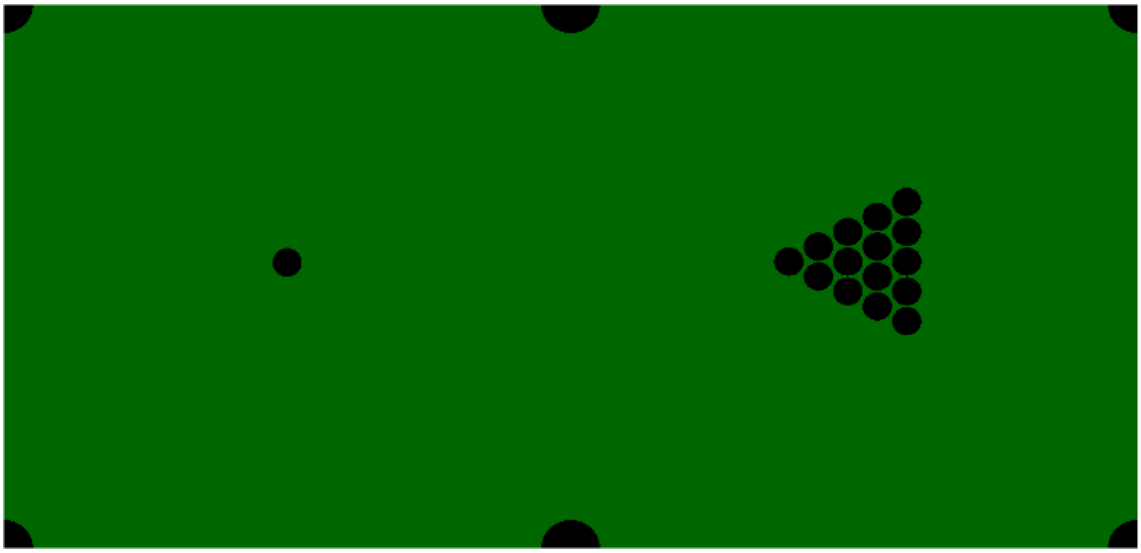
Wenn wir nun k mal die Koordinaten berechnet haben haben wir einen fertigen Kreis, bestehend aus k Dreiecken. Diese werden nun an den 4 Ecken des Spielfeldes angebracht, sowie bei $(w/2h)$ und $(w/2, 0)$, mit w = Breite des Spielfeldes in Pixeln, h = Höhe des Spielfeldes in Pixeln.

Abbildung 3.1: Vollständiger Triangle-Fan mit k=12



Man sieht, dass wenn man nur $k = 12$ benutzt, ist der Kreis viel zu eckig. Deshalb sollte man einen Wert nehmen der für die Ansprüche genügt. Wir haben uns aufgrund der Größe unserer Kreise für $k = 64$ entschieden um die besten Kreise zu erhalten. Dadurch bekommen wir die Illusion eines Kreises, ohne zu viele Ressourcen zu verbrauchen für alle 16 Kugeln und die 6 Löcher.
Diese Form der Kreisberechnung können wir auch für die Billiard-Kugeln anwenden. Somit haben wir schon einmal unser Spielfeld. Nur sieht dieses noch nicht gerade schön aus.

