

Hochschule für angewandte Wissenschaften  
Fachhochschule Würzburg-Schweinfurt  
Fakultät Informatik und Wirtschaftsinformatik

## **Projektarbeit**

# **Naolympics**

**vorgelegt an der Hochschule für angewandte Wissenschaften  
Fachhochschule Würzburg-Schweinfurt in der Fakultät Informatik und  
Wirtschaftsinformatik zum Abschluss eines Studiums im Studiengang  
Informatik**

Nils Göbel  
Johannes Gehring  
Luca Klingert  
Michael Schmitt

Eingereicht am: 27. September 2023

Erstprüfer: Prof. Dr. Arndt Balzer  
Zweitprüfer: Prof. Dr. Daniel Kulesz



# **Selbstständigkeitserklärung**

Hiermit versichere ich, dass ich die vorgelegte Projektarbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

---

Würzburg, den 27. September 2023



# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>vii</b>
<b>Tabellenverzeichnis</b>	<b>ix</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Der NAO Roboter . . . . .	1
1.2 Beschreibung des Projekts . . . . .	1
1.3 Allgemeine Bedienung (JG) . . . . .	3
1.3.1 Einrichtung . . . . .	3
1.3.2 SSH-Verbindung . . . . .	3
1.3.3 Benötigte Python-Bibliotheken . . . . .	3
1.3.4 Ausführen von Skripten . . . . .	4
<b>2 Spielstanderkennung (JG)</b>	<b>7</b>
2.1 Theoretische Grundlagen . . . . .	7
2.1.1 Bildverarbeitung . . . . .	7
2.1.2 Hough-Transformation . . . . .	11
2.1.3 Border Tracing . . . . .	12
2.2 Methodik . . . . .	15
2.2.1 Vorverarbeitung . . . . .	15
2.2.2 TicTacToe . . . . .	15
2.2.3 Vier gewinnt . . . . .	17
2.2.4 Parameter . . . . .	18
2.3 Ergebnisse . . . . .	19
2.3.1 TicTacToe . . . . .	19
2.3.2 Vier gewinnt . . . . .	22
2.3.3 Präzision . . . . .	24
<b>3 Spieltaktiken (NG)</b>	<b>27</b>
3.1 Tictactoe . . . . .	27
3.1.1 Spielfeld . . . . .	27
3.1.2 Spielregeln . . . . .	27
3.1.3 Input . . . . .	27
3.1.4 Output . . . . .	27
3.1.5 Grundkonzept Gamelogic . . . . .	28
3.2 Vier Gewinnt . . . . .	28
3.2.1 Spielfeld . . . . .	28
3.2.2 Spielregeln . . . . .	28

## Inhaltsverzeichnis

3.2.3	Input . . . . .	29
3.2.4	Output . . . . .	29
3.2.5	Grundkonzept: Prioritäten . . . . .	29
<b>4</b>	<b>Movement (NG)</b>	<b>31</b>
4.1	Herausforderungen . . . . .	31
4.2	Ideen und Lösungsansätze . . . . .	32
4.2.1	Touchscreenfähiger Finger . . . . .	32
4.2.2	Arm Movement . . . . .	33
4.2.3	Positionierung NAO und Tablet . . . . .	33
4.3	Lösung: . . . . .	35
4.3.1	Touchscreenfähiger Finger . . . . .	35
4.3.2	Positionierung des Tablets zum NAO . . . . .	36
4.3.3	Arm Movement über fixe Punkte . . . . .	40
<b>5</b>	<b>Spielablauf (NG, JG)</b>	<b>43</b>
5.1	Spielablauf . . . . .	43
5.1.1	Menüführung und Spielauswahl . . . . .	43
<b>6</b>	<b>Flutter app (MS, LK)</b>	<b>47</b>
6.1	Einrichten von Flutter und Erstellen einer APK . . . . .	47
6.1.1	Flutter Installation . . . . .	47
6.1.2	Flutter Prüfung und Konfiguration . . . . .	47
6.1.3	APK erstellen . . . . .	47
6.2	Singleplayer . . . . .	47
6.2.1	Tic Tac Toe . . . . .	47
6.2.2	Vier gewinnt . . . . .	48
6.3	Mulitplayer . . . . .	48
6.3.1	Verbindungsauflbau und Datenübertragung . . . . .	48
6.3.2	Aufbau der Host-Client Verbindung . . . . .	49
6.3.3	Navigator . . . . .	49
6.3.4	Tic Tac Toe . . . . .	50
6.3.5	Vier gewinnt . . . . .	50
<b>7</b>	<b>Zusammenfassung und Probleme</b>	<b>51</b>
	<b>Literaturverzeichnis</b>	<b>53</b>

# Abbildungsverzeichnis

2.1	Beispiel Weichzeichnen . . . . .	9
2.2	Parameterraum Linie Sinusoid . . . . .	11
2.3	Parameterraum Kreise . . . . .	12
2.4	<i>TicTacToe</i> -Spielfeld aus Sicht des <i>NAO</i> . . . . .	19
2.5	<i>TicTacToe</i> Kantenbild . . . . .	20
2.6	<i>TicTacToe</i> Konturen . . . . .	20
2.7	Ausgewertetes <i>TicTacToe</i> -Spielfeld . . . . .	21
2.8	<i>Vier gewinnt</i> -Spielfeld aus Sicht des <i>NAO</i> . . . . .	22
2.9	<i>Vier gewinnt</i> Kantenbild . . . . .	22
2.10	<i>Vier gewinnt</i> Kreise . . . . .	23
2.11	Ausgewertetes <i>Vier gewinnt</i> -Spielfeld . . . . .	23
4.1	NAO vor Tablet . . . . .	31
4.2	Tabletstift . . . . .	32
4.3	Mobile Gaming Finger Sleeve . . . . .	32
4.4	Benötigte Ausstattung . . . . .	35
4.5	Aluminium Folie, Schere und Klebemittel . . . . .	35
4.6	Tripod . . . . .	36
4.7	Höhe Tablet . . . . .	36
4.8	Z-Achse falsch eingestellt . . . . .	37
4.9	Z-Achse richtig eingestellt . . . . .	37
4.10	X-Achse Startposition . . . . .	38
4.11	X-Achse korrekt (seitliche Ansicht) . . . . .	38
4.12	X-Achse korrekt (frontale Ansicht) . . . . .	38
4.13	Y-Achse Startposition . . . . .	39
4.14	Y-Achse falsch eingestellt . . . . .	39
4.15	Y-Achse korrekt eingestellt . . . . .	39
4.16	Grundposition . . . . .	40
5.1	Allgemeines Beispiel für die <i>TextToSpeech</i> -Funktion . . . . .	43
5.2	Allgemeines Beispiel für die <i>Touch</i> -Funktion . . . . .	43



# Tabellenverzeichnis

2.1	Parameter für die Spielstanderkennung von <i>TicTacToe</i> und <i>Vier gewinnt</i> . . .	18
2.2	Ergebnisse für die Präzision der Spielstanderkennung von <i>TicTacToe</i> und <i>Vier gewinnt</i> anhand von jeweils drei Stichproben . . . . .	25



# 1 Einleitung

## 1.1 Der NAO Roboter

Die NAO Roboter von Aldebaran Robotics sind humanoide Roboter, die gerade in der Forschung von Robotik und Informatik Einzug gefunden haben. Die etwa 58 Zentimeter großen Roboter besitzen neben den üblichen Gelenken, wie man sie auch beim Menschen findet (Schulter, Ellbogen, Knie etc.), drei Finger, zwei Kameras in Stirn und Mund sowie mehrere Annäherungssensoren. Im Laufe der Jahre haben sich verschiedene internationale Projekte um die Roboter entwickelt, darunter der RoboCup, bei dem Forschungsteams von Hochschulen rund um die Welt im Fußball gegeneinander antreten. Durch die humanoide Gestalt und die große Auswahl an "Sinnesorganen" können viele verschiedene Anwendungsfälle und Projekte mit den Robotern realisiert werden, so auch das Spielen von einfachen Spielen. In diesem Projekt soll gezeigt werden, dass die *NAOs* eigenständig auf Tablets Spiele erkennen, Spielzüge berechnen und die Touchscreens bedienen können.

## 1.2 Beschreibung des Projekts

Die Grundidee des Projekts umfasst folgende Punkte:

- Logikspiele für zwei Personen auf Tablet
- *NAO vs. Mensch*
- *NAO vs. NAO*
- Mögliche Spielideen: TicTacToe, Vier Gewinnt, Mastermind, Schiffe versenken etc.
- Spielfeld über Bilderkennung (OpenCV)
- CV und Spiellogik soll direkt auf NAO laufen

Anhand dieser allgemeinen Projektidee wurden folgende Anforderungen gestellt, die das Projekt erfüllen soll:

- *NAO* soll eigenständig Klicks auf Touchscreens tätigen können, bspw. durch einen Touch-”Fingerhut” aus entsprechendem Material
- Eine entsprechende Spieleapp für die verwendeten *Android*-Tablets soll entwickelt werden
- Die App soll sowohl lokalen Multiplayer unterstützen, als auch zwischen zwei Geräten in nächster Nähe (bspw. im selben Netzwerk oder via Bluetooth)

## 1 Einleitung

- NAO soll so vor dem Tablet positioniert werden, dass er das Spielen beginnen kann
- NAO soll das jeweilige Spiel und das Spielfeld auf Tablet erkennen können
- *NAO* kann einige Spiele, wie bspw. *Vier gewinnt* erkennen und spielen
- *NAO* besitzt eigene Spieltaktiken und führt diese aus
- *NAO* kann gegen einen anderen *NAO* spielen
- *NAO* kann sich mit seinem Gegenspieler (Mensch oder andere NAO) einigen wer anfängt
- *NAO* erkennt, dass er gewonnen hat und jubelt
- NAO soll Spielsituation auswerten und mit nächstem Zug reagieren können
- NAO soll die richtige Stelle auf dem Tablet finden und drücken können
- NAO soll auf Fehler reagieren können (z.B.: Wenn der Roboter in volle Zeile bei 4 gewinnt wirft; z.B. mit Spielabbruch oder Rücknahme des Zugs)

Verwendet wurden Tablets des Modells Galaxy Tab S7.

## 1.3 Allgemeine Bedienung (JG)

### 1.3.1 Einrichtung

Zur Einrichtung der Python SDK sowie der Software Choregraphe verweisen wir auf den Installation Guide des Herstellers Aldebaran.

