HAW HAMBURG

MASTER THESIS
Arbin Medi

Deep Reinforcement Learning und Bildverarbeitung zur autonomen Steuerung eines Flipperautomaten

FAKULTÄT TECHNIK UND INFORMATIK
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Arbin Medi

Deep Reinforcement Learning und Bildverarbeitung zur autonomen Steuerung eines Flipperautomaten

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang *Master of Science Automatisierung*
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Marc Hensel
Zweitgutachter: Prof. Dr. Marco Grimm

Eingereicht am: 27. Oktober 2025

Arbin Medi

Thema der Arbeit

Deep Reinforcement Learning und Bildverarbeitung zur autonomen Steuerung eines Flipperautomaten

Stichworte

Deep Reinforcement Learning, Bildverarbeitung, Autonome Steuerung, Flippersystem, Virtuelle Trainingsumgebung, Sim-to-Real-Transfer, Maschinelles Lernen

Kurzzusammenfassung

Diese Masterarbeit untersucht, wie ein Flippersystem durch Bildverarbeitung und Deep Reinforcement Learning zu autonomem Spielverhalten befähigt werden kann. Dazu wurde ein Verfahren zur präzisen Zustandserfassung entwickelt und eine virtuelle Trainingsumgebung implementiert, die realitätsnahe physikalische Bedingungen simuliert. Erste Schritte zur Übertragung des trainierten Modells auf einen physischen Demonstrator wurden erfolgreich durchgeführt. Die Arbeit legt damit die Grundlagen für autonome Flippersysteme auf Basis von Deep Reinforcement Learning.

Title of Thesis

Deep Reinforcement Learning and Computer Vision for Autonomous Control of a Pinball Machine

Keywords

Deep Reinforcement Learning, Computer Vision, Autonomous Control, Pinball System, Virtual Training Environment, Sim-to-Real Transfer, Machine Learning

Abstract

This master's thesis investigates how a pinball system can be enabled for autonomous gameplay through computer vision and deep reinforcement learning. A method for precise state detection was developed, and a virtual training environment was implemented that simulates realistic physical conditions. Initial steps toward transferring the trained model to a physical demonstrator were successfully completed. This work thereby establishes the foundation for autonomous pinball systems based on deep reinforcement learning.

Inhaltsverzeichnis

Abbildungsverzeichnis **viii**

Tabellenverzeichnis **ix**

1 Einleitung **1**

1.1 Motivation 1
1.2 Zielsetzung 2
1.3 Aufbau der Arbeit 3

2 Theoretische Grundlagen **5**

2.1 Bildverarbeitung 5
2.1.1 Kameramodell und Verzerrungskorrektur 5
2.1.2 Bildvorverarbeitung und Filterung 6
2.1.3 Gradientenbasierte Kantendetektion 9
2.1.4 Hough-Transformation 10
2.2 Grundlagen des Reinforcement Learning 11
2.2.1 Agent-Environment-Framework 11
2.2.2 Markov-Entscheidungsprozess 12
2.2.3 Value-Functions und die Bellman-Gleichung 13
2.2.4 Exploration versus Exploitation 14
2.3 Deep Q-Networks 15
2.3.1 Architektur 15
2.3.2 Experience Replay Buffer 16
2.3.3 Target Network 16
2.3.4 Verlustfunktion und Gradientenabstieg 17
2.3.5 Exploration-Strategie 17
2.3.6 Trainingsalgorithmus 19

3 Stand der Technik **20**
3.1 Vorgängerarbeit 20
3.2 Tulane University 22
3.3 University of Alberta 22
3.4 Hochschule Reutlingen 23
3.5 Vergleichende Bewertung und Technologieableitung 25

4 Anforderungsanalyse **26**
4.1 Systemumgebung 26
4.2 Stakeholder 28
4.2.1 Auftraggeber 28
4.2.2 Entwickler 29
4.2.3 Anwender 30
4.2.4 Weiterentwickler 30
4.3 Virtuelle Umgebung 30
4.3.1 Systemabstraktion mittels Anwendungsfalldiagramm 30
4.3.2 Anforderungserhebung 32
4.4 Physischer Demonstrator 36
4.4.1 Systemabstraktion mittels Anwendungsfalldiagramm 36
4.4.2 Anforderungserhebung 37

5 Konzept **41**
5.1 Systemarchitektur 41
5.2 Hardware 43
5.3 Software 46
5.3.1 Bildverarbeitungsstrategie 46
5.3.2 Simulationsumgebung 48
5.3.3 Agent 49

6 Entwicklung des Bildverarbeitungssystems **52**
6.1 Kamerawahl und physischer Aufbau 52
6.2 Softwarearchitektur 54
6.2.1 Klassenübersicht 54
6.2.2 High-Level-Datenfluss 56
6.3 Kamerakalibrierung 58
6.4 Spielfelderkennung 60
6.4.1 Funktionsweise 60

6.4.2 Evaluation . . . 63
6.5 Kugelerkennung . . . 65
6.5.1 Funktionsweise . . . 65
6.5.2 Evaluation . . . 67
6.6 Flipperarm-Erkennung . . . 69
6.6.1 Funktionsweise . . . 69
6.6.2 Evaluation . . . 71

7 Entwicklung der virtuellen Trainingsumgebung 72
7.1 Softwarearchitektur . . . 72
7.2 Geometriemodell . . . 74
7.3 Physik-Engine . . . 76
7.3.1 Physikalisches Modell der Kugelbewegung . . . 76
7.3.2 Flipperdynamik und Kollisionserkennung . . . 77
7.3.3 Ablauf eines Simulationsschritts . . . 77
7.4 Kollisionssystem . . . 79
7.4.1 Vergleich von Kollisionserkennungsverfahren . . . 79
7.4.2 Datenstrukturen . . . 80
7.4.3 Kollisionskarte . . . 80
7.4.4 Kollisionserkennung . . . 82
7.4.5 Kollisionsauflösung . . . 83
7.5 Gymnasium-Environment . . . 83
7.5.1 Observation Space und Action Space . . . 84
7.5.2 Reward-Funktion . . . 84
7.5.3 Episode-Lifecycle . . . 85
7.5.4 Implementierung der Gymnasium-Schnittstelle . . . 86
7.6 Environment-Wrapper . . . 87
7.7 Rendering und Visualisierung . . . 90
7.8 Training . . . 90
7.9 Evaluation . . . 91
7.9.1 Funktionale Anforderungen . . . 91
7.9.2 Nicht-funktionale Anforderungen . . . 93

8 Inbetriebnahme des physischen Demonstrators 95

9 Fazit und Ausblick 97

Literaturverzeichnis **100**

Selbstständigkeitserklärung **103**

Abbildungsverzeichnis

2.1 Agent and Environment Framework [1] 12
2.2 Deep Q-Learning 15
2.3 Epsilon-Decay während des DQN-Trainings 18

3.1 Physischer Flipper 21

4.1 Systemumgebung 27
4.2 Anwendungsfalldiagramm der virtuellen Umgebung 31
4.3 Anwendungsfalldiagramm des physischen Demonstrators 36

5.1 Systemarchitektur des autonomen Flippersystems 42

6.1 Physischer Aufbau des Flippersystems 53
6.2 Klassendiagramm der Bildverarbeitung 55
6.3 Vereinfachtes Sequenzdiagramm des Bildverarbeitungssystems 57
6.4 Vereinfachtes Aktivitätsdiagramm der CameraCalib-Klasse 59
6.5 Vereinfachtes Aktivitätsdiagramm der Codesequenz für die Bildentzerrung 60
6.6 Vereinfachtes Aktivitätsdiagramm der FieldDetector-Klasse 62
6.7 Vereinfachtes Aktivitätsdiagramm der BallDetector-Klasse 66
6.8 Vereinfachtes Aktivitätsdiagramm der ArmDetector-Klasse 70

7.1 Klassendiagramm der virtuellen Trainingsumgebung 73
7.2 2D-Darstellung der Flippergeometrie 75
7.3 step()-Methode der Pinball.py 78
7.4 Statische Kollisionskarte mit farbig markierten Kollisionspixeln 81
7.5 Bresenham-Schleife 82
7.6 Zustandsautomat des Lebenszyklus einer Episode 85
7.7 step()-Methode der PinballEnv.py 87
7.8 Objektdiagramm der Wrapper-Kompositionsstruktur 88

Tabellenverzeichnis

3.1 Vergleichende Bewertung der Technologien 25

5.1 Entscheidungsmatrix zwischen 2D-RGB-Kamera und Tiefenkamera 45
5.2 Entscheidungsmatrix der Bildverarbeitungsansätze 47
5.3 Entscheidungsmatrix der Simulationsansätze 49
5.4 Entscheidungsmatrix der RL-Algorithmen 51

6.1 Evaluation der Spielfelderkennung über zehn Testbilder 64
6.2 Evaluation der Kugelerkennung über 50 Testframes 68

7.1 DQN-Hyperparameter 91

1 Einleitung

Die fortschreitende Digitalisierung und Automatisierung moderner Industrieprozesse hat autonome Systeme zu einem zentralen Forschungsgebiet der Ingenieurswissenschaften gemacht. Besonders die Kombination von Reinforcement Learning (RL), Simulation und Bildverarbeitung eröffnet neue Möglichkeiten für intelligente Steuerungssysteme, die in der Lage sind, komplexe Entscheidungen zu treffen. [12]. Die Integration dieser Technologien in einem gemeinsamen System stellt jedoch sowohl konzeptionelle als auch technische Herausforderungen dar.

Die vorliegende Arbeit untersucht diese Thematik in einem praxisorientierten Kontext anhand eines Flipperautomaten, der im Rahmen der Arbeit zu einem autonomen System weiterentwickelt werden soll. Im Folgenden wird zunächst die Motivation für diese Forschungsarbeit dargelegt, anschließend die konkrete Zielsetzung definiert und schließlich der strukturelle Aufbau der Arbeit erläutert.

1.1 Motivation

Die Überführung theoretischer Konzepte in praktische Anwendungen bleibt eine zentrale Herausforderung für das Verständnis und die Weiterentwicklung komplexer Technologien. Demonstratoren spielen dabei eine entscheidende Rolle, da sie komplexe Algorithmen und Verfahren erst greifbar und nachvollziehbar machen. Erfolgreiche Beispiele hierfür sind die Arbeiten zu autonomen Labyrinthen mittels Deep Reinforcement Learning [13, 24, 9], die zeigen, wie virtuelle Trainingsumgebungen und Bildverarbeitung zur Steuerung physischer Systeme kombiniert werden können. Ohne eine anschauliche und funktionsfähige Ausgestaltung bleiben die Potenziale der künstlichen Intelligenz und Bildverarbeitung oft abstrakt und schwer zugänglich.

Die Faszination autonomer Systeme liegt in der Kombination verschiedener Technologien, die gemeinsam intelligentes Verhalten ermöglichen. Reinforcement Learning ermöglicht

es Systemen, durch wiederholte Interaktion mit ihrer Umgebung optimale Handlungsstrategien zu erlernen[14]. Dabei beobachtet eine virtuelle Steuereinheit (sog. Agent) kontinuierlich Zustände, führt Aktionen aus und erhält Belohnungen, um schrittweise eine Strategie zu entwickeln, die langfristig die beste Performance erzielt. Bildverarbeitung wandelt visuelle Informationen in strukturierte Daten um und ermöglicht dadurch die Erkennung und Verfolgung von Objekten [10]. Virtuelle Umgebungen erweitern diese Möglichkeiten, indem sie physikalisch korrekte Simulationen bereitstellen, in denen Agenten beschleunigt trainiert und anschließend auf reale Systeme übertragen werden können [23].

Die Herausforderung besteht darin, diese abstrakten Konzepte in einer Form zu präsentieren, die sowohl technisch anspruchsvoll als auch allgemein verständlich ist. Ein Demonstrator muss die komplexen Wechselwirkungen zwischen maschinellem Lernen und sensorischer Wahrnehmung veranschaulichen und gleichzeitig die praktische Relevanz dieser Technologien für zukünftige Anwendungen aufzeigen.

Aus dieser Notwendigkeit heraus entsteht die Motivation dieser Arbeit: einen solchen Demonstrator zu schaffen, der die Kombination von Deep Reinforcement Learning und Bildverarbeitung in einem anschaulichen und verständlichen Kontext präsentiert. Der Flipperautomat bietet hierfür eine ideale Ausgangslage, da er einerseits ein jedem vertrautes Spielgerät darstellt und andererseits die technischen Herausforderungen moderner autonomer Systeme in sich vereint.

1.2 Zielsetzung

Das primäre Ziel dieser Masterarbeit ist die Entwicklung eines autonomen Flippersystems, das durch die Kombination von Deep Reinforcement Learning und Bildverarbeitung eigenständig spielen kann. Hierfür werden zwei komplementäre Systeme realisiert: eine virtuelle Trainingsumgebung für das beschleunigte Training des Reinforcement-Learning-Agenten (RL-Agenten) und ein physischer Demonstrator für die praktische Anwendung.

Die virtuelle Umgebung soll als digitaler Zwilling fungieren und eine physikalisch korrekte Verhaltensnachbildung des Flipperautomaten bereitstellen, in der ein Reinforcement-Learning-Agent effizient trainiert werden kann. Dabei müssen alle relevanten Komponenten realitätsgetreu modelliert werden, um eine erfolgreiche Übertragung der erlernten

Strategien auf das physische System zu ermöglichen. Der physische Demonstrator soll durch Integration eines Kamerasystems zur Zustandserfassung und einer intelligenten Steuerungseinheit den Spielzustand kontinuierlich überwachen und die Flipperarme entsprechend ansteuern.

Ein zentraler Aspekt der Arbeit liegt in der nahtlosen Übertragbarkeit der in der virtuellen Umgebung trainierten RL-Modelle auf den physischen Demonstrator. Dadurch soll demonstriert werden, wie Simulation-to-Reality-Transfer in praktischen Anwendungen erfolgreich realisiert werden kann. Das entwickelte System soll als anschaulicher Demonstrator dienen, der die Potenziale moderner KI-Technologien in einem vertrauten Kontext präsentiert und sowohl für Lehrzwecke als auch für die Weiterentwicklung autonomer Systeme genutzt werden kann.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich in neun Kapitel, die den Entwicklungsprozess vom theoretischen Fundament bis zur praktischen Realisierung systematisch dokumentieren. Zunächst werden die theoretischen Grundlagen der Bildverarbeitung und des Reinforcement Learning erarbeitet. Daran anschließend erfolgt eine Analyse des Stands der Technik, die verwandte Arbeiten im Bereich autonomer Flippersysteme untersucht und als Grundlage für die technologischen Entscheidungen dieser Arbeit dient.

Auf Basis dieser Vorarbeiten werden die Anforderungen an das zu entwickelnde System definiert, wobei die Analyse getrennt für die virtuelle Trainingsumgebung und den physischen Demonstrator erfolgt. Das darauf aufbauende Konzept präsentiert die Systemarchitektur sowie die grundlegenden Design-Entscheidungen und bildet damit den Übergang von der Planung zur Implementierung.

Die Umsetzung wird in zwei umfangreichen Kapiteln dokumentiert. Zunächst wird die Entwicklung des Bildverarbeitungssystems beschrieben, wobei alle Komponenten hinsichtlich Funktionsweise und Leistungsfähigkeit evaluiert werden. Anschließend folgt die Implementierung der virtuellen Trainingsumgebung mit der Physik-Simulation und der Integration eines trainierbaren Agenten. Die Inbetriebnahme des physischen Demonstrators behandelt die praktische Integration aller Systemkomponenten und diskutiert

die Herausforderungen beim Transfer des trainierten Modells auf das reale System. Abschließend werden die Ergebnisse zusammengefasst, die Limitationen diskutiert und ein Ausblick auf zukünftige Entwicklungsmöglichkeiten gegeben.

2 Theoretische Grundlagen

Dieses Kapitel legt die theoretischen Grundlagen für die Entwicklung des autonomen Flippersystems. Zunächst werden die wesentlichen Konzepte der Bildverarbeitung behandelt, die für die Zustandserfassung des physischen Systems erforderlich sind. Anschließend werden die Grundlagen des Reinforcement Learning eingeführt. Abschließend werden Deep Q-Networks beschrieben.

2.1 Bildverarbeitung

Die Bildverarbeitung bildet die Grundlage für die visuelle Zustandserfassung des Flippersystems. In diesem Abschnitt werden die relevanten Verfahren zur Kameramodellierung, Bildvorverarbeitung sowie zur Detektion von Kanten erläutert.

2.1.1 Kameramodell und Verzerrungskorrektur

Reale Kamerasysteme weichen aufgrund der physikalischen Eigenschaften ihrer Optik vom idealisierten Pinhole-Kameramodell ab und verursachen systematische geometrische Verzerrungen, die für präzise Bildverarbeitungsverfahren korrigiert werden müssen [2]. Das Pinhole-Kameramodell beschreibt die Projektion eines dreidimensionalen Weltpunktes $P = [X, Y, Z]^T$ auf einen zweidimensionalen Bildpunkt $p = [u, v]^T$ durch die intrinsische Kameramatrix

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.1}$$

Die Parameter f_x und f_y repräsentieren die effektive Brennweite der Kamera in horizontaler und vertikaler Richtung, ausgedrückt in Pixeleinheiten ($fx = f/dx, fy = f/dy$),

während c_x und c_y den Hauptpunkt definieren, an dem die optische Achse die Bildebene schneidet. Die Projektion erfolgt durch

$$s \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{2.2}$$

mit dem tiefenabhängigen Skalierungsfaktor s. Reale Objektive verursachen zwei Haupttypen von Verzerrungen. Die Korrektur erfolgt in normalisierten Koordinaten $x = (u - c_x)/f_x$ und $y = (v - c_y)/f_y$. Die radiale Verzerrung entsteht durch Linsenkrümmung und wird durch die Koeffizienten k_1, k_2, k_3 über die Gleichung

$$\begin{bmatrix} x_{\text{korrigiert}} \\ y_{\text{korrigiert}} \end{bmatrix} = \left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) \cdot \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.3}$$

beschrieben, wobei r^2 der quadratische Abstand vom Hauptpunkt ist. Die tangentiale Verzerrung resultiert aus imperfekter Linsenzentrierung und wird durch p_1 und p_2 über die Gleichungen

$$\begin{aligned} x_{\text{korrigiert}} &= x + 2p_1 xy + p_2(r^2 + 2x^2) \\ y_{\text{korrigiert}} &= y + p_1(r^2 + 2y^2) + 2p_2 xy \end{aligned} \tag{2.4}$$

modelliert. Das vollständige Verzerrungsmodell wird durch den Vektor $D = [k_1, k_2, p_1, p_2, k_3]$ parametrisiert und ermöglicht durch mathematische Inversion die Korrektur aufgenommener Bilder zu einer geometrisch korrekten Darstellung.

2.1.2 Bildvorverarbeitung und Filterung

In diesem Abschnitt werden Verfahren zur Bildvorverarbeitung vorgestellt. Dabei wird zunächst der Gauß-Filter zur Rauschunterdrückung erläutert, gefolgt von der Contrast Limited Adaptive Histogram Equalization (CLAHE) zur Kontrastverstärkung.

Gaußsche Glättung

Die Gaußsche Glättung stellt einen fundamentalen Vorverarbeitungsschritt in der digitalen Bildverarbeitung dar, der zur Reduktion von Bildrauschen und zur Vorbereitung nachgelagerter Analyseverfahren eingesetzt wird [20]. Als Tiefpassfilter unterdrückt sie hochfrequente Bildinhalte wie Rauschen und feine Details, während niederfrequente Strukturen erhalten bleiben und somit eine kontrollierte Glättung bei gleichzeitiger Erhaltung der wesentlichen Bildstrukturen ermöglicht wird. $I(x, y)$ sei ein digitales Grauwertbild mit den Dimensionen $M \times N$ Pixel, wobei $I(x, y) \in [0, 255]$ den Intensitätswert an der Position (x, y) bezeichnet.

Ein Filterkern K ist eine kleine, typischerweise quadratische Matrix von Koeffizienten der Größe $(2k + 1) \times (2k + 1)$, wobei k den Radius des Kerns bezeichnet. Bei der Faltungsoperation wird dieser Kern pixelweise über das gesamte Bild bewegt, wobei an jeder Position eine gewichtete Summe der umliegenden Pixelwerte berechnet wird. Diese Operation ermöglicht es, lokale Nachbarschaftsbeziehungen zu berücksichtigen und gezielt bestimmte Bildeigenschaften zu verstärken oder zu unterdrücken. Das Verfahren basiert auf der zweidimensionalen gaußschen Normalverteilung, die als Faltungskern über das Eingangsbild angewendet wird. Die kontinuierliche zweidimensionale Gaußfunktion ist definiert über die Gleichung

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}, \tag{2.5}$$

wobei σ die Standardabweichung der Verteilung bezeichnet, welche die Stärke der Glättung bestimmt. Für die diskrete Bildverarbeitung wird diese kontinuierliche Funktion zu einem endlichen Filterkern diskretisiert. Ein typischer 3×3 - Gaußkern mit $\sigma = 1$ hat folgende normalisierte Koeffizientenmatrix

$$K = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}, \tag{2.6}$$

wobei der Normalisierungsfaktor $1/16$ die Summe aller Kernelemente darstellt und sicherstellt, dass die Gesamthelligkeit des Bildes erhalten bleibt. Die Faltungsoperation wird für jeden Bildpunkt durch die Gleichung

$$I_{\text{gefiltert}}(x,y) = \sum_{m=-k}^{k} \sum_{n=-k}^{k} K(m,n) \cdot I_{\text{original}}(x+m, y+n) \tag{2.7}$$

beschrieben, wobei m und n die lokalen Koordinaten innerhalb des Filterkerns relativ zum Kernzentrum bezeichnen. Die Gaußsche Glättung bietet mehrere entscheidende Vorteile gegenüber anderen Filterverfahren. Die rotationssymmetrischen Eigenschaften des Gaußkerns gewährleisten eine richtungsunabhängige Glättung, während die gewichtete Mittelwertbildung die lokalen Bildstrukturen besser erhält als einfache Mittelwertfilter. Zudem ist das Verfahren separierbar, wodurch die zweidimensionale Faltung durch zwei aufeinanderfolgende eindimensionale Operationen ersetzt werden kann, was die Rechenzeit erheblich reduziert.

CLAHE

CLAHE ist eine verbesserte Form der adaptiven Histogramm-Äqualisierung, die zur lokalen Kontrastverbesserung in digitalen Bildern verwendet wird [25]. Im Gegensatz zur globalen Histogramm-Äqualisierung arbeitet CLAHE mit kleinen Bildregionen und begrenzt dabei die Kontrastverstärkung, um unerwünschte Artefakte zu vermeiden.

Das Verfahren teilt zunächst das Eingangsbild $I(x,y)$, wobei I die Intensitätswerte bezeichnet und x und y die Pixelkoordinaten sind, in $M \times N$ nicht-überlappende, rechteckige Bereiche (Tiles) $T_{i,j}$ auf. Hierbei ist M die Anzahl der Zeilen und N die Anzahl der Spalten, während i und j die Indizes für Zeile und Spalte des jeweiligen Tiles sind. Für jedes Tile wird das lokale Histogramm $H_{i,j}(k)$ über

$$H_{i,j}(k) = \sum_{(x,y) \in T_{i,j}} \delta(I(x,y) - k) \tag{2.8}$$

berechnet, das angibt, wie oft jeder Grauwert k in diesem Bildbereich vorkommt. Dabei repräsentiert δ die Kronecker-Delta-Funktion und die Summation erstreckt sich über den gesamten Definitionsbereich der jeweiligen Subregion. Der entscheidende Mechanismus von CLAHE ist die Begrenzung der Histogramm-Werte durch

$$H'i, j(k) = \min(Hi, j(k), \alpha), \tag{2.9}$$

wobei $H'_{i,j}(k)$ das begrenzte Histogramm darstellt und min die Minimum-Funktion ist. Der Parameter α definiert die maximale erlaubte Häufigkeit für jeden Grauwert und verhindert eine zu starke Kontrastverstärkung.

Die durch den Clipping-Prozess entstehenden überschüssigen Pixel N_{excess} werden, um die Gesamtpixelzahl zu erhalten, gleichmäßig über alle L verfügbaren Grauwertebenen verteilt.

Der Redistribuierungsfaktor β ergibt sich aus der Division dieser überschüssigen Pixel durch die Anzahl der Grauwertebenen:

$$\beta = \frac{N_{\text{excess}}}{L}. \tag{2.10}$$

Das resultierende, amplitudenbegrenzte Histogramm $H''_{i,j}(k)$ wird anschließend durch Addition des Redistribuierungsfaktors zu jedem Histogramm-Bin berechnet:

$$H''_{i,j}(k) = H'_{i,j}(k) + \beta. \tag{2.11}$$

Die lokale Intensitätstransformation $T_{i,j}(g)$ für jede Subregion basiert auf der normierten kumulativen Verteilungsfunktion (CDF) dieses amplitudenbegrenzten Histogramms:

$$T_{i,j}(g) = (L-1) \cdot \frac{\sum_{k=0}^{g} H''_{i,j}(k)}{\sum_{k=0}^{L-1} H''_{i,j}(k)}. \tag{2.12}$$

Um Diskontinuitäten an den Grenzen benachbarter Subregionen zu vermeiden, wird eine bilineare Interpolation durchgeführt, sodass die Übergänge im Bild gleichmäßig und frei von sichtbaren Artefakten verlaufen.

2.1.3 Gradientenbasierte Kantendetektion

Kanten in digitalen Bildern manifestieren sich als signifikante lokale Helligkeitsänderungen und repräsentieren Diskontinuitäten in der Intensitätsfunktion. Die gradientenbasierte Kantendetektion nutzt die ersten partiellen Ableitungen der Bildfunktion $I(x, y)$, um

diese Übergänge zu identifizieren [26]. Der Gradient eines Bildes ist definiert über die Gleichung

$$\nabla I(x,y) = \begin{bmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{bmatrix} = \begin{bmatrix} G_x \\ G_y \end{bmatrix}, \tag{2.13}$$

wobei G_x und G_y die Gradienten in horizontaler bzw. vertikaler Richtung darstellen. In der praktischen Anwendung werden diese partiellen Ableitungen durch finite Differenzen approximiert. Die einfachste diskrete Näherung berechnet die Helligkeitsänderung zwischen benachbarten Pixeln über

$$\begin{aligned} G_x(x,y) &\approx I(x+\delta,y) - I(x,y), \\ G_y(x,y) &\approx I(x,y+\delta) - I(x,y), \end{aligned} \tag{2.14}$$

wobei δ den Pixelabstand definiert. Diese Differenzen quantifizieren die Stärke und Richtung des lokalen Helligkeitsübergangs. Die Gradientenmagnitude, welche die Kantenstärke unabhängig von der Richtung beschreibt, ergibt sich aus

$$|\nabla I(x,y)| = \sqrt{G_x^2 + G_y^2}. \tag{2.15}$$

Nach der Gradientenberechnung erfolgt die Kantendetektion durch Schwellwertbildung. Ein Pixel (x,y) wird als Kantenpixel klassifiziert, wenn die Gradientenmagnitude oder die absolute Helligkeitsdifferenz einen definierten Schwellwert τ überschreitet:

$$|\nabla I(x,y)| > \tau \implies \text{Kante.} \tag{2.16}$$

2.1.4 Hough-Transformation

Die Hough-Transformation ist ein robustes Verfahren zur Detektion parametrischer Kurven in Bildern, das 1962 von Paul Hough entwickelt wurde [19]. Der fundamentale Ansatz besteht in einer Transformation vom Bildraum in einen Parameterraum, wodurch geometrische Strukturen trotz Rauschen oder Fragmentierung detektiert werden können. Eine Gerade im Bildraum wird durch die Hesse-Normalform parametrisiert, welche die Distanz ρ zum Ursprung und den Winkel θ der Normale zur x-Achse verwendet:

$$\rho = x \cos\theta + y \sin\theta \tag{2.17}$$

Der Kerngedanke liegt in der Dualität zwischen Bild- und Parameterraum. Jeder Punkt (x_i, y_i) im Bildraum entspricht einer sinusförmigen Kurve im (ρ, θ)-Parameterraum. Wenn mehrere kollineare Bildpunkte auf einer gemeinsamen Geraden liegen, schneiden sich ihre korrespondierenden Kurven im Parameterraum an einem gemeinsamen Punkt (ρ_0, θ_0), welcher die Parameter der gesuchten Geraden kodiert.

2.2 Grundlagen des Reinforcement Learning

Reinforcement Learning bildet neben dem überwachten und unüberwachten Lernen das dritte fundamentale Paradigma des maschinellen Lernens. Im Gegensatz zum supervised Learning, bei dem ein Algorithmus aus gelabelten Eingabe-Ausgabe-Paaren lernt, oder zum unsupervised Learning, das verborgene Strukturen in unannotierten Daten entdeckt, zeichnet sich RL durch einen interaktiven Lernprozess aus. Ein Agent lernt hierbei durch wiederholte Interaktion mit seiner Umgebung optimale Handlungsstrategien zu entwickeln, indem er für seine Aktionen verzögerte Belohnungssignale erhält.

2.2.1 Agent-Environment-Framework

Das grundlegende Konzept des Reinforcement Learning lässt sich durch das Agent-Environment-Framework beschreiben, welches in Abbildung 2.1 dargestellt ist [1].

Ein Agent befindet sich zu jedem diskreten Zeitschritt t in einem Zustand s_t aus dem Zustandsraum S und wählt basierend auf diesem Zustand eine Aktion a_t aus dem verfügbaren Aktionsraum A. Die Umgebung reagiert auf diese Aktion, indem sie den Agenten in einen Folgezustand s_{t+1} überführt und ihm eine skalare Belohnung R_{t+1} zurückgibt. Dieser zyklische Prozess, der sich kontinuierlich wiederholt, generiert eine Sequenz von Zuständen, Aktionen und Belohnungen, die als Trajektorie bezeichnet wird.

Das Ziel des Agenten besteht darin, eine Policy π zu erlernen, welche die Auswahl von Aktionen basierend auf Zuständen beschreibt. Eine Policy ordnet jedem Zustand eine Aktion oder eine Wahrscheinlichkeitsverteilung über Aktionen zu. Eine optimale Policy

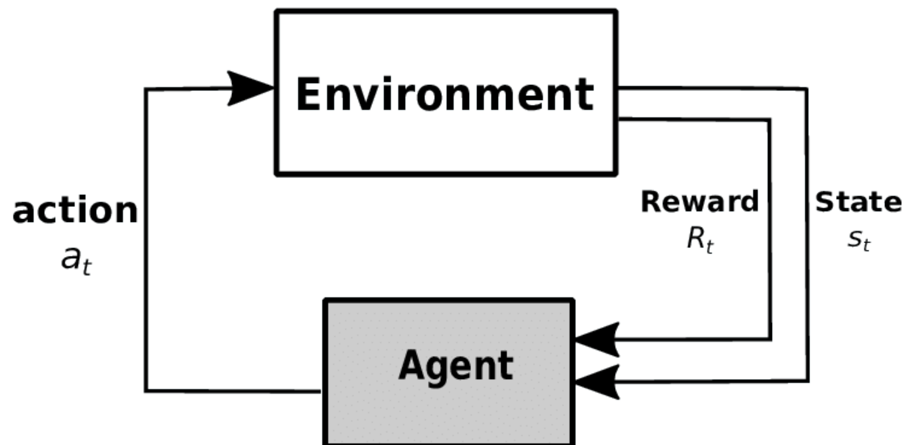


Abbildung 2.1: Agent and Environment Framework [1]

π^* maximiert die erwartete kumulative Belohnung über die Zeit. Da zukünftige Belohnungen typischerweise mit einem Diskontierungsfaktor γ gewichtet werden, ergibt sich der Return G_t als gewichtete Summe aller zukünftigen Belohnungen über

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{2.18}$$

Der Diskontierungsfaktor steuert dabei den Kompromiss zwischen unmittelbaren und zukünftigen Belohnungen. Ein Wert nahe bei eins priorisiert langfristige Gewinne, während ein kleinerer Wert kurzfristige Belohnungen bevorzugt.