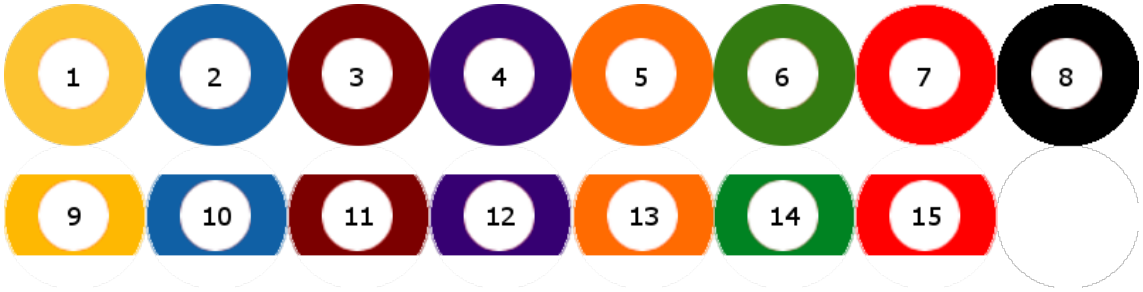
Abbildung 3.2: Spielfeld ohne Texturen



3.1.2 Texturierung

Das Spielfeld als solches benötigt keine Texturierung: Die Löcher können prinzipiell jede beliebige Farbe bekommen, solange man sie noch erkennen kann. Die Kugeln hingegen müssen vom Spieler unterschieden werden können. Man muss schließlich erkennen können, welche Kugeln die Halben und welche die Vollen sind.
Wir müssen also Texturen auf Kreise abbilden. Dazu erstellen wir zunächst eine Textur:
Nun müssen wir die Textur auf einen Kreis bekommen. Dazu verwenden wir u/v-

Abbildung 3.3: Billiardkugel Textur



Koordinaten. Unsere Textur besitzt auf beiden Achsen einen Wertebereich von 0 bis 1. Dabei ist $(0, 0)$ oben links und $(1, 1)$ unten rechts. Da wir die Texturen für alle Kugeln in einer Datei gespeichert haben, können wir jetzt anhand der u/v-Koordinaten der Textur jeder Kugel einen Ausschnitt zuweisen. Dafür werden den Farben der Kugeln werte nach ihrer Reihenfolge auf der Textur gegeben:

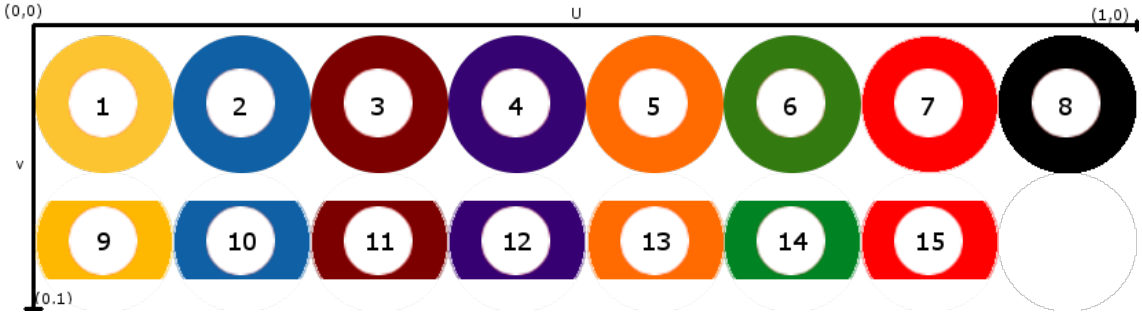
$$c = \begin{cases} 0, & \text{Gelb} \\ 1, & \text{Blau} \\ 2, & \text{Rot} \\ 3, & \text{Lila} \\ 4, & \text{Orange} \\ 5, & \text{Green} \\ 6, & \text{Rot} \\ 7, & \text{Schwarz oder Weiß} \end{cases} \tag{3.3}$$

Außerdem bekommt jede Kugel gesagt, ob sie Halb oder Voll ist, wobei Voll = 0 ist und Halb = 1.

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

Nun können wir uns eine Funktion erstellen, die anhand von Farbe und Fülle die

Abbildung 3.4: Textur mit u/v-Koordinaten



Textur äusschneidetünd 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 x, weil es 8 Kugeln pro Reihe gibt und die u Koordinate genau 1 lang ist, aber $\frac{1}{2}$ für y, weil es nur 2 Reihen gibt und die Textur trotzdem auf der v Achse 1 lang ist. Somit ist der Mittelpunkt der gelben vollen Kugel $(\frac{1}{16}, \frac{1}{4})$, weil dieser bei genau einer Halben Kugel Distanz auf beiden Achsen liegt.
Kommen wir nun zur tatsächlichen Berechnung der Textur-Koordinaten:
Mittelpunkt m der Textur:

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

Nun fehlen noch die Texturkoordinaten für den Kreis um den Mittelpunkt der Textur herum analog zum erstellen der Kreise mit dem Triangle-Fan. Die berechnung läuft dabei genau wie die Berechnung der Koordinaten für die Koordinaten des eigentlichen Kreises, 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 haben wir nicht immer $(0, 0)$ sondern unser vorher

berechnetes x und y in (3.5).