Für naoqi Python SDK Version 2.1: [15]

Choregraphe für Version 2.1: [13]

Für naoqi Version 2.8: [16]

Choregraphe für Version 2.8: [14]

Problematisch stellte sich hierbei die Installation von Choregraphe und der Python SDK auf macOS (M2 Max, macOS 13) heraus, daher wurde dort eine virtuelle Maschine mit Parallels und Windows 11 ARM eingerichtet.

### 1.3.2 SSH-Verbindung

SSH-Verbindungen zum jeweiligen Roboter lassen sich mittels Terminal-SSH oder einem Client wie Bitvise oder PuTTY realisieren, um Zugang auf die Linux Kommandozeile des NAO zu bekommen. Die Zugangsdaten für den Nutzer *nao* sind standardmäßig:

```
Username: nao
Password: nao
Port: 22
```

Hiermit lassen sich Dateien verschieben, Skripte ausführen usw. Falls keine entsprechende Berechtigungen vorliegen kann mittels bash-Befehl

```
su
```

und Passwort "root" zum Nutzer *root* und hiermit *super user*-Berechtigungen erlangt werden. Abweichungen können je nach *NAOqi*-Version sowie Roboter vorkommen.

### 1.3.3 Benötigte Python-Bibliotheken

Für den *Standalone*-Betrieb des Spiels auf dem Roboter werden folgende *Packages* benötigt:

1. NumPy
2. NaoQi
3. OpenCV

Die ersten beiden aufgelisteten Bibliotheken sind auf den *NAOs* mit *NaoQiVersion < 2.2.0* vorinstalliert. Diese Roboter V.5) arbeiten lediglich mit Python 2.7. *OpenCV* konnte "installiert" werden, indem der entpackte Ordner der passenden Version (4.3.0.32) in den

```
/usr/lib/python2.7/site-packages
```

## 1 Einleitung

Ordner gelegt wurde. Hierfür wurde zunächst mit einem *SFTP*-Client wie Bitvise oder FileZilla das entpackte Package in das *home* Verzeichnis transferiert und dann verschoben.

```
mv *opencv-folder* /usr/lib/python2.7/site-packages
```

Bei Robotern V.6 mit Version 2.8 mussten keine Packages zuvor installiert werden, deren Linux Distribution ist auch durch *read-only-filesystem* stärker vor Eingriffen geschützt. Diese unterstützen neben Python 2.7.15 zusätzlich die Version 3.5.6. Es muss beachtet werden, dass es zu Problemen zwischen unterschiedlichen OpenCV-Versionen auf verschiedenen Roboter-Versionen kommen kann, welche adressiert werden müssen. Ein Beispiel aus *vision.py*:

```
if cv2.__version__[0] == "3":  
    _, allContours, hierarchy =  
        cv2.findContours(copy, cv2.RETR_TREE,  
                          cv2.CHAIN_APPROX_SIMPLE)  
else:  
    allContours, hierarchy =  
        cv2.findContours(copy, cv2.RETR_TREE,  
                          cv2.CHAIN_APPROX_SIMPLE)
```

Hier wird geprüft, ob eine *OpenCV*-Version 3.x.x vorliegt, in welcher drei statt zwei Parameter von der Funktion *findContours* zurückgegeben werden. In den Versionen 2.x.x und 4.x.x werden lediglich zwei zurückgegeben.

### 1.3.4 Ausführen von Skripten

Python-Skripte (Dateiendung *.py*) lassen sich über die Kommandozeile der SSH-Verbindung wie folgt ausführen:

```
python example.py
```

Um den Roboter bei Neustart automatisch Skripte bzw. *Executables* ausführen zu lassen, muss ein entsprechender Eintrag in der *autoload.ini* Datei unter folgendem Pfad hinterlegt werden[17]:

```
/home/nao/naoqi/preferences/autoload.ini
```

Entweder können hierbei über SFTP Dateien auf dem Host-System generiert und auf dem Roboter ersetzt, oder direkt auf dem *NAO* über den *nano*-Editor bearbeitet werden. *VI* bzw. *VIM* konnte hierfür im Rahmen des Projekts nicht ausgeführt werden. Beispiel für ein Python-Skript:

```
[python]  
/home/nao/reacting_to_events.py
```

Es ist außerdem möglich, Dateien direkt in einem geeigneten SFTP-Client im Roboter-Dateisystem mit einem Texteditor zu bearbeiten. Der Programm-Eintrittspunkt dieses Projekts ist die Datei *main.py*, wobei der gesamte *naolympics/nao* Ordner in seiner Ordnerstruktur mit *vision* und *movement* Ordner mit auf dem Roboter liegen muss, damit die Importe richtig funktionieren. Es wurde über die *argparse*-Bibliothek die Option hinzugefügt, über Kommandozeilenargumente die IP-Adresse und den Port des Roboters einzufügen. Über das *-h* Flag lässt sich eine Auflistung der Argumente ausgeben:

Eingabe:

```
python main.py -h
```

Ausgabe:

```
usage: main.py [-h] [-i IP] [-p PORT]
```

optional arguments:

-h, --help	show this help message and exit
-i IP, --ip IP	robot IP address
-p PORT, --port PORT	robot Port

Beispiel, um *main.py* auf Roboter 10.30.4.13/9559 zu starten:

```
python main.py --ip 10.30.4.13 --port 9559
```

oder:

```
python main.py -i 10.30.4.13 -p 9559
```

Wenn keine Argumente mitgegeben wurden, wird standardmäßig die IP 10.30.4.13 und der Port 9559 angesprochen.



# 2 Spielstanderkennung (JG)

## 2.1 Theoretische Grundlagen

Wichtiger Teil des Spiel- bzw. Programmablaufs ist die Erkennung des gespielten Spiels und im Anschluss die kontinuierliche Auswertung des Spielstands. Dies soll in diesem Projekt rein über die Stirnkamera der Roboter sowie *Computer Vision (CV)* Algorithmen erfolgen. Eine andere Möglichkeit wäre, die Roboter untereinander über das Netzwerk, bspw. HTTP, über getätigte Spielzüge und damit den Spielstand kommunizieren zu lassen. Problem hierbei ist einerseits, dass man hier ggf. nicht komplett auf *CV* verzichten kann, um z.B. das Spiel zu Beginn zu erkennen, andererseits, dass bei *Mensch vs. NAO* logischerweise eine andere andere Form der Kommunikation gewählt werden müsste, etwa über die *TextToSpeech*-Funktion des *NAO*. Eine zweite Möglichkeit, dieses Problem zu lösen, wäre, wenn die Spiele- App die Kommunikation mit den Robotern übernimmt, oder der menschliche Spieler seinen Zug zusätzlich über einen zweiten Computer einträgt und so mitteilt. Dies würde jedoch gleichzeitig den Gedanken verletzen, dass die Roboter völlig selbstständig spielt und handelt, wozu eben nicht nur die Bedienung des Touchscreens gehören, sondern auch Dinge wie die Spielstanderkennung und -auswertung. Anders könnte man all diese Aufgaben an ein anderes Gerät "outsourcen" und dem *NAO* nur mitteilen, wie er sich zu bewegen hat, was einer Marionette gleicht. Daher die Entscheidung, die Erkennung über *CV* Algorithmen zu realisieren. Hierfür wurden weitestgehend Standardverfahren der *Computer Vision* verwendet, welche nachfolgend theoretisch erörtert werden. Für die Implementierung wurde die Open Source Bibliothek OpenCV verwendet[2].

### 2.1.1 Bildverarbeitung

Unter realen Bedingungen, wie etwa inkonsistenter Beleuchtung, treten einige Probleme auf, welche die Weiterverarbeitung von Bildern deutlich erschweren können. Bei schlechten Lichtbedingungen beispielsweise neigen Aufnahmen zu viel Rauschen (eng. *Noise*) im Bild, also Pixeln mit fehlerhaften Bildwerten[?]. Ein anderes Problem, was aber im Kontext dieses Projektes keine nennenswerte Rolle spielt, ist sog. *Bewegungsunschärfe*, bei der sich das Objekt während der Aufnahme bewegt und so unscharf und "verschmiert" auf dem Bild erscheint[5, S.43]. Bei Bildschirmaufnahmen und computergrafisch gerenderten Szenen treten solche Effekte logischerweise weniger bis gar nicht auf. Ziel ist es also, ein Bild so zu verarbeiten, dass in nachfolgenden Anwendungen, wie etwa Objekterkennung diese negativen Effekte möglichst wenig ins Gewicht fallen. Gängige Vorgehensweise ist hierbei die Kombination von Weichzeichnen (eng. *smoothing*) und ein Kantenerkennungsalgorithmus. Wir erhalten daraus ein verarbeitetes Bild  $J$ .

## Weichzeichnen

Weichzeichnen (engl. *smoothing*) zielt darauf ab, "Ausreißer", also Rauschen im Bild  $I$  zu reduzieren[5]. Hierfür wird allermeistens eine lokale Operation in einem  $(2k + 1) \times (2k + 1)$ ,  $k \in \mathbb{N}$  großen Fenster  $W$  durchgeführt, wobei jeder Pixel des Bildes besucht und als Referenzpixel im Zentrum des Fensters sitzend betrachtet wird. Innerhalb dieses Fensters wird dann eine lokale Operation durchgeführt, und ein neuer Pixelwert  $J(p)$  für das Referenzpixel berechnet wird, wobei nur die Pixel innerhalb des Fensters einbezogen werden. Die Operation kann entweder *linear* oder *nicht-linear* erfolgen. Bei einer *nicht-linearen lokalen Operation* wird ein Algorithmus auf die Pixel angewendet, welcher keiner Faltung entspricht, beispielsweise eine Maximum-Operation[5, S.46].

$$J(p) = \max\{I(x + i, y + j) : -k \leq i \leq k \wedge -k \leq j \leq k\} \quad (2.1)$$

Eine *lineare lokale Operation* lässt sich durch eine diskrete Faltung des *gleitenden Fensters*  $W$  (auch Kernel oder Maske genannt) mit  $I$  beschreiben. Für den Pixel  $p = (x, y)$  lautet die Formel wie folgt:

$$J(p) = W_p * I(p) = \frac{1}{S} \sum_{i=-k}^{i=k} \sum_{j=-k}^{j=k} w_{i,j} \cdot I(x + i, y + j) \quad (2.2)$$

Mit Gewichten  $w_{i,j} \in W$  und Skalierungsfaktor  $S$  (i.d.R.  $\sum w_{i,j}$ )[5, S.47].

