Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Studiengang Master Künstliche Intelligenz

Studienarbeit

von

Helge **Kohl**

4-Detect

Eine auf KI basierte Spielunterstützung mit AR

Bearbeitungszeitraum: von 24. November 2021
bis 2. Februar 2022

1. Prüfer: Prof. Dr.-Ing. Gerald Pirkl

Inhaltsverzeichnis

Abbildungsverzeichnis **ii**

1 Einleitung **1**
1.1 Projektidee 1
1.2 Persönliche Motivation 1

2 Integration von OpenCVSharp in Unity **2**
2.1 OpenCV plus Unity 2
2.2 OpenCVSharp 2

3 Augmented Reality **3**
3.1 Spielfeldstatus erkennen 3
3.1.1 Vorgehensweise 3
3.1.2 Erkennung der Chip-Slots 3
3.1.3 Erkennung der eingeworfenen Chips 6
3.1.4 Aufbereitung des Status zur weiteren Verarbeitung 8
3.2 Zugvorschlag darstellen 8

4 Performanceoptimierung **9**
4.1 Bilateral-Filter 9
4.2 Freigabe von Objekten 9

5 Weitere Probleme **10**
5.1 Belichtung Farberkennung 10

Literaturverzeichnis **11**

Abbildungsverzeichnis

3.1 HSV-Colormap 4
3.2 Spielfeld im Original 4
3.3 Spielfeld maskiert 4
3.4 Spielfeld nach Dilation 5
3.5 Spielfeld nach Erosion 5
3.6 Spielfeld nach Threshold 5
3.7 Spielfeld nach Kantenerkennung 5
3.8 Erkannte Löcher 6
3.9 Original 6
3.10 Erkannte Chips 6
3.11 Erkennung der gelben Chips 7
3.12 Erkennung der roten Chips 7
3.13 Überarbeitete Positionen bei Schieflage 7
3.14 Positionen nach Jennings bei Schieflage 7

Kapitel 1

Einleitung

1.1 Projektidee

Es soll ein System entwickelt werden mit dem mittels Kamera und Bilderkennung der Status eines 4-Gewinnt Spiels ermittelt werden kann. Mithilfe einer künstlichen Intelligenz soll anhand des aktuellen Spielstatus ein Zugvorschlag ermittelt werden, welche dem Nutzer helfen ein Spiel mit höherer Wahrscheinlichkeit zu gewinnen. Der Grundgedanke ist es, Spielern mit weniger Erfahrung so weit zu unterstützen, dass sie mit besseren Spielern mithalten können. Ein Anwendungsfall wäre es, wenn jemand mit jüngeren Geschwistern spielen möchte. So könnte der jüngere Spieler mit dem System unterstützt und seine Gewinnchance gesteigert werden. Der erfahrene Spieler hätte somit einen stärkeren Gegner und der unerfahrene Spieler würde nicht nur verlieren, wodurch der Spielspaß beider Spieler gesteigert wird.

1.2 Persönliche Motivation

Bei der Bearbeitung der Aufgabenstellung konnte ich feststellen, dass viel Raum für eine freie Gestaltung gegeben ist. Dadurch habe ich mich dazu entschlossen, einige für mich noch experimentelle Ansätze zu verfolgen bzw. zu erproben. Da ich im Bachelorstudiengang für Medieninformatik bereits Erfahrung mit Computer-Vision und Bilderkennung sammeln konnte, habe ich mich dazu entschlossen, drauf aufbauend, mittels Bilderkennung den Status eines 4-Gewinnt Spiels zu ermitteln und diesen zur weiteren Verarbeitung vorzubereiten.

Zusätzlich wollte ich in diesem Projekt, Fähigkeiten mit der Entwicklungsumgebung Unity aufbauen, da ich in meinem Studium bisher kaum Erfahrung mit dieser sammeln konnte.