2.2.2 Markov-Entscheidungsprozess

Die theoretische Fundierung des Reinforcement Learning bilden Markov-Entscheidungsprozesse (Markov Decision Processes, MDPs). Ein MDP wird formal durch das Tupel (S, A, P, R, γ) beschrieben, wobei S den Zustandsraum, A den Aktionsraum, P die Übergangsfunktion, R die Belohnungsfunktion und γ den Diskontierungsfaktor darstellt [21]. Die Übergangsfunktion $P(s'|s,a)$ gibt die Wahrscheinlichkeit an, vom Zustand s durch Ausführung der Aktion a in den Folgezustand s' zu gelangen. Die Belohnungsfunktion $R(s,a,s')$ beschreibt die erwartete Belohnung für diesen Übergang. Die zentrale Markov-Eigenschaft besagt, dass der zukünftige Zustand ausschließlich vom aktuellen Zustand und der gewählten Aktion abhängt, nicht jedoch von der Historie vergangener Zustände.

Diese Eigenschaft ermöglicht eine effiziente Darstellung und Lösung des Entscheidungsproblems.

2.2.3 Value-Functions und die Bellman-Gleichung

Zur systematischen Bewertung von Policies werden Value Functions eingesetzt, die den erwarteten zukünftigen Erfolg eines Agenten quantifizieren [27]. Die State-Value Function $V^\pi(s)$ gibt den erwarteten Return an, wenn der Agent von Zustand s aus der Policy π folgt:

$$V^\pi(s) = \mathbb{E}_\pi \left[G_t \mid s_t = s \right]. \tag{2.19}$$

Hierbei bezeichnet $\mathbb{E}_\pi[\cdot]$ den Erwartungswert unter der Policy π, das heißt, die durchschnittliche kumulative Belohnung, die der Agent erwarten kann, wenn er die Policy π befolgt. Komplementär dazu beschreibt die Action-Value Function oder Q-Function $Q^\pi(s,a)$ den erwarteten Return bei Ausführung der spezifischen Aktion a im Zustand s und anschließendem Befolgen der Policy π:

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[G_t \mid s_t = s, a_t = a \right]. \tag{2.20}$$

Die Q-Function bewertet somit nicht nur Zustände, sondern konkrete Zustand-Aktions-Paare und beantwortet die Frage, wie gut es ist, in einem Zustand eine bestimmte Aktion auszuführen und danach der Policy π zu folgen. Die Bellman-Gleichung etabliert eine rekursive Beziehung zwischen dem Wert eines Zustands und den Werten seiner Nachfolgezustände. Diese fundamentale Relation besagt, dass der Wert einer Zustand-Aktions-Kombination der Summe aus unmittelbarer Belohnung r und dem diskontierten erwarteten Wert des Folgezustands s' entspricht:

$$Q^\pi(s,a) = \mathbb{E} \left[r + \gamma Q^\pi(s', a') \right]. \tag{2.21}$$

Hierbei wird über alle möglichen Folgezustände gemittelt, die aus (s,a) resultieren können, und über alle Folgeaktionen a', die gemäß Policy π im Zustand s' gewählt werden. Diese Gleichung bildet die Grundlage für iterative Lösungsverfahren im Reinforcement Learning, da sie es ermöglicht, den Wert einer Zustand-Aktions-Kombination

durch die Werte ihrer Nachfolger auszudrücken. Die optimale Q-Function $Q^*(s, a)$ erfüllt die Bellman-Optimalitätsgleichung, bei der im Folgezustand stets die beste verfügbare Aktion gewählt wird:

$$Q^*(s, a) = \mathbb{E}\left[r + \gamma \max_{a'} Q^*(s', a')\right]. \tag{2.22}$$

Anstatt der Policy π zu folgen, wird im Folgezustand die Aktion mit dem höchsten Q-Wert gewählt. Eine optimale Policy π^* kann direkt aus der optimalen Q-Function abgeleitet werden, indem in jedem Zustand die Aktion mit dem höchsten Q-Wert ausgewählt wird:

$$\pi^*(s) = \arg\max_{a} Q^*(s, a). \tag{2.23}$$

Der $\arg\max$- Operator liefert hierbei die Aktion a, die den Q-Wert maximiert. Sobald die optimale Q-Function bekannt ist, kann somit trivial die optimale Handlungsstrategie bestimmt werden.

2.2.4 Exploration versus Exploitation

Eine fundamentale Herausforderung im Reinforcement Learning stellt das Exploration-Exploitation-Dilemma dar [5]. Der Agent muss kontinuierlich zwischen der Exploitation bereits erlernter Aktionen und der Exploration neuer, potenziell besserer Handlungsoptionen abwägen. Eine unzureichende Exploration kann zu suboptimalen Policies führen, da der Agent möglicherweise in lokalen Optima verharrt. Exzessive Exploration hingegen verzögert die Konvergenz zu einer guten Policy, da der Agent zu viel Zeit mit dem Ausprobieren suboptimaler Aktionen verbringt. Gängige Strategien zur Adressierung dieses Dilemmas sind epsilon-greedy-Verfahren, bei denen mit Wahrscheinlichkeit ϵ eine zufällige Aktion gewählt wird, während ansonsten die beste bekannte Aktion ausgeführt wird. Der Parameter ϵ wird häufig über die Trainingszeit hinweg reduziert, sodass der Agent zunächst exploriert und später zunehmend die erlernte Policy ausnutzt.

2.3 Deep Q-Networks

Klassische Q-Learning-Verfahren speichern die Q-Funktion in tabellarischer Form, wobei für jedes Zustand-Aktions-Paar (s, a) ein separater Q-Wert gespeichert wird. Dieser Ansatz stößt jedoch bei hochdimensionalen Zustandsräumen, wie sie beispielsweise bei visuellen Eingaben auftreten, an praktische Grenzen. Die Anzahl möglicher Zustände wächst exponentiell mit der Dimension des Zustandsraums, wodurch eine tabellarische Repräsentation sowohl speichertechnisch als auch rechnerisch nicht mehr handhabbar wird.

Deep Q-Networks (DQN) adressieren diese Problematik durch den Einsatz neuronaler Netze als Funktions-Approximatoren. Anstatt die Q-Funktion explizit zu speichern, wird ein neuronales Netz $Q(s, a; \theta)$ trainiert, das die Q-Werte für beliebige Zustand-Aktions-Paare approximiert. Hierbei bezeichnet θ die Gewichte des neuronalen Netzes. Diese Parametrisierung ermöglicht es, die Q-Funktion auch für zuvor ungesehene Zustände zu generalisieren, indem das Netzwerk gelernte Muster auf neue Situationen überträgt. Dieses Kapitel nutzt zur Beschreibung der theoretischen Grundlagen von Deep Q-Networks die Quellen [18, 17, 28, 4].

2.3.1 Architektur

Die Architektur eines DQN besteht typischerweise aus mehreren vollständig verbundenen Schichten (Fully Connected Layers) wie in Abbildung 2.2 dargestellt. Das Netzwerk erhält

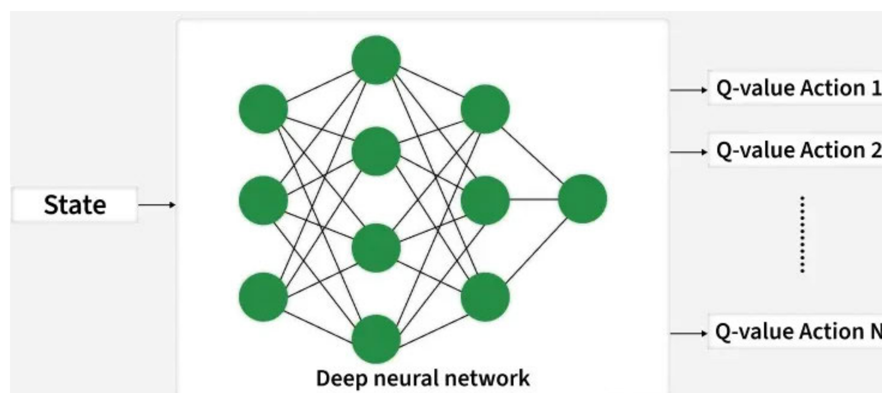


Abbildung 2.2: Deep Q-Learning

den aktuellen Zustand s als Eingabe und gibt für jede mögliche Aktion einen Q-Wert aus.

Formal lässt sich dies als Funktion $Q : S \times A \to \mathbb{R}$ beschreiben, die durch das neuronale Netz mit Parametern θ approximiert wird. Bei diskreten Aktionsräumen ist es effizient, das Netzwerk so zu gestalten, dass es für einen gegebenen Zustand gleichzeitig die Q-Werte für alle Aktionen ausgibt. Dies reduziert die Anzahl erforderlicher Forward-Passes im Vergleich zu einer separaten Evaluierung jeder Aktion und ermöglicht die direkte Anwendung des $\arg\max$-Operators zur Aktionswahl (siehe Gleichung 2.23).

2.3.2 Experience Replay Buffer

Eine zentrale Innovation von DQN ist der Experience Replay Buffer, ein Speicher der Kapazität N, in dem vergangene Erfahrungen in Form von Tupeln $(s_t, a_t, r_{t+1}, s_{t+1})$ gespeichert werden. Während der Interaktion mit der Umgebung sammelt der Agent kontinuierlich diese Transitionen und speichert sie im Buffer. Beim Training werden anschließend zufällige Mini-Batches aus diesem Speicher gezogen.

Dieser Mechanismus adressiert zwei fundamentale Probleme beim Training mit neuronalen Netzen im Reinforcement Learning-Kontext. Erstens weisen aufeinanderfolgende Erfahrungen eine hohe zeitliche Korrelation auf, da sie aus derselben Trajektorie stammen. Diese Korrelation kann zu instabilem Training führen, da das neuronale Netz Schwierigkeiten hat, aus stark korrelierten Daten zu lernen. Durch das zufällige Sampling aus dem Replay Buffer werden diese Korrelationen aufgebrochen. Zweitens ermöglicht Experience Replay eine deutlich effizientere Nutzung der gesammelten Daten, da jede Erfahrung mehrfach zum Training verwendet werden kann, anstatt nur einmal genutzt und dann verworfen zu werden.

2.3.3 Target Network

Eine weitere Stabilisierungsmaßnahme stellt das Target Network dar. Beim Training wird die Bellman-Gleichung als Verlustfunktion verwendet, wobei der Zielwert (Target) für den Q-Wert berechnet wird über

$$y_t = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-). \tag{2.24}$$

Hierbei bezeichnet θ^- die Parameter eines separaten Target Networks, das eine verzögerte Kopie des Haupt-Q-Netzwerks darstellt. Ohne diesen Mechanismus würde derselbe

Parametersatz θ sowohl zur Berechnung des aktuellen Q-Werts als auch des Zielwerts verwendet, was zu einem "moving target problem"führt. Der Zielwert würde sich mit jedem Update ändern, wodurch das Training instabil wird.

Das Target Network wird in regelmäßigen Intervallen, typischerweise alle C Trainingsschritte, mit den Parametern des Haupt-Networks aktualisiert. Zwischen diesen Updates bleiben die Target-Parameter konstant, sodass der Zielwert während mehrerer Trainingsiterationen stabil bleibt. Dies führt zu einem deutlich robusteren Lernprozess.

2.3.4 Verlustfunktion und Gradientenabstieg

Die Parameter θ des Q-Netzwerks werden mittels Gradientenabstieg optimiert, indem die quadratische Differenz zwischen dem vorhergesagten Q-Wert und dem Zielwert minimiert wird. Die Verlustfunktion für einen Mini-Batch von Transitions lautet:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim D}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right]. \tag{2.25}$$

Dabei bezeichnet $\mathcal{D}$ den Replay Buffer, aus dem die Transitions gesampelt werden. Der Gradient dieser Verlustfunktion wird mittels Backpropagation berechnet und zur Aktualisierung der Netzwerkgewichte verwendet:

$$\theta \leftarrow \theta - \alpha \, \nabla_\theta L(\theta), \tag{2.26}$$

wobei α die Lernrate ist und $\nabla_\theta L(\theta)$ den Gradienten der Verlustfunktion bezüglich der Parameter θ bezeichnet.

2.3.5 Exploration-Strategie

Für die Exploration wird bei DQN typischerweise eine ϵ-greedy-Strategie mit Decay eingesetzt. Der Parameter ϵ gibt die Wahrscheinlichkeit an, mit der eine zufällige Aktion gewählt wird und wird über den Trainingsverlauf von einem initialen Wert (typischerweise 1.0) auf einen finalen Wert (typischerweise 0.01 bis 0.05) reduziert. Die Aktion wird zu jedem Zeitschritt gemäß folgender Regel gewählt:

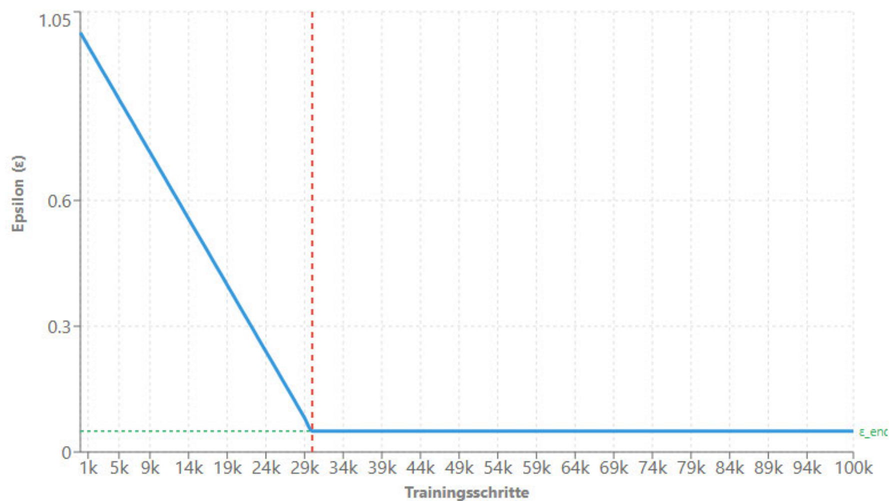


Abbildung 2.3: Epsilon-Decay während des DQN-Trainings

$$a_t = \begin{cases} \text{zufällige Aktion aus } A & \text{mit Wahrscheinlichkeit } \epsilon, \\ \arg\max_a Q(s_t, a; \theta) & \text{mit Wahrscheinlichkeit } 1 - \epsilon. \end{cases} \tag{2.27}$$

Der Verfall (Decay) von ϵ erfolgt üblicherweise linear über einen definierten Anteil der gesamten Trainingsschritte, der als Exploration Fraction bezeichnet wird. Sei T_{total} die Gesamtzahl der Trainingsschritte und f_{explore} die Exploration Fraction (z.B. 0.3 für 30% der Trainingszeit), dann wird ϵ bei jedem Schritt t aktualisiert gemäß

$$\epsilon_t = \max\left(\epsilon_{\text{end}},\ \epsilon_{\text{start}} - \frac{(\epsilon_{\text{start}} - \epsilon_{\text{end}}) \cdot t}{f_{\text{explore}} \cdot T_{\text{total}}}\right). \tag{2.28}$$

Abbildung 2.3 visualisiert diesen Verlauf beispielhaft. In der frühen Trainingsphase bei $\epsilon = 1.0$ wählt der Agent ausschließlich zufällige Aktionen und exploriert damit den gesamten Zustandsraum. Mit fortschreitendem Training sinkt ϵ linear ab, sodass der Agent zunehmend die erlernten Q-Werte ausnutzt (Exploitation). Nach Erreichen des finalen Wertes bleibt dieser Wert konstant, sodass auch in späteren Trainingsphasen ein geringes Maß an Exploration erhalten bleibt, um potenzielle Verbesserungen der Policy nicht zu verpassen.

Dieser schrittweise Übergang von Exploration zu Exploitation ermöglicht es dem Agenten, in der frühen Trainingsphase den Zustandsraum umfassend zu erkunden und in

späteren Phasen die erlernte Policy zunehmend auszunutzen, während durch das verbleibende $\epsilon_{\text{end}} > 0$ weiterhin gelegentlich exploriert wird.

2.3.6 Trainingsalgorithmus

Der vollständige DQN-Trainingsalgorithmus lässt sich wie folgt zusammenfassen: Zunächst wird das Q-Network mit zufälligen Gewichten θ initialisiert und das Target Network mit denselben Gewichten kopiert. Der Replay Buffer wird mit der Kapazität N initialisiert. In jeder Episode interagiert der Agent mit der Umgebung, wobei Aktionen gemäß der ϵ -greedy-Policy gewählt werden. Jede Transition wird im Replay Buffer gespeichert.

Sobald der Buffer eine ausreichende Anzahl von Erfahrungen enthält, wird in regelmäßigen Abständen ein Mini-Batch von Transitionen gesampelt. Für jede Transition im Batch wird der Zielwert berechnet, die Verlustfunktion evaluiert und ein Gradientenschritt durchgeführt. Nach jeweils C Trainingsschritten werden die Target Network Parameter aktualisiert. Dieser Prozess wird über viele Episoden hinweg wiederholt, bis die Policy konvergiert.

3 Stand der Technik

Die Analyse des aktuellen Stands der Technik bildet eine unverzichtbare Grundlage für die wissenschaftliche Einordnung dieser Masterarbeit. Durch die systematische Betrachtung bestehender Forschungsarbeiten im Bereich der autonomen Flippersteuerung lassen sich bewährte Ansätze identifizieren und Forschungslücken aufdecken, die durch das vorliegende Projekt adressiert werden können. Die folgende Untersuchung beginnt mit der direkten Vorgängerarbeit als Ausgangspunkt für die vorliegende Entwicklung und erweitert den Blick anschließend auf drei wegweisende internationale Forschungsarbeiten. Dabei werden sowohl die frühen Pionierarbeiten der Tulane University als auch die innovativen Bildverarbeitungsansätze der University of Alberta sowie moderne Deep Learning-Methoden der Hochschule Reutlingen analysiert. Eine abschließende vergleichende Bewertung leitet schließlich die technologischen Grundlagen für die konzeptionelle Ausrichtung des eigenen Systems ab.

3.1 Vorgängerarbeit

Die vorliegende Masterarbeit baut auf der Bachelorarbeit von [29] auf, welche bereits einen prototypischen Demonstrator entwickelte. Diese Vorgängerarbeit schuf die grundlegende Hardware-Infrastruktur und etablierte erste Ansätze für die autonome Steuerung des Flipper-Systems mittels maschinellen Lernens.

Aus der Vorgängerarbeit werden zentrale Hardware-Komponenten des prototypischen Demonstrators übernommen und weiterverwendet. Abbildung 3.1 zeigt den bestehenden Aufbau mit seinen wesentlichen Elementen. Das Spielfeld mit Rahmen und Elementen basiert auf einem skalierten Aufbau (50x34 cm), konstruiert aus einer Aluverbundplatte mit einem Aluprofil-Rahmen (20x20 mm). Das System umfasst 3D-gedruckte Spielelemente wie Banden, Flipperarme und Kugelrückführungskomponenten, die eine realitätsnahe

Abbildung 3.1: Physischer Flipper

Spielumgebung schaffen. Zur Ansteuerung der Flipperarme werden NEMA 17 Schrittmotoren (Modell 17HS19-2004S1) eingesetzt, die präzise Bewegungen der Schlagarme ermöglichen. Ein zusätzlicher Schrittmotor steuert den Startarm für die Kugelrückführung ins Spielfeld. Das Motorinterface für die Ansteuerung erfolgt über TB6600 Motortreiber in Verbindung mit einem Arduino Uno Mikrocontroller, der als Schnittstelle zwischen der übergeordneten Steuerung und den physischen Aktoren fungiert.

Die vorliegende Masterarbeit erweitert das bestehende System um drei wesentliche Komponenten. Erstens wird ein robustes Bildverarbeitungssystem zur Erfassung der Zustandsinformationen mittels klassischer Computer-Vision-Algorithmen entwickelt. Zweitens erfolgt die Konzeption und Implementierung einer physikalisch korrekten virtuellen Umgebung, die eine effiziente Vorab-Trainingsphase für RL-Agenten ermöglicht. Drittens wird die Übertragung des RL-Agenten von der virtuellen zur physischen Umgebung etabliert, um eine nahtlose Parameterübertragung trainierter Modelle zwischen Simulation und realem Demonstrator zu gewährleisten.

3.2 Tulane University

Die erste dokumentierte Arbeit zur Flipperautomatisierung stammt von Winstead und Christiansen der Tulane University [30]. Ihre 1994 veröffentlichte Studie „Pinball: Planning and Learning in a Dynamic Real-Time-Environment“ behandelt die Entwicklung autonomer Lernagenten für zeitkritische, dynamische Umgebungen.

Die Autoren klassifizieren das Flipperproblem als intermittierendes Steuerungsproblem, bei dem Systemeingriffe ausschließlich möglich sind, wenn sich die Kugel im Wirkungsbereich der Flipperarme befindet. Diese Charakteristik führt zu erheblichen Herausforderungen bei der Zuordnung von Belohnungen zu Aktionen (Credit Assignment Problem).

Die technische Implementierung basiert auf einer C++-Physiksimulation mit diskretisierter Zeitintegration und elastischen Kollisionsmodellen. Als Lernverfahren wurde tabellenbasiertes Q-Learning mit einer 57×30-Zellen-Diskretisierung des Zustandsraums im Flipperarmbereich eingesetzt. Der Aktionsraum umfasst vier diskrete Aktionen: simultane Bewegung beider Flipperarme nach oben oder unten.

Die experimentellen Resultate zeigen moderate Erfolge. Nach 10.000 Trainingsepisoden erzielten die Agenten durchschnittlich 102-108 Punkte und übertrafen damit menschliche Probanden (77-86 Punkte), erreichten jedoch nur marginale Verbesserungen gegenüber einer primitiven Flail-Baseline-Strategie (kontinuierliche Flipperbetätigung) mit 102 Punkten.

Ungeachtet der begrenzten quantitativen Erfolge erbrachten die Autoren bedeutsame konzeptionelle Beiträge: Sie etablierten die Flippersteuerung als validen Testfall für Reinforcement Learning und identifizierten charakteristische Problemstellungen wie intermittierendes Steuerungsverhalten. Gleichzeitig offenbart die Arbeit systematische Limitierungen früher RL-Ansätze, insbesondere bezüglich der niedrig-dimensionalen Zustandsrepräsentation und der mangelnden Integration verfügbaren Modellwissens.

3.3 University of Alberta

Eine erste wegweisende Arbeit im Bereich der physischen Flipperautomatisierung wurde von Adam Metcalf an der University of Alberta entwickelt [15]. Seine Master-Thesis „Pinball: High-Speed Real-Time Tracking and Playing“ aus dem Jahr 2011 stellt eine erste

umfassende Implementierung eines KI-Systems dar, das auf einem realen Flipperautomaten operiert und dabei computerbasierte Kugelerkennung mit physikalischer Simulation kombiniert.

Das entwickelte Framework nutzt für die Bildverarbeitung eine Hochgeschwindigkeitskamera, die 122 Bilder pro Sekunde erfasst. Das System identifiziert die Kugel durch Trennung von Vorder- und Hintergrund, wobei sich bewegende Objekte vom statischen Spielfeld unterschieden werden. Zur Objekterkennung werden zusammenhängende weiße Pixelgruppen zu Objekten gruppiert und nach Größe und Form gefiltert, um die Kugel zu identifizieren.

Für die Physiksimulation setzt die Arbeit auf die Box2D-Engine, die einen kontinuierlichen Kollisionserkennungsansatz verfolgt. Die Flipperarme werden als dynamische Körper mit Revolute Joints modelliert, die eine realistische Rotation um den Drehpunkt ermöglichen. Das System berechnet sowohl elastische als auch inelastische Kollisionen und berücksichtigt Gravitationskräfte auf der geneigten Spielfläche. Die Simulation erfolgt dabei parallel zur Kugelerkennung und dient sowohl der Vorhersage von Kugelbewegungen als auch der Validierung erkannter Positionen.

Die KI-Komponente basiert nicht auf Reinforcement Learning, sondern auf einem regelbasierten, reaktiven Ansatz. Das System definiert Überwachungszonen um die Flipperarme und löst vordefinierte Aktionen aus, sobald die Kugel diese Bereiche erreicht oder durchquert. Dabei wird der Cyrus-Beck-Algorithmus verwendet, um auch schnell bewegende Kugeln zu erfassen.

Die experimentellen Ergebnisse zeigen vielversprechende Leistungen: Das System erreichte eine durchschnittliche Kugelüberlebenszeit von etwa 30 Sekunden, vergleichbar mit menschlichen Anfängern. Besonders bemerkenswert ist die deutlich schnellere Reaktionszeit von 24 Millisekunden gegenüber menschlichen Spielern. Diese Arbeit etablierte erstmals einen funktionsfähigen Rahmen und demonstrierte die Machbarkeit computergesteuerter Flipper-Systeme.

3.4 Hochschule Reutlingen

Ein aktuelles und hochrelevantes Beispiel für die autonome Steuerung physischer Spielsysteme mittels Deep Reinforcement Learning stellt das an der Hochschule Reutlingen entwickelte Projekt „Kicker: An Industrial Drive and Control Foosball System automated

with Deep Reinforcement Learning“ dar [6]. Diese 2021 realisierte Arbeit implementierte einen vollständig automatisierten Tischkicker, dessen Steuerung durch einen RL-Agenten erfolgt und somit industrielle Antriebstechnologien mit modernen Lernverfahren kombiniert.

Die Wahl des Tischkickers als Anwendungsszenario basiert auf vergleichbaren systemtheoretischen Überlegungen wie beim Flipperautomaten. Beide Domänen zeichnen sich durch hochdynamische Zustandsräume, komplexe physikalische Wechselwirkungen sowie ausgeprägte Störanfälligkeit aus. Diese Charakteristika etablieren sie als anspruchsvolle, jedoch didaktisch geeignete Demonstratoren für Reinforcement Learning unter Echtzeitbedingungen.

Das System basiert auf einem Deep-Reinforcement-Learning-Agenten, der zunächst in einer Unity-basierten Simulationsumgebung trainiert und anschließend auf die reale Hardware übertragen wurde. Zur Minimierung der Reality Gap implementierten die Autoren Domain Randomization, wobei während des Trainings physikalische Parameter wie Reibungskoeffizienten, Massenverteilungen und Systemlatenzen stochastisch variiert wurden. Dies erhöht die Robustheit des Agenten gegenüber Modellungenauigkeiten und Umgebungsunsicherheiten im realen System. Als Lernverfahren wurde ein Policy-Gradient-Ansatz gewählt, der kontinuierliche Aktionsräume unterstützt und sich für die direkte Ansteuerung physischer Antriebssysteme eignet. Im Unterschied zum bildverarbeitungsbasierten Ansatz dieser Arbeit nutzt das Kicker-System direkte sensorische Zustandserfassung zur Bestimmung von Kugelposition und Stangenstellung.

Die experimentellen Resultate demonstrieren, dass der trainierte Agent nach erfolgreichem Sim2Real-Transfer in der Lage war, komplexe Spielsituationen zu bewältigen, einschließlich gezielter Kugelabwehr und taktischer Passspiele. Dabei wurden Reaktionszeiten deutlich unterhalb menschlicher Reflexleistung erreicht. Obwohl das System noch nicht auf professionellem Spielniveau agierte, konnte die prinzipielle Machbarkeit autonomer Echtzeitsteuerung unter industriellen Randbedingungen überzeugend nachgewiesen werden. Das Kicker-Projekt liefert somit einen wertvollen Beitrag zur Erforschung von DRL-Anwendungen in dynamischen, physikalischen Systemen. Die methodischen Parallelen zu den Zielsetzungen dieser Arbeit – insbesondere bezüglich Sim2Real-Transfer, DRL-Methodik und Echtzeit-Hardware-Kopplung – machen es zu einer bedeutsamen Ergänzung des technischen Standes für die Entwicklung autonomer Spielsysteme.

3.5 Vergleichende Bewertung und Technologieableitung

Die Analyse der verschiedenen Ansätze zur autonomen Flippersteuerung zeigt eine deutliche Evolution der verwendeten Technologien und Methoden. Während frühe Arbeiten wie die der Tulane University primär auf tabellenbasierten Lernverfahren mit stark diskretisierten Zustandsräumen basierten, demonstrieren moderne Implementierungen wie das Kicker-Projekt der Hochschule Reutlingen die Leistungsfähigkeit von Deep Reinforcement Learning in Kombination mit hochentwickelten Simulationsumgebungen. Die systematische Gegenüberstellung der untersuchten Ansätze (siehe Tabelle 3.1) verdeutlicht charakteristische Unterschiede in der technologischen Herangehensweise. Aus dieser

Tabelle 3.1: Vergleichende Bewertung der Technologien

Kriterium	**Tulane University**	**University of Alberta**	**Hochschule Reutlingen**
Lernverfahren	Tabellen-basiertes Q-Learning	Regelbasierte KI	DRL
Umgebung	Simulation (C++)	Simulation + Physischer Flipper	Simulation + Physisches System
Sensorik	Keine	Hochgeschwindigkeitskamera	Direkte Sensorik
Zustandsrepräsentation	57×30 Diskretisierung	Kontinuierlich	Kontinuierlich
Sim-to-Real-Transfer	Nicht relevant	Nicht relevant	Domain Randomization
Performance	Marginal über Baseline	30s Kugelüberlebenszeit	Professionelles Niveau

Analyse lassen sich drei wesentliche Erkenntnisse für die vorliegende Arbeit ableiten. Erstens erweist sich die Kombination aus virtueller Trainingsumgebung und physischem Demonstrator als erfolgversprechendster Ansatz, da sie die Vorteile beschleunigten Lernens mit realer Anwendbarkeit verbindet. Zweitens zeigt sich die Überlegenheit kontinuierlicher Zustandsrepräsentationen gegenüber groben Diskretisierungen für die Bewältigung hochdynamischer Steuerungsaufgaben. Drittens bestätigt sich die Notwendigkeit robuster Bildverarbeitungssysteme für die zuverlässige Zustandserfassung in physischen Umgebungen.

4 Anforderungsanalyse

Die Anforderungsanalyse bildet die Grundlage für die Konzeption und Entwicklung des Systems, das im Rahmen dieser Masterarbeit entwickelt werden soll. Die präzise Definition von Anforderungen stellt einen essenziellen Schritt dar, um sicherzustellen, dass das System den funktionalen und nicht-funktionalen Erwartungen entspricht und nachhaltig im vorgesehenen Kontext eingesetzt werden kann. Die Anforderungsanalyse gliedert sich in die folgenden zentralen Bereiche: die Systemumgebung, die Stakeholder, die virtuelle Umgebung und den physischen Demonstrator.

Im Abschnitt Systemumgebung werden die technischen und organisatorischen Rahmenbedingungen definiert, die das Umfeld des Systems beschreiben. Hierbei wird deutlich, welche Systemkomponenten bei der Aufstellung von Anforderungen beachtet werden müssen und wie ihre Schnittstellen aussehen. Der Abschnitt Stakeholder identifiziert und analysiert die beteiligten Interessengruppen sowie deren spezifische Erwartungen an das System. Stakeholder umfassen sowohl direkte Nutzer als auch externe Interessengruppen, die den Entwicklungs- und Einsatzprozess beeinflussen. Zuletzt werden die Anwendungsfälle und Anforderungen für die virtuelle Umgebung und den physischen Demonstrator analysiert und definiert. Dabei werden die virtuelle Umgebung und der physische Demonstrator separat als voneinander unabhängige Systeme betrachtet. An den festgelegten Anforderungen kann das entwickelte System (oder Teilsysteme) evaluiert werden.