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

Die Textur wird dann beim erstellen des Triangle-Fans auf die Dreiecke gezeichnet. Dabei bekommen die Dreiecke für den Startknoten $(0, 0)$ immer (x, y) als Texturkoordinaten ((3.5)) und der Randpunkt 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

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 Grenzen 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 wieder h = Höhe des Spielfeldes und w = Breite des Spielfeldes.
Wenn nun einer der Fälle eintritt, wird die Geschwindigkeit auf der Achse, die getroffen wurde, invertiert. Beispiel: Die Kugel stößt an die linke Wand (x=0). Dann hat diese vorher eine negative Geschwindigkeit auf der x-Achse, weil sie Richtung x=0 sich bewegt hat. Wenn wir diese Geschwindigkeit nun umkehren, rollt die Kugel Richtung +x und somit wieder Weg von der Wand. So simulieren wir die Kollision.

Nun kommen die Löcher als Einschränkung hinzu: Damit die Kugeln in die Löcher fallen können, müssen wir Lücken in den Wänden erzeugen. Diese simulieren wir durch weitere Bedingungen. Jeder der 4 vorherigen Fälle, bekommt nun eine weitere Einschränkungen. Die Kugeln müssen einmal an allen Ecken, und dann bei $x = \frac{w}{2}$ durch die Wand können. Dass heißt wir überprüfen bei den x-Achsen Kollisionen ($x = 0$ und $x = w$), ob wir auf der y-Achse zwischen beiden Löchern sind und nicht am Rand. Das lässt sich darstellen als:

$$vy = \begin{cases} -vy, & \text{falls } y \leq (h - l) \land y \geq l, \text{ mit } l = \text{größe der Löcher} \\ vy, & sonst \end{cases} \tag{3.7}$$