Beliebte Kernel für das Weichzeichen sind der Mittelwert-Filter, auch Box-Filter genannt, sowie der Gaussfilter. Ersterer berechnet innerhalb von  $W$  den Mittelwert und somit auf das ganze Bild gesehen den gleitenden Mittelwert. Der Gaussfilter approximiert eine zweidimensionale Gaussfunktion, üblicherweise mit  $\mu_x = 0, \mu_y = 0$ , sodass der Peak der Funktion im Zentrum des Fensters liegt[5, S.57f.].

$$\begin{array}{c} \left[ \begin{array}{ccc} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{array} \right] \bar{9} \\ \text{Box-Filter}(k = 1) \end{array} \quad \begin{array}{c} \left[ \begin{array}{ccccc} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 27 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{array} \right] \overline{273} \\ \text{Gaussfilter}(k = 2, \sigma = 1) \end{array} \quad (2.3)$$

## Kantenerkennung

Eine Kante in einem Bild wird im sog. *Stufenkantenmodell* als rapider Intensitätsunterschied innerhalb eines kleinen räumlichen Bereichs definiert. Hierbei muss man jedoch beachten, dass das gleiche auch für Rauschen gilt. Kanten bzw. Kantenbilder (engl. *edge maps*) können den Inhalt eines Bildes einfangen und simplifizieren, indem nur die Konturen der Objekte behalten werden und nicht-Kanten verworfen werden. So lassen sich auch Umwelteinflüsse wie wechselnde Beleuchtung reduzieren bis eliminieren[5, S.10f.] Viele Kantenerkennungsalgorithmen basieren auf dem Stufenkantenmodell (*Sobel*, *Canny*, *Laplace*,

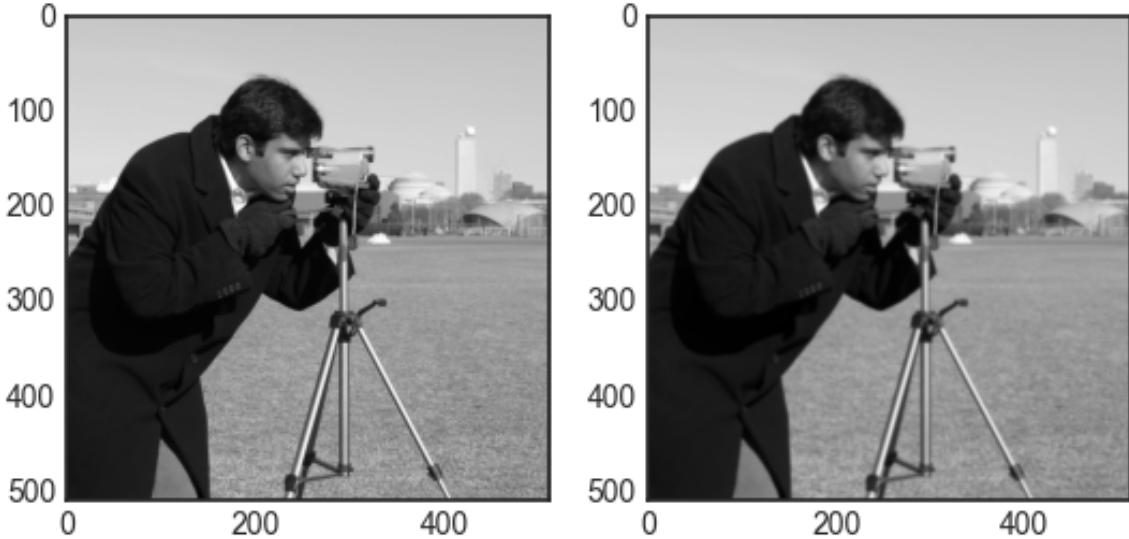


Abb. 2.1: Links: Unbearbeitetes Bild, Rechts: Weichgezeichnetes Bild (Gaussfilter,  $k = 2, \sigma = 1$ )

*Meer-Georgescu), um nur die Bildbereiche zu erfassen, welche eine solche Stufenkante repräsentieren, und nutzen hierfür den approximierten Gradienten des Pixels  $p = (x, y)$ . Der Gradient*

$$\nabla I = \mathbf{grad} I = \left[ \frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right]^T \approx [I_x, I_y] \quad (2.4)$$

welcher die approximierten ersten Ableitungen nach  $x$  und  $y$ ,  $I_x \approx \frac{\partial I}{\partial x}$ ,  $I_y \approx \frac{\partial I}{\partial y}$  enthält. Diese lassen sich mit verschiedenen Kerneln ermitteln, beispielsweise dem Sobel-Operator:

$$\begin{array}{c} \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \\ \text{Sobel-Op. } x\text{-Richtung}(k=1) \end{array} \quad \begin{array}{c} \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \\ \text{Sobel-Op. } y\text{-Richtung}(k=1) \end{array} \quad (2.5)$$

Wichtige Parameter für Kantenerkennung mit Gradienten sind die Länge bzw. der Betrag des Gradienten  $g(p)$  sowie dessen Richtung  $\Theta(p)$ :

$$g(p) = \|\mathbf{grad} I(p)\|_2 = \sqrt{I_x^2 + I_y^2} \quad (2.6)$$

$$\Theta(p) = \text{atan2}(I_y, I_x) \quad (2.7)$$

Ein weiteres Modell für die Definition einer Kante, hier nicht näher besprochen, in einem Bild ist das *Phasenkongruenzmodell*, beispielsweise im *Kovesi*-Algorithmus[5, S.79ff.].

**Canny-Algorithmus** Der Canny-Algorithmus verbindet die bereits vorgestellten Bildverarbeitungskonzepte zu einem robusten Kantenerkennungsalgorithmus, wobei er ein binäres Kantenbild erzeugt. Hierfür nutzt der Algorithmus den Gaussfilter, den Gradienten, eine sog. *non-maxima suppression* und einen Hysterese-basierten *edge-following* Ansatz.

## 2 Spielstanderkennung (JG)

**Data :** Image  $I$ , Lower Threshold  $T_{low}$ , Upper Threshold  $T_{high}$ ,  $\sigma > 0$   
**Result :** edge map  $J$

```

1 Smooth  $I$  with Gaussian filter  $G_\sigma$ ;
2 Calculate approx. gradient  $\text{grad}I(p)$  as well as  $\Theta(p)$  and  $g(p)$ ;
3 Round  $\Theta(p)$  to multiples of  $\frac{\pi}{4}$ ;
4 non-maxima suppression:
5 for every pixel  $p$  do
6   compare  $g(p)$  with the two neighbors'  $g(q), g(q')$  in direction  $\Theta(p)$ ;
7   if  $g(p) \leq g(q)$  or  $g(p) \leq g(q')$  then
8      $| g(p) = 0$ 
9   end
10 end
11 edge-following:
12 for every pixel  $p$  not visited do
13   if  $g(p) \geq T_{high}$  then
14     Mark  $p$  as edge pixel;
15     Visit all 8-adjacent neighbors  $q$  of  $p$ ;
16     if  $g(q) > T_{low}$  then
17       Mark  $q$  as edge pixel;
18        $q = p$ ;
19       Go to 15;
20     end
21   end
22 end
```

**Algorithmus 1 :** Canny-Algorithmus nach [5, S.64]

Bei der *non-maxima suppression* vergleichen wir den gerade besuchten Pixel  $p = (x, y)$  mit seinen direkten Nachbarn in der gerundeten Richtung des Gradienten  $\Theta(p)$ . Wenn beispielsweise  $\Theta(p) = \frac{\pi}{2}$ , so zeigt der Gradient in die vertikale Richtung und wir vergleichen  $g(p) = g(x, y)$  mit  $g(x, y - 1)$  und  $g(x, y + 1)$ .

Beim *edge-following* wenden wir einen Hysterese-Ansatz an, indem wir die "Pfade" an Pixeln verfolgen, deren direkte Nachbarn einen der beiden Schwellenwerte  $T_{low}$  oder  $T_{high}$  überschreiten. Man erhält durch den Canny-Algorithmus somit kein vollständig binärisiertes Bild, aber durch die *non-maxima suppression* werden lediglich die relevanten Kantenpixel in  $J$  erhalten[5, S.64].

**Morphologische Operationen** OpenCV spricht bei einer lokalen Minimum-Operation von *erosion* und bei einer lokalen Maximum-Operation von *dilation*. Diese können dabei helfen, Kanten in Kantenbildern zu verbreitern oder zu schmälern und somit die Konturen zu "schärfen "[1].

## 2.1.2 Hough-Transformation

Die Hough-Transformation ist eine einfache und zuverlässige Methode, Linien und Kreise in Bildern zu erkennen. Hierfür wird für das Bild  $I$  im *Bildraum* ( $(x, y)$ -Koordinaten) ein *Parameterraum* aufgespannt. Die Art und Anzahl der Parameter wird dabei von der zu erkennenden Geometrie bestimmt:

**Linien** Linien werden durch die Formel

$$y = ax + b \quad (2.8)$$

parametrisiert. Betrachtet man nun einen Punkt bzw. ein Pixel  $p$  auf einer Linie, so erhält man:

$$y_p = ax_p + b \quad (2.9)$$

$$b = -ax_p + y_p \quad (2.10)$$

Eine weitere Möglichkeit, Linien zu parametrisieren, welche für Bilder besser geeignet ist, da sie die das Intervall der Parameter begrenzt, ist über die sinusoidale Darstellung:

$$0 = x \sin \Theta - y \cos \Theta + \rho \quad (2.11)$$

mit dem Winkel zwischen  $x$ -Achse und Linie  $\Theta$  und dem Abstand  $\rho$  zwischen Pixel  $p = (x, y)$  und dem Ursprung  $(0, 0)$ .

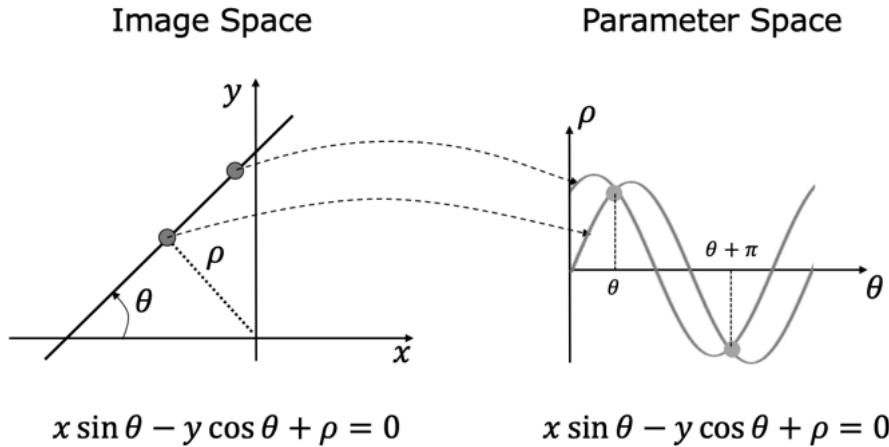


Abb. 2.2: Linie als Parameter von Sinusoiden dargestellt [6]

Es lässt sich erkennen, dass eine Linie im Parameterraum  $(a, b)$  bzw.  $(\Theta, \rho)$  alle Linien im Bildraum darstellt, die den Pixel  $p$  schneiden[6].