4.1 Systemumgebung

Um präzise Anforderungen aufstellen zu können, ist es wichtig, die relevanten Systemkomponenten innerhalb dieses Projekts zu identifizieren und in Kontext zu setzen. Diesem Zweck dient Abbildung 4.1. Kernkomponente ist hierbei der physische Demonstrator (Flipper). Dieser kann über seine Antriebe angesteuert und so automatisiert werden. Das zentrale Ziel soll sein, dass eine künstliche Intelligenz (KI) in Form eines Reinforcement-Learning-Agenten selbständig Steuerbefehle erzeugt, sodass der Flipper ohne direkten

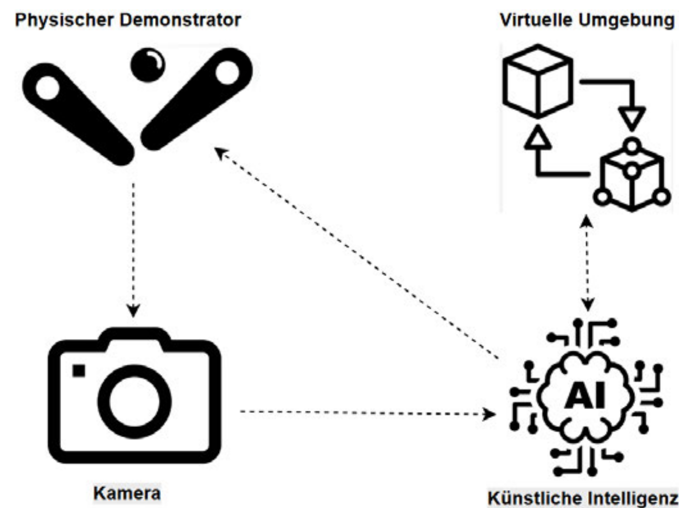


Abbildung 4.1: Systemumgebung

menschlichen Einfluss gespielt wird. Damit auch sinnvolle Steuerbefehle entsendet werden können (damit Kugel im Spiel gehalten wird), muss die KI eine Entscheidungsgrundlage besitzen. Diese wird durch ein Kamerasystem in Form von Bilddaten bereitgestellt. Als letztes stellt sich noch die Frage, wie die KI lernen soll, sinnvolle Steuerbefehle zu erzeugen. Der alleinige Einsatz des physischen Demonstrators eignet sich nicht, da das Training zu viel Zeit beanspruchen würde. Daher kommt eine virtuelle Umgebung (digitaler Zwilling) zum Einsatz. Diese stellt eine möglichst präzise Verhaltensnachbildung des physischen Flippers in Form einer Simulation dar. Mit dieser können große Mengen an Spieldurchläufen in kurzer Zeit simuliert werden und die KI kann viel schneller trainiert und getestet werden. Die Auswahl der genannten Systemkomponenten, beispielsweise des Kamerasystems zur sensorischen Erfassung, wird an dieser Stelle als Anforderung definiert. Die detaillierte Konzeption sowie das Design dieser Komponenten und ihrer Schnittstellen erfolgen im weiteren Verlauf der Arbeit.

Mithilfe der vorangegangenen Betrachtung kann eine Strategie extrahiert werden, um die Anforderungen sinnvoll aufzustellen. Eine sinnvolle Vorgehensweise ist hierbei die Separierung der Anforderungsanalyse für die virtuelle Umgebung und den physischen Demonstrator. Der Grund dafür ist, dass beide Komponenten dadurch als voneinander unabhängige Systeme betrachtet werden können und die schlussendliche Evaluation auch separat durchgeführt werden kann. Auch die Wechselwirkungen der anderen Komponenten (z. B. KI, Kamera, Schnittstelle Antriebssteuerung) mit der physischen und virtuellen Umgebung können so in separaten Anforderungen erfasst werden.

4.2 Stakeholder

Im Rahmen der Anforderungsanalyse hat die Stakeholderanalyse das Ziel, alle relevanten Akteure zu identifizieren, die Einfluss auf die Anforderungen des Projekts haben oder von diesen betroffen sind. Dabei werden sowohl die Bedürfnisse und Erwartungen der Stakeholder als auch deren Einfluss auf die Anforderungsdefinition erfasst. Die Analyse soll dazu beitragen, ein vollständiges und realistisches Anforderungsprofil zu erstellen, das die Anforderungen aller relevanten Parteien berücksichtigt. Die Parteien, die in diesem Kapitel analysiert werden, sind der Auftraggeber, der Entwickler, verschiedene Anwender und Personen im Bereich Weiterentwicklung.

4.2.1 Auftraggeber

Prof. Hensel übernimmt in diesem Projekt die Rolle des Erstprüfers und Betreuers der Masterarbeit und ist somit ein maßgeblicher Akteur im gesamten Verlauf der Arbeit. Als Auftraggeber verfolgt er unterschiedliche Zielsetzungen, die sowohl den wissenschaftlichen Anspruch der Arbeit als auch ihre praktische Relevanz betreffen.

Ein zentrales Anliegen von Prof. Hensel ist die Entwicklung eines funktionalen Modells, das die Anwendung von Reinforcement Learning und Bildverarbeitung in einem Flipperspiel demonstriert. Durch diese praxisorientierte Umsetzung soll veranschaulicht werden, wie fortschrittliche KI-Technologien eingesetzt werden können, um ein autonomes und effizientes System zu schaffen. Diese praktische Anwendung ist für ihn ein wesentliches Ziel, um das Potenzial von KI in der Lösung realer Probleme zu verdeutlichen.

Neben der praktischen Relevanz der Arbeit ist ihm auch die langfristige Perspektive der Forschung und Lehre wichtig. Das Projekt soll als Grundlage dienen, um weitere Entwicklungen im Bereich der künstlichen Intelligenz, Bildverarbeitung und Softwareentwicklung zu fördern sowie Studierende für die Thematik zu begeistern. So kann das Flipperspiel als Anwendungsbeispiel für die Weiterentwicklung und Verbesserung genutzt werden, was auch in zukünftigen Projekten von Studierenden eine wertvolle Basis bieten könnte. Prof. Hensel betrachtet die Masterarbeit daher nicht nur als abgeschlossenes Einzelprojekt, sondern als Teil eines kontinuierlichen Prozesses. Inhaltlich legt er großen Wert auf die wissenschaftliche Tiefe und die Qualität der durchgeführten Analysen. Für ihn ist es entscheidend, dass die Arbeit nicht nur eine funktionierende Lösung liefert,

sondern auch die verwendeten Methoden und Algorithmen kritisch hinterfragt und reflektiert werden. Besonders wichtig ist ihm, dass die Masterarbeit eine solide Basis für zukünftige Entwicklungen im Bereich der künstlichen Intelligenz schafft und zu einem vertieften Verständnis für die Funktionsweise solcher Systeme beiträgt.

4.2.2 Entwickler

Der Entwickler trägt die Hauptverantwortung für die erfolgreiche Umsetzung der Masterarbeit und sorgt dafür, dass die definierten Projektziele erreicht werden. Ein zentraler Aspekt seiner Aufgabe ist die Entwicklung einer funktionalen Lösung, die die Anforderungen an Software, Hardware und künstliche Intelligenz erfüllt. Dabei ist es entscheidend, dass alle Systemkomponenten, von der Programmierung der Software bis zur Integration der Hardware, effektiv zusammenarbeiten.

Ein wichtiger Fokus liegt auf der Implementierung von Reinforcement Learning, um dem Flipper autonomes und optimiertes Verhalten zu ermöglichen. Gleichzeitig ist die Entwicklung einer leistungsfähigen Bildverarbeitung ein wesentlicher Bestandteil der Aufgabe, da diese die Grundlage für die Zustandserfassung des Spiels bildet. Der Entwickler muss daher fundierte Kenntnisse sowohl in den Bereichen maschinelles Lernen und KI-Algorithmen als auch in der Verarbeitung und Analyse von Bilddaten besitzen. Dies umfasst die Auswahl geeigneter Bildverarbeitungsansätze, die Optimierung der Algorithmen für Echtzeitanforderungen sowie die nahtlose Integration der Bildverarbeitung mit dem Reinforcement-Learning-Agenten. Der Entwickler muss komplexe Modelle erstellen und anpassen, um sicherzustellen, dass das System sowohl qualitativ als auch funktional den Anforderungen gerecht wird.

Zudem ist der Entwickler für die kontinuierliche Optimierung des Systems verantwortlich. Dabei werden auftretende technische Herausforderungen mit kreativen Lösungen angegangen, um die Performance des Systems zu steigern und eine reibungslose Ausführung zu gewährleisten. Die Aufgabe des Entwicklers ist es, die gesamte technische Umsetzung zu verantworten und sicherzustellen, dass alle Anforderungen in Bezug auf Funktionalität und Qualität erfüllt werden.

4.2.3 Anwender

Im Rahmen des Projekts lassen sich Anwender identifizieren, deren Bedürfnisse und Interessen entscheidend für die Ausgestaltung der Lösung sind. Es gibt Studierende und weitere Interessierte. Diese sind vor allem daran interessiert, einen praxisnahen Einblick in die Ergebnisse studentischer Arbeiten zu erhalten. Das Projekt bietet eine ideale Gelegenheit, das Interesse an künstlicher Intelligenz und Bildverarbeitung in der Automatisierung zu fördern. Durch die Darstellung der praktischen Anwendung von KI werden theoretische Konzepte greifbar und verdeutlicht, wie sie in realen Szenarien implementiert werden können.

4.2.4 Weiterentwickler

Die Automatisierung des Flippers kann eine spannende Grundlage für Folgearbeiten sein. Sowohl Studierende als auch externe Personen können davon profitieren. Dabei werden viele verschiedene technische Bereiche wie z. B. künstliche Intelligenz, Bildverarbeitung, Softwareentwicklung und Simulation abgedeckt.

4.3 Virtuelle Umgebung

Die virtuelle Umgebung bildet ein unabhängiges System. Daher wird die Anforderungsanalyse, wie in Kapitel 4.1 beschrieben, auch unabhängig vom physischen Demonstrator durchgeführt. Die nächsten Schritte in diesem Kapitel sind die Analyse der Anwendungsfälle (Anwendungsfalldiagramm) und die Aufstellung der Anforderungen.

4.3.1 Systemabstraktion mittels Anwendungsfalldiagramm

Anwendungsfälle (Use Cases) und das Anwendungsfalldiagramm (Use-Case Diagram) sind essenzielle Bestandteile der Anforderungsanalyse, da sie helfen, die funktionalen Anforderungen an ein System strukturiert zu erfassen und zu beschreiben (Systemabstraktion) [8]. Dabei werden die Interaktionen zwischen Akteuren (Benutzern und externen Systemen) und dem System selbst dargestellt. Abbildung 4.2 stellt das Anwendungsfalldiagramm der virtuellen Umgebung dar. Die im Diagramm dargestellten Akteure (Strichmännchen), die mit der virtuellen Umgebung interagieren, sind der Anwender,

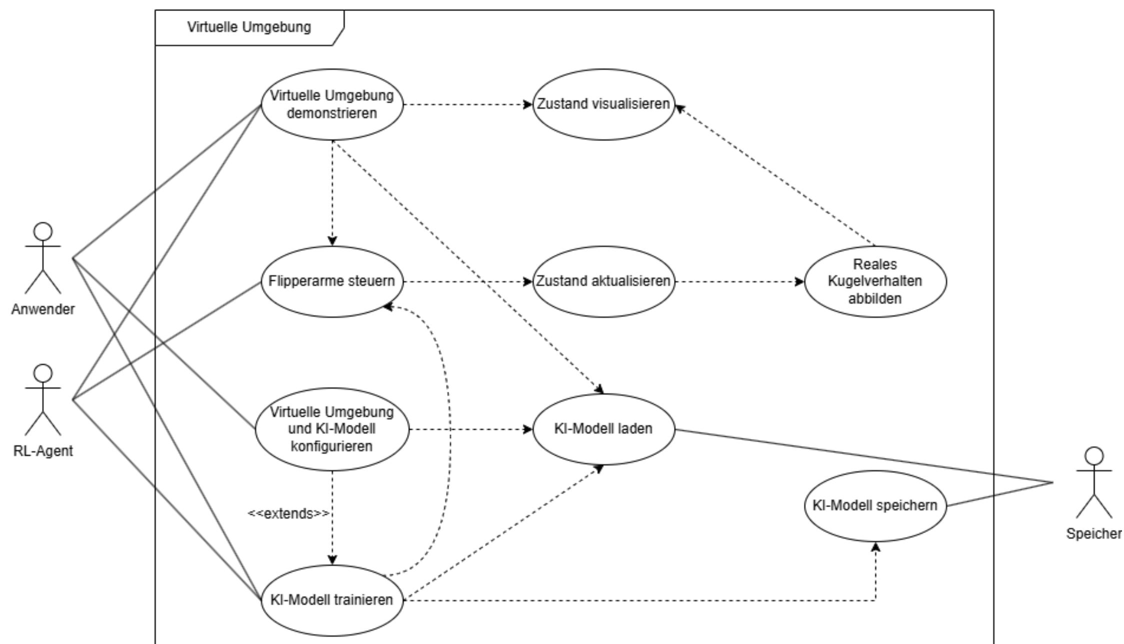


Abbildung 4.2: Anwendungsfalldiagramm der virtuellen Umgebung

der RL-Agent und der Speicher. Die durchgezogenen Linien repräsentieren Assoziationen zwischen den Akteuren und den Anwendungsfällen und veranschaulichen, welche Akteure welche Anwendungsfälle nutzen. Die gestrichelten Pfeile kennzeichnen Abhängigkeitsbeziehungen (dependency relationships) zwischen den Anwendungsfällen. Zur besseren Übersicht sind die meisten Abhängigkeitsbeziehungen nicht beschriftet. Solche stellen sogenannte zwangsläufige Abhängigkeiten («include») dar. Das bedeutet, dass bestimmte Anwendungsfälle andere Anwendungsfälle zwingend aufrufen. Zusätzlich können optionale Abhängigkeiten durch «extend»-Beziehungen modelliert werden. Diese kommen zum Einsatz, wenn ein Anwendungsfall unter bestimmten Bedingungen um eine erweiterte Funktionalität ergänzt wird.

Anwender sollen in der Lage sein, die virtuelle Umgebung zu demonstrieren. Dies umfasst insbesondere die Interaktion mit zwei zentralen Anwendungsfällen. Erstens muss der aktuelle Zustand der virtuellen Umgebung visualisiert werden. Dieser Zustand umfasst das Flipper-Spielfeld, die Position der Flipperarme sowie die Position der Kugel. Zweitens handelt es sich bei dem Flippersystem um ein dynamisches System, in dem sich sowohl die Kugel als auch die Flipperarme bewegen. Während die Kugel nicht direkt beeinflusst werden kann, erfolgt die Steuerung der Flipperarme durch einen Reinforcement-Learning-Agenten, der in die Systemdynamik eingreift. Die Steuerung der Flipperarme und die Be-

wegung der Kugel erfordert eine kontinuierliche Aktualisierung des Systemzustands, um die physikalische Konsistenz der Simulation sicherzustellen. Dies beinhaltet die Berücksichtigung von Krafteinwirkungen, Kollisionen und Bewegungsdynamiken, sodass das Zusammenspiel zwischen Flipperarmen, Spielfeld und Kugel realitätsnah nachgebildet wird. Durch diese fortlaufende Aktualisierung entsteht ein kohärentes dynamisches Verhalten, das die Wechselwirkungen innerhalb des Flippersystems präzise abbildet.

Zusätzlich zu den bereits beschriebenen Anwendungsfällen besitzen sowohl der Anwender als auch der RL-Agent zwei weitere zentrale Interaktionsmöglichkeiten mit der virtuellen Umgebung. Erstens kann der Anwender die virtuelle Umgebung und das KI-Modell konfigurieren. Dies umfasst die Anpassung relevanter Parameter wie physikalische Eigenschaften der Umgebung, Simulationsbedingungen oder Hyperparameter des KI-Modells sowie das Modell selbst. Zweitens kann der Anwender den RL-Agenten bzw. das KI-Modell in eine Trainingsphase überführen. Während des Trainings erfolgt eine kontinuierliche Steuerung der Flipperarme, da der RL-Agent durch wiederholte Interaktionen mit dem dynamischen System optimiert wird. So wird sichergestellt, dass das Modell aus den Aktionen lernt und seine Steuerstrategie schrittweise verbessert.

Abschließend wird der Speicher betrachtet. Als externes System dient er dazu, KI-Modelle zu laden und zu speichern. Verschiedene Anwendungsfälle müssen in der Lage sein, diese nutzen zu können. So kann ein Modell während einer Simulation gesichert oder ein bestehendes Modell für weitere Interaktionen genutzt werden.

4.3.2 Anforderungserhebung

Basierend auf der Analyse der Systemumgebung, der Stakeholder sowie des Anwendungsfalldiagramms der virtuellen Umgebung erfolgt nun die Erhebung der Anforderungen. Diese werden in funktionale (FA) und nicht-funktionale Anforderungen (NFA) unterteilt [11], um eine strukturierte und nachvollziehbare Anforderungsdefinition zu gewährleisten. Zudem wird jede Anforderung mit einer eindeutigen ID versehen, um die Nachverfolgbarkeit und spätere Evaluation zu erleichtern.

Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben die konkreten Funktionen, die das System erfüllen muss. Diese Anforderungen ergeben sich aus den identifizierten Anwendungsfällen und den Interaktionen der Akteure mit dem System.

- **VU-FA1: Visualisierung der virtuellen Umgebung**
  Die virtuelle Umgebung muss eine visuelle Darstellung des Flippers, der Flipperarme und der Kugel bereitstellen. Dies bildet die Grundlage für die Demonstration eines RL-Agenten. Außerdem kann der Nutzer die Spielqualität beobachten.
- **VU-FA2: Aktualisierung des Systemzustands**
  Das System muss den Zustand des Flippers (Kugelposition und Armstellung) kontinuierlich berechnen und physikalisch korrekt aktualisieren. Dabei sind das Roll- und Kollisionsverhalten der Kugel sowie die Bewegungen der Flipperarme realitätsgetreu zu modellieren, indem Reibung und Schwerkraft berücksichtigt werden. Dies gewährleistet, dass ein in der virtuellen Umgebung trainierter RL-Agent auch einen physischen Flipper zielgerichtet steuern kann.
- **VU-FA3: Bereitstellung von Zustandsinformationen für den RL-Agenten**
  Die virtuelle Umgebung muss dem RL-Agenten kontinuierlich den aktuellen Zustand des Spiels zur Verfügung stellen.
- **VU-FA4: Steuerung der Flipperarme**
  Der RL-Agent muss in der Lage sein, Steuerbefehle an die Flipperarme zu senden, um den Spielverlauf aktiv zu beeinflussen. Winkel und Geschwindigkeit der Flipperarme sollten variabel einstellbar sein, um sie an die realen Gegebenheiten anpassen zu können. Diese Größen müssen nicht fest gefordert werden, daher werden sie im Entwicklungskapitel definiert.
- **VU-FA5: Spielflusssicherung und Episoden-Management**
  Sobald die Kugel durch die Lücke zwischen den Flipperarmen rollt, in die Startbahn rollt oder stecken bleibt und das Spiel endet bzw. nicht mehr spielbar wird, soll ein automatischer Neustart erfolgen, um den Spielfluss aufrechtzuerhalten. Dadurch wird sichergestellt, dass das Training des RL-Agenten ohne Unterbrechung fortgesetzt wird.
- **VU-FA6: Training des RL-Agenten**
  Die virtuelle Umgebung muss es ermöglichen, ein Reinforcement-Learning-Modell

zu trainieren und Trainingsparameter wie z. B. die durchschnittliche Episodendauer oder Reward-Kurve aufzuzeichnen.

- **VU-FA7: Evaluation des RL-Agenten**
  Die virtuelle Umgebung soll die Funktion bereitstellen, den RL-Agenten auszuwerten und auch zu demonstrieren. Für die Auswertung werden die Spielzeit (Wie lange Ball im Spiel gehalten?) und die Trefferquote (Ball bei Armbewegung getroffen?) als Indikatoren aufgezeichnet.

- **VU-FA8: Speichern und Laden von KI-Modellen**
  Die virtuelle Umgebung muss die Möglichkeit bieten, trainierte Modelle zu speichern und wieder in das System zu laden. Hierdurch können erfolgreiche RL-Modelle weiterbenutzt werden.

Nicht-funktionale Anforderungen

Nicht-funktionale Anforderungen sind in der System- und Softwareentwicklung Anforderungen, die die Qualitätseigenschaften eines Systems unabhängig von dessen konkreter Funktionalität beschreiben [16]. Sie definieren Rahmenbedingungen und Kriterien, die für ein System relevant sind. NFA dienen der Bewertung und Sicherstellung der Systemqualität und beeinflussen maßgeblich Konzeptentscheidungen sowie den Entwicklungs- und Testprozess.

Für die systematische Aufstellung der NFA an einem etablierten Qualitätsmodell (hier: ISO 25010) wird das Paper [16] als Hilfestellung herangezogen. Dieses vergleicht verschiedene Qualitätsmodelle miteinander. Die Anforderungserhebung wird jedoch nicht in der Ausführlichkeit präsentiert, wie sie im Paper beschrieben ist.

Funktionale Eignung

- **VU-NFA1: Physikalische Modellierung und Genauigkeit**
  Die virtuelle Umgebung muss den physischen Flipper als 3D-Modell darstellen und dabei alle relevanten Komponenten (Spielfeld, Flipperarme, Hindernisse, Kugel) visuell erkennbar abbilden. Desweiteren muss die Simulation die Physik genau und abhängig von der Spielfeldneigung darstellen. Außerdem muss die Bewegung der Flipperarme den mechanischen Einschränkungen des realen Flippers entsprechen (maximaler Auslenkwinkel, Bewegungsgeschwindigkeit, Beschleunigung).

- **VU-NFA2: Proportionale Abbildung**
  Die virtuelle Umgebung muss die Geometrie des physischen Flippers maßstabsgetreu nachbilden, um eine realistische Simulation und Vergleichbarkeit zu gewährleisten.

- **VU-NFA3: Zustandsraum-Definition**
  Der Zustandsraum für den RL-Agenten muss alle relevanten Informationen (Kugelposition, Flipperarm-Stellungen) in einem für den RL-Agenten geeigneten Format bereitstellen.

- **VU-NFA4: Belohnungssystem**
  Das Belohnungssystem muss so gestaltet sein, dass es dem RL-Agenten ermöglicht, die Kugel mindestens 15 Sekunden im Spiel zu halten. Diese Zeitspanne wurde bewusst gewählt: Sie ist einerseits erreichbar und stellt andererseits sicher, dass der Erfolg auf erlernten Strategien und nicht auf bloßem Zufall beruht.

Performance-Effizienz

- **VU-NFA5: Echtzeitvisualisierung**
  Bei aktivierter Visualisierung muss die Simulation flüssig mit mindestens 30 FPS darstellbar sein. Diese Framerate entspricht dem Standard etablierter RL-Frameworks und ist für die menschliche Wahrnehmung einer flüssigen Bewegung ausreichend.

Zuverlässigkeit

- **VU-NFA6: Simulationsstabilität**
  Die Simulation muss stabil und ohne physikalisch unmögliche Zustände (z. B. Kugel durchdringt Wände) ablaufen.

Wartbarkeit und Übertragbarkeit

- **VU-NFA7: Code-Qualität**
  Die Implementierung muss objektorientiert, strukturiert und angemessen dokumentiert sein.

- **VU-NFA8: Modellübertragbarkeit**
  Trainierte RL-Modelle müssen zwischen virtueller Umgebung und physischem Demonstrator ohne Anpassungen übertragbar sein. Außerdem muss der RL-Agent in der Simulation zeitlich identisch arbeiten wie in der realen Umgebung.

4.4 Physischer Demonstrator

Der physische Demonstrator bildet ein unabhängiges System. Daher wird die Anforderungsanalyse, wie in Kapitel 4.1 beschrieben, auch unabhängig von der virtuellen Umgebung durchgeführt. Die nächsten Schritte in diesem Kapitel sind die Analyse der Anwendungsfälle (Anwendungsfalldiagramm) und die Aufstellung der Anforderungen.

4.4.1 Systemabstraktion mittels Anwendungsfalldiagramm

Wie bereits in Kapitel 4.3.1 für die virtuelle Umgebung dargelegt, erfolgt in diesem Abschnitt die äquivalente Analyse des Anwendungsfalldiagramms für den physischen Demonstrator. Dieses ist in Abbildung 4.3 dargestellt.

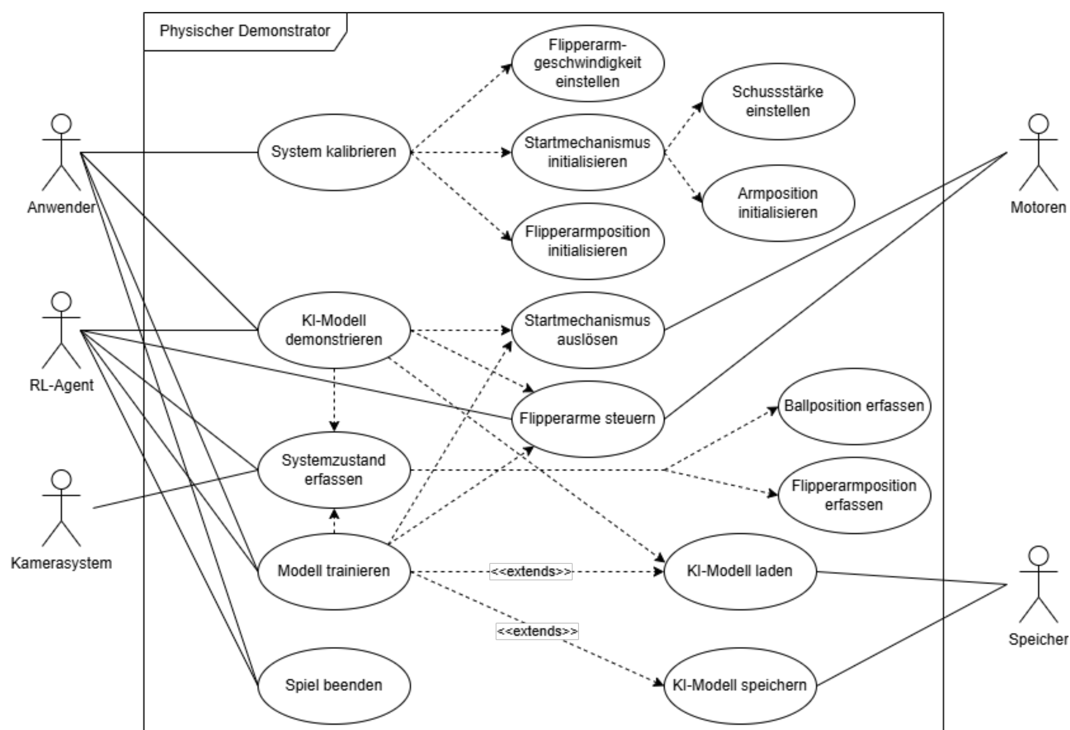


Abbildung 4.3: Anwendungsfalldiagramm des physischen Demonstrators

Die im Diagramm aufgeführten Akteure, die mit dem physischen Demonstrator interagieren, sind der Anwender, der RL-Agent, das Kamerasystem, die Motoren sowie der Speicher. Dem Anwender obliegt die Möglichkeit, den physischen Demonstrator zu kalibrieren. Dabei ist zu beachten, dass die vorgenommenen Konfigurationen einen maß-

geblichen Einfluss auf die Leistungsfähigkeit des Gesamtsystems ausüben können. Es ist daher empfehlenswert, die gewählten Einstellungen über sämtliche Projektphasen hinweg, darunter Training, Evaluation, Testbetrieb und produktivem Einsatz, konsistent beizubehalten. Die Kalibrierung bzw. die Einstellung umfasst unter anderem die Festlegung der Anfangsposition und -geschwindigkeit der Flipperarme sowie die Parametrierung des Startmechanismus zur Kugelbeförderung in das Spielfeld, einschließlich der Armstellung und Abschussstärke (Anwendungsfälle).

Um dem RL-Agenten eine effektive Ausführung seiner Steuerungsaufgabe (Kugel im Spiel halten) zu ermöglichen, ist eine Trainingsphase erforderlich. Analog zur virtuellen Umgebung soll der Anwender auch beim physischen Demonstrator das Modell trainieren bzw. vortrainierte Modelle weitertrainieren können. Hierbei besteht die Möglichkeit, bestehende Modelle in das System zu laden sowie neue Modelle zu speichern (Anwendungsfälle). Darüber hinaus soll der Anwender in der Lage sein, den Flipper, präziser die Funktionsfähigkeit des KI-Modells, in einer Demonstration vorzuführen. Dazu ist zum einen die kontinuierliche Erfassung des Systemzustands (Positionen von Kugel und Flipperarmen) erforderlich. Zum anderen muss beim Start der Demonstration die Kugel durch einen Motor in das Spielfeld befördert werden. Im weiteren Verlauf übernimmt der RL-Agent die Ansteuerung der Motoren zur Bewegung der Flipperarme, um die Kugel im Spiel zu halten. Abschließend muss dem Anwender die Möglichkeit eingeräumt werden, den Spielvorgang zu beenden.

4.4.2 Anforderungserhebung

Basierend auf der Analyse der Systemumgebung, der Stakeholder sowie des Anwendungsfalldiagramms des physischen Demonstrators erfolgt nun die Erhebung der Anforderungen in gleicher Weise wie in Kapitel 4.3.2 für den physischen Demonstrator.

Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben die konkreten Funktionen, die das System erfüllen muss. Diese Anforderungen ergeben sich aus den identifizierten Anwendungsfällen und den Interaktionen der Akteure mit dem System.

- **PD-FA1: Systemkalibrierung und -initialisierung**
  Es muss Anwendern die Möglichkeit gegeben werden, die Geschwindigkeit und Initialposition der Flipperarme einzustellen. Auch müssen Anwender die Schussstärke und Position des Startarms einstellen können. Durch diese Möglichkeiten ist die Testung verschiedener Systemkonfigurationen möglich.
- **PD-FA2: Erfassung des Systemzustands**
  Das Kamerasystem muss den Systemzustand (Position der Kugel und der Flipperarme) kontinuierlich erfassen und an den RL-Agenten übergeben. Dies bildet die Basis für die Entscheidungen des RL-Agenten.
- **PD-FA3: Steuerung der Flipperarme**
  Der RL-Agent muss in der Lage sein, Steuerbefehle an die Flipperarme bzw. an die jeweiligen Motoren zu senden, um den Spielverlauf aktiv zu beeinflussen.
- **PD-FA4: Spielflusssicherung**
  Sobald die Kugel in das Spielfeld befördert wird, soll der Startarm automatisch in seine Ausgangslage zurückgebracht werden. Dies bereitet einen Neustart des Spiels vor ohne den Eingang der Kugel zu blockieren.
- **PD-FA5: Training des RL-Agenten**
  Anwender müssen in der Lage sein, vortrainierte RL-Agenten mit dem physischen Demonstrator weiter zu trainieren. Die Performance des RL-Agenten sollte dadurch gesteigert werden.
- **PD-FA6: Demonstration des RL-Agenten**
  Der physische Demonstrator soll die Funktion bereitstellen, den RL-Agenten zu demonstrieren, ohne den Lernfortschritt zu beeinflussen. Die Demonstration beinhaltet das Spiel zu starten (Kugel in das Spielfeld befördern) und die Kugel im Spiel zu halten.
- **PD-FA7: Speichern und Laden von KI-Modellen**
  Der physische Demonstrator muss die Möglichkeit bieten, trainierte Modelle zu speichern und wieder in das System zu laden. Hierdurch können erfolgreiche RL-Modelle weiterbenutzt werden.
- **PD-FA8: Stoppen des Spiels**
  Anwender müssen das Spiel jederzeit beenden können. Beim Spielstopp müssen alle Motoren instantan ausgeschaltet werden.

Nicht-funktionale Anforderungen

Äquivalent zu Kapitel 4.3.2 wird in diesem Kapitel die Aufstellung der NFA für den physischen Demonstrator durchgeführt. Auch hier wird das Paper [16] als Hilfestellung für eine systematische Anforderungserhebung herangezogen.

Funktionale Eignung

- **PD-NFA1: Kompatibilität Kamerasystem**
  Das zu integrierende Kamerasystem muss mit dem bestehenden Flipper-System kompatibel sein und darf die mechanischen Eigenschaften des Flippers nicht beeinträchtigen. Die Kamera muss so positioniert oder eingestellt werden, dass sie das gesamte Spielfeld erfasst.

- **PD-NFA2: Bilderfassungsqualität**
  Das Kamerasystem muss eine räumliche Auflösung des Flippers von mindestens 0,5 mm/Pixel erreichen, um Kugel und Flipperarme präzise zu detektieren. Bei einem Kugeldurchmesser von 16 mm entspricht dies einer Abbildung mit ca. 32 Pixeln, was für robuste Kantendetektion und Formerkennung ausreichend ist. Eine gröbere Auflösung würde insbesondere bei schnellen Bewegungen zu Detektionsproblemen führen.