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

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

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

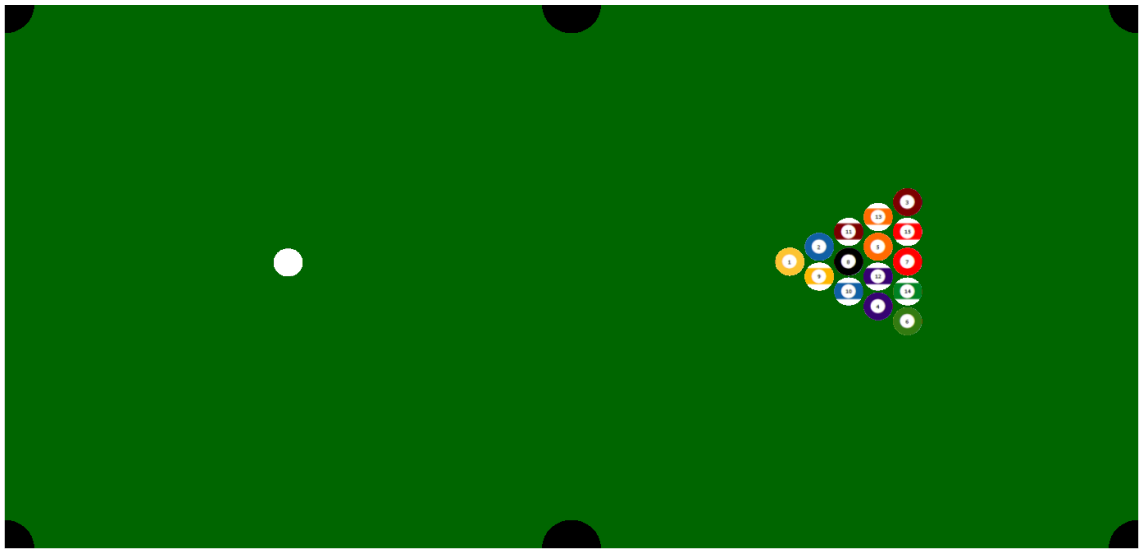
Kollision mit anderen Kugeln

Für die Kollision zwischen den Kugeln müssen wir in jedem Frame berechnen, wie sich jede Kugel zu jeder anderen verhält und wenn sich diese treffen, in welche Richtung sie weiter Rollen. Dazu schauen wir uns immer nur 2 Kugeln an, wobei immer nur die erste verändert wird. Die Kollisionsberechnung war uns bereits vorgegeben aus einem Beispiel für ein AirHockey Spiel. Der Unterschied lag nur darin, dass im Airhockey nur eine Kugel aktiv ist und diese mit zwei nicht beweglichen Schlägern kollidiert. Somit haben wir, statt zu testen, ob die Kugel mit einem der beiden Schläger zusammenstößt, die Kollision von jeder Kugel mit jeder anderen berechnet. Dadurch, dass die jede Kugel in Relation zu jeder anderen Kugel berechnet wird, müssen wir die Kollisionsmethode nicht abändern und können einfach die Berechnung einseitig berechnen. Damit wird in jedem Frame überprüft, ob eine Kugel mit einer anderen Kollidiert und das für alle Kombinationsmöglichkeiten von 2 Kugeln. Dabei muss natürlich darauf geachtet werden, dass keine Kugel mit sich selbst kollidiert.

Zusammenfassung

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

Abbildung 3.5: Fertiges Spielfeld



3.2 Kamerakalibrierung

Dieses Kapitel beschäftigt sich mit der Kamerakalibrierung, worunter neben der Entzerrung des Kamerabildes von Bild mit Tiefe in ein 2D-Bild auch die Transformation eines gegebenen XY-Punktes, welches die aktuelle Position des Queues repräsentiert, in das Spielfeldkoordinatensystem fällt. Die Kamerakalibrierung wird durch die Erkennung eines dargestellten und durch das Kamerabild erfassten Schachbrettmusters, welches ein bereits bekanntes Muster zur Erkennung von Fixpunkten im Kamerabild darstellt, bewerkstelligt. Dies stellt gleichzeitig die Erkennung des Spielfeldes dar, da das Schachbrettmuster auf dem Spielfeldbereich angezeigt wird. Im Folgenden wird zwischen Entzerrung des Kamerabildes und Transformieren eines Punktes unterteilt.

3.2.1 Entzerrung der Kamerabildes

Sobald die Anwendung gestartet wird, wird die Kamera verbunden und ein Timer fragt alle 16ms ab, ob die Kalibrierung gestartet werden soll. Wenn dem so ist, werden Fotos aufgenommen (weiteres dazu s. 'Entzerrung des Kamerabildes'), ansonsten passiert nichts.

Darstellung des Erkennungsmusters

Berechnung der Kachelabmessungen vertikal und horizontal:
H_k Höhe Kachel, W_k Breite Kachel, H_p Höhe Anzeigebereich, W_p Breite Anzeigebereich, Hor = Horizontale Anzahl Kacheln, $Vert$ = Vertikale Anzahl Kacheln
$H_k = \frac{H_p}{Vert}, W_k = \frac{W_p}{Hor}$

Darstellung von Kacheln:
$i \in [1, Hor], j \in [1, Vert] : glRecti(i * H_k, j * W_k, (i+1) * H_k, (j+1) * W_k)$

Entzerrung des Kamerabildes/Punktes

Bildkoordinaten = BK, Weltkoordinaten = WK, Patterneckpunkte = PE, Vector 2/3 Point = 2/3V

Nimmt 20 Fotos auf und wertet diese dann in der *Calibration* aus:
$patternWorldCoordinates$ = 3V mit $(x * a, y * a, 0)$ für alle $x \in Hor - 1$ für alle $y \in Vert - 1$ und a = Kantenlänge für Kachel in WK