**Kreise** Kreise lassen sich durch die Formel

$$(x_p - a)^2 + (y_p - b)^2 = r^2 \quad (2.12)$$

beschreiben, mit dem Mittelpunkt  $(a, b)$  und Radius  $r$ . Somit ergibt sich bei bekanntem Radius  $r$  der Parameterraum  $(a, b)$  und für unbekannte Radii  $(a, b, r)$ .

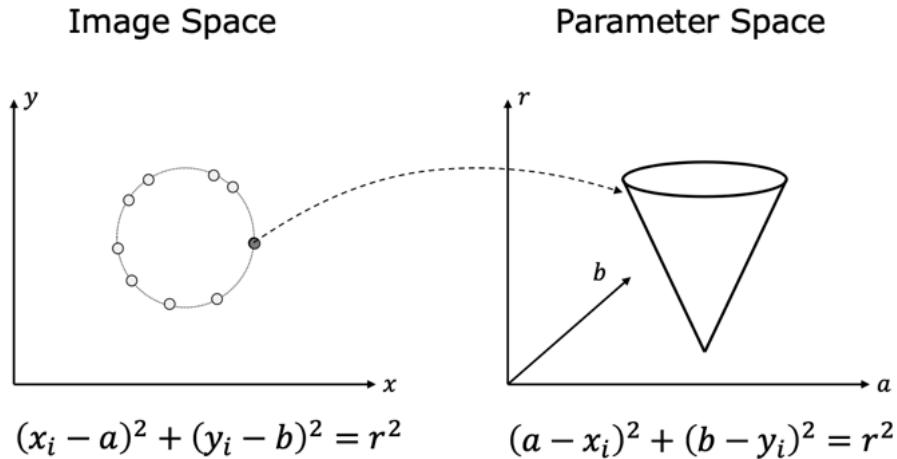


Abb. 2.3: Kreiserkennung mit unbekanntem Radius im Parameterraum dargestellt [6]

Jeder Kreis bzw. Schicht des mit  $r$  wachsenden Kegels, wie in 2.3 zu sehen, im Parameterraum (je nach dem ob  $r$  bekannt ist) ergibt somit einen Kreis im Bildraum, welcher den Punkt  $p$  schneidet[6].

**Akkumulator-Array** Mit diesem Vorwissen über Kreise und Linien lässt sich nun eine Methodik aufstellen, wie man Linien und Kreise in Bildern erkennt. Punkte auf einer Geraden bzw. Kreis in einem Bild werden alle als eigene Linie bzw. Kreis im Parameterraum dargestellt. Diese werden sich in dem Punkt im Parameterraum schneiden, welcher die Linie oder den Kreis im Bildraum darstellt. Wir diskretisieren nun den Parameterraum  $(a, b)$  (und ggf.  $r$ ) in geeigneter Weise und erzeugen ein zwei- bzw. dreidimensionales Array mit entsprechender Größe, Akkumulator-Array  $A(a, b)$  genannt, welches den diskreten Parameterraum repräsentiert und mit Zellen mit dem Wert 0 initialisiert wird. Jeder Punkt im Bildraum ist eine Linie/Kreis im Parameterraum und verläuft diese durch die Zelle in  $A$ , so wird der Wert der Zelle inkrementiert. Wenn man nun für jeden Punkt  $p$  in  $I$  (üblicherweise ist  $p$  ein Kantenpixel in einem Kantenbild) eine Linie/Kreis in  $A$  "einträgt", so ergeben die Zellen in  $A$  mit den höchsten Werten die Parameter der Linien/Kreise in  $I$ . Es ist üblich, einen Schwellenwert  $T$  für die Entscheidung Linie oder keine Linie heranzuziehen[6] [5, S.121ff.].

## 2.1.3 Border Tracing

*Border Tracing* oder – *Following* beschreibt eine fundamentale Technik der Analyse von Binärbildern, bei der man miteinander verbundene, "1"-Komponenten ermittelt und somit topologische Eigenschaften des Bildes analysiert. Hierbei erhält man meist die Konturen von Objekten im Bild. Es ist weiterhin möglich bzw. erwünscht, auch Sub-Konturen, also Grenzen innerhalb von Grenzen zu ermitteln. Die Anwendungsbereiche liegen hier bei Bild- bzw. Objekterkennung, aber auch Kompressionsmethoden[18]. Ein Ansatz hierfür ist das Folgen von Grenz-Pixeln in einer definierten Nachbarschaft, z.B. Vier (engl. 4-adjacency), sowie einer Reihenfolge für das Folgen (Uhrzeigersinn oder dagegen). Diesen Ansatz nutzt beispielsweise der Voss-Algorithmus, wie hier beschrieben[5, S.115f.]. Der

## 2.1 Theoretische Grundlagen

von *OpenCV* verwendete Algorithmus ist der von *Suzuki et al.* vorgeschlagene Ansatz, welcher Hierarchien zwischen Konturen berücksichtigt. Eine grafische Repräsentation eines Iterationsschrittes des *Suzuki*-Algorithmuses findet sich hier [4].

## 2 Spielstanderkennung (JG)

**Data :** Binary Image  $I$

**Result :** Hierarchical structured contours  $C$

```

1 Set newly found border NBD = 1;
2 Step 1:
3 Scan  $I$  from left to right, top to bottom:
4 for row  $i$  in  $I$  do
5   for column  $J$  in  $I$  do
6     if  $I(i, j - 1) == 1$  and  $I(i, j) == 0$  then
7       Increment NBD;
8       Set  $(i_2, j_2)$  as  $(i, j - 1)$ ;
9     else if  $I(i, j) \geq 1$  and  $I(i, j + 1) == 0$  then
10      Increment NBD;
11      Set  $(i_2, j_2)$  as  $(i, j + 1)$ ;
12      if  $I(i, j) > 1$  then
13        | Set Last NBD LNDB = 1;
14      end
15    else
16      | Go to Step 3;
17  end
18 end

```

**19 Step 2:**

```

20 1. Start from  $(i_2, j_2)$ , look around clockwise in the pixel neighborhood;
21 if Non-zero pixel in neighborhood found then
22   | Denote it as  $(i_1, j_1)$ ;
23 else if No non-zero pixel found then
24   | Set  $I(i, j) = -NBD$ ;
25 2. Set  $(i_2, j_2) = (i_1, j_1)$  and  $(i_3, j_3) = (i, j)$ ;
26 3. Start from the next element of pixel  $(i_2, j_2)$  in counterclockwise order, traverse
   the neighborhood of  $(i_3, j_3)$ 
27 Find the first non-zero pixel and set it to  $(i_4, j_4)$ ;
28 4. Change the value of current pixel  $(i_3, j_3)$ :
29 if  $I(i_3, j_3 + 1)$  is a zero-pixel belonging to region outside the boundary then
30   | Set current pixel value to  $-NBD$ ;
31 else if  $I(i_3, j_3 + 1)$  is not a zero-pixel and current pixel value = 1 then
32   | Set current pixel value to  $NBD$ ;
33 else
34   | Don't change current pixel value;
35 5. If in (2.3), return to starting point  $(i_4, j_4) = (i, j)$  and  $(i_3, j_3) = (i_1, j_1)$  and go to
   Step 3;
36 Else set  $(i_2, j_2) = (i_3, j_3)$  and  $(i_3, j_3) = (i_4, j_4)$  and go to (2.3);

```

**37 Step 3:**

```

38 if  $I(i, j) \neq 1$  then
39   | Set LNBD =  $|I(i, j)|$ ;
40   | Start scanning from next pixel  $(i, j + 1)$ ;
41 end

```

42 The algorithm terminates when reaching the bottom right corner of  $I$ ;

14 **Algorithmus 2 :** Border following Algorithmus nach Suzuki und Abe [18][4]

## 2.2 Methodik

### 2.2.1 Vorverarbeitung

**Data :** Image  $I$ , gaussian kernel size  $k_g$ , canny lower threshold  $T_{low}$ , canny upper threshold  $T_{high}$ , dilate iterations  $d$ , erode iterations  $e$

**Result :** processed image  $J$

1 Convert  $I$  from RGB to grayscale:

2  $I \xrightarrow{\text{RGB to Gray}} I'$ ;

3 Smooth  $I'$  with Gaussian Blur and parameters  $k_g, \sigma = 0$ :

4  $I' \xrightarrow{\text{Gaussian Blur}} I''$ ;

5 Apply Canny edge detection with parameters  $T_{low}, T_{high}$ :

6  $I'' \xrightarrow{\text{Canny}} E$ ;

7 Dilate  $E$   $d$  times with kernel  $[[1, 1], [1, 1]]$ :

8  $E \xrightarrow{\text{Dilate}} E'$ ;

9 Erode  $E'$   $e$  times with kernel  $[[1, 1], [1, 1]]$ :

10  $E' \xrightarrow{\text{Erode}} J$ ;

11 Return  $J$ ;

**Algorithmus 3 :** Bildverarbeitungsroutine nach geg. Parametern

Diese generalisierte Bildverarbeitungsroutine nach 3 liefert ein geglättetes binäres Kan tenbild. In Abhängigkeit der Parameter bleiben mehr oder weniger Details im Bild erhalten und die Kanten sind je nach Anzahl der Iterationen von *Dilate* und *Erode* dicker oder dünner. Dementsprechend müssen die Parameter je nach Beleuchtung/Spiegelung auf dem Bildschirm und dem zu erkennenden Spiel angepasst werden. Eine Auflistung der empfohlenen Parameter findet sich in Tabelle 2.2

### 2.2.2 TicTacToe

Die Methode zur Erkennung und Einordnung der Konturen für die Spielstanderkennung von *TicTacToe* basiert auf der *OpenCV*-Funktion *findContours*, welche mit entsprechenden Parametern die Kontouren hierarchisiert von  $-1$  (Äußerste) bis  $n$  (Innerste Kontur) nach dem Algorithmus von *Suzuki et al.* zurückgibt. Der Gedanke hierbei ist, den Rahmen des *Tic-Tac-Toe*-Feldes als äußerste Kontur  $C_o$  zu erkennen, und von dieser aus die neun Teilstücke des Spielfeldes als die inneren Konturen zu bestimmen. Der Schwellenwert für die Fläche einer inneren Kontur soll zusätzlich die Wahrscheinlichkeit verringern, dass "Ausreißer" als Teilstück erkannt werden.