Performance-Effizienz

- **PD-NFA3: Echtzeitfähigkeit**
  Die Gesamtlatenz von der Bilderfassung bis zur Motoransteuerung muss unter 50 ms betragen, um eine effektive Kugelkontrolle zu gewährleisten. Bei einer maximalen Kugelgeschwindigkeit von 10 m/s legt die Kugel in 50 ms eine Strecke von 50 cm zurück – dies entspricht der gesamten Spielfeldlänge. Eine höhere Latenz würde bedeuten, dass der RL-Agent auf veraltete Zustandsinformationen reagiert, wodurch präzise Flipperarm-Aktionen zur Kugelkontrolle unmöglich werden.

- **PD-NFA4: Bildverarbeitungsgeschwindigkeit**
  Das Kamerasystem muss mindestens 60 FPS liefern und die Bildverarbeitung muss diese Framerate verarbeiten können. Bei einer maximalen Kugelgeschwindigkeit von 10 m/s und 60 FPS legt die Kugel zwischen zwei Frames etwa 16,7 cm zurück. Dies gewährleistet eine ausreichend dichte räumliche Abtastung der Kugelbahn und rechtzeitige Reaktion des RL-Agenten. Eine niedrigere Framerate würde zu

größeren Sprüngen in der Kugelposition führen und könnte schnelle Bewegungen unzureichend erfassen.

Kompatibilität und Übertragbarkeit

- **PD-NFA5: Hardware-Kompatibilität**
  Das System muss mit gängigen Recheneinheiten betrieben werden können. Die Schnittstellen müssen standardkonform sein.

- **PD-NFA6: Modellübertragbarkeit**
  In der virtuellen Umgebung trainierte RL-Modelle müssen auf dem physischen Demonstrator lauffähig sein.

Wartbarkeit

- **PD-NFA7: Modularität**
  Alle Systemkomponenten müssen modular aufgebaut und einzeln austauschbar sein.

- **PD-NFA8: Dokumentation**
  Der Quellcode muss vollständig dokumentiert sein.

Sicherheit und Wirtschaftlichkeit

- **PD-NFA9: Mechanische Sicherheit**
  Die Motorsteuerung muss Sicherheitsgrenzen implementieren, um mechanische Beschädigungen zu verhindern. Maximale Winkelgeschwindigkeiten und Beschleunigungen müssen begrenzt werden.

- **PD-NFA10: Kostenrahmen**
  Die Gesamtkosten für Neuanschaffungen (z. B. Kamerasystem inklusive notwendiger Halterungen) dürfen 300€ nicht überschreiten.

5 Konzept

Das vorliegende Konzeptkapitel bildet die zentrale Brücke zwischen der in Kapitel 4 durchgeführten Anforderungsanalyse und der technischen Umsetzung des autonomen Flippersystems. Basierend auf den identifizierten funktionalen und nicht-funktionalen Anforderungen werden in diesem Kapitel die wesentlichen Designentscheidungen getroffen und konzeptionell begründet. Dazu wird zunächst die Architektur des Gesamtsystems festgelegt und dargestellt, wie die verschiedenen Systemkomponenten interagieren sollen. Anschließend erfolgt eine systematische Unterteilung in Hardware und Software, um die unterschiedlichen Anforderungen und Designkriterien strukturiert behandeln zu können. Alle Designentscheidungen werden dabei unter Berücksichtigung der in der Anforderungsanalyse definierten Kriterien getroffen, um die Erfüllung der Systemanforderungen sicherzustellen.

5.1 Systemarchitektur

Das in dieser Masterarbeit entwickelte autonome Flippersystem besteht aus mehreren interdependenten Komponenten, die sowohl hardware- als auch softwareseitig präzise aufeinander abgestimmt werden müssen. Die Systemarchitektur bildet das konzeptionelle Fundament, das die strukturierte Integration aller Teilsysteme ermöglicht und deren Zusammenspiel definiert.

Zur strukturierten Darstellung wird ein Komponentendiagramm verwendet, das die modularen Bestandteile des Systems und deren Schnittstellen übersichtlich visualisiert. Diese Darstellungsform eignet sich besonders für interdisziplinäre Projekte, da sie sowohl Hardware- als auch Softwarekomponenten in einem einheitlichen Format abbildet und die Kommunikationswege zwischen den verschiedenen Subsystemen transparent macht. Abbildung 5.1 zeigt die Systemarchitektur des autonomen Flippersystems in Form eines Komponentendiagramms.

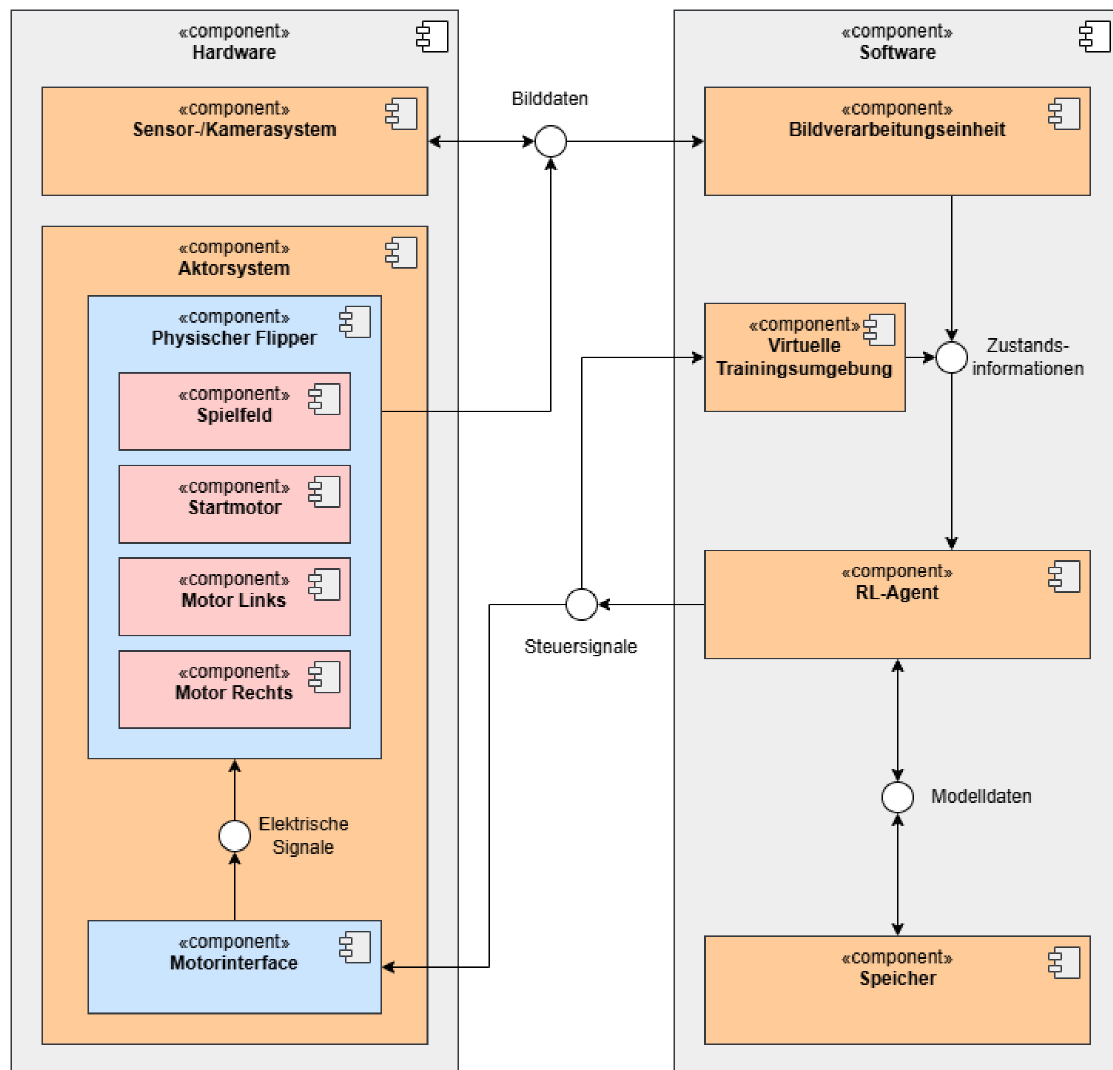


Abbildung 5.1: Systemarchitektur des autonomen Flippersystems

Das Gesamtsystem lässt sich in zwei zentrale Komponenten untergliedern: Hardware und Software. Die Hardware-Komponente setzt sich aus dem Sensorsystem, welches durch ein Kamerasystem realisiert wird, sowie dem Aktorsystem zusammen. Letzteres umfasst sowohl den physischen Flipper als auch das zugehörige Motorinterface. Der physische Flipper besteht aus dem Spielfeld und drei Motoren, die den linken Arm, den rechten Arm sowie den Startarm antreiben. Das Motorinterface übernimmt dabei die Aufgabe, elektrische Steuersignale an die entsprechenden Motoren zu übertragen.

Die Software-Komponente gliedert sich in vier Hauptmodule: die Bildverarbeitungseinheit, die virtuelle Trainingsumgebung, den RL-Agenten sowie den Speicher. Innerhalb

dieses Systems übermittelt die Bildverarbeitungseinheit Zustandsinformationen des physischen Flippers – bestehend aus Kugelposition und Armstellung – an den RL-Agenten. Parallel dazu stellt die virtuelle Trainingsumgebung dem RL-Agenten Zustandsinformationen der Simulation zur Verfügung. Diese beiden Datenübertragungsprozesse erfolgen unabhängig voneinander und nicht simultan. Der RL-Agent verfügt über die Flexibilität, sowohl in der virtuellen als auch in der realen Umgebung trainiert und demonstriert zu werden. Zur Gewährleistung der Persistenz können bereits trainierte RL-Modelle im Speicher abgelegt und bei Bedarf wieder abgerufen werden.

Die Schnittstellen zwischen Hardware- und Softwarekomponente ermöglichen eine nahtlose Kommunikation beider Systemteile. Das Kamerasystem erfasst kontinuierlich Bilddaten des physischen Flippers und leitet diese an die Bildverarbeitungseinheit weiter. Der RL-Agent nutzt die erhaltenen Zustandsinformationen, um je nach Anwendungskontext entsprechende Steuersignale zu generieren – entweder für die virtuelle Trainingsumgebung oder für das Motorinterface der realen Flipper-Hardware.

Ergänzend ist zu erwähnen, dass eine zentrale Systemkomponente in der dargestellten Architektur konzeptionell nicht explizit aufgeführt wird: die Recheneinheit, auf welcher die gesamte Softwarekomponente ausgeführt wird. Diese Hardware-Plattform stellt die notwendige Rechenleistung und Speicherkapazität zur Verfügung und bildet somit die technische Grundlage für die Ausführung aller softwarebasierten Prozesse. Obwohl diese Komponente aufgrund ihrer übergeordneten Rolle nicht als separates Element im Systemdiagramm dargestellt werden kann, ist sie für das Funktionieren des Gesamtsystems von essentieller Bedeutung.

5.2 Hardware

Die Hardware-Komponenten bilden das physische Fundament des autonomen Flippersystems und müssen präzise auf die in der Anforderungsanalyse definierten Spezifikationen abgestimmt werden. Wie in Kapitel 5.1 dargestellt, gliedert sich die Hardware in zwei Hauptbereiche: das Sensorsystem, realisiert durch ein Kamerasystem, sowie das Aktorsystem, bestehend aus dem physischen Flipper und dem Motorinterface. In diesem Abschnitt wird die Konzeption der Hardware beschrieben. Da das Aktorsystem sowie die zugehörige Schnittstelle bereits vorgegeben sind, konzentriert sich der folgende Abschnitt ausschließlich auf das Kamerasystem.

Das Kamerasystem stellt die sensorische Schnittstelle zwischen dem physischen Flipper und der Bildverarbeitungseinheit dar. Als zentrales Element der Wahrnehmungskette erfasst es kontinuierlich den Spielzustand und ermöglicht dem RL-Agenten fundierte Entscheidungen zur Steuerung der Flipperarme. Die Qualität und Leistungsfähigkeit des Kamerasystems hat dabei direkten Einfluss auf die Gesamtperformance des autonomen Flippersystems.

Aus den in Kapitel 4.4.2 definierten Anforderungen lassen sich spezifische Kriterien für das Kamerasystem ableiten. Neben diesen gibt es zusätzliche Aspekte wie Tiefenmessung, Empfindlichkeit gegenüber wechselnden Lichtverhältnissen und Kompatibilität mit Bibliotheken.

Die Echtzeitfähigkeit (PD-NFA3) erfordert eine Gesamtlatenz von der Bilderfassung bis zur Motoransteuerung, die eine effektive Kugelkontrolle ermöglicht. Bei einer maximalen Kugelgeschwindigkeit von etwa 10 m/s auf dem 500 mm x 340 mm großen Spielfeld muss das System Reaktionszeiten unter 50 ms gewährleisten (Erklärung in der Anforderung). Die Bildverarbeitungsgeschwindigkeit (PD-NFA4) muss daher mindestens 60 FPS betragen, um bei maximaler Kugelgeschwindigkeit noch eine ausreichende räumliche Auflösung der Kugelbahn zu erreichen.

Die Bilderfassungsqualität (PD-NFA2) muss ausreichend sein, um die metallische Kugel auf dem matten schwarzen Spielfeld zuverlässig zu detektieren. Dies erfordert eine Auflösung, die die Kugel mit mindestens 15-20 Pixeln Durchmesser abbildet, sowie einen ausreichenden Kontrast zwischen Kugel und Hintergrund. Die Kompatibilität (PD-NFA1) verlangt eine mechanisch unaufdringliche Integration ohne Beeinträchtigung des Spielbetriebs. Schließlich begrenzt die Kostenrestriktion (PD-NFA10) das Budget auf maximal 300€. Für die Lösung der Aufgabe kommen zwei Kategorien von Technologien in Frage:

- **RGB-Kameras** (2D-Kameras) erfassen zweidimensionale Farbbilder des Spielfelds. Sie nutzen Sensoren zur Aufzeichnung der Lichtintensität in verschiedenen Spektralbereichen (Rot, Grün, Blau). Für die Flipper-Anwendung bieten sie mehrere Vorteile. Die metallische Kugel hebt sich durch seinen Helligkeitsunterschied deutlich vom matten schwarzen Hintergrund ab. Die Positionen der Relevanten Elemente (Kugel, Arme, Spielfeld) können zusätzlich durch Farbmarker identifiziert werden. RGB-Kameras sind in verschiedenen Preisklassen verfügbar, von günstigen Webcams bis zu spezialisierten Industriekameras, die höhere Frameraten und deterministische Latenzzeiten bieten.

- **Tiefenkameras** (3D-Kameras) erfassen zusätzlich zur Bildinformation die Entfernung jedes Bildpunkts zur Kamera. Preislich liegen sie zwischen 150-600€. Theoretisch könnten Tiefenkameras die Entfernung der Kugel messen und damit zusätzliche Informationen für die Trajektorienschätzung liefern. Außerdem sind Tiefenkameras besonders robust gegenüber wechselnden Lichtverhältnissen. In der Praxis zeigen sich jedoch mehrere Nachteile. Die Auflösung von Tiefenkameras ist typischerweise geringer als bei RGB-Kameras (z. B. 640x480). Die Framerate liegt oft unter 30 FPS, was für schnelle Kugelbewegungen unzureichend ist. Zudem haben kleine, metallische Objekte wie der Flipperkugel oft schlechte Reflexionseigenschaften für die verwendeten Infrarot-Signale, was zu unzuverlässigen Tiefenmessungen führt.

Zur systematischen Bewertung der Kameraoptionen wird eine gewichtete Entscheidungsmatrix mit den vorher genannten Technologien verwendet. Diese ist in der Tabelle 5.1 dargestellt. Die Bewertung erfolgt anhand einer Skala von eins (ungenügend) bis fünf (sehr gut).

Tabelle 5.1: Entscheidungsmatrix zwischen 2D-RGB-Kamera und Tiefenkamera

Kriterium	**Gewichtung**	**2D-RGB-Kamera**	**Tiefenkamera**
Framerate und Latenz	30%	5	3
Auflösung	20%	5	3
Tiefenmessung	10%	1	4
Lichtempfindlichkeit	10%	3	5
Bibliothekskompatibilität	10%	5	3
Integrationsaufwand	10%	4	3
Kosten	10%	4	3
Gewichtete Summe	**100%**	**4,2**	**3,3**

Die 2D-RGB-Kamera erzielt eine deutlich bessere Bewertung. Die höhere Framerate und Auflösung sind für die schnelle Kugelverfolgung essentiell. Die zusätzliche Tiefeninformation einer 3D-Kamera bietet keinen deutlichen Mehrwert, da die relevanten Bewegungen in der Spielebene stattfinden. Außerdem sei gesagt, dass die dargelegte Betrachtung in diesem Kapitel nicht ausschließlich relevant für die Differenzierung zwischen 2D- und 3D-Kamera ist, sondern eine Grundlage für die spezifische Kamerawahl schafft. Die vorausgegangenen Betrachtungen werden in der Entwicklung des Systems wiederkehrend wichtig sein.

5.3 Software

Die Software-Komponenten bilden die intelligente Steuerungsebene des autonomen Flippersystems und ermöglichen die Verarbeitung der Sensordaten sowie die autonome Entscheidungsfindung. Wie in Kapitel 5.1 dargestellt, umfasst die Software vier zentrale Module: die Bildverarbeitungseinheit, die virtuelle Trainingsumgebung, den RL-Agenten sowie den Speicher. Dieses Kapitel widmet sich der konzeptionellen Ausgestaltung dieser Softwaremodule unter Berücksichtigung der in Kapitel 4 definierten funktionalen und nicht-funktionalen Anforderungen.

5.3.1 Bildverarbeitungsstrategie

Die Bildverarbeitungsstrategie stellt ein zentrales Bindeglied zwischen der physikalischen Welt des Flipperautomaten und der Entscheidungslogik des RL-Agenten dar. Ziel dieser Softwarekomponente ist es, aus den durch das Kamerasystem gelieferten Rohbildern relevante Zustandsinformationen (Position der Kugel, Stellung der Flipperarme) in Echtzeit zu extrahieren und dem Agenten in geeigneter Form bereitzustellen.

Die Anforderungen an diese Softwarekomponente ergeben sich unmittelbar aus den zuvor definierten funktionalen und nicht-funktionalen Anforderungen des physischen Demonstrators (vgl. Kapitel 4.4.2). Eine zentrale funktionale Forderung ist die kontinuierliche und zuverlässige Erfassung des Systemzustands (PD-FA2). Die Bildverarbeitung muss die Position der Kugel sowie die Stellung der Flipperarme präzise und robust erkennen. Da es sich um ein physikalisch hochdynamisches System handelt, in dem die Kugel hohe Geschwindigkeiten erreicht, darf die Objekterkennung nicht anfällig für Rauschen, Bildartefakte oder wechselnde Lichtverhältnisse sein. Ein entscheidender Aspekt ist die Echtzeitfähigkeit des Gesamtsystems. Die Anforderung PD-NFA3 legt eine maximale Gesamtlatenz von 50 ms fest. Zudem wird mit PD-NFA4 gefordert, dass die Bildverarbeitung mindestens 60 Bilder pro Sekunde verarbeiten kann. Auch die langfristige Perspektive spielt bei der Auswahl eine Rolle: Die Lösung muss modular aufgebaut (PD-NFA7), auf verschiedenen Recheneinheiten einsetzbar (PD-NFA5) und gut dokumentiert sein (PD-NFA8). Aus diesen Anforderungen lassen sich sieben zentrale Kriterien ableiten:

1. Genauigkeit der Objekterkennung
2. Robustheit gegenüber Lichtschwankungen

3. Verarbeitungslatenz
4. Framerate-Kompatibilität
5. Modularität und Portierbarkeit
6. Wartbarkeit und Nachvollziehbarkeit
7. Kosten- und Entwicklungsaufwand

Für die Realisierung der Bildverarbeitungsstrategie kommen drei grundlegende Technologieansätze in Betracht:

- **Klassische Bildverarbeitung:** Dieser Ansatz nutzt deterministische Verfahren. Die Vorteile liegen in der vollständigen Kontrolle über den Verarbeitungsprozess und der geringen Rechenleistungsanforderung. Die Algorithmen sind transparent und ihr Verhalten vorhersagbar, was die Fehlerdiagnose erleichtert.
- **Deep Learning-basierte Bildverarbeitung:** Neuronale Netze, insbesondere Convolutional Neural Networks (CNNs), können komplexe Muster lernen und sind robust gegenüber Variationen.
- **Hybride Verfahren:** Diese kombinieren klassische Bildverarbeitung mit Deep Learning-Komponenten. Dieser Ansatz verspricht die Vorteile beider Welten, erhöht jedoch die Systemkomplexität und den Wartungsaufwand erheblich.

Zur systematischen Bewertung der Bildverarbeitungsansätze wird eine gewichtete Entscheidungsmatrix verwendet. Diese ist in der Tabelle 5.2 dargestellt. Die Bewertung erfolgt anhand einer Skala von eins (ungenügend) bis fünf (sehr gut).

Tabelle 5.2: Entscheidungsmatrix der Bildverarbeitungsansätze

Kriterium	**Gewichtung**	**Klassisch**	**DL**	**Hybrid**
Genauigkeit der Objekterkennung	20%	4	5	5
Verarbeitungslatenz	20%	5	3	3
Robustheit ggü. Lichtschwankungen	15%	3	5	4
Framerate-Kompatibilität	15%	5	3	4
Modularität und Portierbarkeit	10%	5	3	3
Wartbarkeit und Nachvollziehbarkeit	10%	5	2	3
Kosten- und Entwicklungsaufwand	10%	5	2	2
Gewichtete Summe	**100%**	**4,35**	**3,35**	**3,5**

Die klassische Bildverarbeitung erzielt mit 4,35 Punkten die beste Bewertung. Die Entscheidung wird durch mehrere Faktoren gestützt: Die deterministischen Verfahren bieten die geringste Latenz und höchste Framerate-Kompatibilität, was für die Echtzeitanforderungen des Flippersystems essentiell ist. Der strukturierte Aufbau des Spielfelds mit klar definierten Objekten begünstigt klassische Ansätze. Die vollständige Transparenz der Algorithmen ermöglicht präzise Fehlerdiagnose und Optimierung. Zudem ist keine aufwendige Datensammlung und kein Training erforderlich, was den Entwicklungsaufwand minimiert.

Basierend auf dieser Analyse wird für die Bildverarbeitungsstrategie ein klassischer Ansatz gewählt, der optimal auf die spezifischen Anforderungen des autonomen Flippersystems abgestimmt ist. Auch hier sei gesagt, dass die vorausgegangenen Betrachtungen dieses Kapitels nicht nur für die Wahl, sondern auch für die Umsetzung der Bildverarbeitungsstrategie wichtig sind.

5.3.2 Simulationsumgebung

Die virtuelle Trainingsumgebung bildet das Kernstück für die Entwicklung und das Training des RL-Agenten und muss die physikalischen Eigenschaften des realen Flippers hinreichend genau nachbilden. Die Anforderungen an diese Komponente ergeben sich aus Kapitel 4.3.2 und umfassen sowohl die physikalisch korrekte Simulation der Spielmechanik als auch die nahtlose Integration mit RL-Frameworks.

Aus den funktionalen Anforderungen leitet sich ab, dass die Simulationsumgebung den Systemzustand kontinuierlich berechnen (VU-FA2), dem RL-Agenten Zustandsinformationen bereitstellen (VU-FA3), Steuerbefehle der Flipperarme verarbeiten (VU-FA4) sowie ein automatisches Episoden-Management implementieren muss (VU-FA5). Die nichtfunktionalen Anforderungen fordern eine physikalisch genaue Modellierung mit Berücksichtigung der Spielfeldneigung (VU-NFA1), maßstabsgetreue Abbildung der Geometrie (VU-NFA2), geeignete Zustandsraum-Definition (VU-NFA3) sowie ein Belohnungssystem, das dem Agenten ermöglicht, die Kugel mindestens 15 Sekunden im Spiel zu halten (VU-NFA4).

Für die Realisierung der virtuellen Trainingsumgebung kommen drei grundlegende Ansätze in Betracht. Game-Engine-basierte Simulationen wie Unity oder Unreal Engine bieten hochentwickelte Physik-Engines mit fotorealistischer Visualisierung und schneller Prototypenentwicklung. Nachteile sind eingeschränkte Kontrolle über Physikparameter,

Performance-Probleme bei parallelen Simulationen und potenzielle Lizenzkosten. Spezialisierte Physik-Simulatoren wie MuJoCo oder PyBullet sind explizit für RL-Anwendungen entwickelt und bieten deterministische Physikberechnung sowie effiziente Parallelisierung. Sie erfordern jedoch umfangreiche Einarbeitung, bieten eingeschränkte Visualisierung und sind primär für Roboter-Kinematik ausgelegt. Custom-Implementierungen ermöglichen vollständige Kontrolle über alle Simulationsaspekte mit optimaler Anpassung an die spezifischen Flipper-Anforderungen. Der Nachteil liegt im höheren Entwicklungsaufwand und der eigenständigen Implementierung aller Physikalgorithmen.

Zur systematischen Bewertung wird eine gewichtete Entscheidungsmatrix verwendet, die in Tabelle 5.3 dargestellt ist.

Tabelle 5.3: Entscheidungsmatrix der Simulationsansätze

Kriterium	**Gew.**	**Game Eng.**	**Physik-Sim.**	**Custom**
Physikalische Genauigkeit	25%	3	4	5
Anpassbarkeit an realen Flipper	20%	2	3	5
RL-Framework Integration	20%	3	5	4
Performance für Training	15%	3	5	4
Entwicklungsaufwand	10%	5	3	2
Transparenz	10%	2	3	5
Gewichtete Summe	**100%**	**3,05**	**4,05**	**4,40**

Die Custom-Implementierung erzielt mit 4,40 Punkten die beste Bewertung und bietet vollständige Kontrolle über Physikparameter für exakte Nachbildung des realen Flippers. Die Architektur folgt dem Gymnasium-Standard als De-facto-Interface für RL-Umgebungen. Diese Standardisierung ermöglicht nahtlose Integration mit RL-Bibliotheken wie Stable-Baselines3 und trennt Datenmodelle, Physiksimulation, Kollisionserkennung und Visualisierung modular.

5.3.3 Agent

Der Reinforcement-Learning-Agent bildet die zentrale Entscheidungskomponente des autonomen Flippersystems und muss basierend auf Zustandsinformationen optimale Steuerungsentscheidungen für die Flipperarme treffen. Die Wahl des RL-Algorithmus hat maßgeblichen Einfluss auf Trainingseffizienz, Konvergenzverhalten und die letztendliche Performance des Systems.

Aus den Anforderungen ergibt sich, dass der Agent mit einem diskreten Aktionsraum arbeitet, da die Flipperarme entweder aktiviert oder deaktiviert werden (VU-FA4). Der Zustandsraum umfasst kontinuierliche Variablen wie Flipper-Winkel und Kugelposition, ist jedoch niedrigdimensional. Die Anforderung VU-NFA4 verlangt, dass der Agent die Kugel mindestens 15 Sekunden im Spiel halten kann, was sowohl Sample-Effizienz als auch Stabilität beim Training erfordert.

Für die Auswahl des RL-Algorithmus kommen drei etablierte Verfahren in Betracht. Deep Q-Networks sind wertbasierte Methoden, die für diskrete Aktionsräume entwickelt wurden. Sie nutzen Experience Replay zur Entkopplung zeitlicher Korrelationen und ein Target Network zur Stabilisierung des Trainings. DQN ist sample-effizient, da Erfahrungen mehrfach verwendet werden, und eignet sich besonders für niedrigdimensionale Zustandsräume. Als Off-Policy-Algorithmus kann DQN aus beliebigen Erfahrungen lernen. Nachteile sind die Beschränkung auf diskrete Aktionen und potenzielle Instabilitäten bei komplexen Value-Funktionen.

Proximal Policy Optimization (PPO) ist ein Policy-Gradient-Verfahren, das direkt eine stochastische Policy lernt. PPO zeichnet sich durch robustes Konvergenzverhalten und einfache Hyperparameter-Tuning aus. Es funktioniert sowohl mit diskreten als auch kontinuierlichen Aktionsräumen und ist weniger anfällig für katastrophales Vergessen. Als On-Policy-Algorithmus ist PPO jedoch weniger sample-effizient, da Erfahrungen nur einmal verwendet werden. PPO erfordert zudem mehr Trainingsschritte bis zur Konvergenz.

Soft Actor-Critic (SAC) ist ein moderner Off-Policy-Algorithmus, der Maximum-Entropy-RL nutzt. SAC ist hochgradig sample-effizient und zeigt stabiles Trainingsverhalten. Der Algorithmus ist primär für kontinuierliche Aktionsräume konzipiert und kann zwar auf diskrete Räume adaptiert werden, verliert dabei jedoch Effizienzvorteile. Die Implementierung ist komplexer als DQN oder PPO.

Zur systematischen Bewertung wird eine gewichtete Entscheidungsmatrix verwendet, die in Tabelle 5.4 dargestellt ist.

DQN erzielt mit 4,55 Punkten die beste Bewertung. Diese Entscheidung wird durch mehrere Faktoren gestützt: Die native Unterstützung diskreter Aktionsräume macht DQN ideal für die binäre Flipper-Steuerung. Die hohe Sample-Effizienz durch Experience Replay ermöglicht effektives Lernen mit begrenzten Trainingsressourcen. Der niedrigdimensionale Zustandsraum des Flippers vermeidet die Komplexitätsprobleme, für die DQN

Tabelle 5.4: Entscheidungsmatrix der RL-Algorithmen

Kriterium	**Gew.**	**DQN**	**PPO**	**SAC**
Eignung für diskreten Action Space	25%	5	4	2
Sample-Effizienz	20%	5	3	5
Trainingsstabilität	20%	4	5	4
Konvergenzgeschwindigkeit	15%	4	3	4
Implementierungskomplexität	10%	4	5	3
Verfügbarkeit etablierter Implementierungen	10%	5	5	4
Gewichtete Summe	**100%**	**4,55**	**4,00**	**3,65**

anfällig sein kann. Etablierte Implementierungen in Stable-Baselines3 bieten geprüfte, optimierte Algorithmen mit umfangreicher Dokumentation.

6 Entwicklung des Bildverarbeitungssystems

Dieses Kapitel beschreibt die Entwicklung des Bildverarbeitungssystems zur Zustandserfassung des physischen Flipperautomaten. Zunächst wird die Auswahl des Kamerasystems sowie dessen physischer Aufbau dokumentiert. Anschließend wird die Softwarearchitektur mit der Klassenstruktur und dem Datenfluss zwischen den Komponenten dargestellt. Die folgenden Abschnitte behandeln die Kamerakalibrierung zur Korrektur optischer Verzerrungen sowie die drei zentralen Erkennungsmodule: die Spielfelderkennung zur Etablierung der Perspektivtransformation, die Kugelerkennung zur kontinuierlichen Verfolgung der Ballposition und die Flipperarm-Erkennung zur Bestimmung der Armwinkel. Jedes Modul wird hinsichtlich seiner Funktionsweise erläutert und anhand definierter Metriken evaluiert.

6.1 Kamerawahl und physischer Aufbau

Die Auswahl des Kamerasystems erfolgt auf Basis der in Kapitel 4.4.2 definierten Anforderungen sowie der konzeptionellen Überlegungen aus Kapitel 5.2. Für das autonome Flippersystem wird die Daheng Imaging VEN-161-61U3C-M01 Industriekamera gewählt, eine USB3.0-basierte RGB-Kamera mit 1,6 Megapixel Auflösung (1440 × 1080 Pixel) und einer maximalen Framerate von 163 FPS bei Vollauflösung.

Die Entscheidung für diese spezifische Kamera wird durch mehrere Faktoren bestimmt. Zunächst übertrifft die Kamera deutlich die in PD-NFA4 geforderten 60 FPS und ermöglicht damit eine hochauflösende Verfolgung schneller Kugelbewegungen. Bei der gegebenen Spielfeldgröße wird die Stahlkugel mit etwa 60-65 Pixeln Durchmesser abgebildet, was die in PD-NFA2 geforderte Mindestqualität für zuverlässige Objekterkennung deutlich

Abbildung 6.1: Physischer Aufbau des Flippersystems

übersteigt. Die USB3.0-Schnittstelle und die kurzen Belichtungszeiten der Industriekamera unterstützen zudem die kritische Anforderung PD-NFA3 einer Gesamtlatenz unter 50 ms.