Kapitel 2

Integration von OpenCVSharp in Unity

2.1 OpenCV plus Unity

Um die Bildverarbeitungs- und Computer Vision-Bibliothek „OpenCV“ in Unity verwenden zu können, wurde das Unity-Asset „OpenCV plus Unity“ verwendet. Diese integriert den .NET-Wrapper „OpenCVSharp“ für „OpenCV“ in Unity. Somit ist es möglich die Funktionen von „OpenCV“ in der Programmiersprache C#, welche in Unity verwendet wird, zu nutzen. Somit können Unity-Objekte in „OpenCV“-Objekte und umgekehrt, umgewandelt werden. Leider ist die Webseite des Entwicklers von „OpenCV plus Unity“ nicht mehr erreichbar, die Dokumentation von „OpenCVSharp“ jedoch schon. Da lediglich die Konvertierung der Objekte mithilfe von „OpenCV plus Unity“ nötig war, stellte dies kein Problem dar.

2.2 OpenCVSharp

„OpenCVSharp“ ist, wie in Abschnitt 2.1 beschrieben, eine Wrapper-Bibliothek von „OpenCV“ für .NET. Somit war es möglich die Bilderkennung in C# zu implementieren. Durch die sehr ausführliche Dokumentation von „OpenCVSharp“ konnten die „OpenCV“-Funktionen nahezu problemlos verwendet werden. Da ich bisher nur mit „OpenCV“ in der Programmiersprache Python gearbeitet habe hatte ich anfangs Probleme mit den zu verwendenden Datentypen, konnte dadurch aber einige Erfahrungen sammeln.

Kapitel 3

Augmented Reality

3.1 Spielfeldstatus erkennen

3.1.1 Vorgehensweise

Um den Spielfeldstatus zu ermitteln, wurde die Erkennung vorerst in „OpenCV“ in der Programmiersprache Python entwickelt, da dort die Umwandlung von Datentypen wegfällt und langes Builden des Programms wegfallen. Somit konnte deutlich schneller Entwickelt werden, da bei der Verarbeitung der Bilder sehr viele empfindliche Parameter eingestellt werden mussten. Für die Erkennung wurde der Ansatz von Jennings, o. D. verfolgt, welcher mittels Kantenerkennung die Löcher des Spielfeldes erkennt und somit die Positionen dieser erkennt und im nächsten Schritt die Farben dieser ermittelt. Meine Vorgehensweise ähnelt diesem Projekt, die Bilderkennung wurde jedoch stark abgewandelt, da die Implementierung sehr empfindlich in Bezug auf Licht und andere Umgebungsmerkmale reagiert und damit sehr instabil, bis gar nicht lauffähig ist.

3.1.2 Erkennung der Chip-Slots

Um die Felder in denen Chips liegen können, müssen zuerst einige Vorverarbeitungsschritte durchgeführt werden. Es spielen einige Faktoren, wie die Beleuchtung der Umgebung, die Ausrichtung und die Position der Lichtquelle. Wird zum Beispiel das Spielfeld direkt angeleuchtet stellen Reflektionen ein Problem dar.

Im ersten Verarbeitungsschritt wird die Umgebung durch eine Farbmaske entfernt, sodass möglichst nur das Spielfeld im Bild vorhanden ist. Hierfür wurde eine Farbmaske im HSV-Farbbereich (Hue, Saturation, Value) definiert. Die Erstellung dieser und weiterer Farbmasken wurde mit der Vorgehensweise von Kinght, 2018 durchgeführt.