Der hier entwickelte Algorithmus zur Erkennung des *TicTacToe*-Spielstands basiert auf der hierarchischen Schachtelung von Konturen mit dem Spielfeldrand als äußerste Kontur und darauffolgend die neun "Kästchen" der Teilstücke. Sollte diese Relation nicht gegeben sein, so wird ein ungültiges Ergebnis (**None**) zurückgegeben. Sind neun Teilstücke gegeben, so werden diese nach  $x$ - und  $y$ -Koordinaten sortiert, sodass diese nach Standard-Schema ("links oben nach rechts unten") behandelt werden. Der *tile offset* soll hierbei verhindern,

## 2 Spielstanderkennung (JG)

**Data :** edge map  $J$ , contour area threshold  $T_c$   
**Result :** Outer contours  $C_o$ , Inner contours  $C_i$

- 1 Process  $I$  with algorithm 3 auf der vorherigen Seite;
- 2  $I \rightarrow J$
- 3 Find all contours in  $J$  with 2 auf Seite 14;
- 4 Iterate over all contours:
- 5 **for**  $c$  in contours **do**
- 6   **if** hierarchy of  $c$  is 0 and contour area of  $c > T_c$  **then**
- 7     | Add  $c$  to  $C_i$ ;
- 8   **else if** hierarchy of  $c$  is -1 **then**
- 9     | Add  $c$  to  $C_o$ ;
- 10 **end**
- 11 Return  $C_o, C_i$ ;

**Algorithmus 4 :** Konturdetektionsroutine nach geg. Parametern

**Data :** Image  $I$ , Tile offset  $O$ , minimal radius  $r_{min}$ , maximal radius  $r_{max}$ , circle accumulator threshold  $T_{acc}$ , minimal line length  $l_{min}$ , maximal line gap  $l_{max}$ , line accumulator threshold  $T_l$   
**Result :** two-dimensional result array  $R$

- 1 Detect outer and inner contours with algorithm 4:
- 2  $J \rightarrow C_o, C_i$
- 3 **if**  $|C_i| == 9$  **then**
- 4   Sort  $C_i$  by  $y$ -Position;
- 5   **for** every row, row index in  $C_i$  **do**
- 6     | Sort row by  $x$ -Position;
- 7     | **for** contour, column index in row **do**
- 8       | Get upper left corner position  $(x, y)$ , height  $h$  and width  $w$  of contour bounding rectangle;
- 9       | Get subimage tile from  $J$ :
- 10      |  $tile = J[x + O : x + w - O, y + O : y + h - O]$
- 11      | Check for circle(s) in tile;
- 12      | **if** Circle present in tile with Hough Circle Transform and parameters  $r_{min}, r_{max}, T_{acc}$  **then**
- 13       |   | Mark  $R[\text{row index}][\text{column index}]$  as circle;
- 14      | **else if** Lines present in tile with Hough Line Transform and parameters  $l_{min}, l_{max}, T_l$  **then**
- 15       |   | Mark  $R[\text{row index}][\text{column index}]$  as cross;
- 16      | **end**
- 17   | **end**
- 18 | Return  $R$ ;
- 19 **else**
- 20 |   Return **None**;

**Algorithmus 5 :** Algorithmus zur TicTacToe-Spielstanderkennung

dass die Umrandungen des Kästchens selbst als Linie eines Kreuzes erkannt wird. Im Anschluss wird zunächst überprüft, ob ein Kreis im Kästchen vorliegt und wenn ja im resultierenden Array an der entsprechenden Stelle eingetragen, andernfalls analog für Linien als Merkmal der Kreuze.

### 2.2.3 Vier gewinnt

**Data :** Image  $I$ , minimal radius  $r_{min}$ , maximal radius  $r_{max}$ , circle accumulator threshold  $T_{acc}$ , circle distance  $d_c$ , white lower threshold  $T_w$

**Result :** two-dimensional result array  $R$

```

1 Process  $I$  with algorithm 3 auf Seite 15:
2  $I \rightarrow J;$ 
3 Detect circles  $c$  in  $J$  with Hough Circle Transform and parameters
    $r_{min}, r_{max}, T_{acc}, d_c, T_w:$ 
4  $J \xrightarrow{\text{Hough}} c;$ 
5 if  $|c| == 42$  then
6   Sort  $c$  by  $y$ -Position;
7   for every row, row index in  $c$  do
8     Sort row by  $x$ -Position;
9     for circle, column index in row do
10       Detect color of circle center with parameter  $T_w$ ;
11        $T'_w = T_w;$ 
12       while Color not detected correctly and  $T'_w \geq 0$  do
13          $T'_w = T'_w - 10;$ 
14         Detect color of circle center with parameter  $T'_w$ ;
15       end
16       if Color is red then
17         Set  $R[\text{row index}][\text{column index}]$  to red;
18       else if Color is yellow then
19         Set  $R[\text{row index}][\text{column index}]$  to yellow;
20       else if Color is white then
21         Continue;
22       end
23     end
24   Return  $R$ ;
25 else
26   Return None;
```

**Algorithmus 6 :** Algorithmus zur *Vier gewinnt*-Spielstanderkennung

Der hier entwickelte Algorithmus zur Spielstanderkennung von *Vier gewinnt* basiert auf der Annahme, dass das Spielfeld aus 42 Kreisen besteht, welche entweder rot, gelb oder weiß gefärbt sind. Für die Erkennung der Farbe eines oder mehrerer Pixel liefert *OpenCV* die *inRange*-Methode, welche für angegebene obere und untere RGB-Werte zurückgibt, ob sich die Farbe des Pixels innerhalb dieser Grenzwerte befindet. So lässt sich auch bei nicht

## 2 Spielstanderkennung (JG)

konstanten Bedingungen, wie etwa die Beleuchtung, die übergeordnete Farbe eines Pixels wie etwa "Rot" in einem Bild zu erkennen.

### 2.2.4 Parameter

Die Spielstanderkennungsalgorithmen basieren schlussendlich hauptsächlich darauf, die richtigen Parameter für die vorgestellten Methoden zu ermitteln, damit diese bspw. alle Kreise im Bild und deren Farbe im Bild korrekt erkennen und somit robuste Ergebnisse liefern. Die Methodik profitiert dabei von einem fest definierten Sichtfeld für den NAO wie in den Anforderungen festgelegt. Durch Testen haben sich folgende Parameter(-bereiche) als robust erwiesen:

Parameter	<i>TicTacToe</i>	<i>Vier gewinnt</i>
gaussian kernel size $k_g$	5-9	9-15
canny lower threshold $T_{low}$	0	0
canny upper threshold $T_{high}$	20-40	20-40
dilate iterations $d$	8-12	0-4
erode iterations $e$	2-6	0-2
contour area threshold $T_c$	500	-
Tile offset $O$	20	-
minimal radius $r_{min}$	75	40
maximal radius $r_{max}$	95	55
circle accumulator threshold $T_{acc}$	15-25	15-25
minimal line length $l_{min}$	50	-
maximal line gap $l_{max}$	5-15	-
line accumulator threshold $T_l$	7-11	-
Boundaries white color (lower is $T_w$ )	-	[210, 210, 210] - [255, 255, 255]
Boundaries red color	-	[10, 0, 0] - [255, 100, 100]
Boundaries yellow color	-	[50, 100, 0] - [255, 255, 100]

Tab. 2.1: Parameter für die Spielstanderkennung von *TicTacToe* und *Vier gewinnt*

## 2.3 Ergebnisse

Nachfolgend werden für beide Spiele jeweils die Ergebnisse der Spielständerkennungsalgorithmen an einem Beispiel präsentiert. Solange der Roboter in seiner kalibrierten Position vor dem Tablet steht und dieses nicht durch direktes Licht zu viel Spiegelung auf dem Bildschirm erzeugt, sind diese Ergebnisse repräsentativ und reproduzierbar. Sollten Spiegelungen oder anderes Rauschen zu Problemen bei der Erkennung führen, so können die nachfolgend gezeigten Bilder innerhalb der Ausführung durch Angabe der entsprechenden *Debug*-Parameter angezeigt werden, um Rückschlüsse auf das Verhalten zu gewinnen. Meist genügt bei Artefakten im Bild ein Anpassen der Bilderarbeitsroutine 3 auf Seite 15 und hier speziell die Größe des Gaussfilters und der obere Schwellenwert des *Canny*-Operators.

### 2.3.1 TicTacToe

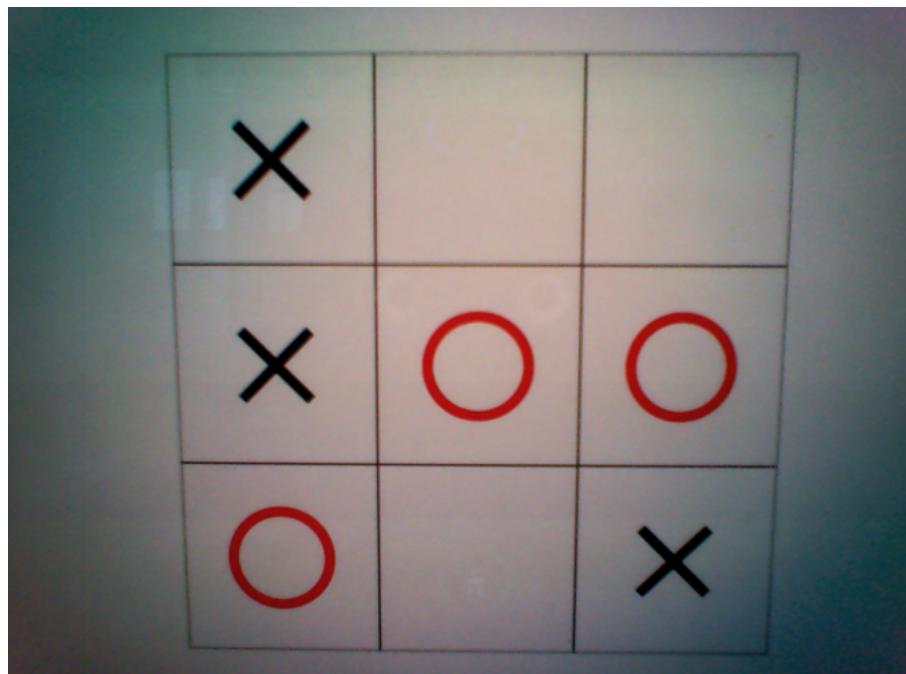


Abb. 2.4: *TicTacToe*-Spielfeld aus Sicht des *NAO*

Wie in 2.4 zu erkennen, hat die Stirnkamera des *NAO* Probleme damit, ein gleichmäßig belichtetes Bild zu erzeugen. Gerade in den Ecken links oben und rechts unten zeigen sich dunklere Regionen als in der Mitte des Bildes oder links unten und rechts oben. Die Helligkeit des Tablets war bei der Aufnahme auf der höchsten Stufe. Ebenfalls erkennt man links oben und in der Mitte des Bilds Spiegelungen von Fenstern und dem Roboter selbst. Diese beiden Anomalien stellten Herausforderungen für die Bildverarbeitsroutine dar, welche essentiell für den Erfolg der anschließenden Erkennung ist.