Ein entscheidender Faktor für die Kamerawahl ist, dass die Kamera durch vorherige Anschaffung verfügbar ist. Dies ermöglichte den Einsatz einer hochwertigen Industriekamera, die bei regulärem Erwerb die in PD-NFA10 definierte Budgetgrenze von 300 € deutlich überschritten hätte. Darüber hinaus erfüllt die Kamera die Kompatibilitätsanforderung PD-NFA1 durch ihre standardkonforme USB3.0-Schnittstelle und die verfügbaren Python-SDK-Bibliotheken, die eine nahtlose Integration in ein OpenCV-basiertes Bildverarbeitungssystem ermöglichen. Die Wahl einer Industriekamera gegenüber einer Consumer-Webcam wird letztendlich durch die höhere Bildqualität, Bildfrequenz und die Möglichkeit präziser Parametersteuerung von Belichtungszeit und Verstärkung begründet, die für die Echtzeitanforderungen des autonomen Flippersystems essenziell sind.

Abbildung 6.1 zeigt die Anbringung der Kamera am Flipper. Die Kamera ist mittels eines modularen Aluminiumprofil-Gestells direkt über dem Flipperautomaten montiert. Das Gestell besteht aus 20×20 mm Aluminiumprofilen und ermöglicht eine stabile, vibrationsfreie Positionierung der Kamera in einem Abstand von 45 cm zum Spielfeld. Die Kamera ist über einen verstellbaren Haltearm am horizontalen Querträger befestigt, wodurch eine senkrechte Ausrichtung auf das Spielfeld gewährleistet wird. Diese Anordnung erfüllt die Kompatibilitätsanforderung PD-NFA1, da das Gestell den mechanischen Spielbetrieb nicht beeinträchtigt und gleichzeitig eine vollständige Erfassung des Spielfelds ermöglicht. Die USB3.0-Verbindung zur Recheneinheit erfolgt über ein 2 Meter langes Kabel, das ausreichend Bewegungsfreiheit für Wartungsarbeiten bietet. Die robuste Konstruktion gewährleistet eine konstante Kameraposition auch bei dynamischen Bewegungen der Flipperarme und verhindert Bildverzerrungen durch mechanische Schwingungen.

6.2 Softwarearchitektur

In diesem Abschnitt wird die Softwarearchitektur des Bildverarbeitungssystems dargestellt. Zunächst wird die Klassenstruktur erläutert. Anschließend wird der High-Level-Datenfluss zwischen den Komponenten beschrieben.

6.2.1 Klassenübersicht

Ein Klassendiagramm bietet eine übersichtliche Darstellung der Softwarearchitektur und verdeutlicht die Beziehungen zwischen den verschiedenen Komponenten. Es ermöglicht eine strukturierte Darstellung der Verantwortlichkeiten und erleichtert das Verständnis der modularen Softwareaufteilung. Das in Abbildung 6.2 dargestellte Klassendiagramm zeigt die objektorientierte Struktur des entwickelten Bildverarbeitungssystems.

Die Architektur folgt dem Prinzip der Kapselung funktionaler Einheiten, wodurch eine modulare und erweiterbare Implementierung erreicht wird. Im Zentrum des Systems steht die FlipperTracker-Klasse, die als Koordinator für die gesamte Bildverarbeitungspipeline fungiert und die verschiedenen Detektionsmodule orchestriert.

Die Struktur beginnt mit der Klasse CameraCalib, die die grundlegende Funktionalität für die Kamerakalibrierung bereitstellt. Von der FlipperTracker-Klasse aus verzweigt sich das System in drei spezialisierte Detektorklassen. Der FieldDetector ist für die initiale

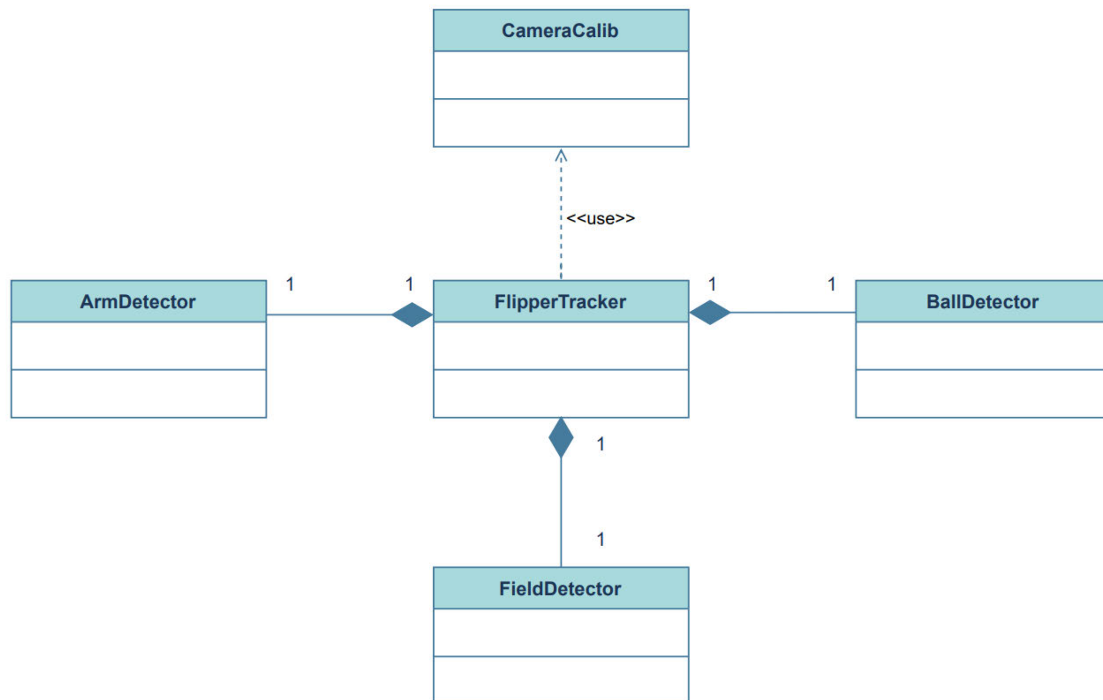


Abbildung 6.2: Klassendiagramm der Bildverarbeitung

Erkennung und perspektivische Transformation des Spielfelds verantwortlich, während der BallDetector die kontinuierliche Verfolgung der Kugel realisiert. Der ArmDetector komplettiert die Funktionalität durch die Winkelbestimmung der Flipperarme. Diese Gliederung der Detektionsaufgaben ermöglicht eine effiziente Verarbeitung der verschiedenen Spielelemente.

Die Komposition der Klassen zeigt sich in den 1:1-Beziehungen. Jede FlipperTracker-Instanz komponiert genau eine Instanz jeder Detektorklasse. Außerdem wird eine temporäre CameraCalib-Instanz genutzt. Diese Architektur gewährleistet eine klare Trennung der Verantwortlichkeiten und erleichtert die Wartbarkeit des Systems. Zur Wahrung der Übersichtlichkeit wurden im Diagramm bewusst die Attribute und Methoden der einzelnen Klassen ausgelassen. Deren detaillierte Funktionalitäten werden in den nachfolgenden Abschnitten dieses Kapitels ausführlich erläutert.

6.2.2 High-Level-Datenfluss

Die Verständlichkeit eines komplexen Bildverarbeitungssystems hängt maßgeblich von der Nachvollziehbarkeit der Datenströme zwischen seinen Komponenten ab. Während die in Kapitel 6.2 dargestellte Klassenstruktur die statischen Beziehungen und Verantwortlichkeiten aufzeigt, bedarf es einer ergänzenden Betrachtung der dynamischen Aspekte des Systems. Der High-Level-Datenfluss visualisiert die zeitliche Abfolge der Methodenaufrufe zwischen den Klassen und schafft damit das konzeptionelle Verständnis für die Orchestrierung der verschiedenen Bildverarbeitungsmodule. Das in Abbildung 6.3 dargestellte Sequenzdiagramm veranschaulicht den Datenfluss des Bildverarbeitungssystems von der Initialisierung bis zur Objektverfolgung. Das Diagramm gliedert den Systemablauf in vier wesentliche Abschnitte, die die Struktur des Bildverarbeitungssystems widerspiegeln.

Die Systeminitialisierung bildet den Ausgangspunkt der Verarbeitung, wobei der externe Aufruf von *run()* (in Main-Programm) die Orchestrierung des gesamten Systems in eine Hauptschleife versetzt (Schleife zur Übersichtlichkeit nicht dargestellt). Die Kamerainitialisierung durch *initialize_camera()* etabliert die sensorische Schnittstelle, während die nachfolgende Kalibrierung über *setup_calibration()* die geometrischen Verzerrungsparameter aus der CameraCalib-Klasse lädt und die erforderlichen Korrekturmatrizen bereitstellt.

Die Thread-Verarbeitung implementiert eine Producer-Consumer-Architektur, bei der *grab_frames()* als Producer-Thread kontinuierlich Bild-Frames von der Kamera akquiriert und kalibriert, während *process_frames()* als Consumer-Thread die eigentliche Bildverarbeitung übernimmt. Diese Parallelisierung soll die Echtzeitfähigkeit des Systems durch die Entkopplung von Datenakquisition und -verarbeitung gewährleisten.

Der Bildverarbeitungsprozess selbst unterteilt sich in zwei sequenzielle Phasen. Die Felderkennungsphase nutzt eine Verarbeitungsschleife, die iterativ *detect_field_edges()* aufruft, bis die Spielfeldbegrenzungen erfolgreich identifiziert sind und eine stabile Perspektivtransformation etabliert wurde. Diese einmalige Kalibrierungsphase schafft die geometrische Grundlage für alle nachfolgenden Objekterkennungsoperationen.

Nach erfolgreicher Felderkennung übernimmt die Objektverfolgungsphase mit einer kontinuierlichen Verarbeitungsschleife. Jeder Verarbeitungszyklus beginnt mit der Perspektivtransformation durch *transform_to_field_view()*, die das Kamerabild auf die inneren

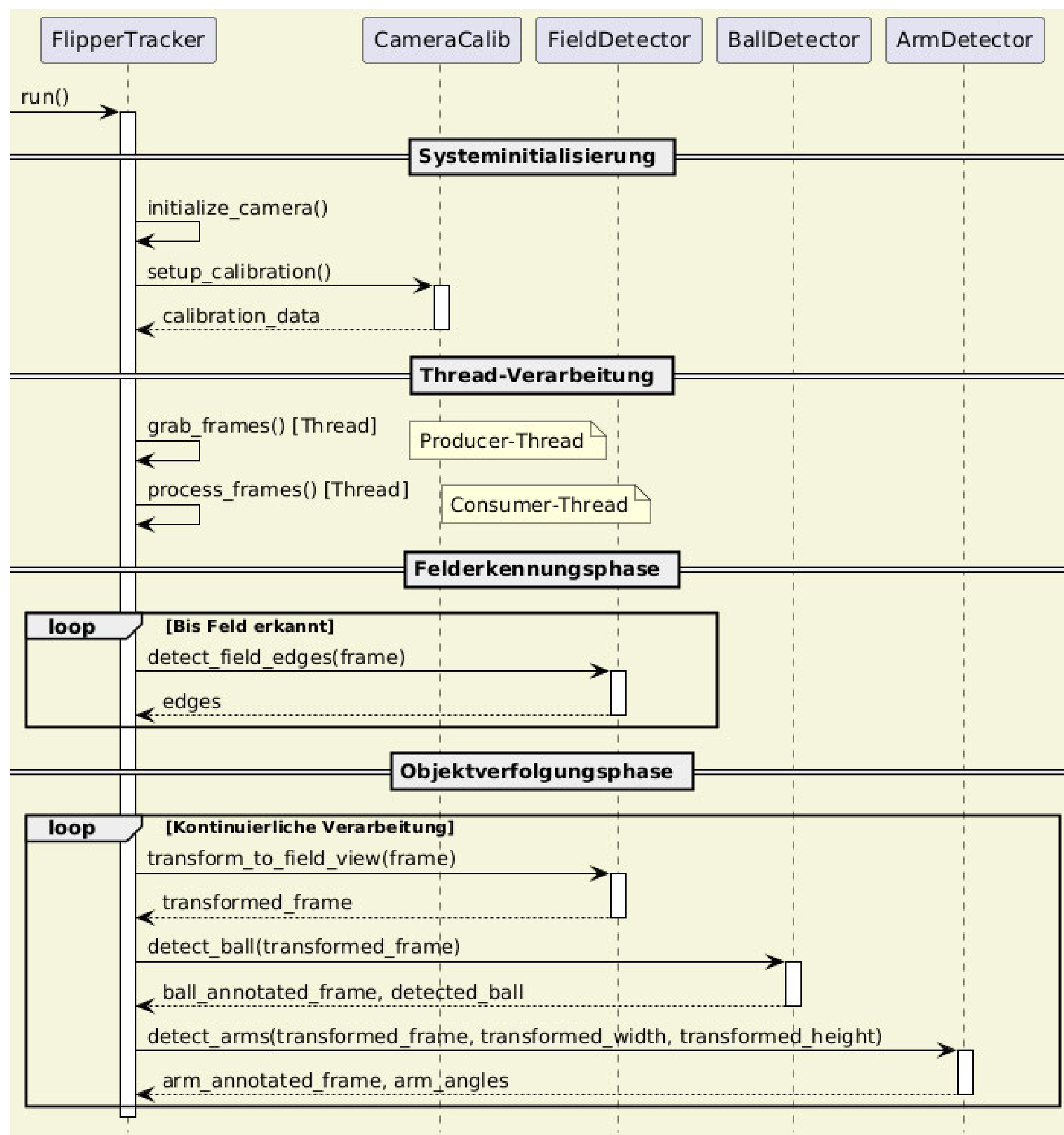


Abbildung 6.3: Vereinfachtes Sequenzdiagramm des Bildverarbeitungssystems

Spielfeldgrenzen begrenzt. Die Verarbeitung durch *detect_ ball()* und *detect_ arms()* extrahiert die Kugelposition sowie die Orientierung der Flipperarme aus dem transformierten Bildmaterial.

Es ist wichtig zu betonen, dass dieses Sequenzdiagramm eine abstrahierte Darstellung des Systems bietet, die bewusst auf implementierungsspezifische Details verzichtet. Die Darstellung fokussiert sich auf die wesentlichen Methodenaufrufe zwischen den Klassen

und abstrahiert von internen Verarbeitungsschritten und algorithmischen Details. Diese bewusste Vereinfachung dient dem Zweck, ein klares Verständnis für die grundlegende Programmstruktur und den High-Level-Datenfluss zu schaffen. Das Diagramm fungiert somit als konzeptionelle Brücke zwischen der statischen Klassenarchitektur und der detaillierten Implementierung. Die spezifischen Funktionalitäten der einzelnen Klassen und Module werden in den folgenden Abschnitten ausführlich behandelt.

6.3 Kamerakalibrierung

Die Kamerakalibrierung stellt einen fundamentalen Vorverarbeitungsschritt dar, der die geometrische Genauigkeit des gesamten Bildverarbeitungssystems maßgeblich beeinflusst. Ohne eine präzise Korrektur der kameraspezifischen Verzerrungen würden systematische Geometriefehler die nachgelagerte Objekterkennung und Positionsbestimmung erheblich beeinträchtigen. Für die Kamerakalibrierung wird ein ChArUco-Board (Chessboard + ArUco) eingesetzt, das die Vorteile von Schachbrettmustern mit der Robustheit von ArUco-Markern (Artificial Recognizable Markers Using OpenCV) kombiniert [2]. Das ChArUco-Board ermöglicht eine besonders präzise Eckenerkennung auch bei teilweiser Verdeckung und bietet durch die eindeutigen ArUco-Marker eine zuverlässige Identifikation der Kalibrierpunkte. Die Bestimmung der Kameraparameter erfolgt durch die spezialisierte Klasse CameraCalib, deren Funktionsweise mithilfe eines Aktivitätsdiagramms in Abbildung 6.4 dargestellt ist.

Das System initialisiert zunächst die Kamera und zeigt ein Kalibrierungsfenster an. Der Benutzer hält das ChArUco-Board vor die Kamera, wobei das System kontinuierlich nach ArUco-Markern und Schachbrett-Ecken sucht. Bei erfolgreicher Erkennung werden die Eckkoordinaten gesammelt. Dieser Prozess wiederholt sich, bis ausreichend Kalibrierpunkte aus verschiedenen Positionen und Orientierungen erfasst wurden. Anschließend berechnet das System mittels *cv2.aruco.calibrateCameraCharuco()* die Kameramatrix K und die Verzerrungskoeffizienten als Vektor D (siehe Kapitel 2.1.1). Die ermittelten Parameter werden schließlich als JSON-Datei gespeichert. Die praktische Anwendung der Kalibrierungsdaten zur Bildentzerrung wird durch das in Abbildung 6.5 dargestellte Aktivitätsdiagramm veranschaulicht. Das System lädt zunächst die zuvor gespeicherten Kalibrierungsdaten und berechnet mit *cv2.getOptimalNewCameraMatrix()* eine optimierte Kameramatrix, die den verfügbaren Bildbereich maximal ausnutzt. Anschließend werden mit *cv2.initUndistortRectifyMap()* die Entzerrungskarten vorbereitet, wel-

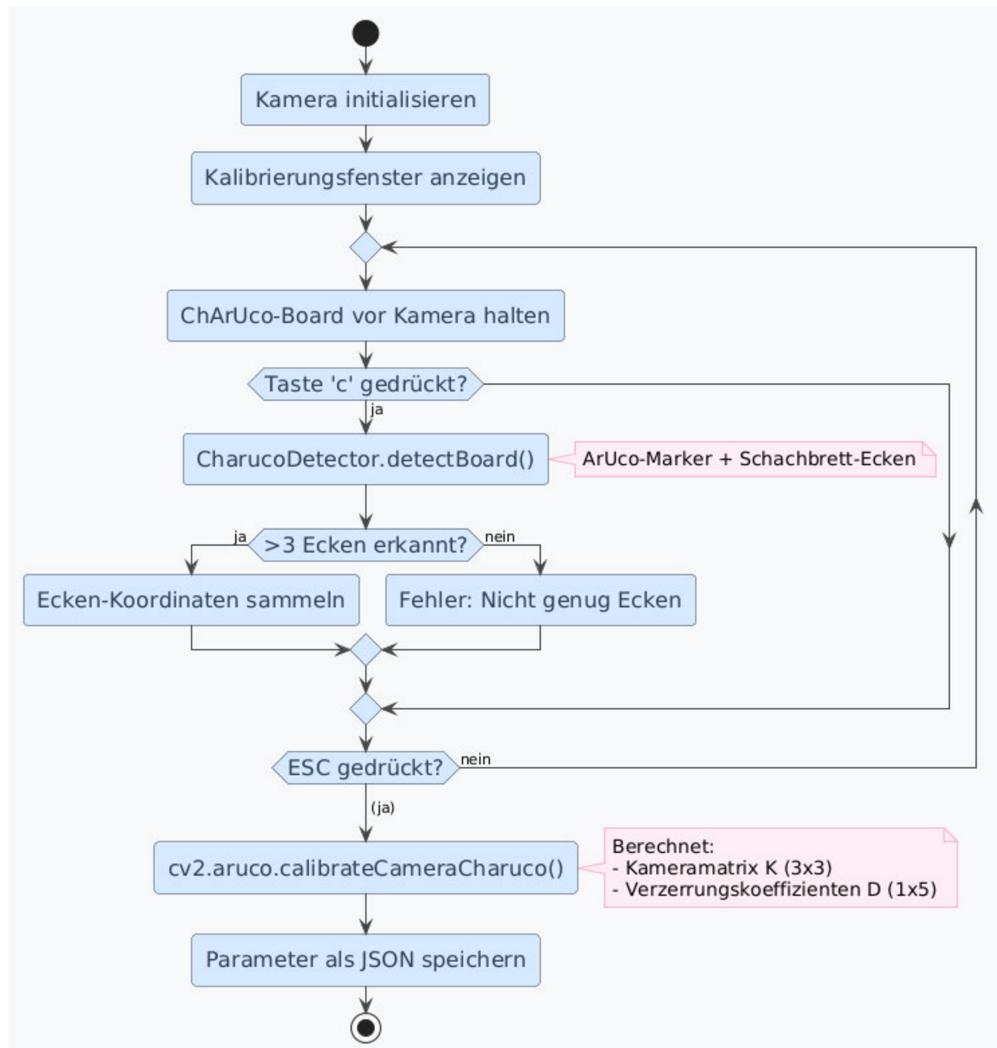


Abbildung 6.4: Vereinfachtes Aktivitätsdiagramm der CameraCalib-Klasse

che die pixelgenaue Zuordnung zwischen verzerrtem und korrigiertem Bild definieren. Während der Laufzeit wird jedes von der Kamera erfasste Rohbild mittels *cv2.remap()* unter Verwendung der vorberechneten Karten effizient entzerrt (innerhalb Grab-Thread in FlipperTracker-Klasse). Dieser Ansatz gewährleistet eine optimale Performance, da die rechenintensive Berechnung der Transformationsparameter nur einmalig während der Initialisierung erfolgt und die Echtzeitverarbeitung durch die hochoptimierte Remap-Operation realisiert wird.

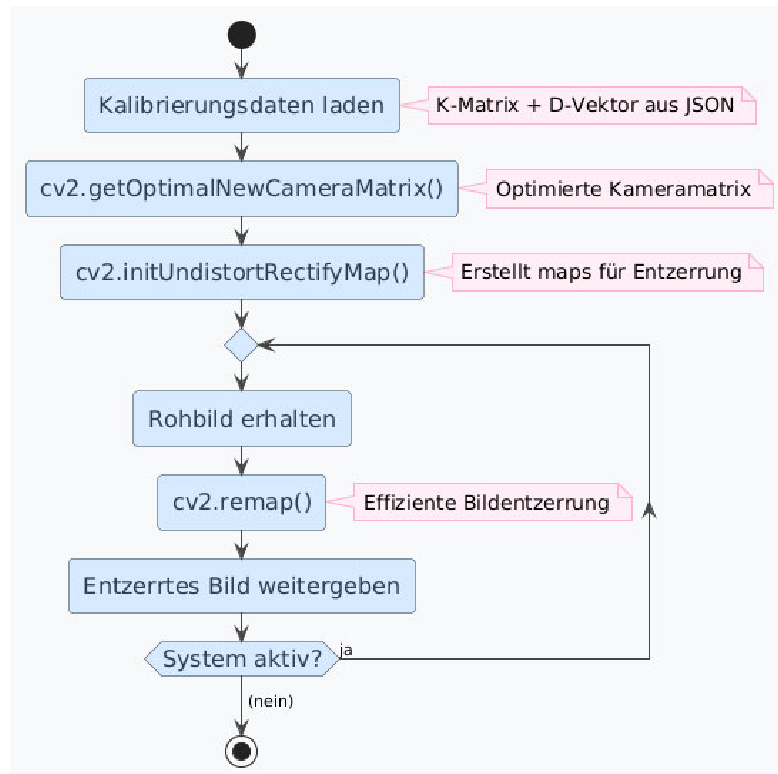


Abbildung 6.5: Vereinfachtes Aktivitätsdiagramm der Codesequenz für die Bildentzerrung

6.4 Spielfelderkennung

Die Spielfelderkennung bildet die Grundlage für alle nachfolgenden Bildverarbeitungsschritte. In diesem Abschnitt wird zunächst die Funktionsweise des Erkennungsalgorithmus beschrieben. Anschließend erfolgt die Evaluation anhand definierter Metriken zur Bewertung der geometrischen Genauigkeit.

6.4.1 Funktionsweise

Die Spielfelderkennung bildet eine wichtige Vorverarbeitungskomponente des Bildverarbeitungssystems, da sie die geometrische Grundlage für alle nachfolgenden Objekterkennungsoperationen schafft. Durch die automatische Identifikation und perspektivische Transformation des Bildbereichs wird eine normierte Arbeitsebene etabliert. Diese Normierung ist besonders relevant für den späteren Einsatz des RL-Agenten, der auf konsis-

tente Zustandsrepräsentationen angewiesen ist. Darüber hinaus ermöglicht die Spielfeldnormierung eine deutliche Reduktion des zu verarbeitenden Bildbereichs, wodurch die Recheneffizienz des Gesamtsystems optimiert wird.

Die Implementierung der Spielfelderkennung erfolgt durch die spezialisierte Klasse FieldDetector, deren algorithmischer Ablauf mittels eines Aktivitätsdiagramms in Abbildung 6.6 strukturiert dargestellt ist. Das Diagramm veranschaulicht den sequenziellen Erkennungsprozess von der initialen Bildaufnahme bis zur finalen Perspektivtransformation. Der Spielfelderkennungsprozess gliedert sich in vier wesentliche Phasen, die iterativ durchlaufen werden, bis eine stabile Felderkennung etabliert ist. Die Bildvorverarbeitung wandelt das Eingangsbild in ein Grauwertbild um und wendet eine gaußsche Glättung (siehe Kapitel 2.1.2) zur Rauschunterdrückung an.

Aufgrund der Kamerapositionierung entspricht die Bildgeometrie einer um 90° im Uhrzeigersinn rotierten Darstellung der in Abbildung 3.1 dokumentierten physischen Anordnung. Die algorithmisch als linke Kante referenzierte Komponente korreliert mit der unteren Spielfeldkante des realen Aufbaus. Die Kantendetektion fokussiert sich gezielt auf die Identifikation der linken und rechten vertikalen Spielfeldkanten (obere und untere im realen Aufbau), die als primäre Orientierung dienen. Für die linke Kante wird der komplette freie Bereich zwischen dem hellen Spielfeldrand und dem dunklen Spielfeld ausgenutzt. Diese Kante bietet optimale Erkennungsbedingungen, da sie über ihre gesamte Länge unverdeckt ist und einen ausgeprägten Helligkeitskontrast aufweist (Silber-zu-Schwarz-Übergang). Die rechte Kante hingegen wird nur in ihrem freien, nicht von Spielelementen verdeckten Bereich detektiert.

Dieser partielle Erkennungsansatz ist ausreichend, da die rechte Kante primär zur Bestimmung der horizontalen Spielfeldausdehnung benötigt wird und somit die Länge des Spielfelds definiert. Die Kantenerkennung verwendet eine adaptive Schwellwertanalyse, bei der für jede x-Position entlang der linken Bildhälfte eine vertikale Linie abgetastet wird. Dabei wird nach charakteristischen Helligkeitsübergängen gesucht, die den Übergang vom metallischen Spielfeldrand zum matten schwarzen Spielfeld kennzeichnen (siehe Kapitel 2.1.3). Für die rechte Kante erfolgt eine entsprechende Analyse in der rechten Bildhälfte, wobei nach dem umgekehrten Übergang (Schwarz-zu-Silber) gesucht wird. Die Auswahl der richtigen Kanten wird mittels einer Bewertung des Helligkeitsabfalls, der Anzahl an schwarzen Pixeln neben der Kante und der Länge der Kante durchgeführt. Die Eckpunktebestimmung erfolgt durch horizontale Interpolation zwischen den detektierten vertikalen Kanten. Unter Verwendung der längeren linken Kante als Refe-

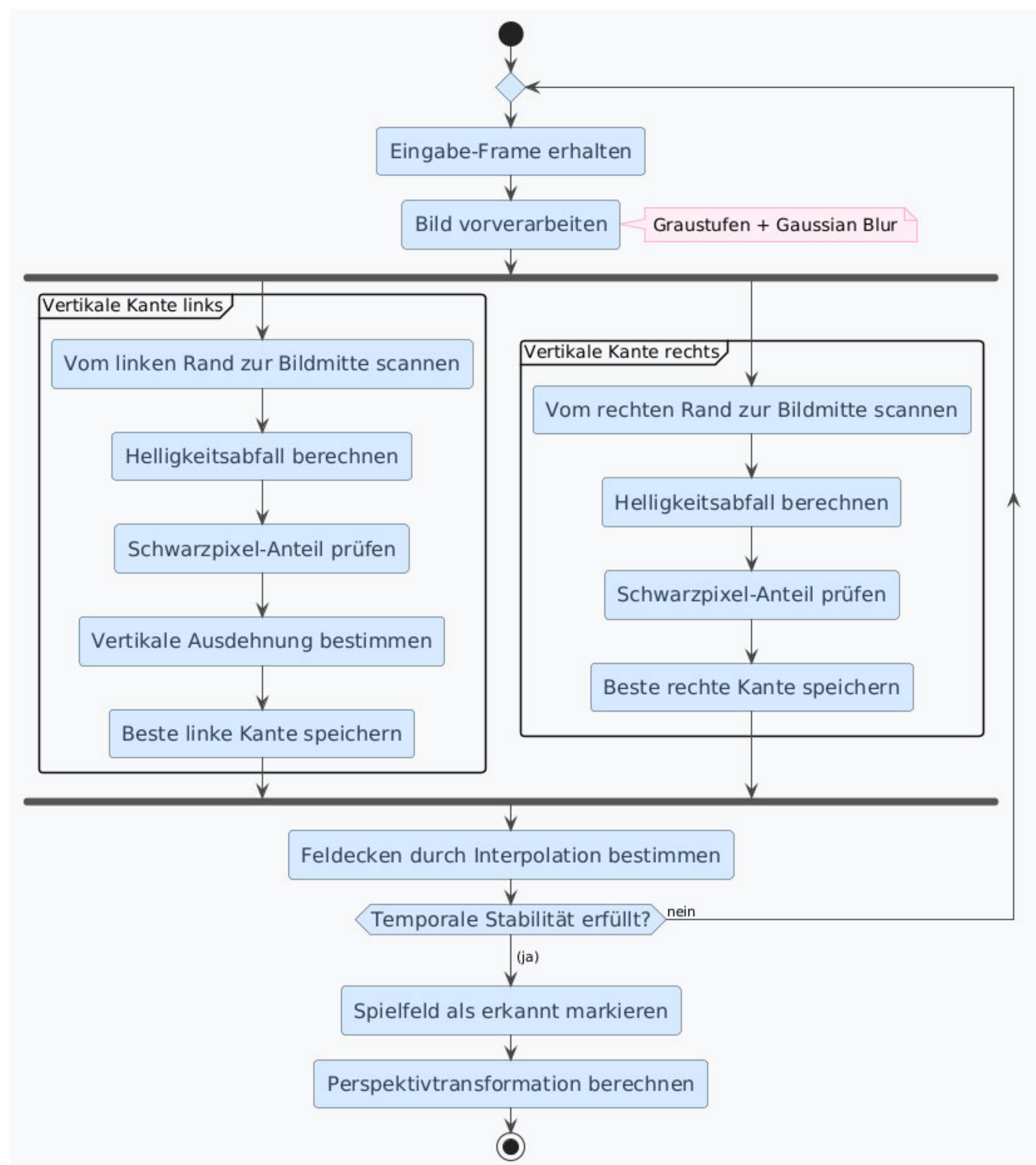


Abbildung 6.6: Vereinfachtes Aktivitätsdiagramm der FieldDetector-Klasse

renz werden die vier Eckpunkte des Spielfelds durch Projektion der vertikalen Grenzen der linken Kante auf die Position der rechten Kante berechnet.

Zur Sicherstellung der Erkennungsstabilität ist eine temporale Filterung durch Akkumulation mehrerer aufeinanderfolgender Erkennungsergebnisse implementiert. Erst wenn

mindestens fünf konsistente Feldbegrenzungen innerhalb einer definierten Toleranz (Eckpunkte innerhalb von 30 Pixeln wiederkehrend erkannt) erkannt wurden, wird die Spielfelderkennung als bestätigt klassifiziert. Diese Phase verhindert falsch-positive Erkennungen aufgrund von Bildrauschen oder temporären Störungen.

Nach erfolgreicher Felderkennung wird die Perspektivtransformationsmatrix mittels der OpenCV-Funktion *cv2.getPerspectiveTransform()* berechnet. Diese Matrix ermöglicht die Transformation des erfassten Kamerabilds in eine orthogonale Draufsicht auf das Spielfeld mit definierten Abmessungen. Alle nachfolgenden Bildverarbeitungsoperationen arbeiten ausschließlich auf diesem normalisierten Spielfeldbereich.