$patternCorners$ = 2V Speicher für PE in BK
$patternWorldBuffer$ = 3V Speicher für PE in WK
$pointBuffer$ = 2V Zwischenspeicher für PE in BK (wird für jedes Bild neu initialisiert)

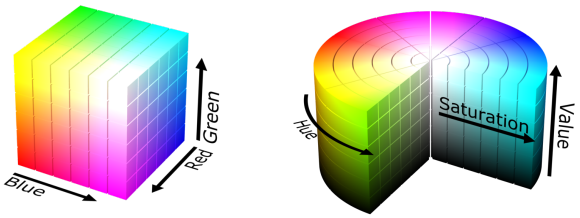
3.2.2 Mappen eines Kamerapunktes in das Spielfeld

3.3 Queue-Detektion

Wie schon in der Einleitung beschrieben, soll es dem Spieler möglich sein, mit dem realen Queue die simulierten Kugeln zu stoßen. Dazu wird das Spielfeld, und somit auch der Queue, von einer Kamera erfasst. Um die Interaktion des Queues mit den Kugeln zu ermöglichen, muss in dem Eingabebild der Kamera der Queue, bzw. die für die Kollisionsberechnung relevanten Punkte, 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 wird dann unter Verwendung der Hauptkomponentenanalyse ein Kollisionspunkt ermittelt. Schließlich wird die Einbindung des Kollisionspunktes in die Kollisionsberechnung erläutert, welche die Detektion des Queues abschließt.

3.3.1 Segmentierung

Sei $\mathbf{I} \in \{0, \ldots, 255\}^{n \times m \times 3}$ das von der Kamera aufgenommene $n \times m$ große RGB-Farbbild. Ziel der Segmentierung ist es, ein Binärbild $\mathbf{B} \in \{0, 1\}^{n \times m}$ 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 mit seinem Farbton $H \in [0, 360]$, Sättigung $S \in [0, 1]$ und Helligkeitswert $V \in [0, 1]$ dargestellt (s. Abb. 3.6).



(a) (b)

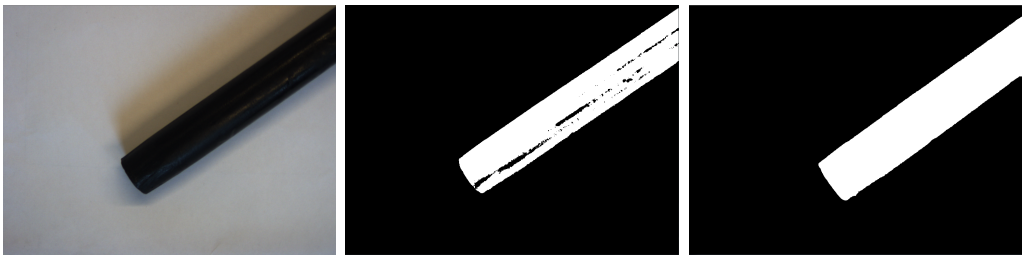
Abbildung 3.6: Darstellung von Farbtönen im (a) RGB-Farbraumes durch Rot-, Grün- und Blauanteileund und im (b) HSV-Farbraumes durch Farbton (Hue), Sättigung (Saturation) und Helilgkeitswert (Value). Dunkle Farben im HSV-Farbraum haben einen kleinen Helligkeitswert.

Dies ermöglicht es, lediglich einen Schwellwert θ für den Helligkeitswert festzulegen, um den schwarzen Anteil des Bildes und damit den Queue zu segmentieren. 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 mittels eines morphologischen Closings diese Lücken gefüllt. Bei einem morphologischen Closing wird zunächst eine Dilatation und dann eine Erosion auf dem Bild mit durch ein $k \times k$ großes, rechteckiges Fenster durchgeführt. Für die Dilatation wird 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.



(a) (b) (c)

Abbildung 3.7: Ergebnis der Segmentierung des Eingabebildes. (a) Eingebild mit 1024×768 Pixeln, (b) Binärbild aus Schwellwertverfahren ($\theta = 30$), (c) Ergebnis des morphologischen Closings ($k = 15$)

3.3.2 Erkennung des Kollisionspunktes

Um nicht mit jedem im Binärbild als **1** klassifizierten Punkt kollidieren zu müssen, wird nur mit der Spitze des Queues kollidiert. Die Spitze des Queues wird dabei durch die Hauptkomponentenanalyse bestimmt.