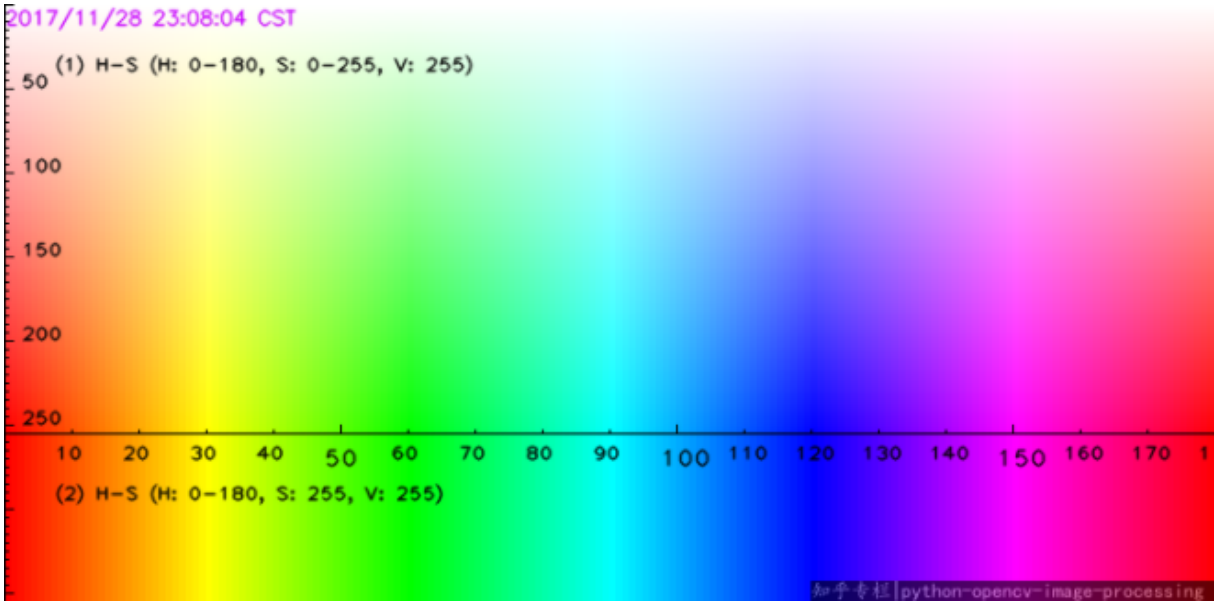

Abbildung 3.1: HSV-Colormap

Hierfür werden ein oberer und unterer Farbbereich gewählt, zwischen dem, die zu erkennenden Farbwerte liegen sollen. In Abbildung 3.1 ist die Colormap von Kinght zu sehen. Die X-Achse stellt den Hue-Wert (Farbton) dar. Die Y-Achse stellt die Saturation (Sättigung) dar. Der Value (Hellwert) wird im Bereich von 20-255 festgelegt. Für eine Blaumaske könnte man also beispielsweise die Werte (100, 100, 20) als unteren Schwellwert und (140, 255, 255) als oberen Schwellwert verwenden.

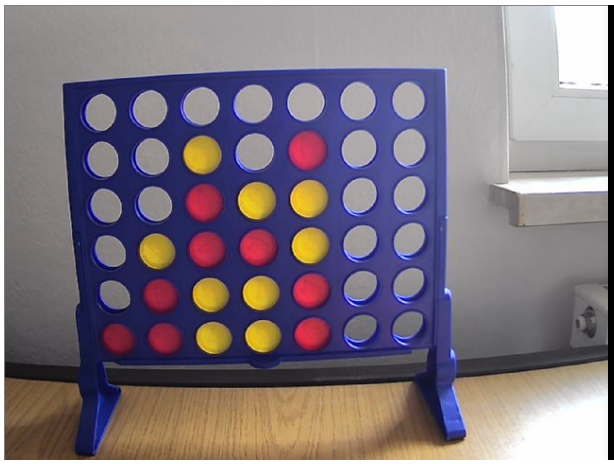

Abbildung 3.2: Spielfeld im Original

Abbildung 3.3: Spielfeld maskiert

Um nun die Konturen der Chip-Felder zu extrahieren, wird das Bild des Spielfelds in ein Graustufenbild umgewandelt. Störungen bei der Erkennung des Spielfelds werden durch den Dilate-Algorithmus entfernt. Durch ein Erode werden weitere Reflektionen entfernt und die Kanten geglättet.