6.4.2 Evaluation

Die Evaluation der Spielfelderkennung konzentriert sich auf die Bewertung der geometrischen Genauigkeit unter kontrollierten Bedingungen. Da die Spielfelderkennung als einmaliger Initialisierungsschritt ausgelegt ist und nicht kontinuierlich unter variablen Umgebungsbedingungen operieren muss, erfolgt die Bewertung unter standardisierten Testbedingungen mit konstanter Beleuchtung, fixierter Kameraposition und unverdecktem Spielfeld. Diese Herangehensweise ermöglicht eine präzise Quantifizierung der algorithmischen Leistungsfähigkeit ohne Störeinflüsse durch externe Variablen. Die Evaluation basiert auf einem Vergleich zwischen automatisch erkannten Spielfeldgrenzen und manuell erstellten Ground-Truth-Referenzdaten, wobei die tatsächlichen Spielfeldecken pixelgenau annotiert werden.

Als primäre Bewertungsmetrik wird die Intersection over Union (IoU) verwendet, die das Verhältnis der Überschneidungsfläche zur Vereinigungsfläche zwischen erkanntem und tatsächlichem Spielfeld berechnet. Diese flächenbasierte Metrik erfasst die Gesamtqualität der Spielfeldabgrenzung als einzelne Kennzahl und spiegelt direkt die Auswirkungen auf nachgelagerte Bildverarbeitungsoperationen wider. Ergänzend wird die Zentroid-Abweichung zwischen den Mittelpunkten der erkannten und tatsächlichen Spielfelder ermittelt. Mithilfe dieser Werte kann eine Aussage über die Anforderung PD-NFA6 (Modellübertragbarkeit) getroffen werden.

Zur Bewertung der geometrischen Genauigkeit wurden zehn Testbilder verwendet. Die in der Tabelle 6.1 Ergebnisse zeigen eine durchschnittliche IoU von etwa 0.98, was auf eine nahezu vollständige Überdeckung der automatisch erkannten Spielfeldflächen mit

den Referenzdaten hinweist. Damit bestätigt sich, dass die Flächenabgrenzung des Spielfeldes durch den Algorithmus äußerst zuverlässig gelingt und kaum Abweichungen zur manuellen Annotation bestehen.

Tabelle 6.1: Evaluation der Spielfelderkennung über zehn Testbilder

Bild-Nr.	**IoU**	**Zentroid-Abweichung [px]**
1	0.981	4.8
2	0.975	5.2
3	0.979	5.6
4	0.983	4.9
5	0.976	5.4
6	0.982	4.7
7	0.977	5.3
8	0.980	4.6
9	0.984	5.1
10	0.978	5.0

Ergänzend wurde die Zentroid-Abweichung zwischen den Mittelpunkten der erkannten und tatsächlichen Spielfelder berechnet. Mit einem Mittelwert von etwa fünf Pixeln bleibt dieser Abstand deutlich innerhalb der Dimensionen des Spielfelds und ist für nachgelagerte Bildverarbeitungsaufgaben vernachlässigbar. Die Zentroid-Analyse belegt damit, dass die erkannten Spielfelder nicht nur in ihrer Form, sondern auch in ihrer Positionierung hochpräzise sind.

Zusammenfassend bestätigt die Evaluation, dass die implementierte Spielfelderkennung eine robuste und exakte Grundlage für die perspektivische Normalisierung des Spielfelds liefert. Die ermittelten Werte verdeutlichen, dass sowohl Flächenkongruenz als auch Lagegenauigkeit die Anforderungen an eine konsistente und stabile Verarbeitung erfüllen. Außerdem zeigt die Zentroid-Abweichung, dass die Anforderung PD-NFA6 (Modellübertragbarkeit) an dieser Stelle nicht verletzt wird. Eine Abweichung von fünf Pixeln sollte kein Problem bei der Übertragung des RL-Agenten auf die physische Umgebung darstellen, da die Kugelauflösung bei etwa 60 Pixeln (Durchmesser) liegt und die Verschiebung hierbei vernachlässigbar klein sein sollte.

6.5 Kugelerkennung

Die Kugelerkennung ermöglicht die kontinuierliche Verfolgung der Ballposition und stellt damit eine zentrale Komponente für die Zustandsrepräsentation des Systems dar. In diesem Abschnitt wird zunächst die Funktionsweise des Erkennungsalgorithmus beschrieben. Anschließend erfolgt die Evaluation.

6.5.1 Funktionsweise

Die Kugelerkennung stellt eine zentrale Komponente des Bildverarbeitungssystems dar, da sie die kontinuierliche Verfolgung der Kugelposition in Echtzeit ermöglicht und damit die Grundlage für die Entscheidungsfindung des RL-Agenten bildet. Die robuste und präzise Detektion der sich schnell bewegenden Stahlkugel auf dem schwarzen Spielfeld erfordert einen spezialisierten Algorithmus, der sowohl bewegungsbasierte als auch formanalytische Kriterien kombiniert.

Die Implementierung der Kugelerkennung erfolgt durch die spezialisierte Klasse BallDetector, deren algorithmischer Ablauf mittels eines Aktivitätsdiagramms in Abbildung 6.7 dargestellt ist. Das erhaltene Bild wird zunächst in ein Grauwertbild umgewandelt, geglättet (siehe Kapitel 2.1.2) und anschließend wird mithilfe von CLAHE (siehe Kapitel 2.1.2) der Kontrast verbessert. Das System implementiert einen bewegungsbasierten Ansatz mit Fallback-Mechanismus, der sowohl die Detektion bewegter als auch stationärer Kugeln gewährleistet. Die Differenzbildberechnung bildet das Herzstück der Kugelerkennung, da sie gezielt bewegte Objekte vom statischen Hintergrund separiert. Diese Methode ist besonders geeignet für die Flipper-Anwendung, da sich die Kugel typischerweise mit hohen Geschwindigkeiten bewegt und somit deutliche Intensitätsunterschiede zwischen aufeinanderfolgenden Bildern erzeugt. Durch die Subtraktion des Referenzbildes vom aktuellen Bild werden ausschließlich Bildregionen mit zeitlichen Veränderungen hervorgehoben, wodurch statische Spielelemente wie Banden, Hindernisse und Flipperarme in Ruheposition effektiv ausgeblendet werden.

Die nachfolgende Schwellwertbildung konvertiert das Differenzbild in ein binäres Format (Schwarz-Weiß-Bild), gefolgt von einer morphologischen Opening-Operation, die kleine Störungen entfernt und die Objektkonturen glättet. Die Konturenerkennung identifiziert zusammenhängende Regionen, die anschließend anhand von Zirkularität, Seitenverhältnis und Intensitätswerten bewertet werden. Bei erfolgreicher Konturenerkennung wird

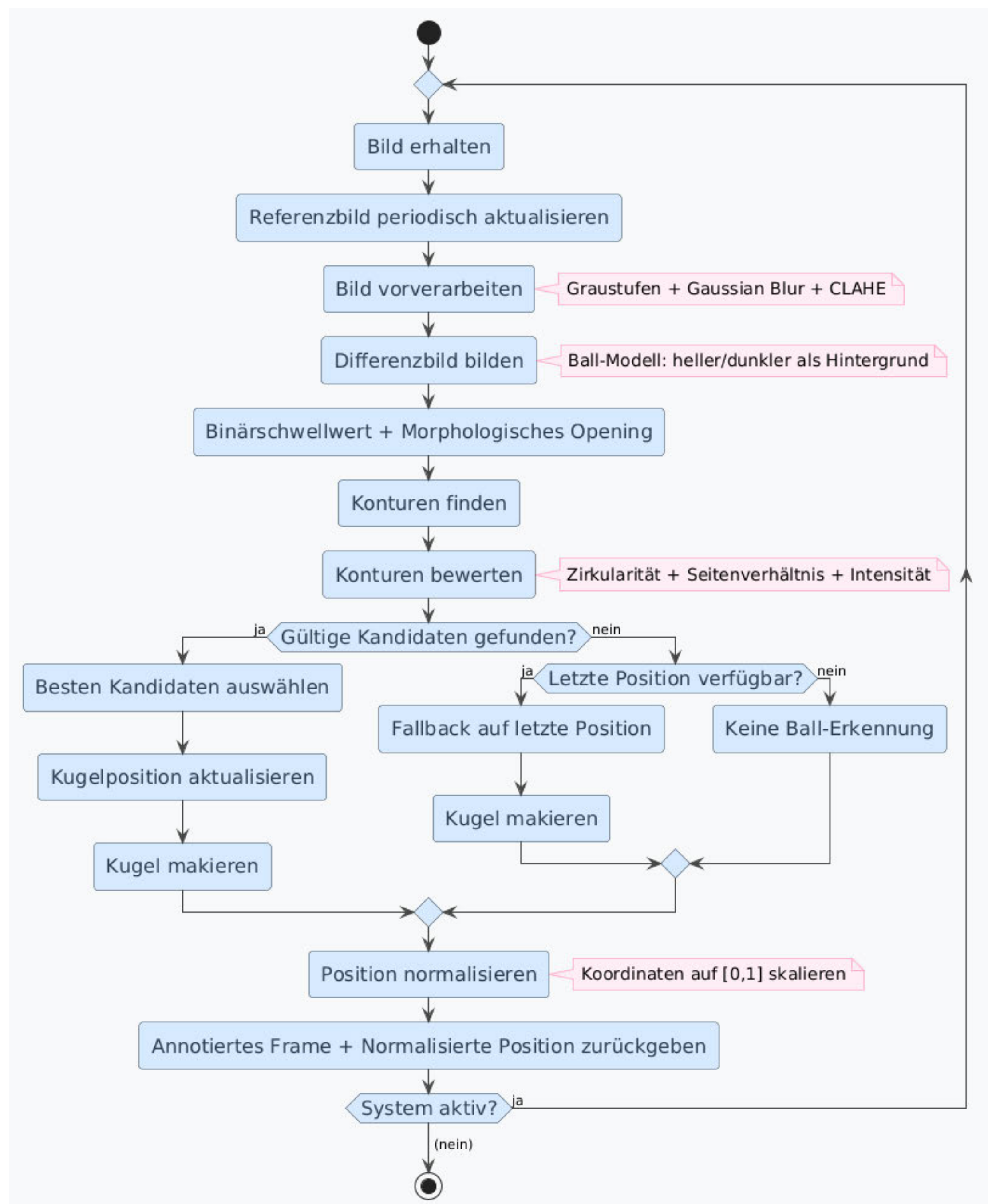


Abbildung 6.7: Vereinfachtes Aktivitätsdiagramm der BallDetector-Klasse

der beste Kandidat ausgewählt und die Kugelposition aktualisiert. Falls keine gültigen Kandidaten gefunden werden, greift ein Fallback-Mechanismus auf die letzte bekannte

Position zurück, sofern diese verfügbar ist. Abschließend erfolgt eine Positionsnormalisierung auf den Wertebereich von null bis eins, bevor das annotierte Bild zusammen mit der normalisierten Position zurückgegeben wird.

6.5.2 Evaluation

Die Evaluation der Kugelerkennung ist essentiell für die Bewertung der Gesamtsystemleistung, da präzise Kugelpositionsdaten die Grundlage für erfolgreiche RL-Agent-Entscheidungen bilden. Eine fehlerhafte oder unzuverlässige Kugeldetektion würde direkt die Trainingsqualität und Performance des autonomen Flippersystems beeinträchtigen. Bei der Evaluation müssen insbesondere die Anforderungen PD-NFA3 (Echtzeitfähigkeit) und PD-NFA6 (Modellübertragbarkeit) berücksichtigt werden. Die Echtzeitfähigkeit erfordert, dass die Kugelerkennung innerhalb der 50 ms Gesamtlatenz funktioniert, während die Modellübertragbarkeit voraussetzt, dass Erkennungsungenauigkeiten nicht zu systematischen Abweichungen zwischen virtueller Trainingsumgebung und physischem System führen.

Die Evaluation basiert auf einem Ground-Truth-Dataset, das durch manuelle Aufzeichnung und Annotation von 50 repräsentativen Frames erstellt wird. Die Frames werden in einem Intervall von 0,04 Sekunden (25 FPS) aufgenommen und zeigen verschiedene Kugelgeschwindigkeiten und -positionen. Jedes Frame wird manuell annotiert, indem die tatsächliche Kugelposition genau markiert wird. Als Bewertungskriterium für eine korrekte Detektion wird definiert, dass der Mittelpunkt der automatisch erkannten Kugel innerhalb der Grenzen der tatsächlichen Kugel liegen muss. Dieses geometrische Kriterium stellt sicher, dass nur ausreichend präzise Detektionen als erfolgreich gewertet werden.

Die quantitative Bewertung erfolgt anhand von drei Standardmetriken der Objekterkennung. Die True-Positive-Rate berechnet sich als Verhältnis korrekt erkannter Kugeln zu allen tatsächlich vorhandenen Kugeln und quantifiziert die Erkennungsgenauigkeit des Systems. Die False-Positive-Rate ermittelt das Verhältnis fälschlicherweise als Kugel klassifizierter Objekte zu allen Detektionen und bewertet die Neigung des Systems zu Fehlalarmen. Die False-Negative-Rate bestimmt das Verhältnis übersehener Kugeln zu allen vorhandenen Kugeln und charakterisiert die Vollständigkeit der Erkennung. Diese Metriken ermöglichen eine umfassende Bewertung sowohl der Präzision als auch der

Zuverlässigkeit der implementierten Kugelerkennungsalgorithmen unter realistischen Betriebsbedingungen.

Zur objektiven Bewertung der Kugelerkennungsleistung werden spezifische Schwellenwerte für jede Metrik definiert. Für die True-Positive-Rate wird ein Mindestwert von 95% als erforderlich erachtet. Diese Anforderung ergibt sich aus der kritischen Rolle der Kugelerkennung für die RL-Agent-Performance: Fehlende Kugeldetektionen führen zu unvollständigen Zustandsinformationen und damit zu suboptimalen Agenten-Entscheidungen. Bei einer Erkennungsrate unter 95% würde jeder zwanzigste Frame ohne gültige Kugelposition verarbeitet, was bei der hohen Dynamik des Flipperspiels zu erheblichen Performanceeinbußen führen könnte.

Die False-Positive-Rate sollte 2% nicht überschreiten, da Fehldetektionen den RL-Agenten mit falschen Zustandsinformationen versorgen und dadurch das Lernverhalten negativ beeinflussen können. Phantom-Kugeln an inkorrekten Positionen würden systematische Verzerrungen in der Strategieentwicklung verursachen. Die False-Negative-Rate wird analog zur True-Positive-Rate mit maximal 5% toleriert, wobei dieser Grenzwert die akzeptable Obergrenze für übersehene Kugeln definiert, ohne die Echtzeitfähigkeit des Systems zu gefährden. Die quantitative Auswertung der Kugelerkennung basiert auf der systematischen Klassifikation der 50 Test-Frames und ist in Tabelle 6.2 zusammengefasst.

Tabelle 6.2: Evaluation der Kugelerkennung über 50 Testframes

Klassifikation	**Frames**
True Positives (TP)	1–20, 22–35, 37–50 (48 Frames)
False Positives (FP)	Keine (0 Frames)
False Negatives (FN)	21, 36 (2 Frames)

Metrik	**Wert**
True-Positive-Rate	96.0%
False-Positive-Rate	0.0%
False-Negative-Rate	4.0%

Die experimentellen Ergebnisse demonstrieren eine hervorragende Erkennungsleistung, die alle definierten Schwellenwerte übertrifft. Mit einer True-Positive-Rate von 96% wird die geforderte Mindesterkennungsrate von 95% erreicht und um einen Prozentpunkt überschritten. Die Abwesenheit jeglicher False-Positive-Detektionen unterstreicht die hohe Spezifität des Algorithmus und eliminiert das Risiko systematischer Fehlführung des RL-Agenten durch Phantom-Objekte. Die False-Negative-Rate von 4% liegt unter dem Toleranzgrenzwert von 5% und betrifft lediglich zwei von 50 Frames. Wichtig ist zu erwähnen,

dass die Kugelerkennung an dieser Stelle bei Stillstand der Flipperarme getestet wurde und daher einige Testszenarien nicht abgedeckt sind. Bei der späteren Integration der Systemkomponenten könnten weitere Probleme auftreten, die an dieser Stelle nicht testbar sind.

Hinsichtlich der kritischen Systemanforderungen zeigt die Evaluation eine vollständige Konformität. Die Echtzeitfähigkeit (PD-NFA3) wird durch die geringe Anzahl fehlgeschlagener Detektionen nicht gefährdet. Die Modellübertragbarkeit (PD-NFA6) profitiert von der hohen Erkennungsgenauigkeit, da konsistente Kugelpositionsdaten zwischen virtueller Trainingsumgebung und physischem System eine wesentliche Voraussetzung für erfolgreichen Transfer darstellen. Die systematische Analyse bestätigt, dass keine Gefahr einer Anforderungsverletzung besteht.

6.6 Flipperarm-Erkennung

Die Flipperarm-Erkennung ermöglicht die Bestimmung der aktuellen Winkelstellung beider Flipperarme und vervollständigt damit die Zustandserfassung des Systems. In diesem Abschnitt wird zunächst die Funktionsweise des Erkennungsalgorithmus beschrieben. Anschließend erfolgt die Evaluation der Erkennungsqualität.

6.6.1 Funktionsweise

Die Flipperarm-Erkennung stellt eine spezialisierte Komponente des Bildverarbeitungssystems dar, die für die kontinuierliche Bestimmung der Orientierung beider Flipperarme verantwortlich ist. Diese Information ist essenziell für den RL-Agenten, da die aktuelle Armstellung sowohl für die Zustandsrepräsentation als auch für die Aktionsplanung relevant ist.

Die Implementierung der Flipperarm-Erkennung erfolgt durch die spezialisierte Klasse ArmDetector, deren algorithmischer Ablauf mittels eines Aktivitätsdiagramms in Abbildung 6.8 dargestellt ist. Das System implementiert einen kontinuierlichen Erkennungsansatz, der die Orientierung beider Flipperarme quasi-parallel verarbeitet und deren Winkelstellung in Echtzeit bestimmt. Der Erkennungsprozess beginnt mit der Definition von Regions of Interest (ROI), die auf die spezifischen Bereiche der Flipperarme fokussiert

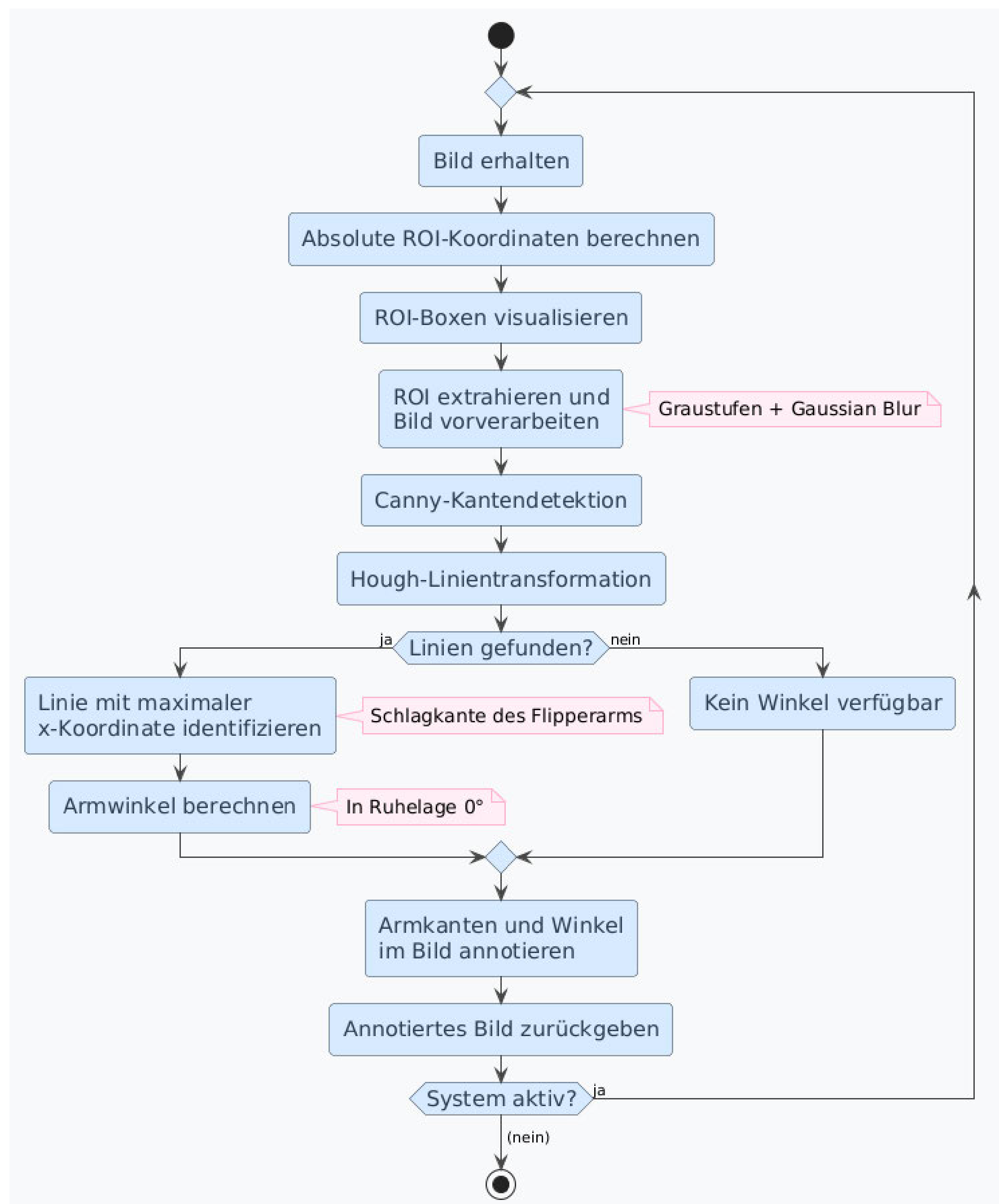


Abbildung 6.8: Vereinfachtes Aktivitätsdiagramm der ArmDetector-Klasse

sind. Diese räumliche Beschränkung reduziert sowohl die Rechenzeit als auch die Wahrscheinlichkeit von Fehldetektionen durch irrelevante Spielelemente. Die normalisierten

ROI-Koordinaten werden zunächst in absolute Pixel-Koordinaten transformiert, bevor die Erkennungsbereiche zur Visualisierung im Hauptbild eingezeichnet werden.

Die Kantendetektion erfolgt in zwei sequenziellen Schritten. Zunächst wird eine Canny-Kantendetektion angewendet, die charakteristische Helligkeitsübergänge entlang der Flipperarm-Konturen identifiziert. Diese Methode liefert jedoch nur einzelne Kantenpixel ohne strukturelle Informationen über zusammenhängende Linien. Die nachfolgende Hough-Linien-transformation (siehe Kapitel 2.1.4) interpretiert diese Kantenpixel als gerade Linienabschnitte und ermöglicht damit die Rekonstruktion der linearen Flipperarm-Geometrie.

Für die Winkelbestimmung wird gezielt die Linie mit der maximalen x-Koordinate identifiziert, die der Schlagkante des Flipperarms entspricht. Diese Kante ist funktional relevant, da sie die primäre Kontaktfläche zur Kugelsteuerung darstellt. Die Winkelberechnung erfolgt relativ zur Ruhelage des Arms, wobei 0° der horizontalen Ausgangsposition entspricht. Das System annotiert abschließend sowohl die erkannten Armkanten als auch die berechneten Winkelwerte im Hauptbild, wodurch eine visuelle Rückmeldung über die Erkennungsqualität ermöglicht wird.

6.6.2 Evaluation

Die Evaluation der Flipperarm-Erkennung fokussiert sich auf die korrekte Identifikation der Schlagkanten beider Flipperarme. Zur Bewertung wurden während der Armbewegungen insgesamt 30 Testbilder aufgenommen, die unterschiedliche Armstellungen von der Ruheposition bis zur vollständigen Aktivierung abdecken. Jedes Bild wurde manuell analysiert, um zu überprüfen, ob die durch den Algorithmus erkannten Kanten auf der Schlagkante des jeweiligen Flipperarms liegen.

Die qualitative Auswertung zeigt, dass die Hough-Linien-Transformation in allen 30 Testbildern die korrekten Schlagkanten beider Arme identifiziert. Die implementierte Strategie, gezielt die rechteste erkannte Linie innerhalb der definierten Regions of Interest auszuwählen, erweist sich als robust. Sowohl der linke als auch der rechte Flipperarm werden konsistent erkannt.

Die erfolgreiche Kantenerkennung bildet damit eine solide Grundlage für die nachfolgende Winkelberechnung und erfüllt die Anforderungen an die Zustandsrepräsentation für den RL-Agenten. Die visuelle Validierung bestätigt, dass die Arm-Detektion unter realistischen Betriebsbedingungen zuverlässig funktioniert.

7 Entwicklung der virtuellen Trainingsumgebung

Dieses Kapitel beschreibt die Entwicklung der virtuellen Trainingsumgebung als digitaler Zwilling des physischen Flipperautomaten. Zunächst wird die Softwarearchitektur mit der Klassenstruktur dargestellt. Anschließend werden das Geometriemodell und die Physik-Engine erläutert. Das Kollisionssystem wird detailliert beschrieben, bevor die Integration als Gymnasium-Environment erfolgt. Die Environment-Wrapper zur Vorverarbeitung der Trainingsdaten sowie die Visualisierungskomponente werden daraufhin vorgestellt. Abschließend werden die Trainingskonfiguration und die Evaluation der implementierten Umgebung anhand der definierten Anforderungen aufgezeigt.

7.1 Softwarearchitektur

Die Implementierung der virtuellen Trainingsumgebung basiert auf einer modularen, objektorientierten Architektur, die eine klare Trennung der Verantwortlichkeiten zwischen Datenstrukturen, Simulationslogik, Visualisierung und Training ermöglicht. Abbildung 7.1 zeigt das vereinfachte Klassendiagramm mit den wichtigsten Klassen und ihren Beziehungen.

Die Klassen PinballGeometry und State bilden die Datengrundlage des Systems. PinballGeometry kapselt alle geometrischen Parameter des Spielfelds als unveränderliche Datenstruktur, während State den aktuellen Spielzustand mit Flipper-Winkeln, Kugelposition und -geschwindigkeit repräsentiert. Die Klasse Pinball hält beide Datenklassen durch Kompositionsbeziehungen und orchestriert die gesamte Physiksimulation. PinballCollision wird ebenfalls als Komposition von Pinball gehalten und behandelt Kollisionen zwischen Kugel und statischen Spielfeldelementen. PinballCollision greift über eine Abhängigkeitsbeziehung auf PinballGeometry zu, um geometrische Parameter für die Kollisionsberechnungen zu erhalten. Die Klasse PinballRender3D visualisiert das Spielfeld in

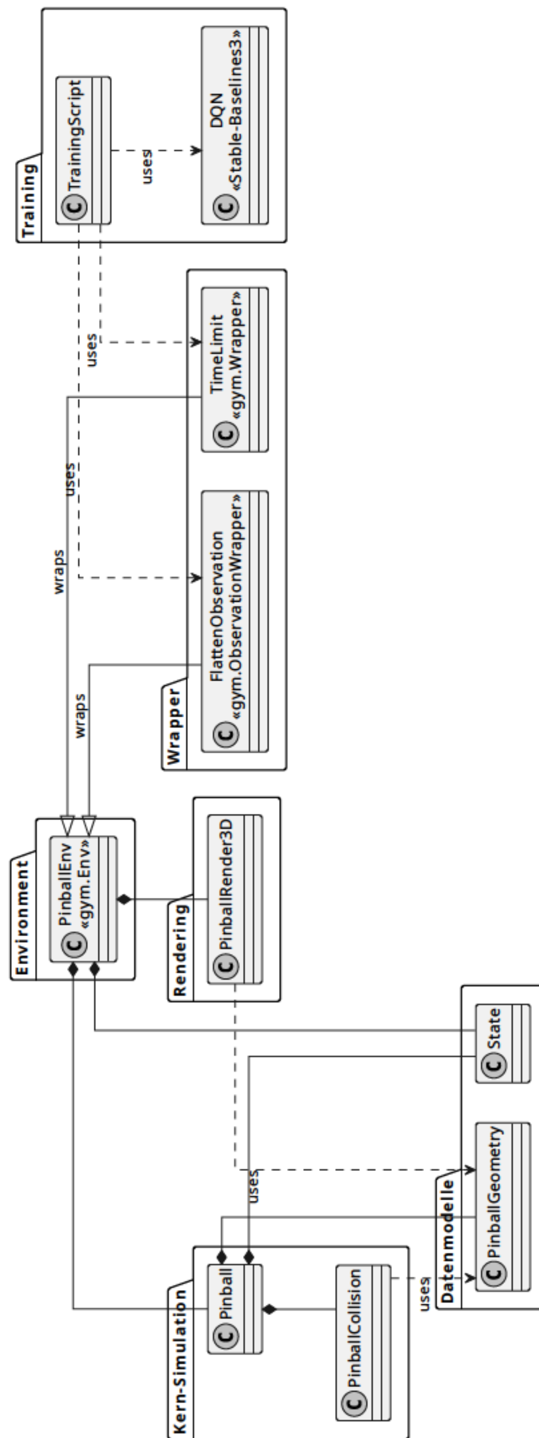


Abbildung 7.1: Klassendiagramm der virtuellen Trainingsumgebung

3D und nutzt PinballGeometry über eine Abhängigkeitsbeziehung zur Konstruktion der 3D-Szene.

Die Klasse PinballEnv implementiert die Gymnasium-Schnittstelle und bildet das zentrale Environment für das Reinforcement Learning. Sie hält Pinball, State und PinballRender3D als Kompositionen und definiert den Observation Space, Action Space sowie die Reward-Funktion. PinballEnv wird von zwei Wrapper-Klassen umschlossen, die im Diagramm durch gestrichelte Linien mit dem Stereotyp *wraps* dargestellt sind. FlattenObservation transformiert die Dictionary-Observation in einen flachen Vektor für die Kompatibilität mit DQN, während TimeLimit die maximale Episoden-Länge begrenzt. Die Klasse TrainingScript orchestriert den gesamten Trainingsprozess und nutzt die gewrappten Environments über Abhängigkeitsbeziehungen. TrainingScript instanziiert und konfiguriert den DQN-Algorithmus aus Stable-Baselines3, was ebenfalls durch eine gestrichelte Abhängigkeitslinie repräsentiert wird. Der Datenfluss verläuft hierarchisch von den Datenstrukturen über die Simulation zum Environment und schließlich zum Training, wobei durchgezogene Linien starke Kopplungen durch Komposition oder direkte Nutzung und gestrichelte Linien schwache Kopplungen durch Abhängigkeiten oder Wrapper-Beziehungen kennzeichnen.

7.2 Geometriemodell

Das Geometriemodell bildet die strukturelle Grundlage der Physiksimulation und definiert alle räumlichen Parameter des Spielfelds sowie der dynamischen Elemente. Die Klasse PinballGeometry kapselt sämtliche geometrischen Informationen in einer unveränderlichen Datenstruktur, die von mehreren Komponenten des Systems referenziert wird. Diese zentrale Verwaltung der Geometrieparameter gewährleistet Konsistenz zwischen Simulation, Kollisionserkennung und Visualisierung. Abbildung 7.2 zeigt die schematische Darstellung des Spielfelds mit allen relevanten geometrischen Komponenten.

Die Klasse PinballGeometry definiert das Spielfeld durch einen rechteckigen Rahmen (Frame) mit Parametern für die Länge und Breite, die die äußeren Grenzen des Spielbereichs festlegen. Das Koordinatensystem ist so definiert, dass der Ursprung in der Mitte des Spielfelds liegt, die x-Achse von links nach rechts verläuft und die y-Achse von unten nach oben zeigt. Die oberen Ecken des Spielfelds werden durch zwei Bögen (TopCorners) geformt, die jeweils durch Radius, Mittelpunkt und Winkelbereich charakterisiert sind. Die linke Seitenwand verfügt über eine horizontale Verbindung (LeftConnection) zum

Abbildung 7.2: 2D-Darstellung der Flippergeometrie

Bogen, während die rechte Seite die Startbahn (LauncherLane) mit definierter Breite und Länge beherbergt. Die beiden Flipper werden durch ihre Rotationspunkte relativ zum Spielfeldrahmen, Größe, Dicke sowie maximalen Öffnungswinkel (slope) spezifiziert. Die Kugel wird durch Radius und Masse charakterisiert, wobei der Radius für Kollisionsberechnungen und die Masse für die Impulsdynamik relevant sind. Alle geometrischen Parameter werden in Zentimetern gespeichert, was eine physikalisch konsistente Berechnung mit realistischen Größenverhältnissen ermöglicht.