Dazu werden zunächst die Koordinaten i, j der Punkte aus $\mathbf{B}$ mit $\mathbf{B}[i,j] = 1$ als Zeilenvektoren in einer $l \times 2$ Matrix $\mathbf{X}$ zusammengeführt. Daraufhin wird der Mittelwert

für beiden Dimensionen berechnet und die Koordinaten normalisiert:

$$\mathbf{u} = (u_1, u_2) \text{ mit } u_j = \frac{1}{k} \sum_{i=1}^{k} \mathbf{X}_{i,j}, j \in \{1, 2\}$$

$$\mathbf{A} = \mathbf{B} - \mathbf{h}\mathbf{u}^T \text{ mit } \mathbf{h}[i] = 1 \text{ für } i = 1, \ldots, l$$

Binärbild $\mathbf{B} \in \{0, 1\}^{n \times m}$ Ausgabe der Segmentierung
Da der Queue
Hauptkomponentenanalyse:
Koordinaten in $\mathbf{B}$ der als Queue klassifizierten Pixel als Zeilenvektoren:
$X = \{(x, y) \mid \mathbf{B}_{x,y} = 1, x \in [1, n], y \in [1, m]\}$, $k = |\mathbf{B}|$
Sei $\mathbf{X}$ die $k \times 2$ Matrix der k Zeilenvektoren aus X
Berechnung des Mittelwertes für beide Dimensionen:
Normalisierung mittels des Mittelwertes:
$\mathbf{A} = \mathbf{B} - \mathbf{h}\mathbf{u}^T$ mit dem $k \times 1$ Spaltenvektor $\mathbf{h}, h_i = 1$ für $i = 1, \ldots, k$
Bestimmung der 2×2 Kovarianzmatrix:
$\mathbf{C} = \frac{1}{k-1}\mathbf{B}^T \cdot \mathbf{B}$

3.3.3 Bewegungsinterpolation und Kollisionsberechnung

3.4 Benutzerinteraktion

3.4.1 Spielregeln

3.4.2 Benutzerinteraktion

4 Ergebnisse

4.1 Aufbau der Umfrage

4.2 Darstellung der Ergebnisse

4.2.1 Benutzerfreundlichkeit

4.2.2 Genauigkeit der Spielsteuerung

5 Diskussion

5.1 Verbesserungsvorschläge

A Weitere Informationen

One morning, when Gregor Samsa woke from troubled dreams, he found himself transformed in his bed into a horrible vermin. He lay on his armour-like back, and if he lifted his head a little he could see his brown belly, slightly domed and divided by arches into stiff sections. The bedding was hardly able to cover it and seemed ready to slide off any moment. His many legs, pitifully thin compared with the size of the rest of him, waved about helplessly as he looked. „What's happened to me?"he thought. It wasn't a dream. His room, a proper human room although a little too small, lay peacefully between its four familiar walls. A collection of textile samples lay spread out on the table - Samsa was a travelling salesman - and above it there hung a picture that he had recently cut out of an illustrated magazine and housed in a nice, gilded frame. It showed a lady fitted out with a fur hat and fur boa who sat upright, raising a heavy fur muff that covered the whole of her lower arm towards the viewer. Gregor then turned to look out the window at the dull weather. Drops of rain could be heard hitting the pane, which made him feel quite sad. „How about if I sleep a little bit longer and forget all this nonsense", he thought, but that was something he was unable to do because he was used to sleeping on his right, and in his present state couldn't get into that position. However hard he threw himself onto his right, he always rolled back to where he was. He must have tried it a hundred times, shut his eyes so that he wouldn't have to look at the floundering legs, and only stopped when he began to feel a mild, dull pain there that he had never felt before. „Oh, God, he thought, what a strenuous career it is that I've chosen!"Travelling day in and day out.

Literaturverzeichnis