Durch geeignete Wahl der Parameter konnten diese Störfaktoren weitestgehend entfernt werden. Lediglich im Feld rechts oben taucht ein Punkt auf, der fälschlicherweise als Kante erkannt und durch die *Dilation* zusätzlich vergrößert wurde.

## 2 Spielstanderkennung (JG)

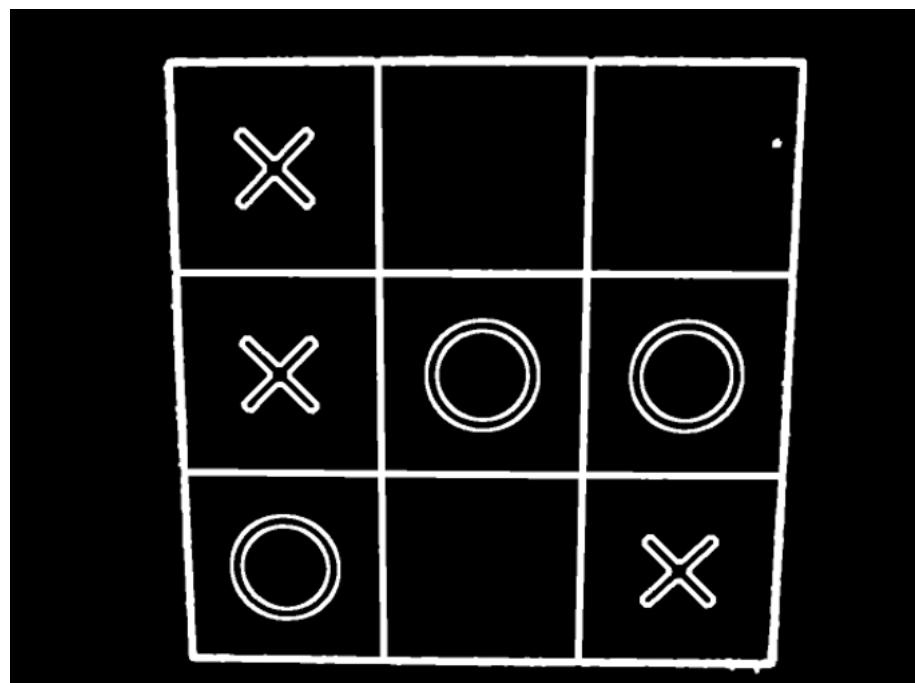


Abb. 2.5: Nach Algorithmus 3 auf Seite 15 und Parametern 2.2 zu Kantenbild verarbeitetes *TicTacToe*-Spielfeld

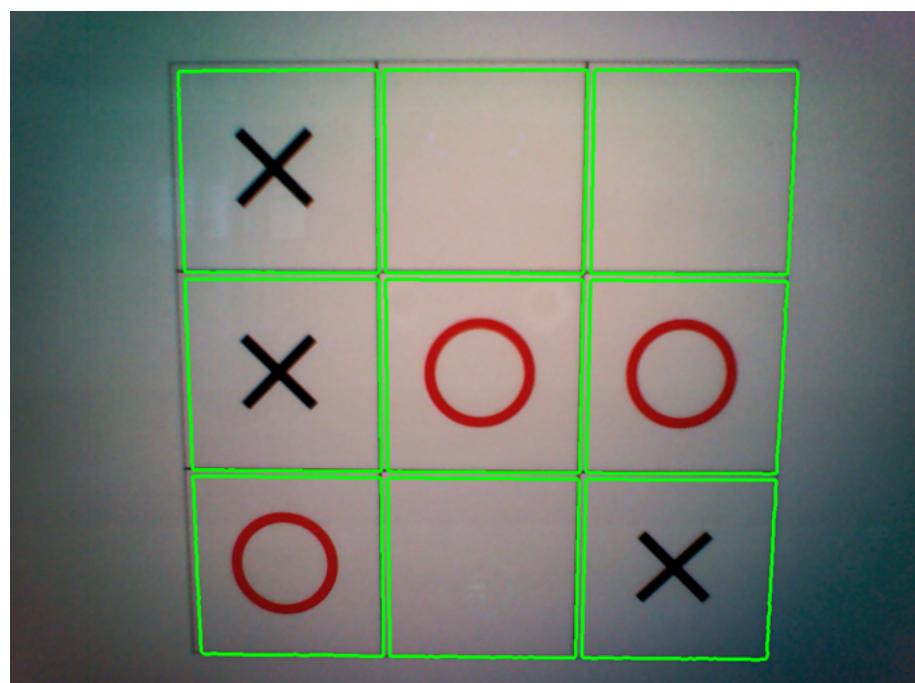


Abb. 2.6: Nach Algorithmus 4 auf Seite 16 erkannte Teilstücke (grün) im *TicTacToe*-Spielfeld

Durch den Schwellenwert der Fläche der inneren Konturen, wie in 4 auf Seite 16 beschrieben, werden jedoch lediglich die Ränder der Teilfelder wie gewünscht erkannt.

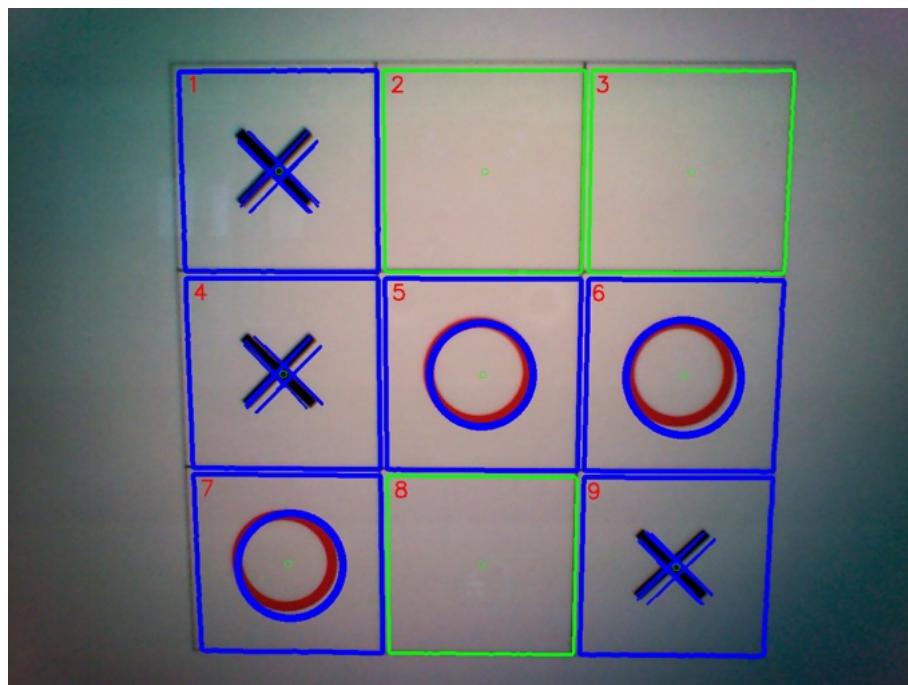


Abb. 2.7: Nach Algorithmus 5 auf Seite 16 erkannter Spielstand im *TicTacToe*-Spielfeld, von links oben nach rechts unten nummeriert; Leere Teilfelder und Mittelpunkte in grün, Teilfelder mit Inhalt in blau.

Durch geeignete Parameter-Wahl nach 2.2 auf Seite 25 wird der Inhalt des Spielfeldes korrekt erkannt:

```
[ [ "X",  "-",  "-"],  
[ "X",  "O",  "O"],  
[ "O",  "-",  "X"] ]
```