7.3 Physik-Engine

Die Physik-Engine bildet das Kernstück der Simulation und implementiert die realitätsnahe Bewegungsdynamik der Kugel sowie die Interaktion mit den Flippern. Die Klasse Pinball orchestriert alle physikalischen Berechnungen und integriert die Kollisionserkennung für statische Objekte von PinballCollision. Die Simulation basiert auf einem deterministischen Zeitschrittverfahren mit einer festen Schrittweite von $\Delta t = 0{,}01\,\mathrm{s}$. Die physikalische Modellierung der Kugeldynamik und Kollisionen orientiert sich an den Grundprinzipien der klassischen Mechanik und den Verfahren der Spielephysik nach [7].

7.3.1 Physikalisches Modell der Kugelbewegung

Die Bewegung der Kugel wird als rollende Vollkugel auf einer geneigten Ebene modelliert, wobei das Spielfeld um einen Winkel θ zur Horizontalen geneigt ist. Für eine rollende Kugel ergibt sich aufgrund des Trägheitsmoments $I = \frac{2}{5}mr^2$ (mit Masse m und Radius r) eine reduzierte Beschleunigung gegenüber der reinen Gleitbewegung. Die effektive Beschleunigung beträgt $a_{\mathrm{eff}} = \frac{5}{7}\,\mathrm{g}$, wobei $g = 9{,}81\,\mathrm{m/s^2}$ die Erdbeschleunigung bezeichnet. In y-Richtung, entlang der Hauptneigung, wirken die Hangabtriebskraft und die Reibungskraft mit dem Reibungskoeffizienten $\mu = 0{,}00118$. Die resultierende Beschleunigung lautet für abwärts gerichtete Bewegung

$$a_y = -\frac{5}{7}\,\mathrm{g}\,(\sin\theta - \mu\cos\theta). \tag{7.1}$$

Bei aufwärts gerichteter Bewegung kehrt sich das Vorzeichen der Reibung um. In x-Richtung, senkrecht zur Hauptneigung, wirkt ausschließlich Reibung. Die zeitliche Integration erfolgt nach dem Euler-Verfahren mit der kinematischen Gleichung

$$\vec{s}(t + \Delta t) = \vec{s}(t) + \vec{v}(t) \cdot \Delta t + \frac{1}{2}\vec{a} \cdot \Delta t^2 \tag{7.2}$$

wobei $\vec{s}$ die Position, $\vec{v}$ die Geschwindigkeit und $\vec{a}$ die Beschleunigung repräsentieren. Eine Sonderbehandlung verhindert numerische Oszillationen beim Stillstand durch Nullsetzen der Geschwindigkeit bei Vorzeichenwechsel.

7.3.2 Flipperdynamik und Kollisionserkennung

Die Flipper bewegen sich mit konstanter Winkelgeschwindigkeit zwischen zwei Extrempositionen. Jeder Flipper durchläuft drei diskrete Zustände: aktiviert, rückkehrend und in Ruhe. Die Kollisionserkennung erfolgt analytisch durch Projektion der Kugelposition auf das Flippersegment. Der Parameter $t \in [0, 1]$ beschreibt die Position des nächstgelegenen Punktes auf dem Flipper über

$$t = \frac{(\vec{p}_{\text{ball}} - \vec{p}_{\text{pivot}}) \cdot \vec{d}_{\text{flipper}}}{l^2_{\text{flipper}}} \tag{7.3}$$

Hierbei bezeichnet $\vec{p}_{\text{ball}}$ die Kugelposition, $\vec{p}_{\text{pivot}}$ den Rotationspunkt des Flippers, $\vec{d}_{\text{flipper}}$ den normierten Richtungsvektor des Flippers und l_{flipper} die Flipperlänge. Eine Kollision liegt vor, wenn der euklidische Abstand zwischen dem projizierten Punkt und der Kugelposition die Summe aus Kugelradius r_{ball} und halber Flipperdicke $d_{\text{flipper}}/2$ unterschreitet. Bei detektierter Kollision wird die Geschwindigkeit des Flippers am Kollisionspunkt $\vec{v}_{\text{flipper}} = \omega \cdot r_{col} \cdot \vec{t}$ berechnet, wobei $r_{col} = t \cdot l_{\text{flipper}}$ der Abstand vom Pivot-Punkt und $\vec{t}$ der Tangentialvektor ist. Die Reflexion basiert auf der Relativgeschwindigkeit $\vec{v}_{\text{rel}} = \vec{v}_{\text{ball}} - \vec{v}_{\text{flipper}}$ und einem geschwindigkeitsabhängigen Restitutionskoeffizienten $e = 0{,}5 + 0{,}2 \cdot \frac{|\omega|}{\omega_{\max}}$, wodurch aktivierte Flipper stärkere Impulse übertragen. Der resultierende Geschwindigkeitsimpuls lautet

$$\Delta\vec{v} = -(1 + e)\,(\vec{v}_{\text{rel}} \cdot \vec{n})\,\vec{n} \tag{7.4}$$

wobei $\vec{n}$ den Normalenvektor vom Flipper zur Kugel bezeichnet. Die Kugelposition wird anschließend um die Penetrationstiefe plus einen Sicherheitsabstand verschoben.

7.3.3 Ablauf eines Simulationsschritts

Die step()-Methode implementiert einen vollständigen Physikschritt in einer definierten Sequenz. Abbildung 7.3 zeigt die vollständige Implementierung dieser zentralen Methode.

```python
def step(self, state, time_sec=None, activate_left=False, activate_right=False):
    """
    Perform one physics time step including flipper movement and collision.

    Parameters
    ----------
    state : State
        Current ball state (will be modified).
    time_sec : float, optional
        Time step in [s]. If None, uses self.dt.
    activate_left : bool, optional
        Activate left flipper. Default is False.
    activate_right : bool, optional
        Activate right flipper. Default is False.
    """
    dt = time_sec if time_sec is not None else self.dt

    # Handle flipper activation
    if activate_left or activate_right:
        self.activate_flipper(state, left=activate_left, right=activate_right)

    # Update flipper positions
    self._update_flippers(state, dt)

    # Store old position for collision detection
    old_pos = state.position.copy()

    # Execute physics step
    self._calculate_motion(state, dt)

    # Check for flipper collision first (analytical)
    flipper_collision = self._check_flipper_collision(state, dt)

    if flipper_collision:
        state.velocity = state.velocity * 1

    # Only check static collisions if no flipper collision occurred
    if not flipper_collision:
        collision_info, collision_pos = self.collision.check_trajectory(
            old_pos, state.position
        )

        if collision_info:
            # Resolve collision
            state.position, state.velocity = self.collision.resolve_collision(
                collision_pos if collision_pos is not None else state.position,
                state.velocity,
                collision_info,
                dt
            )
            state.velocity = state.velocity
```

Abbildung 7.3: step()-Methode der Pinball.py

Die Methode verarbeitet zunächst eingehende Flipper-Aktivierungen durch *activate_flipper()* und aktualisiert die Flipper-Positionen mittels *_update_flippers()*. Die alte Kugelposition wird vor der Bewegungsberechnung gespeichert, um anschließend eine trajektorienbasierte Kollisionsprüfung zu ermöglichen. Die Berechnung der Kugelbewegung unter Berücksichtigung von Gravitation und Reibung erfolgt durch *_calculate_motion()*. Die Kollisionsprüfung ist hierarchisch organisiert. Flipper-Kollisionen werden durch

check_flipper_collision() priorisiert, da sie dynamische Objekte betreffen und eine präzise analytische Behandlung ermöglichen. Statische Kollisionen mit Wänden und Bögen werden nur bei Abwesenheit einer Flipper-Kollision geprüft, um Doppelbehandlungen zu vermeiden. Die Trajektorienprüfung durch *check_trajectory()* detektiert auch Hochgeschwindigkeitskollisionen mittels des Bresenham-Algorithmus, bei denen die Kugel in einem Zeitschritt mehrere Kollisionspixel durchquert. Die Methode *resolve_collision()* berechnet die reflektierte Geschwindigkeit unter Berücksichtigung materialspezifischer Restitutionskoeffizienten und verschiebt die Kugelposition aus der Kollisionszone.

7.4 Kollisionssystem

Die Kollisionserkennung für statische Spielfeldelemente stellt eine kritische Komponente der Physiksimulation dar, da sie sowohl Echtzeitfähigkeit als auch hohe Präzision gewährleisten muss. Die Klasse PinballCollision ist ausschließlich für die Kollisionserkennung mit unveränderlichen Geometrieelementen wie Wänden, Bögen und statischen Flipper-Boards zuständig, während dynamische Flipper-Kollisionen analytisch in der Klasse Pinball behandelt werden.

7.4.1 Vergleich von Kollisionserkennungsverfahren

Für die Kollisionserkennung in zweidimensionalen Spielumgebungen existieren mehrere etablierte Verfahren. Die analytische Kollisionserkennung berechnet für jede Spielfeldkomponente explizit den Abstand zur Kugelposition und prüft geometrische Bedingungen. Dieser Ansatz bietet theoretisch perfekte Präzision, erfordert jedoch für jedes Objekt individuelle Berechnungsroutinen und skaliert schlecht mit der Anzahl der Spielfeldelemente. Die Bounding-Box-Methode approximiert komplexe Geometrien durch rechteckige oder kreisförmige Hüllen und ermöglicht effiziente Kollisionsprüfungen durch einfache Abstandsberechnungen. Die Approximation führt jedoch zu ungenauen Kollisionen, insbesondere bei schrägen Wänden und Bögen. Physik-Engines wie Box2D oder PyMunk bieten vollständige Kollisionssysteme mit fortgeschrittenen Features, bringen jedoch erheblichen Overhead und Abhängigkeiten mit sich, die für eine kontrollierte Simulationsumgebung unerwünscht sind. Der pixelbasierte Ansatz diskretisiert das Spielfeld in

eine Kollisionskarte, wobei jedes Pixel Kollisionsinformationen speichert. Diese Methode kombiniert hohe Präzision mit konstantem Zeitaufwand für Kollisionsabfragen und ermöglicht eine einheitliche Behandlung aller Geometrietypen.

Die Implementierung verwendet einen pixelbasierten Ansatz mit Sparse-Storage-Optimierung, da dieser die spezifischen Anforderungen der Pinball-Simulation optimal erfüllt. Die konstante Zeitkomplexität für Punktabfragen gewährleistet deterministisches Echtzeitverhalten unabhängig von der Spielfeldkomplexität. Die einheitliche Datenstruktur eliminiert die Notwendigkeit geometriespezifischer Kollisionsroutinen und vereinfacht die Wartbarkeit. Durch Sparse-Storage werden ausschließlich Kollisionspixel gespeichert, wodurch der Speicheraufwand trotz feiner Diskretisierung akzeptabel bleibt. Die Pixelgröße von 0,1 cm bietet millimetergenaue Auflösung bei moderatem Speicherbedarf.

7.4.2 Datenstrukturen

Die Klasse PinballCollision implementiert das Kollisionssystem durch drei Kernkomponenten. Das Enum CollisionType klassifiziert neun verschiedene Kollisionsobjekte (Wände, Bögen, Flipper-Boards, Launcher-Lane), wodurch unterschiedliche physikalische Eigenschaften pro Objekttyp definierbar sind. Die Datenklasse CollisionInfo kapselt für jeden Kollisionspixel den Kollisionstyp, den Normalenvektor $\vec{n}$ sowie den Restitutionskoeffizienten e. Die zentrale Datenstruktur ist ein Dictionary der Form *collision_map: Dict[Tuple[int, int], CollisionInfo]*, das Pixelkoordinaten auf Kollisionsinformationen abbildet. Diese Sparse-Storage-Implementierung speichert ausschließlich Kollisionspixel, während der Großteil des leeren Spielfelds keinen Speicher beansprucht.

7.4.3 Kollisionskarte

Die statische Kollisionskarte wird während der Initialisierung konstruiert und bleibt anschließend unveränderlich. Abbildung 7.4 zeigt die visualisierte Collision-Map mit farbcodierter Darstellung aller Kollisionstypen.

Die Koordinatentransformation erfolgt durch die Hilfsfunktion *_world_to_map()*, die Weltkoordinaten (x, y) in diskrete Pixelindizes über

$$\text{map}_x = \left\lfloor \frac{x + \text{offset}_x}{\text{pixel_size}} \right\rfloor, \quad \text{map}_y = \left\lfloor \frac{y + \text{offset}_y}{\text{pixel_size}} \right\rfloor \tag{7.5}$$

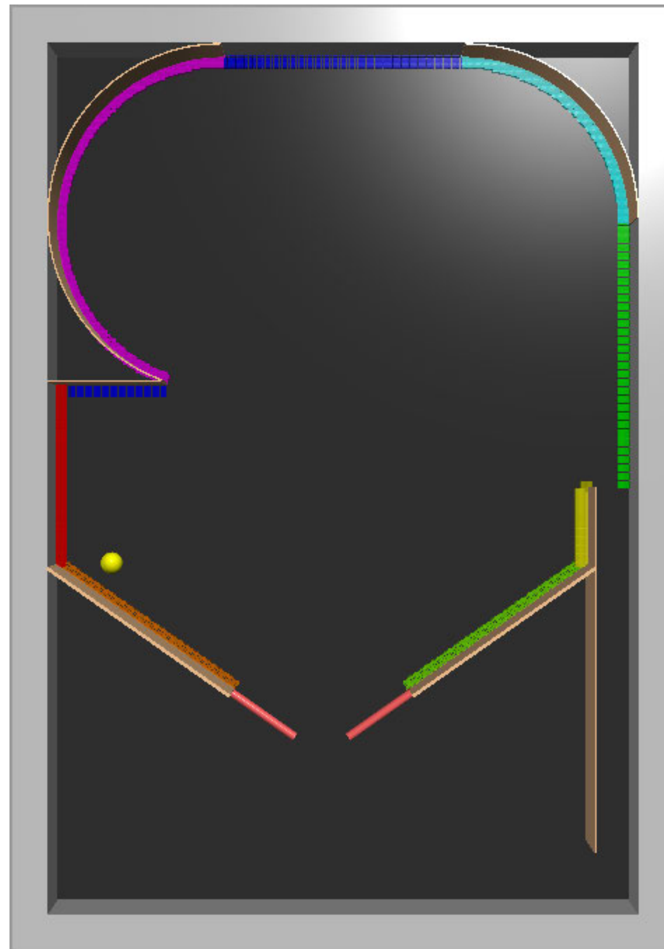


Abbildung 7.4: Statische Kollisionskarte mit farbig markierten Kollisionspixeln

umrechnet, wobei die Offsets die Verschiebung vom weltkoordinatenzentrierten System in ein positiv-indiziertes Pixelarray kompensieren. Das zentrale Konstruktionsprinzip basiert auf einem mehrschichtigen Layer-System. Anstatt ausschließlich die geometrische Oberfläche zu diskretisieren, werden mehrere parallele Schichten an Kollisionspixeln generiert, die sich über eine Distanz entsprechend dem Kugelradius erstrecken. Dieses Layer-System ermöglicht frühzeitige Kollisionserkennung, noch bevor der Kugelmittelpunkt die geometrische Oberfläche erreicht, und approximiert damit die räumliche Ausdehnung der Kugel.

Für die oberen Bögen wird eine winkelbasierte Diskretisierung mit anschließender radialer Layer-Generierung verwendet. Die statischen Flipper-Boards, die die V-förmige Struktur unterhalb der beweglichen Flipper bilden, werden entlang der geneigten Linie

```
# Bresenham's line algorithm
dx = abs(x1 - x0)
dy = abs(y1 - y0)
sx = 1 if x0 < x1 else -1
sy = 1 if y0 < y1 else -1
err = dx - dy

x, y = x0, y0

while True:
    # Check for collision at this pixel
    if (x, y) in self.collision_map:
        # Convert back to world coordinates
        world_x = x * self.pixel_size - self.offset_x
        world_y = y * self.pixel_size - self.offset_y
        return self.collision_map[(x, y)], np.array([world_x, world_y])

    # Check if we've reached the end
    if x == x1 and y == y1:
        break

    # Move to next pixel
    e2 = 2 * err
    if e2 > -dy:
        err -= dy
        x += sx
    if e2 < dx:
        err += dx
        y += sy

return None, None
```

Abbildung 7.5: Bresenham-Schleife

mit entsprechend rotierten Normalenvektoren diskretisiert. Diese statischen Boards sind geometrisch fest und unterscheiden sich fundamental von den dynamischen Flippern, deren Kollisionen analytisch in der Pinball-Klasse behandelt werden.

7.4.4 Kollisionserkennung

Die Kollisionserkennung bietet zwei Schnittstellen mit unterschiedlichen Anwendungsfällen. Die Methode *check_collision(x, y)* prüft durch direkte Dictionary-Abfrage, ob eine gegebene Position mit einem statischen Element der Kollisionskarte kollidiert. Die Methode *check_trajectory(start_pos, end_pos)* detektiert Kollisionen entlang einer linearen Trajektorie und ist essentiell für die Erkennung von Hochgeschwindigkeitskollisionen, bei denen die Kugel in einem Zeitschritt mehrere Pixel durchquert. Nach der Konvertierung der Start- und Endpositionen in Map-Koordinaten durchläuft die Implementierung die Trajektorie mittels Bresenham-Algorithmus. Abbildung 7.5 zeigt die zentrale Durchlaufschleife. Der Algorithmus verwendet Fehlerakkumulation zur inkrementellen Bestimmung

der Pixelsequenz entlang der Linie. Die Variablen dx und dy repräsentieren die absoluten Distanzen, während sx und sy die Bewegungsrichtung kodieren. Die Fehlervariable err steuert, ob der nächste Schritt horizontal, vertikal oder diagonal erfolgt. Bei jedem durchlaufenen Pixel erfolgt eine Dictionary-Abfrage, ob dieser Pixel zur Kollisionskarte gehört. Bei detektierter Kollision wird die Rückkonvertierung der Map-Koordinaten in Weltkoordinaten durchgeführt und die Methode liefert sowohl die CollisionInfo als auch die präzise Kollisionsposition.

7.4.5 Kollisionsauflösung

Die Methode *resolve_collision()* berechnet die physikalische Reaktion auf eine detektierte Kollision mit statischen Objekten. Die reflektierte Geschwindigkeit wird durch Projektion der Eingangsgeschwindigkeit auf den Normalenvektor über die Gleichung

$$\vec{v}_\perp = \vec{v} \cdot \vec{n} \tag{7.6}$$

berechnet. Die Kollision wird nur aufgelöst, wenn $v_\perp < 0$ gilt, was einer Annäherung an die Oberfläche entspricht. Die neue Geschwindigkeit berechnet sich dann über

$$\vec{v}_{\text{neu}} = \vec{v} + \Delta\vec{v}, \quad \Delta\vec{v} = -(1+e)\,\vec{v}_\perp \cdot \vec{n}. \tag{7.7}$$

wobei e der materialspezifische Restitutionskoeffizient aus der CollisionInfo ist. Parallel zur Geschwindigkeitskorrektur wird die Kugelposition aus der Kollisionszone verschoben, um Penetration zu verhindern. Die initiale Verschiebung beträgt $3 \cdot$ pixel_size in Normalenrichtung. Eine iterative Sicherheitsprüfung stellt sicher, dass die neue Position nicht erneut mit einem statischen Element kollidiert, wobei bei Bedarf die Verschiebedistanz inkrementell erhöht wird. Diese robuste Behandlung verhindert numerische Instabilitäten, die durch diskrete Zeitschritte und Pixelisierung entstehen können.

7.5 Gymnasium-Environment

Die Integration der Physiksimulation in das Reinforcement-Learning-Framework erfolgt durch die Klasse PinballEnv, die das Gymnasium-Interface implementiert und damit

die Schnittstelle zwischen Agent und Simulationsumgebung bildet. Gymnasium ist der offizielle Nachfolger des OpenAI-Gym-Frameworks und definiert einen standardisierten API-Vertrag, der es ermöglicht, verschiedene RL-Algorithmen mit beliebigen Umgebungen zu kombinieren [3].

7.5.1 Observation Space und Action Space

Der Observation Space definiert die Zustandsrepräsentation, die dem Agenten zur Verfügung steht. Die Implementierung verwendet einen Dictionary Space, der zwei Komponenten umfasst: Die Flipper-Winkel werden im Intervall $[-\theta_{\text{max}}, \theta_{\text{max}}]$ angegeben, wobei $\theta_{\text{max}} = 35$ dem maximalen Öffnungswinkel entspricht. Die Kugelposition wird auf den normierten Bereich $[-1, 1]$ für beide Achsen skaliert, wobei die Transformation $x_{\text{norm}} = x/x_{\text{max}}$ und $y_{\text{norm}} = y/y_{\text{max}}$ die physikalischen Koordinaten in dimensionslose Größen überführt. Diese Normalisierung gewährleistet numerische Stabilität beim Training neuronaler Netze und macht die Observation unabhängig von den absoluten Spielfeldabmessungen.

Der Action Space ist als diskreter Raum mit vier Aktionen definiert, die alle Kombinationen der Flipper-Aktivierung abbilden. Die binäre Kodierung ermöglicht eine effiziente Repräsentation: Aktion null (00_2) entspricht keiner Bewegung, Aktion eins (01_2) aktiviert den rechten Flipper, Aktion zwei (10_2) den linken Flipper und Aktion drei (11_2) beide Flipper simultan. Diese Diskretisierung reduziert die Komplexität des Aktionsraums gegenüber kontinuierlichen Steuerungen und erleichtert die Konvergenz wertbasierter Lernverfahren wie DQN.

7.5.2 Reward-Funktion

Die Belohnungsstruktur implementiert ein dichtes Reward-Signal, das sowohl Verhaltensanreize als auch Bestrafungen für suboptimale Aktionen kombiniert. Eine Basis-Belohnung von $r_{\text{survival}} = +0{,}02$ wird bei jedem Zeitschritt vergeben und incentiviert das Aufrechterhalten langer Episoden. Bei erfolgreicher Flipper-Ball-Kollision erhält der Agent eine Belohnung von $r_{\text{hit}} = +1{,}0$, während die Aktivierung eines Flippers ohne nachfolgende Kollision mit $r_{\text{miss}} = -0{,}23$ bestraft wird. Diese asymmetrische Gewichtung verhindert blindes Flipper-Spamming und fördert gezieltes Timing. Terminiert die Episode durch Verlust der Kugel, wird eine Strafe von $r_{\text{term}} = -1{,}0$ addiert.

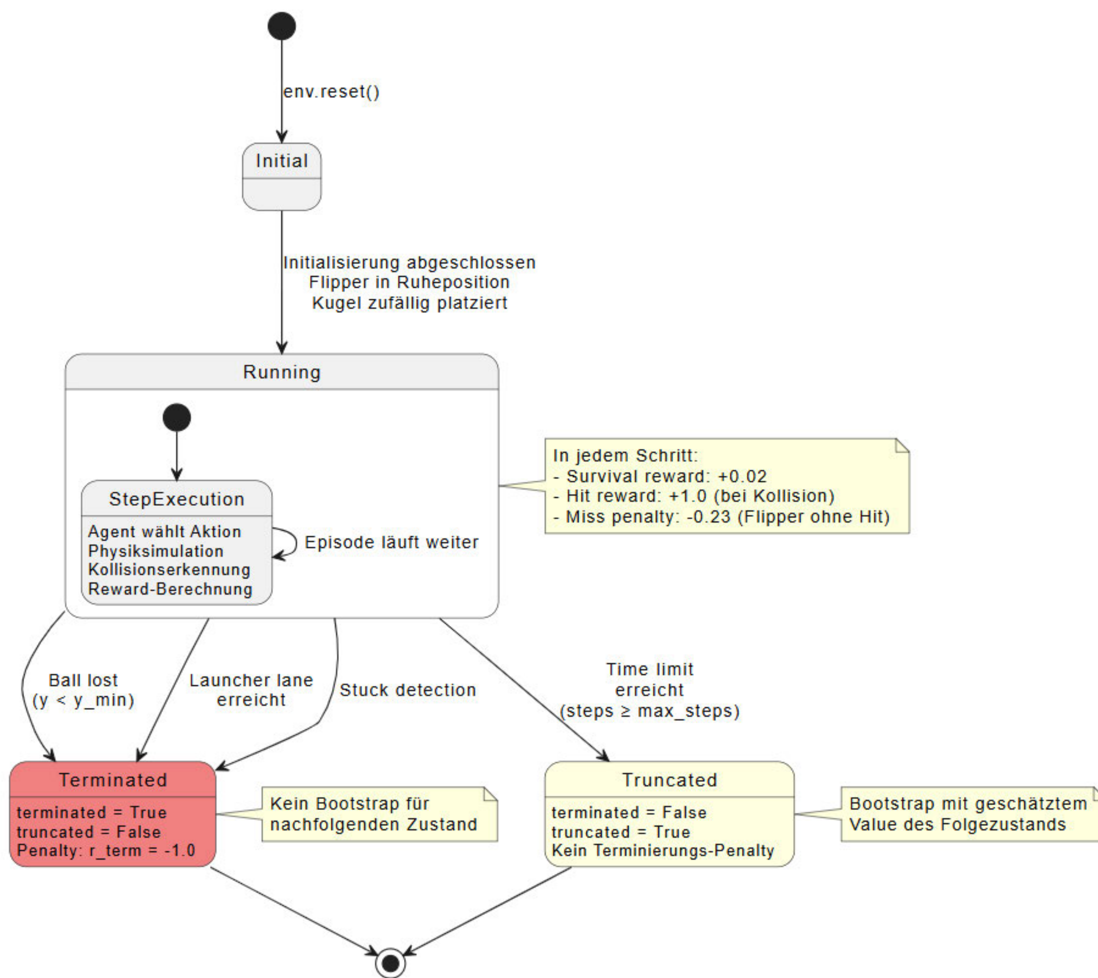


Abbildung 7.6: Zustandsautomat des Lebenszyklus einer Episode

7.5.3 Episode-Lifecycle

Der Lebenszyklus einer Episode folgt einem klar definierten Zustandsautomaten, der in Abbildung 7.6 dargestellt ist. Nach der Initialisierung durch *reset()* befindet sich das Environment im Zustand *Running*, in dem der Agent kontinuierlich Aktionen ausführen kann. Die Episode kann durch verschiedene Bedingungen beendet werden, wobei zwischen natürlicher Terminierung und erzwungenem Abbruch durch externe Constraints unterschieden wird.

Eine natürliche Terminierung tritt ein, wenn die Kugel unterhalb der Flipper-Ebene fällt ($y < y_{\min}$), wobei $y_{\min}$ analytisch aus der Flipper-Geometrie und dem maximalen

Öffnungswinkel berechnet wird. Zusätzlich wird die Episode beendet, wenn die Kugel die Startbahn (Launcher Lane) im unteren rechten Spielfeldbereich erreicht, was physikalisch dem Verlust der Kugel entspricht. Eine Stuck-Detection terminiert die Episode nach mehr als 100 Kollisionen innerhalb eines Zeitfensters, um Situationen zu behandeln, in denen die Kugel in einer Anomalie gefangen ist und nicht mehr produktiv gespielt werden kann.

Der Zustandsautomat visualisiert diese Übergänge und zeigt die verschiedenen Terminierungspfade. Die Unterscheidung zwischen 'terminated' und 'truncated' ist für moderne RL-Algorithmen von Bedeutung, da sie unterschiedliche Bootstrap-Strategien für die Value-Funktion erfordern. Bei natürlicher Terminierung wird kein nachfolgender Zustand für das Bootstrapping verwendet, während bei 'Truncation' der erwartete Wert des Folgezustands geschätzt werden muss.

7.5.4 Implementierung der Gymnasium-Schnittstelle

Die Klasse PinballEnv implementiert die drei zentralen Methoden des Gymnasium-Interfaces: *reset()*, *step()* und *render()*. Die reset()-Methode initialisiert eine neue Episode, indem die Flipper in die untere Ruheposition gesetzt werden und die Kugel zufällig im oberen Spielfeldbereich platziert wird.

Die step()-Methode orchestriert die Ausführung einer Aktion in einer klar definierten Sequenz, die in Abbildung 7.7 dargestellt ist. Zunächst wird der Action-Index in binäre Flipper-Aktivierungen dekodiert, wobei das Mapping durch ein vordefiniertes Dictionary erfolgt. Anschließend wird die alte Kugelposition gespeichert, bevor die Physiksimulation aufgerufen wird. Nach der Bewegungsberechnung erfolgt die Kollisionsprüfung zwischen Flipper und Ball. Basierend auf der Flipper-Aktivierung und dem Kollisionsergebnis wird der Reward gemäß der definierten Belohnungsstruktur berechnet. Abschließend prüft die Methode die Terminierungsbedingungen und ergänzt bei Episodenende die Terminierungsstrafe. Die Rückgabewerte entsprechen dem Gymnasium-Standard (observation, reward, terminated, truncated, info), wobei das info-Dictionary zusätzliche Diagnoseinformationen wie Flipper-Aktivierung und detektierte Kollisionen enthält.

```python
# Decode action to flipper movements
action = int(action) if isinstance(action, np.ndarray) else action
movements = self.__action_to_movements[action]
activate_left = movements[0]
activate_right = movements[1]

flipper_activated = activate_left or activate_right

# Store old position for collision detection
old_pos = self.state.position.copy()

# Execute physics step
self.pinball.step(self.state, time_sec=0.01,
                  activate_left=activate_left,
                  activate_right=activate_right)

# Check for collisions
collision_info, _ = self.pinball.collision.check_trajectory(old_pos, self.state.position)
if collision_info:
    self.collision_count_window += 1

# Check for flipper collision (hit detection)
flipper_hit = self._check_flipper_collision(old_pos, self.state.position)

# Calculate reward
reward = self.__calculate_reward(flipper_activated, flipper_hit)

# Check termination conditions
done = self.__is_terminated(self.collision_count_window)

# Apply termination penalty if episode ended
if done:
    reward += self.PENALTY_TERMINATION

# Render if active
if self.render_mode == '3D':
    self.render()

# Return step results
observation = self.__get_observation()
truncated = False
info = {
    'flipper_activated': flipper_activated,
    'flipper_hit': flipper_hit,
    'collisions': self.collision_count_window
}
return observation, reward, done, truncated, info
```

Abbildung 7.7: step()-Methode der PinballEnv.py

7.6 Environment-Wrapper

Das Gymnasium-Framework ermöglicht die modulare Erweiterung von Environments durch Wrapper-Klassen, die das Verhalten der Umgebung modifizieren, ohne den Quellcode des Base-Environments anzutasten. Wrapper ermöglichen die schrittweise Transformation von Observations, Actions und Rewards sowie die Erweiterung der Environment-Logik. Die Implementierung nutzt vier spezialisierte Wrapper, die hierarchisch verschachtelt werden und deren Kompositionsstruktur in Abbildung 7.8 als Objektdiagramm dargestellt ist.