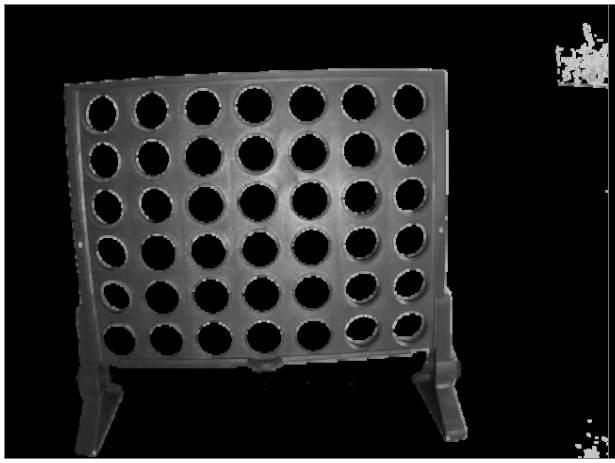

Abbildung 3.4: Spielfeld nach Dilation

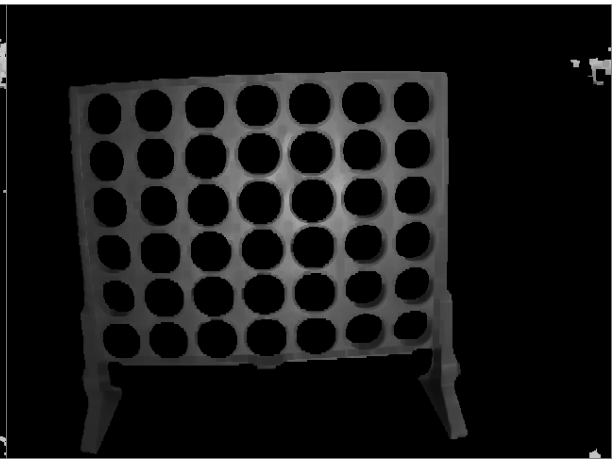

Abbildung 3.5: Spielfeld nach Erosion

Im nächsten Schritt bleibt nur die Kontur des Spielfeldes durch einen Threshold übrig. Beim Threshold werden alle nicht vollständig schwarzen Pixel in weiße Pixel umgewandelt. Durch die Canny-Kantenerkennung werden dann die Konturen, bestehend aus Rahmen und Löchern, des Spielfeldes erkannt.

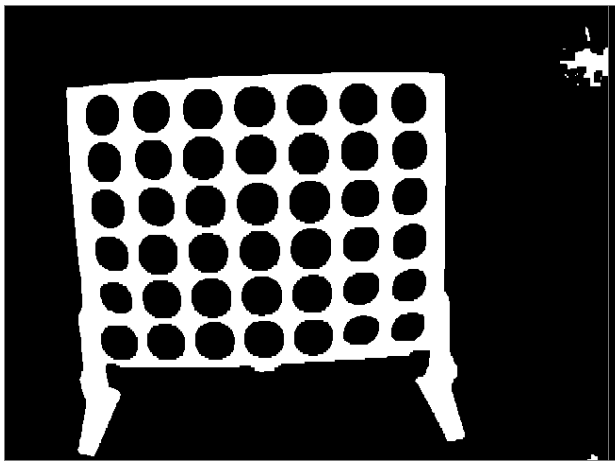

Abbildung 3.6: Spielfeld nach Threshold

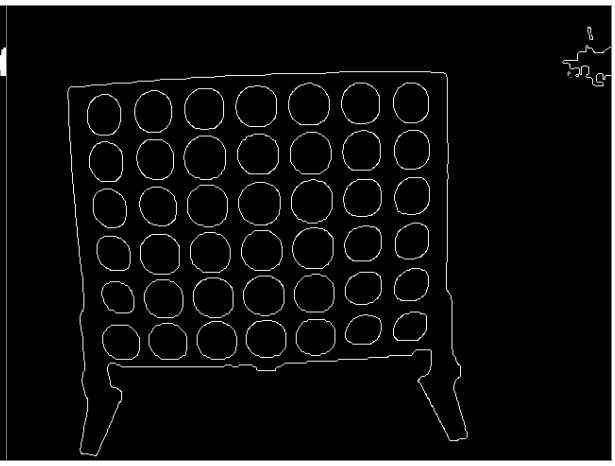

Abbildung 3.7: Spielfeld nach Kantenerkennung

Mit der „findContours“-Funktion von „OpenCV“ können im nächsten Schritt die Löcher detektiert werden. Dabei werden aneinander liegende Punkte derselben Farbe oder Intensität erkannt. Wie in Abbildung 3.7 zu sehen, wird dabei auch die Kontur des Spielfeldes selbst erkannt sowie die Störung oben rechts, deshalb ist es nötig diese herauszufiltern.

Mithilfe der „boundingRect“-Funktion wird ein Rechteck um die Kontur gezeichnet. Anhand des Rechtecks kann dann der Mittelpunkt des Loches ermittelt werden. Zuletzt legen wir alle Konturen, sowie deren Mittelpunkt und dem einschließenden Rechteck in Listen ab um diese später für die Erkennung des Spielbrettstatus wiederverwenden zu können.