### 2.3.2 Vier gewinnt

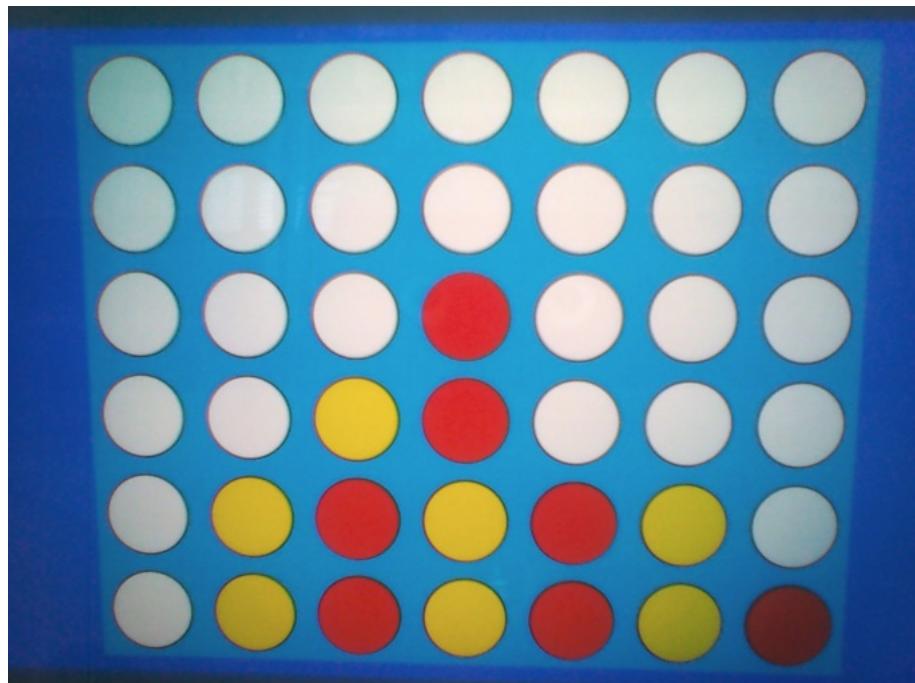


Abb. 2.8: *Vier gewinnt*-Spielfeld aus Sicht des NAO

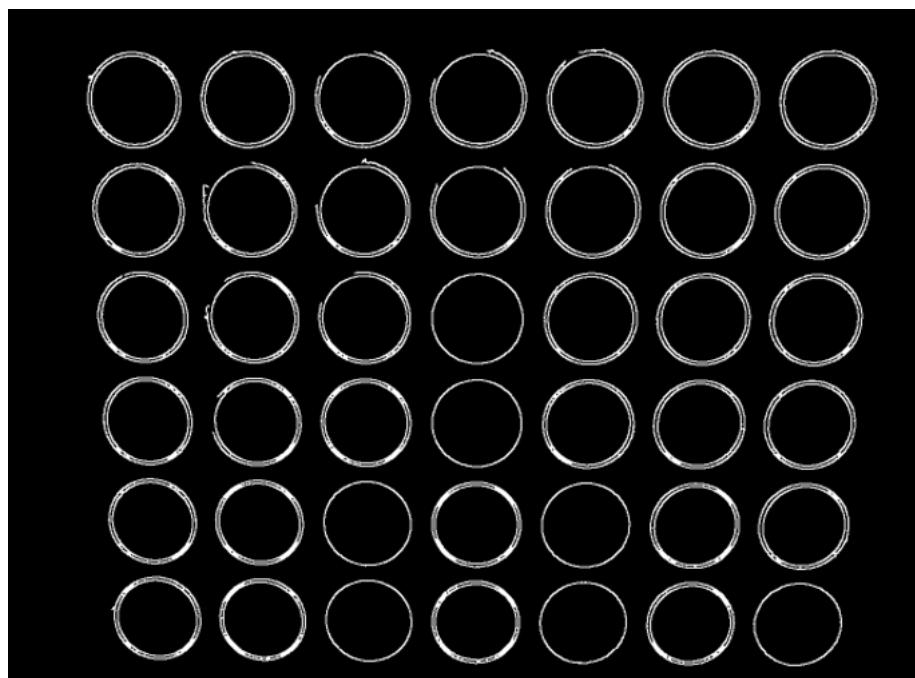


Abb. 2.9: Nach Algorithmus 3 auf Seite 15 und Parametern 2.2 zu Kantenbild verarbeitetes *Vier gewinnt*-Spielfeld

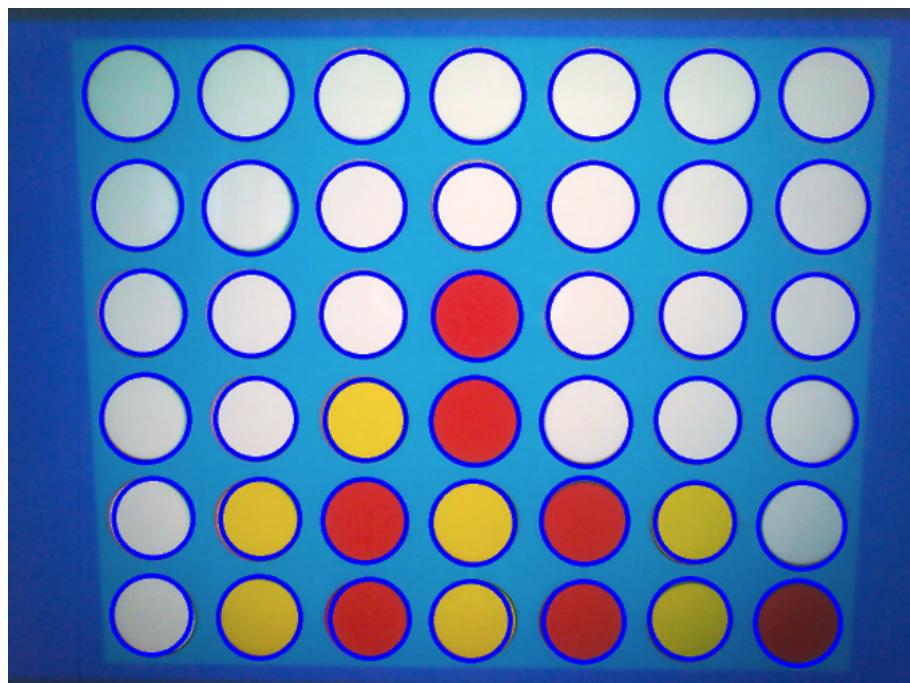


Abb. 2.10: Mit Parametern 2.2 erkannte Teilstufen im *Vier gewinnt*-Spielfeld

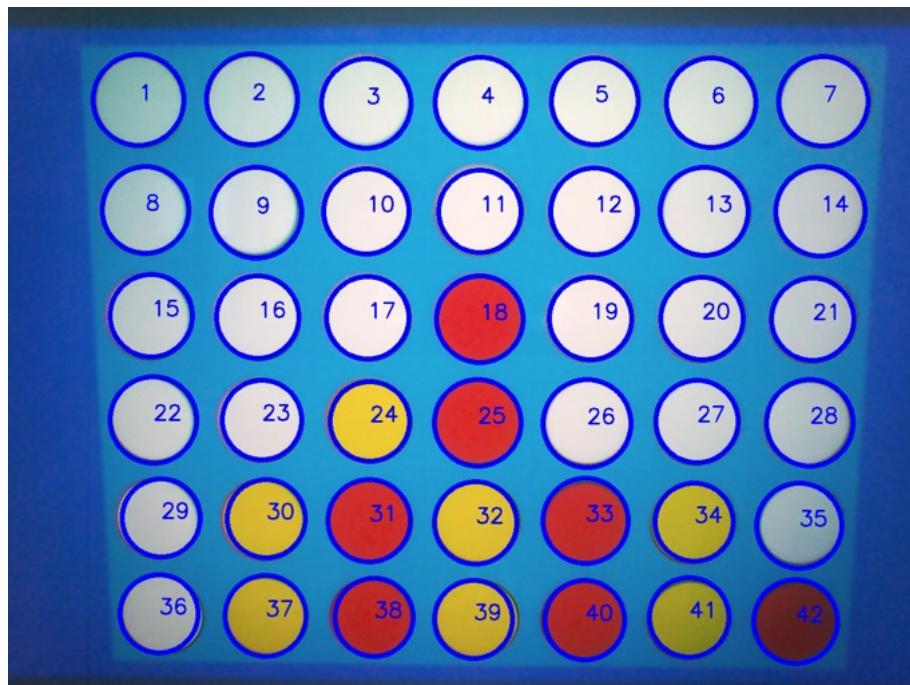


Abb. 2.11: Nach Algorithmus 6 auf Seite 17 erkannter Spielstand im *Vier gewinnt*-Spielfeld

## 2 Spielstanderkennung (JG)

Durch geeignete Parameter-Wahl nach 2.2 auf der nächsten Seite wird der Inhalt des Spielfeldes korrekt erkannt:

```
[['-', '-', '-', '-', 'R', 'Y', '-'],
['Y', 'R', '-', 'R', 'Y', 'R', 'R'],
['Y', 'R', 'Y', 'R', 'Y', 'Y', 'Y'],
['R', 'Y', 'Y', 'R', 'R', 'Y', 'Y'],
['Y', 'R', 'R', 'Y', 'R', 'Y', 'R'],
['Y', 'R', 'R', 'Y', 'Y', 'R', 'R']]
```

### 2.3.3 Präzision

Die Präzision der Methodik wurde anhand dieser Spielfeldkonfiguration stichprobenartig gemessen, indem standardmäßig 1000 Bilder aufgenommen, der ermittelte mit dem tatsächlichen Spielstand verglichen und bei Unterschieden ein Zähler inkrementiert wurde. Beispiel für *TicTacToe*, analog für *Vier gewinnt*:

```
def tictactoe_error_count(robotIP, PORT, field_after_move,
                           iterations=1000):
    wrong_count = 0
    for i in range(iterations):
        fail = vision.get_image_from_nao(robotIP, PORT)
        field = vision.detect_tictactoe_state(fail, minRadius=65,
                                               maxRadius=85, acc_thresh=15, canny_upper_thresh=25,
                                               dilate_iterations=8, erode_iterations=4,
                                               gaussian_kernel_size=9)
        if field != field_after_move:
            print(i)
            if field != None:
                for row in field:
                    print(row)
            wrong_count += 1
    print(wrong_count)
```

Konfiguration 1:

```
[['X', '-', '-'],
 ['X', 'O', 'O'],
 ['O', '-', 'X']];
```

Konfiguration 2:

```
[['O', 'O', '-'],
 ['X', '-', 'X'],
 ['O', 'O', 'X']];
```

Konfiguration 3:

```
[['-', 'X', '-'],
 ['O', 'X', 'O'],
```

```

['X', 'O', 'O']];
Konfiguration 4:
[[['-', '-', '-', '--', 'R', 'Y', '-'],
['Y', 'R', '--', 'R', 'Y', 'R', 'R'],
['Y', 'R', 'Y', 'R', 'Y', 'Y', 'Y'],
['R', 'Y', 'Y', 'R', 'R', 'Y', 'Y'],
['Y', 'R', 'R', 'Y', 'R', 'Y', 'R'],
['Y', 'R', 'R', 'Y', 'Y', 'R', 'R']];
Konfiguration 5:
[[['-', 'R', '--', '--', '--', '--', '--'],
['-', 'Y', '--', 'Y', '--', '--', '--'],
['-', 'R', 'Y', 'R', '--', '--', '--'],
['-', 'Y', 'R', 'Y', '--', 'R', 'R'],
['R', 'R', 'R', 'Y', '--', 'Y', 'Y'],
['Y', 'Y', 'Y', 'R', '--', 'R', 'Y']];
Konfiguration 6:
[[['-', '--', '--', '--', '--', '--', '--'],
['--', '--', '--', '--', '--', '--', '--'],
['--', '--', '--', '--', '--', '--', '--'],
['--', '--', '--', 'R', '--', '--', '--'],
['Y', 'R', 'R', 'Y', 'R', 'Y', '--'],
['Y', 'Y', 'R', 'R', 'Y', 'R', 'Y']]);

```

Datum	Konfiguration	Anzahl Fehler	Anteil fehlerhafte Erkennung
05.09.23	1	2	0.002
11.09.23	2	2	0.002
11.09.23	3	5	0.005
05.09.23	4	1	0.001
11.09.23	5	0	0.000
11.09.23	6	1	0.001

Tab. 2.2: Ergebnisse für die Präzision der Spielstanderkennung von *TicTacToe* und *Vier gewinnt* anhand von jeweils drei Stichproben

Die gewählten Ansätze zur Erkennung von *TicTacToe* und *Vier gewinnt* erscheinen primiv im Vergleich zu Ansätzen wie neuronalen Netzen und sind stark abhängig von der korrekten Position des Roboters vor dem Bildschirm bzw. Tablet. Jedoch wurde sich darauf geeinigt, dass sich der NAO direkt vor dem Tablet befindet, daher konnte man von dieser Bedingung ausgehen und davon profitieren. Des Weiteren sprechen die Messungen der Präzision für den gewählten Ansatz. Man sieht bei den Stichproben und besonders im tatsächlichen Spielablauf eine gewisse Anzahl an Fehlern. Daher sind *Unit Tests* nur begrenzt möglich bzw. sinnvoll, da mit den minimal abweichenden Positionen des Roboters vor dem Bildschirm und möglichen Beleuchtungsartefakten auch keine hundertprozentige Reproduzierbarkeit garantiert ist. Somit könnten nicht bestandene *Unit Tests* auftreten, welche jedoch nicht die Fehlerhaftigkeit der Methodik als solches demonstrieren und nur eine geringe Fehlerwahrscheinlichkeit besitzen. Daher zeigen diese stichprobenartigen Messungen

## 2 Spielstanderkennung (JG)

die Zuverlässigkeit der Methodik.