Der FlattenObservation-Wrapper transformiert den Dictionary-basierten Observation Space in einen flachen Vektor der Dimension vier. Die ursprüngliche Observation wird in einen eindimensionalen Array $[f_L, f_R, b_x, b_y]$ konvertiert, wobei f_L und f_R die Winkel des lin-

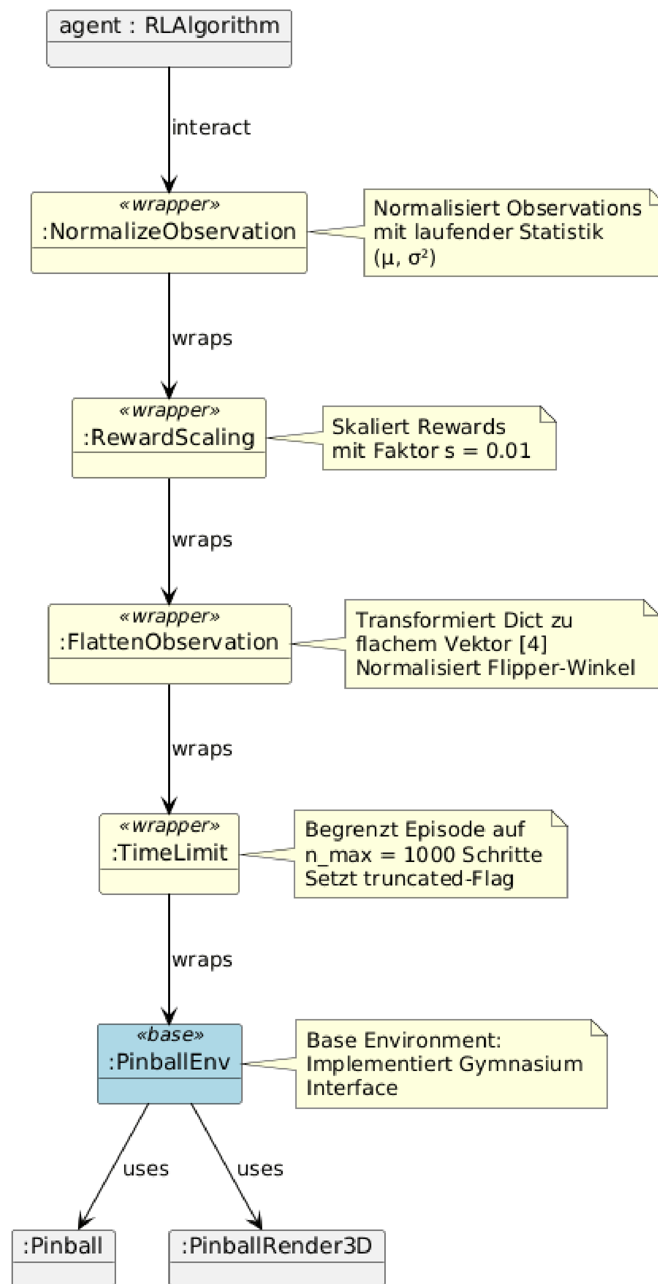


Abbildung 7.8: Objektdiagramm der Wrapper-Kompositionsstruktur

ken bzw. rechten Flippers und b_x und b_y die Koordinaten der Kugelposition bezeichnen. Die Flipper-Winkel werden zusätzlich auf das Intervall $[-1, 1]$ normalisiert durch die Transformation

$$f_{\text{norm}} = \frac{f}{\theta_{\max}}, \tag{7.8}$$

wobei f den ursprünglichen Flipper-Winkel bezeichnet. Diese Konvertierung ist zwingend erforderlich für die Kompatibilität mit DQN-Algorithmen, da Q-Netzwerke flache Vektoreingaben erwarten und Dictionary Spaces nicht direkt verarbeiten können. Die Wrapper-Klasse erbt von *gym.ObservationWrapper*.

Der TimeLimit-Wrapper begrenzt die maximale Episodenlänge auf $n_{\max} = 1000$ Schritte und verhindert damit potenziell unendlich lange Episoden. Der RewardScaling-Wrapper multipliziert alle Rewards mit einem Skalierungsfaktor s, um die numerische Stabilität beim Training neuronaler Netze zu verbessern. Typische Werte liegen bei $s = 0{,}01$, wodurch große Reward-Variationen gedämpft werden.

Der NormalizeObservation-Wrapper führt eine laufende Normalisierung der Observations durch, indem Mittelwert und Varianz über alle bisherigen Observations geschätzt und die Transformation

$$o_{\text{norm}} = \frac{o - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{7.9}$$

angewendet wird, wobei o die ursprüngliche Observation, μ den laufenden Mittelwert, σ^2 die laufende Varianz und ϵ eine kleine Konstante zur Vermeidung von Division durch Null bezeichnet. Diese Normalisierung verhindert, dass einzelne Observation-Dimensionen durch ihre Größenordnung das Training dominieren.

Die Reihenfolge der Wrapper-Anwendung ist semantisch relevant und folgt der Regel, dass Transformationen von innen nach außen erfolgen. Das Base-Environment PinballEnv wird zunächst mit TimeLimit gewrappt, um die Episodenlänge zu kontrollieren. Anschließend transformiert FlattenObservation die Dictionary-Observation in einen flachen Vektor. Die äußeren Wrapper NormalizeObservation und RewardScaling normalisieren schließlich die Observations und Rewards für das Training. Diese hierarchische Struktur ermöglicht eine klare Trennung von Zuständigkeiten, wobei jeder Wrapper eine spezifische Verantwortlichkeit trägt und unabhängig aktiviert oder deaktiviert werden kann.

7.7 Rendering und Visualisierung

Die visuelle Darstellung der Simulationsumgebung erfolgt durch die Klasse PinballRender3D, die auf dem Panda3D-Framework basiert und eine dreidimensionale Visualisierung des Spielfelds ermöglicht. Die Rendering-Komponente ist vollständig entkoppelt von der Simulationslogik und wird optional über einen Parameter beim Instanziieren des Environments aktiviert.

Die 3D-Szene wird aus der PinballGeometry-Klasse konstruiert, wobei alle geometrischen Parameter direkt in visuelle Primitive übersetzt werden. Das Spielfeld wird als geneigte Ebene mit dem konfigurierten Neigungswinkel dargestellt, die Flipper als rechteckige Körper an ihren Pivot-Punkten und die Kugel als Sphäre mit dem definierten Radius. Die Kamera ist in einer festen isometrischen Perspektive positioniert, die das gesamte Spielfeld überblickt.

Die *render()-Methode* des Environments aktualisiert die Positionen der dynamischen Objekte durch Aufrufe von *move_ball()* und *set_flipper_angles()*, wobei die Transformationen direkt aus dem aktuellen State übernommen werden. Die Visualisierung erfolgt synchron mit der Physiksimulation und ermöglicht sowohl die Beobachtung des Trainingsprozesses als auch die Evaluation trainierter Agenten. Für das Training ohne Visualisierung kann *render_mode=None* gesetzt werden, was die Ausführungsgeschwindigkeit signifikant erhöht.

7.8 Training

Das Training des DQN-Agenten erfolgt unter Verwendung der Stable-Baselines3-Bibliothek [22], die eine optimierte Implementierung des in Abschnitt 2.3 beschriebenen Deep Q-Network-Algorithmus bereitstellt. Die Trainingskonfiguration umfasst die Orchestrierung mehrerer paralleler Environments, die Festlegung der Hyperparameter sowie die Integration von Callbacks für Monitoring und Modell-Persistierung.

Die Erzeugung der Trainingsumgebungen erfolgt über eine Factory-Funktion, die die konsistente Instanziierung und Konfiguration der Environment-Pipeline aus Abschnitt 7.6 gewährleistet. Der Monitor-Wrapper protokolliert Episode-Statistiken wie kumulative Rewards und Episodenlängen für die spätere Analyse.

Die Konfiguration des DQN-Algorithmus basiert auf den in Abschnitt 2.3 eingeführten theoretischen Grundlagen. Tabelle 7.1 listet die verwendeten Hyperparameter mit ihren Werten. Wichtig ist zu erwähnen, dass kein Parameter-Tuning durchgeführt worden ist.

Parameter	Wert	Beschreibung
`learning_rate`	1×10^{-4}	Schrittweite α für Gradientenabstieg
`buffer_size`	50000	Kapazität N des Replay Buffers
`batch_size`	32	Größe der Mini-Batches beim Training
`gamma`	0.99	Diskontierungsfaktor γ
`learning_starts`	1000	Schritte vor Beginn des Trainings
`target_update_interval`	1000	Update-Intervall C des Target Networks
`exploration_fraction`	0.3	Anteil f_{explore} für ϵ-Decay
`exploration_initial_eps`	1.0	Initialer Explorationswert ϵ_{start}
`exploration_final_eps`	0.05	Finaler Explorationswert ϵ_{end}

Tabelle 7.1: DQN-Hyperparameter

Die Lernrate wurde empirisch gewählt, um stabiles Konvergenzverhalten zu ermöglichen. Der Replay Buffer speichert ausreichend Erfahrungen, um zeitliche Korrelation aufzubrechen (siehe Abschnitt 2.3.2). Der Diskontierungsfaktor priorisiert langfristige Belohnungen, was für die Pinball-Aufgabe essentiell ist. Die Exploration-Strategie folgt der in Gleichung 2.28 definierten epsilon-greedy-Policy mit linearem Decay über die ersten 30% der Trainingsschritte. Das Target Network wird alle 1000 Trainingsschritte durch Hard Update mit den Q-Network-Parametern synchronisiert. Die Netzwerkarchitektur verwendet die MlpPolicy mit zwei Hidden Layers à 64 Neuronen und ReLU-Aktivierungen.

7.9 Evaluation

Die Evaluation der virtuellen Trainingsumgebung erfolgt anhand der in Abschnitt 4.3.2 definierten funktionalen und nicht-funktionalen Anforderungen. Ziel ist es, zu überprüfen, ob die implementierte Umgebung alle gestellten Anforderungen erfüllt und somit als Grundlage für das Training eines Reinforcement-Learning-Agenten geeignet ist.

7.9.1 Funktionale Anforderungen

Visualisierung der virtuellen Umgebung (VU-FA1): Die Anforderung nach visueller Darstellung des Spielfelds, der Flipperarme und der Kugel wird durch die Pinball-

Render3D-Klasse erfüllt. Die 3D-Visualisierung basiert auf dem Panda3D-Framework und stellt alle relevanten Komponenten in Echtzeit dar. Die Darstellung ermöglicht die Beobachtung des Agentenverhaltens während des Trainings und der Evaluation.

Aktualisierung des Systemzustands (VU-FA2): Die kontinuierliche Berechnung und physikalisch korrekte Aktualisierung des Systemzustands erfolgt durch die Pinball-Klasse mit der Physik-Engine. Die Implementierung berücksichtigt das Rollverhalten der Kugel auf geneigter Ebene mit Reibungskoeffizienten, die Schwerkraftkomponente entlang der Neigung sowie Kollisionen mit statischen und dynamischen Objekten. Die Flipper-Bewegung folgt einer realistischen Kinematik mit konstanter Winkelgeschwindigkeit und definierten Grenzen. Die Simulation läuft deterministisch mit festem Zeitschritt von 0,01 Sekunden.

Bereitstellung von Zustandsinformationen (VU-FA3): Die PinballEnv-Klasse stellt dem RL-Agenten über den Observation Space kontinuierlich den aktuellen Zustand zur Verfügung. Dieser umfasst die Flipper-Winkel sowie die normalisierte Kugelposition und wird nach jedem Simulationsschritt aktualisiert. Die Dictionary-Struktur ermöglicht einen klar strukturierten Zugriff auf alle relevanten Zustandsinformationen.

Steuerung der Flipperarme (VU-FA4): Der RL-Agent kann über den Action Space mit vier diskreten Aktionen beide Flipper unabhängig voneinander aktivieren. Die Flipper bewegen sich mit einer Winkelgeschwindigkeit von 600 Grad pro Sekunde bis zu einem maximalen Öffnungswinkel von 35 Grad. Die Parameter sind in der Pinball-Klasse konfigurierbar und können an unterschiedliche Anforderungen angepasst werden.

Spielflusssicherung und Episoden-Management (VU-FA5): Die Terminierungsbedingungen in PinballEnv erkennen automatisch, wenn die Kugel verloren geht, die Startbahn erreicht oder das System in einen nicht mehr spielbaren Zustand gerät. Bei Episodenende wird automatisch ein Neustart durch die reset()-Methode initiiert, die die Flipper in Ruheposition setzt und die Kugel zufällig im oberen Spielfeldbereich platziert. Dies gewährleistet einen unterbrechungsfreien Trainingsablauf.

Training des RL-Agenten (VU-FA6): Die Integration mit Stable-Baselines3 ermöglicht das Training von DQN-Modellen auf der virtuellen Umgebung. Trainingsparameter wie Episode-Länge, kumulative Rewards und Kollisionszähler werden durch den Monitor-Wrapper kontinuierlich aufgezeichnet. Die TensorBoard-Integration visualisiert den Trainingsverlauf in Echtzeit und ermöglicht die Analyse von Konvergenzverhalten und Lernfortschritt.

Evaluation des RL-Agenten (VU-FA7): Der EvalCallback führt regelmäßige Evaluationen während des Trainings durch und misst dabei Episode-Länge sowie kumulative Rewards über mehrere Episoden. Die Trefferquote wird durch die Reward-Komponente für erfolgreiche Flipper-Ball-Kollisionen implizit erfasst. Separate Evaluations-Skripte ermöglichen detaillierte Auswertungen trainierter Modelle mit beliebig vielen Test-Episoden.

Speichern und Laden von KI-Modellen (VU-FA8): Die Persistierung trainierter Modelle erfolgt über die Stable-Baselines3-API durch die save()-Methode, die sowohl Netzwerkgewichte als auch Hyperparameter speichert. Der CheckpointCallback erstellt automatisch Zwischenstände während des Trainings. Gespeicherte Modelle können über die load()-Methode wieder geladen und für weitere Trainings- oder Evaluationszwecke verwendet werden.

7.9.2 Nicht-funktionale Anforderungen

Physikalische Modellierung und Genauigkeit (VU-NFA1): Die 3D-Darstellung bildet alle Komponenten des physischen Flippers visuell erkennbar ab. Die Physiksimulation modelliert die Kugelbewegung als rollende Vollkugel auf geneigter Ebene mit korrektem Trägheitsmoment. Die Flipper-Kinematik respektiert die mechanischen Einschränkungen mit maximalem Auslenkwinkel von 35 Grad und realistischer Bewegungsgeschwindigkeit.

Proportionale Abbildung (VU-NFA2): Die PinballGeometry-Klasse definiert alle geometrischen Parameter in Zentimetern und bildet die Dimensionen des physischen Demonstrators maßstabsgetreu nach. Die Spielfeldgröße, Flipper-Positionen und Kugel-Radius entsprechen den realen Verhältnissen.

Zustandsraum-Definition (VU-NFA3): Der Observation Space stellt dem RL-Agenten alle relevanten Informationen in normalisiertem Format bereit. Die Flipper-Winkel liegen im Intervall vom negativen bis positiven maximalen Öffnungswinkel, die Kugelposition ist auf den Bereich minus eins bis eins skaliert. Diese Normalisierung gewährleistet numerische Stabilität beim Training neuronaler Netze.

Belohnungssystem (VU-NFA4): Das implementierte Reward-System kombiniert Survival-Reward, Hit-Reward und Miss-Penalty zu einem dichten Signal, das gezieltes Flipper-Timing incentiviert. Evaluationen zeigen, dass trainierte Agenten die Kugel deutlich län-

ger als 15 Sekunden im Spiel halten können, was die Effektivität des Belohnungssystems bestätigt.

Echtzeitvisualisierung (VU-NFA5): Die Panda3D-basierte Visualisierung läuft flüssig mit mehr als 30 FPS bei aktiviertem Rendering. Die Aktualisierung der Flipper-Positionen und Kugel-Position erfolgt synchron mit der Physiksimulation ohne merkbare Verzögerungen.

Simulationsstabilität (VU-NFA6): Die Kollisionserkennung durch PinballCollision verhindert zuverlässig das Durchdringen von Wänden durch die Kugel. Die trajektorienbasierte Kollisionsprüfung detektiert auch Hochgeschwindigkeitskollisionen. Die Stuck-Detection terminiert Episoden bei physikalisch inkonsistenten Zuständen und verhindert damit unendliche Episoden.

Code-Qualität (VU-NFA7): Die Implementierung folgt objektorientierten Prinzipien mit klarer Trennung von Datenstrukturen, Simulationslogik und Visualisierung. Das modulare Design mit PinballGeometry, State, Pinball, PinballCollision, PinballEnv und PinballRender3D ermöglicht eine wartbare und erweiterbare Codebasis. Alle Klassen und Methoden sind dokumentiert.

Modellübertragbarkeit (VU-NFA8): Die virtuelle Umgebung verwendet denselben Zeitschritt von 0,01 Sekunden wie der physische Demonstrator. Die Geometrieparameter und Flipper-Kinematik sind konfigurierbar und können exakt an die reale Hardware angepasst werden. Trainierte Modelle können ohne Modifikation auf dem physischen System ausgeführt werden, sofern die Action- und Observation-Spaces identisch implementiert sind.

8 Inbetriebnahme des physischen Demonstrators

Die Übertragung des in der virtuellen Umgebung trainierten RL-Agenten auf den physischen Flipperautomaten stellt eine der zentralen Herausforderungen im Bereich des maschinellen Lernens dar, da sich simulierte und reale Systeme in wesentlichen Aspekten unterscheiden. Für die praktische Erprobung wurde eine Gymnasium-Umgebung implementiert, die alle notwendigen Hardwarekomponenten integriert. Die Systemarchitektur umfasst dabei die serielle Kommunikation mit dem Arduino zur Flipper-Ansteuerung, eine Threading-basierte Kameraintegration für die kontinuierliche Bildverarbeitung sowie die Inferenz des trainierten DQN-Modells. Der Observation Space besteht aus normalisierten Flipper-Winkeln und Ball-Koordinaten, während der diskrete Action Space vier Flipper-Kombinationen ermöglicht.

Bei der praktischen Durchführung zeigte sich, dass die grundlegende Integration aller Komponenten erfolgreich war. Die Bildverarbeitung lieferte stabile Detektionsergebnisse, die Flipper ließen sich zuverlässig ansteuern und der Agent konnte Aktionen generieren. Allerdings offenbarte sich dabei ein gravierendes Problem: Die Flipper schlugen nahezu kontinuierlich, was auf ein fundamentales Verhalten des Agenten hindeutet, das nicht mit sinnvollem Spielverhalten übereinstimmt. Diese Beobachtung lässt sich auf mehrere systematische Diskrepanzen zwischen simulierter Trainingsumgebung und physischem System zurückführen.

Ein zentraler Faktor ist die Timing-Diskrepanz zwischen virtueller und physischer Umgebung. In der Simulation erfolgt die Aktionsausführung praktisch verzögerungsfrei zum beobachteten Zustand, während im physischen System eine Gesamtlatenz zwischen Bildaufnahme und tatsächlicher Flipper-Bewegung entsteht. Diese setzt sich zusammen aus der Belichtungszeit der Kamera, der Bildverarbeitung zur Extraktion von Ball- und Flipper-Positionen, der Inferenzzeit des neuronalen Netzes sowie der mechanischen Reaktionszeit der Motoren. Während in der Simulation Zustand und Aktion nahezu synchron sind,

existiert im realen System eine signifikante zeitliche Entkopplung. Da der Agent niemals mit dieser Latenz trainiert wurde, kann er nicht adäquat darauf reagieren.

Das zweite wesentliche Problem betrifft die Bildverarbeitung, die im Zusammenspiel aller Komponenten noch nicht ausreichend getestet und optimiert werden konnte. Während die einzelnen Bildverarbeitungsalgorithmen zur Ball-Detektion und Flipper-Tracking in isolierten Tests zufriedenstellende Ergebnisse lieferten, zeigten sich im Echtzeitbetrieb des gesamten Systems Schwächen, die eine zuverlässige Zustandserfassung beeinträchtigen.

Die Experimente demonstrieren die grundsätzliche Machbarkeit der Hardware-Integration und zeigen gleichzeitig die Herausforderungen des Sim-to-Real-Transfers auf. Die identifizierten Problemfelder bieten klare Ansatzpunkte für zukünftige Arbeiten. Diese Arbeit legt damit die konzeptionelle und technische Grundlage für iterative Verbesserungen in Richtung eines vollständig autonomen Flippersystems, auch wenn eine vollständige funktionale Integration im Rahmen dieser Masterarbeit aufgrund der Komplexität des Sim-to-Real-Problems nicht abgeschlossen werden konnte.

9 Fazit und Ausblick

Die vorliegende Masterarbeit hatte zum Ziel, ein autonomes Flippersystem durch die Integration von Deep Reinforcement Learning und Bildverarbeitung zu realisieren. Im Zentrum stand dabei die Entwicklung einer virtuellen Trainingsumgebung als digitaler Zwilling sowie die Konzeption eines Bildverarbeitungssystems zur Zustandserfassung, mit der übergeordneten Absicht, den trainierten RL-Agenten auf einen physischen Demonstrator zu übertragen.

Im Bereich der Bildverarbeitung konnte ein vollständig funktionsfähiges System entwickelt werden, das alle relevanten Spielelemente zuverlässig erkennt und präzise lokalisiert. Die Spielfelderkennung liefert eine stabile geometrische Grundlage für alle nachfolgenden Verarbeitungsschritte, während die Kugelerkennung auch bei hohen Geschwindigkeiten eine robuste Positionsbestimmung gewährleistet. Die Flipperarm-Erkennung ermöglicht die kontinuierliche Erfassung der Armstellungen und vervollständigt damit die Zustandsrepräsentation des Systems. Die durchgeführten Evaluationen bestätigen, dass das Bildverarbeitungssystem die gestellten Anforderungen an Genauigkeit und Zuverlässigkeit in kontrollierten Testszenarien erfüllt.

Die virtuelle Trainingsumgebung wurde als physikalisch realistische Simulation implementiert, die eine effiziente Entwicklung und Erprobung von RL-Agenten ermöglicht. Die Umgebung bildet die wesentlichen Dynamiken des Flippersystems hinreichend genau nach und stellt eine geeignete Grundlage für das Training dar. Trainierte Agenten zeigen in der Simulation ein funktionales Spielverhalten und können die Kugel über längere Zeiträume im Spiel halten. Die konsequente Ausrichtung der Simulation an den Parametern des physischen Demonstrators legt die konzeptionelle Grundlage für eine spätere Modellübertragung.

Die Integration aller Systemkomponenten zum physischen Demonstrator gelang technisch erfolgreich. Alle Hardwarekomponenten kommunizieren wie vorgesehen, die Bildverarbeitung liefert kontinuierliche Zustandsinformationen und der Agent kann Aktionen generieren. Die praktische Erprobung offenbarte jedoch fundamentale Herausforderungen beim

Sim-to-Real-Transfer: Der Agent zeigt ein inadäquates Spielverhalten, das sich durch nahezu kontinuierliches Schlagen der Flipper manifestiert und nicht mit den erlernten sinnvollen Spielstrategien in der virtuellen Umgebung korrespondiert.

Die Ursache dieses Problems liegt in systematischen Diskrepanzen zwischen simulierter und physischer Umgebung. Ein zentraler Faktor ist die zeitliche Entkopplung von Beobachtung und Aktion im realen System. Während in der Simulation Zustand und Reaktion praktisch synchron erfolgen, entsteht im physischen Aufbau eine signifikante Gesamtlatenz durch die sequenzielle Verarbeitung von Bildaufnahme, Zustandsextraktion, Inferenz und mechanischer Ausführung. Da der Agent während des Trainings ausschließlich mit verzögerungsfreien Aktionen konfrontiert wurde, verfügt er nicht über die notwendigen Strategien zum Umgang mit dieser Latenz. Das zweite wesentliche Problem betrifft die Robustheit der Bildverarbeitung im integrierten Systemkontext. Obwohl die einzelnen Komponenten in isolierten Tests zufriedenstellende Resultate lieferten, zeigten sich im Echtzeitbetrieb unter realen Bedingungen Stabilitätsprobleme, die eine zuverlässige Zustandserfassung beeinträchtigen.

Für die Weiterentwicklung des Systems ergeben sich mehrere zentrale Ansatzpunkte. Die Integration der Systemlatenz in die virtuelle Trainingsumgebung stellt eine prioritäre Maßnahme dar. Durch explizite Modellierung der zeitlichen Verzögerung während des Trainings könnte der Agent Strategien entwickeln, die mit der inhärenten Latenz des physischen Systems kompatibel sind. Komplementär dazu sollte die Optimierung der Gesamtsystemlatenz verfolgt werden, um die zeitliche Entkopplung von Beobachtung und Aktion zu minimieren. Ein weiterer erfolgversprechender Ansatz besteht darin, das in der Simulation vortrainierte Modell direkt in der physischen Umgebung weiterzutrainieren. Dieses Fine-Tuning würde es dem Agenten ermöglichen, sich an die spezifischen Charakteristika des realen Systems anzupassen und die Diskrepanzen zwischen Simulation und Realität durch kontinuierliches Lernen zu kompensieren.

Das zweite wesentliche Problem betrifft die Bildverarbeitung im integrierten Systemkontext. Obwohl die einzelnen Komponenten in isolierten Tests zufriedenstellende Resultate lieferten, zeigten sich im Echtzeitbetrieb, wenn alle Systemkomponenten parallel arbeiten, Stabilitätsprobleme, die eine zuverlässige Zustandserfassung beeinträchtigen. Die Bildverarbeitung muss dahingehend optimiert werden, dass sie auch unter den Anforderungen des vollständig integrierten Systems robust und präzise funktioniert.

Die im Rahmen dieser Arbeit entwickelte modulare Systemarchitektur bietet eine solide Grundlage für diese zukünftigen Erweiterungen. Die klare Trennung der Komponenten,

die umfassende Dokumentation sowie die Integration etablierter Frameworks gewährleisten die Wartbarkeit und Erweiterbarkeit des Systems. Zusammenfassend lässt sich konstatieren, dass diese Masterarbeit die konzeptionellen und technischen Grundlagen für ein autonomes Flippersystem erfolgreich etabliert hat. Die entwickelten Komponenten funktionieren in ihren jeweiligen Domänen wie spezifiziert, und die virtuelle Trainingsumgebung ermöglicht das Training funktionsfähiger Agenten. Die Herausforderungen beim Sim-to-Real-Transfer sind charakteristisch für diese Problemklasse und verdeutlichen die inhärente Komplexität der Übertragung simulierten Verhaltens auf physische Systeme. Die identifizierten Problemfelder bieten klare Ansatzpunkte für iterative Verbesserungen. Das geschaffene System stellt damit einen wertvollen Beitrag zur praktischen Anwendung von Reinforcement Learning in physikalischen Systemen dar und legt eine fundierte Basis, auf der zukünftige Entwicklungen aufbauen können, um die Vision eines vollständig autonomen Flippersystems schrittweise zu verwirklichen.

Literaturverzeichnis

[1] Amiri, Roohollah ; Mehrpouyan, Hani ; Fridman, Lex ; Mallik, Ranjan ; Nallanathan, Arumugam ; Matolak, David: A Machine Learning Approach for Power Allocation in HetNets Considering QoS. (2018), 03

[2] An, Gwon H. ; Lee, Siyeong ; Seo, Min-Woo ; Yun, Kugjin ; Cheong, Won-Sik ; Kang, Suk-Ju: Charuco board-based omnidirectional camera calibration method. In: *Electronics* 7 (2018), Nr. 12, S. 421

[3] Brockman, Greg ; Cheung, Vicki ; Pettersson, Ludwig ; Schneider, Jonas ; Schulman, John ; Tang, Jie ; Zaremba, Wojciech: Openai gym. In: *arXiv preprint arXiv:1606.01540* (2016)

[4] Charu C, Aggarwal: *Neural networks and deep learning: a textbook*. Springer, 2018

[5] Coggan, Melanie: Exploration and exploitation in reinforcement learning. In: *Research supervised by Prof. Doina Precup, CRA-W DMP Project at McGill University* 51 (2004)

[6] De Blasi, Stefano ; Klöser, Sebastian ; Müller, Arne ; Reuben, Robin ; Sturm, Fabian ; Zerrer, Timo: Kicker: An industrial drive and control foosball system automated with deep reinforcement learning. In: *Journal of Intelligent & Robotic Systems* 102 (2021), Nr. 1, S. 20

[7] Eberly, David H.: *Game physics*. CRC Press, 2010

[8] Goll, Joachim: Einführung in standardisierte Diagrammtypen nach UML. In: *Methoden des Software Engineering*. Springer, 2012

[9] Hensel, Marc ; Lassahn, Sanda V.: Virtual Environment and Automated Physical Rolling Maze as Experimental Platform for Deep Reinforcement Learning. In: *Proceedings of the International Symposium on Ambient Intelligence and Embedded Systems (AmiEs-2025)*, 2025. – https://international-symposium.org/amies_2025/proceedings_2025/Hensel_AmiEs_2025_Paper.pdf

[10] Huang, Thomas S. ; Schreiber, William F. ; Tretiak, Oleh J.: Image processing. In: *Proceedings of the IEEE* 59 (2005), Nr. 11, S. 1586–1609

[11] Kleuker, Stephan ; Kleuker, Stephan: Grundkurs Software-Engineering mit UML: Der pragmatische Weg zu erfolgreichen Softwareprojekten. (2013), S. 55–92

[12] Kober, Jens ; Bagnell, J A. ; Peters, Jan: Reinforcement learning in robotics: A survey. In: *The International Journal of Robotics Research* 32 (2013), Nr. 11, S. 1238–1274

[13] Lassahn, Sandra V.: *3D-Simulation und prototypischer Aufbau eines durch Reinforcement Learning gesteuerten Labyrinths*, HAW Hamburg, Masterarbeit, 2024. – https://reposit.haw-hamburg.de/handle/20.500.12738/18266

[14] Li, Yuxi: Deep reinforcement learning: An overview. In: *arXiv preprint arXiv:1701.07274* (2017)

[15] Metcalf, Adam: Pinball: High-speed real-time tracking and playing. (2011)

[16] Miguel, José P ; Mauricio, David ; Rodríguez, Glen: A review of software quality models for the evaluation of software products. In: *arXiv preprint arXiv:1412.2977* (2014)

[17] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Graves, Alex ; Antonoglou, Ioannis ; Wierstra, Daan ; Riedmiller, Martin: Playing atari with deep reinforcement learning. In: *arXiv preprint arXiv:1312.5602* (2013)

[18] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Rusu, Andrei A. ; Veness, Joel ; Bellemare, Marc G. ; Graves, Alex ; Riedmiller, Martin ; Fidjeland, Andreas K. ; Ostrovski, Georg u. a.: Human-level control through deep reinforcement learning. In: *nature* 518 (2015), Nr. 7540, S. 529–533

[19] Mukhopadhyay, Priyanka ; Chaudhuri, Bidyut B.: A survey of Hough Transform. In: *Pattern Recognition* 48 (2015), Nr. 3, S. 993–1010

[20] Nischwitz, Alfred ; Fischer, Max ; Haberäcker, Peter ; Socher, Gudrun: *Bildverarbeitung: Band II des Standardwerks Computergrafik und Bildverarbeitung.* Springer-Verlag, 2019

[21] Puterman, Martin L.: Markov decision processes. In: *Handbooks in operations research and management science* 2 (1990), S. 331–434

[22] Raffin, Antonin ; Hill, Ashley ; Gleave, Adam ; Kanervisto, Anssi ; Ernestus, Maximilian ; Dormann, Noah: Stable-baselines3: Reliable reinforcement learning implementations. In: *Journal of machine learning research* 22 (2021), Nr. 268, S. 1–8

[23] Ranaweera, Mahesh ; Mahmoud, Qusay H.: Bridging the reality gap between virtual and physical environments through reinforcement learning. In: *IEEE Access* 11 (2023), S. 19914–19927

[24] Rastagar, Shabir: *Deep Reinforcement Learning und Bildverarbeitung zur generalisierbaren Steuerung physischer Labyrinthe*, HAW Hamburg, Masterarbeit, 2025

[25] Reza, Ali M.: Realization of the contrast limited adaptive histogram equalization (CLAHE) for real-time image enhancement. In: *Journal of VLSI signal processing systems for signal, image and video technology* 38 (2004), Nr. 1, S. 35–44

[26] Saif, Jamil A. ; Hammad, Mahgoub H. ; Alqubati, Ibrahim A.: Gradient based image edge detection. In: *International Journal of Engineering and Technology* 8 (2016), Nr. 3, S. 153

[27] Sutton, Richard S. ; Barto, Andrew G.: Reinforcement learning: An introduction. In: *MIT Press* (2018). ISBN 9780262039246

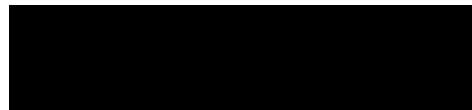
[28] Sutton, Richard S. ; Barto, Andrew G. u. a.: *Reinforcement learning: An introduction.* MIT press Cambridge, 1998

[29] Wessels, Alexander: *Entwicklung und prototypischer Aufbau eines durch Reinforcement Learning gesteuerten Flipper-Automaten.* March 2024. – Bachelor's thesis, Hamburg University of Applied Sciences

[30] Winstead, Nathaniel S. ; Christiansen, Alan D.: Pinball: Planning and learning in a dynamic real-time environment. In: *AAAI-94 Fall Symposium on Control of the Physical World by Intelligent Agents*, 1994, S. 153–157

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.



Ort	Datum	Unterschrift im Original