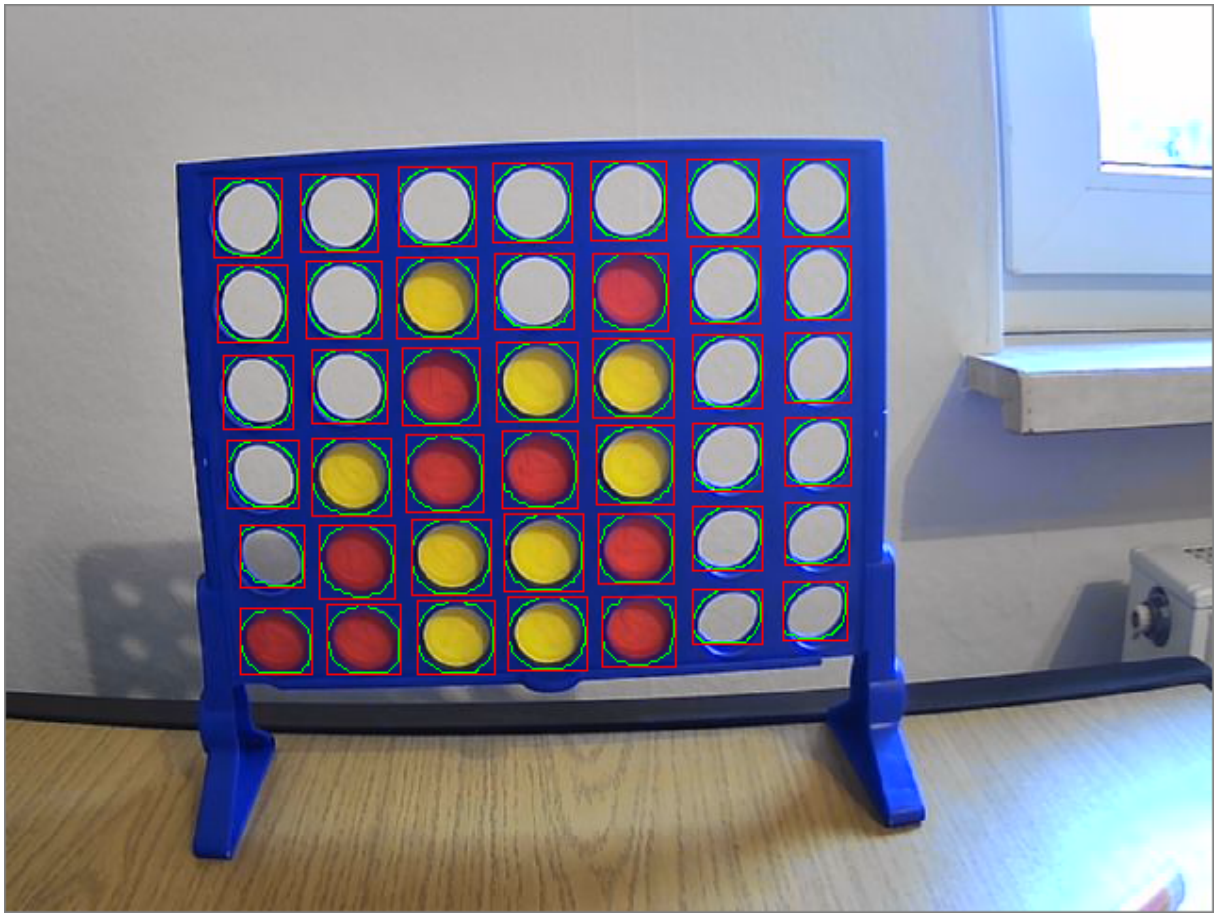

Abbildung 3.8: Erkannte Löcher

3.1.3 Erkennung der eingeworfenen Chips

Um die eingeworfenen Chips zu erkennen, wird nun, sofern alle Felder erkannt wurden, die durchschnittliche Lochgröße errechnet und die Liste der Positionen so sortiert, dass die zuerst die X-Positionen aufsteigend und dann die Y-Positionen aufsteigend folgen.

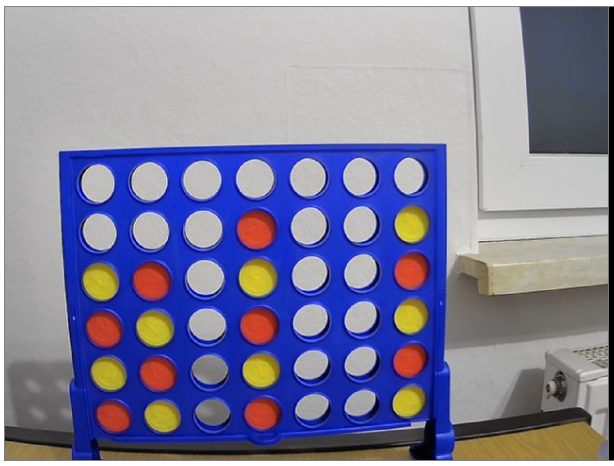

Abbildung 3.9: Original

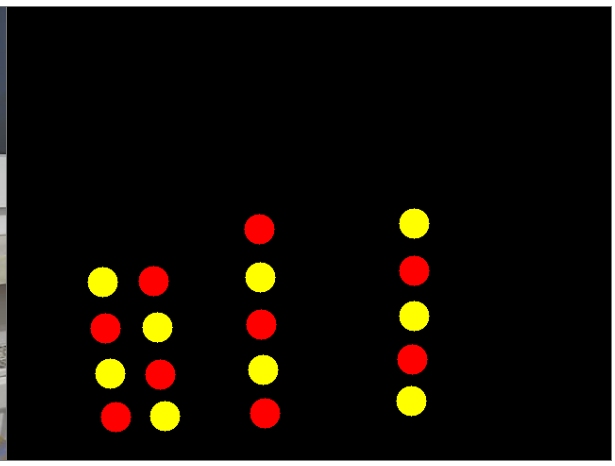

Abbildung 3.10: Erkannte Chips

Um die Chips zu unterscheiden, werden Farbmasken für Rot und Gelb definiert. Nun werden die Löcher geprüft ob etwas rotes bzw. gelbes gefunden wird, wenn ja dann wurde ein farbiger Chip erkannt. Die Ergebnisse werden in einer 2D-Matrix gespeichert um diese an die KI weiterzugeben.

Abbildung 3.11: Erkennung der gelben Chips

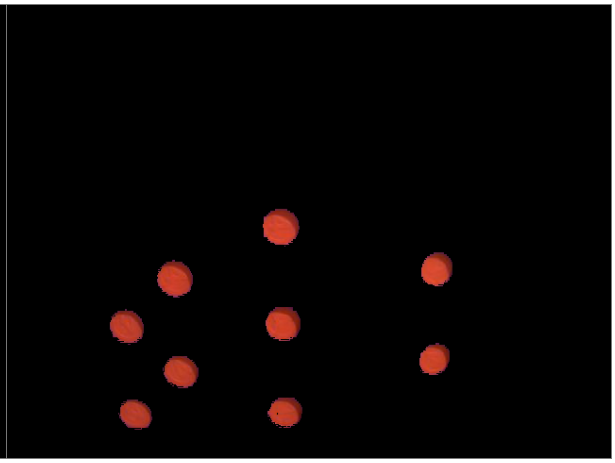

Abbildung 3.12: Erkennung der roten Chips

Die Erkennung der Chips wurde wieder auf Basis von Jennings's Algorithmus entworfen jedoch weiter angepasst. So war es bei Jennings's nicht möglich das Board schief zu halten. Jennings's Algorithmus verwendete die Position des ersten Lochs und ermittelte anhand der Breite und Höhe des Spielfeldes die weiteren Positionen. Die neue Implementierung verwendet die zuvor erstellte Positionenliste. Zusätzlich hat die Verwendung einer Kamera mit Weitwinkelobjektiv Probleme verursacht, weil die Löcher keine perfekte Reihe nach unten bilden.

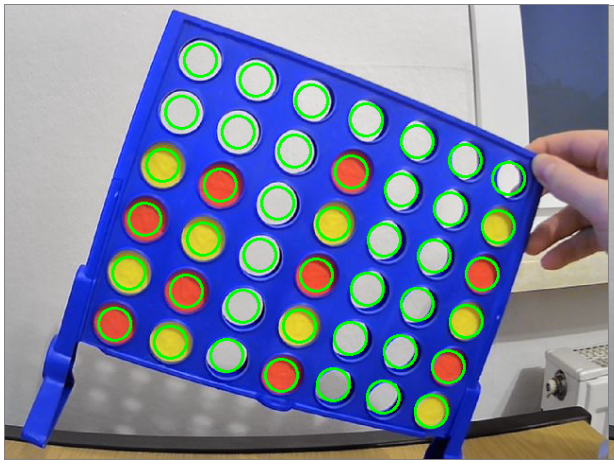

Abbildung 3.13: Überarbeitete Positionen bei Schieflage

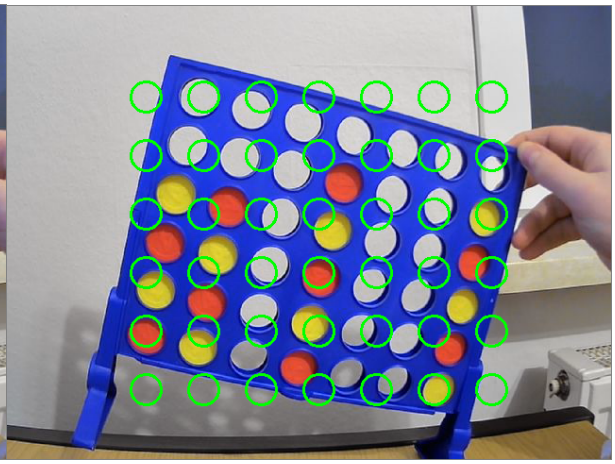

Abbildung 3.14: Positionen nach Jennings bei Schieflage

3.1.4 Aufbereitung des Status zur weiteren Verarbeitung

Für die Weiterverarbeitung wurde eine StateResult-Klasse entworfen, da nicht nur die erkannten Felder benötigt werden. Objekte der StateResult-Klasse enthalten zum einen die in Unterabschnitt 3.1.3 erwähnte 2D-Matrix des Spielfeldstatus aber auch zusätzliche Informationen wie die Positionen der einzelnen Löcher, die Positionen der Spalten, die Anzahl der eingeworfenen Chips jeder Farbe, die durchschnittliche Chipgröße sowie eine Variable zur Erkennung, ob der erkannte Status gültig ist.

3.2 Zugvorschlag darstellen

Um den Zugvorschlag darzustellen, wurde von der KI die Spalte in welche, der nächste Chip eingeworfen werden sollte angeliefert. Zunächst wird ermittelt welcher Spieler als nächstes einwerfen muss, um zu entscheiden welcher Chip dargestellt werden wird. Anhand der durchschnittlichen Chipgröße wird errechnet, wie groß der Chip sein wird. Anhand der Spalte wird nun die Position des Vorschlags errechnet und sofern die Position im Kamerabereich ist wird dieser dort abgebildet.

Zunächst wird ein schwarzes Bild erstellt, das dieselben Maße wie das Kamerabild enthält. Der Chip wird an der Stelle eingefügt, an der er auch im Originalbild sein wird. Dieses Bild wird dann auch als Maske verwendet, um die Pixel an der Stelle im Kamerabild, an der der Chip sein wird auf 0 zu setzen. Hierfür wurde die erstellte Maske gethresholded um ein Bild zu erhalten, das nur schwarze (0) und weiße (1) Pixel enthält. Von den Pixeln des Kamerabildes werden nun alle Pixel, derselben Position, der Maske entfernt. Nun kann das Bild, in dem nur der Chip zu sehen ist über das Kamerabild gelegt werden, indem man die Matrizen der Bilder und somit die Pixel addiert.

Dies war notwendig, da durch die Konvertierung von Unity-Assets mit „OpenCV plus Unity“ der Alphakanal verloren geht. Dieser wird bei transparenten Bildern dazu verwendet, um den transparenten Anteil auch tatsächlich transparent zu machen. Ohne die oben beschriebene Vorgehensweise könnte der runde Chip nicht richtig dargestellt werden, da die transparenten Pixel einen farbigen Wert erhalten.

Kapitel 4

Performanceoptimierung

4.1 Bilateral-Filter

In einer der ersten Versionen des Systems wurde zur Erkennung der Löcher noch ein Bilateralfilter verwendet. Dieser Entrauscht bzw. lässt ein Bild verschwimmen und lässt dieses glatter erscheinen. Jedoch sind starke Performanceeinbrüche durch das wiederholte Anwenden des Filters auf das Kamerabild entstanden. Somit konnten wir anfangs durch das Entfernen des Filters einen FPS-Anstieg von 2-10 FPS auf 150-200 FPS feststellen. Nach den Optimierungen aus Abschnitt 4.2 ist der Bilateralfilter zwar wieder verwendbar, jedoch wurde die Pipeline so umgebaut, dass dieser nicht weiter notwendig ist und der Ansatz mit dem Filter wurde verworfen. Wie Porikli, 2008, S. 2 in seiner Veröffentlichung erwähnt, ist der Bilateralfilter kein linearer Filter und sein Ergebnis wird nicht durch einfache Matrixmultiplikation ermittelt, was der Grund dafür ist warum er rechnerisch sehr anspruchsvoll ist.

4.2 Freigabe von Objekten

Weiter wurde festgestellt, dass bei der Verwendung von „OpenCV plus Unity“ in Unity offenbar der Garbage Collector nicht richtig funktioniert. In den ersten Versionen kam es bei einer längeren Laufzeit zu Memory Leaks und daraus resultierenden Abstürzen. Die Abstürze wurden zuerst mit der Unity-Funktion „System.UnloadUnusedAssets“behoben, welche jedoch sehr inperformant ist. Durch Deaktivieren des Garbage Collector und manuelles freigeben der Objekte konnten jedoch die FPS auf 150+ gesteigert werden.

Kapitel 5

Weitere Probleme

5.1 Belichtung Farberkennung

Durch die Verwendung von Farbmasken bei der Spielbretterkennung wurde festgestellt, dass die in den Farbmasken definierten Parameter sehr empfindlich gegenüber der Belichtung und dem Hintergrund sind. So haben wir mehrfach feststellen können das die Erkennung bei einem Projektmitglied problemlos funktioniert, während der andere keinen Chip erkennen konnte. Gerade bei schwachen Lichtquellen wurden Chips teilweise gar nicht erkannt, bei Überbelichtung und weißem Hintergrund wurden gelbe Chips gerne an leeren Löchern erkannt, da die Farbe des Hintergrunds als gelb erkannt wurde. Durch das Anpassen der Filter, wurden dann aber stark beleuchtete Chips durch die Reflektion des Lichts nicht mehr erkannt.

Die in der finalen Implementierung verwendeten Filter sind relativ stabil, solange keine Über- bzw. Unterbelichtung vorliegt. Im besten Fall sollte bei Überbelichtung kein weißer Hintergrund verwendet werden.

Problematisch ist es weiterhin, wenn der Hintergrund farbig ist. So sollte roter und gelber Hintergrund vermieden werden, da diese sonst als Chips erkannt werden könnten. Dies könnte korrigiert werden indem spezielle Chips verwendet werden würden und die Erkennung nicht auf den Farben beruhen würde.

Literatur

Jennings, M. (o. D.). ConnectFour-ComputerVisionAI. Zugriff 28. Januar 2022 unter https://github.com/Matt-Jennings-GitHub/ConnectFour-ComputerVisionAI

Kinght. (2018). Kommentar in Choosing the correct upper and lower HSV boundaries for color detection with'cv::inRange' (OpenCV). Zugriff 28. Januar 2022 unter https://stackoverflow.com/questions/10948589/choosing-the-correct-upper-and-lower-hsv-boundaries-for-color-detection-withcv

OpenCV. (o. D.). *OpenCV*. Zugriff 28. Januar 2022 unter https://opencv.org/

OpenCV plus Unity. (2019). *Paper Plane Tools*. Zugriff 28. Januar 2022 unter https://assetstore.unity.com/packages/tools/integration/opencv-plus-unity-85928#publisher

OpenCVSharp: OpenCV wrapper for .NET. (o. D.). *shimat*. Zugriff 28. Januar 2022 unter https://github.com/shimat/opencvsharp

Porikli, F. (2008). Constant time O(1) bilateral filtering. *2008 IEEE Conference on Computer Vision and Pattern Recognition*, 1–8. https://doi.org/10.1109/CVPR.2008.4587843