Weitere Möglichkeiten zur Implementierung der Spielstanderkennung wären *Machine-* bzw. *Deep Learning* basierte Ansätze [11, 9], oder aber Methoden wie *Template Matching*, bei der Referenzbilder und der tatsächliche Input "verglichen" werden[10]. Zusätzlich beinhaltet  *naoqi* ein eigenes Bilderkennungsmodul (*ALVisionRecognition*), welches ebenfalls mit Referenzbildern arbeitet. Nachteil ist hierbei jedoch, dass man Referenzbilder bzw. Trainingsdaten benötigt, welche jedoch nicht immer gleich dem aktuellen Sichtfeld des Roboters sind. Zudem können diese Ansätze potentiell rechenaufwändiger als der hier gewählte Ansatz sein, was bei dem ohnehin nicht sehr leistungsstarken Prozessor des *NAO* ein Problem sein kann.

# 3 Spieltaktiken (NG)

## 3.1 Tictactoe

Ziel ist es die aktuellen Spielsituation, die durch die Vision erhalten wird, auszuwerten und den daraus resultierenden Zug zu berechnen. Dabei soll es auch möglich sein zwischen mehreren Schwierigkeitsstufen auszuwählen.

### 3.1.1 Spielfeld

Feld Bezeichnungen:

0	1	2
3	4	5
6	7	8

Beispiel Feld:

x	o	
x		o

### 3.1.2 Spielregeln

Zwei Spieler treten gegeneinander an. Der eine Spieler erhält das Zeichen x und der Andere o. Abwechselnd dürfen die Spieler eines der 9 Felder mit ihrem Zeichen markieren. Der Spieler der zuerst 3 Felder in einer Reihe, Spalte oder einer der beiden Diagonalen hat, hat gewonnen. Sind alle Felder voll ohne, dass ein Spieler gewonnen hat, endet das Spiel in einem Unentschieden.

### 3.1.3 Input

Als Input werden das aktuelle Spielfeld, das eigene Zeichen (in unserem Fall x oder o), das Zeichen des Gegenspielers (in unserem Fall x oder o), ein Zeichen für die nicht belegten Felder (in unserem Fall -) und der Schwierigkeitsgrad (1 -> leicht; 2 -> mittel; 3 -> schwer; 4 -> unmöglich)

### 3.1.4 Output

Die Methode soll am Ende zurückgeben, in welches Feld der Roboter spielen sollte. Die Felder sind dabei so definiert, wie sie oben bei Feld Bezeichnungen benannt sind.

### 3.1.5 Grundkonzept Gamelogic

Bei Tictactoe gibt es nur eine beschränkte Möglichkeiten an Zügen und Spielsituationen und viele sind gleichbedeutend. Beispielsweise ist beim ersten Zug jede Ecke gleich gut. Deshalb wird im ersten Zug in eine zufällige Ecke gespielt Kanten sind schlechter, da sie nur für die Spalte und Reihe, aber nicht für die Diagonale nutzbar sind. Welche der Spielsituationen gerade vorhanden ist, wird an der Anzahl an freien, eigenen und Feldern des Gegenspielers für jede Reihe, Spalte und Diagonale erkannt. Daraus resultiert dann der Spielzug. Für jede Spielsituation gibt es dann einen perfekt Zug. Abhängig von der Schwierigkeit wird eine Spielsituation erkannt und der beste Zug ausgewählt. Eine schlechtere Schwierigkeit führt dazu, dass manche kluge Züge nicht durchgeführt werden und stattdessen simplere Züge zum Teil zufälligere Züge oder bewusst schlechte Züge durchgeführt werden. Wenn bereits zwei Steine nebeneinander liegen wird immer der 3. daneben gelegt. Verteidigt wird diese Situation von allen Schwierigkeiten außer leicht. Bei der gewählten Umsetzung ist es möglich jedes Spielfeld zu übergeben und die Logik berechnet daraus das beste Feld. Dadurch ist die Logik nicht von den vorherigen Zügen abhängig und kann für jede aktuelle Situation ein Ergebnis zurückgeben.

## 3.2 Vier Gewinnt

Ziel ist es die aktuellen Spielsituation, die durch die Vision erhalten wird, auszuwerten und den daraus resultierenden Zug zu berechnen. Dabei soll es auch möglich sein zwischen mehreren Schwierigkeitsstufen auszuwählen.

### 3.2.1 Spielfeld

Das Spielfeld ist 6 Felder hoch und 7 Felder breit. Beispiel Feld:

		Spalte2				
	Spalte1	Y	Spalte3			
	R	Y	R	Spalte4	Spalte5	
Spalte0	Y	R	Y	Y	R	
R	Y	Y	R	R	Y	Spalte6

### 3.2.2 Spielregeln

Zwei Spieler treten gegeneinander an. Beide Spieler erhalten unterschiedliche Farben oder Zeichen zugeteilt. In unserem Fall R (red) und Y (yellow). Abwechselnd wählt ein Spieler eine Spalte in die er spielen möchte. Dabei fällt der Spielstein bis auf das unterst mögliche Feld der Spalte. Ziel des Spiels ist es vier der eigenen Steine in eine Reihe, eine Spalte oder eine Diagonale zu setzen. Der Spieler, der das zuerst schafft hat gewonnen, falls zuvor alle Felder belegt sind, endet das Spiel im Unentschieden.

### 3.2.3 Input

Als Input werden das aktuelle Spielfeld, das eigene Zeichen (in unserem Fall R oder Y), das Zeichen des Gegenspielers (in unserem Fall R oder Y), ein Zeichen für die nicht belegten Felder (in unserem Fall -) und der Schwierigkeitsgrad (1 -> leicht; 2 -> mittel; 3 -> schwer; 4 -> unmöglich) benötigt.

### 3.2.4 Output

Die Methode soll am Ende zurückgeben, in welche Spalte der Roboter spielen sollte. Die Spalten sind dabei so definiert, wie sie oben im Beispiel Feld benannt sind.

### 3.2.5 Grundkonzept: Prioritäten

Die Spiellogik funktioniert grundsätzlich über Prioritäten. Dabei wird für jede Spalte berechnet, wie wichtig es in der jeweiligen Runde ist, den eigenen Spielstein in diese zu setzen. Dabei wird zuerst berechnet, wo in der Spalte der Spielstein landen wird. Danach wird einzeln für die Zeile, Spalte und beide Diagonalen berechnet, wie bedeutend der Spielstein dort für die Offensive und Defensive ist. Beispielsweise Spalte1, Offensive, Diagonale1. Es werden also pro Spalte 8 Werte berechnet. Eigene Steine führen zu einer höheren Priorität in der Offensive, aber zu einer geringeren in der Defensive, da dieses Feld dann bereits in dieser Richtung verteidigt wird. Neutrale Felder erhöhen die Priorität leicht für die Offensive. Das ist besonders relevant beim ersten Spielzug, da dort keine fremden oder eigenen Steine gelegt wurden, somit wird im besten Zug bei einem leeren Feld in die mittlere Spalte gespielt, da diese noch die meisten Möglichkeiten für vier Steine nebeneinander bietet. Steine vom Gegenspieler führen zu einer verringerten Punktzahl für die Offensive für diese Richtung und zu einer erhöhten Priorität für die Defensive. Sowohl für die Offensive als auch Defensive wird berücksichtigt, ob eigene oder gegnerische Steine für die Richtung überhaupt noch von Relevanz sind oder nicht bereits abgeblockt sind.

#### Ausnahmefall bei winning move und defend

In manchen Ausnahmefällen kann eine Möglichkeit das Spiel zu beenden von der Priorität nicht den höchsten Wert erhalten. Beispielsweise, wenn der Gegner auch den Sieg androht oder Zwickmühlen angedroht werden. Deshalb wird zusätzlich zu den Prioritäten überprüft, ob das Spiel bereits beendet werden kann. Ist dies der Fall erhält diese Spalte in dieser Richtung eine Priorität von 3000 damit dieser Zug auf jeden Fall gespielt wird.

Äquivalent dazu kann dasselbe passieren, wenn der Gegner drei Steine in einer Richtung hat, aber beispielsweise selbst eine Zwickmühle gebildet werden kann, dann erhält diese Spalte für diese Richtung eine Priorität von 2000.

#### Vorlegen für den Gegner

Außerdem gilt zu berücksichtigen, dass das Legen eines Steins für diese Spalte bedeutet, dass der Gegner auf das Feld darüber ein Stein legen kann. Dieser Stein könnte für den Gegner wichtiger sein als das Feld, dass wir belegt haben.

## Schwierigkeitsgrad

Der Schwierigkeitsgrad wird über einen Fehlerfaktor gesteuert. Bei der Berechnung der Priorität für jede Spalte erhält jede eine zufällige Zahl zwischen 0 und dem Fehlerfaktor. Je höher der Fehlerfaktor ist desto größer ist der Einfluss des Zufalls im Verhältnis zu der vom Algorithmus bestimmten Priorität der verschiedenen Felder.

## Gesamtpriorität für eine Spalte

Die Gesamtpriorität einer Spalte ergibt sich dann aus der höchsten Priorität einer der berechneten Richtungen. Darauf addiert wird wenn die höchste Priorität eine offensive bzw. defensive Aktion war die Summe aller offensiven bzw. defensiven Aktionen multipliziert mit 0,1. Davon wird die höchste Priorität für den Gegenspieler für das darüber liegende Feld multipliziert mit 0,4 abgezogen. Am Ende muss nur noch abgeglichen werden, welche Spalte die höchste Priorität hat und dorthin wird der Stein gespielt.

# 4 Movement (NG)

Der Roboter soll die Fähigkeit besitzen ein beliebiges Feld auf dem Bildschirm anzuklicken.

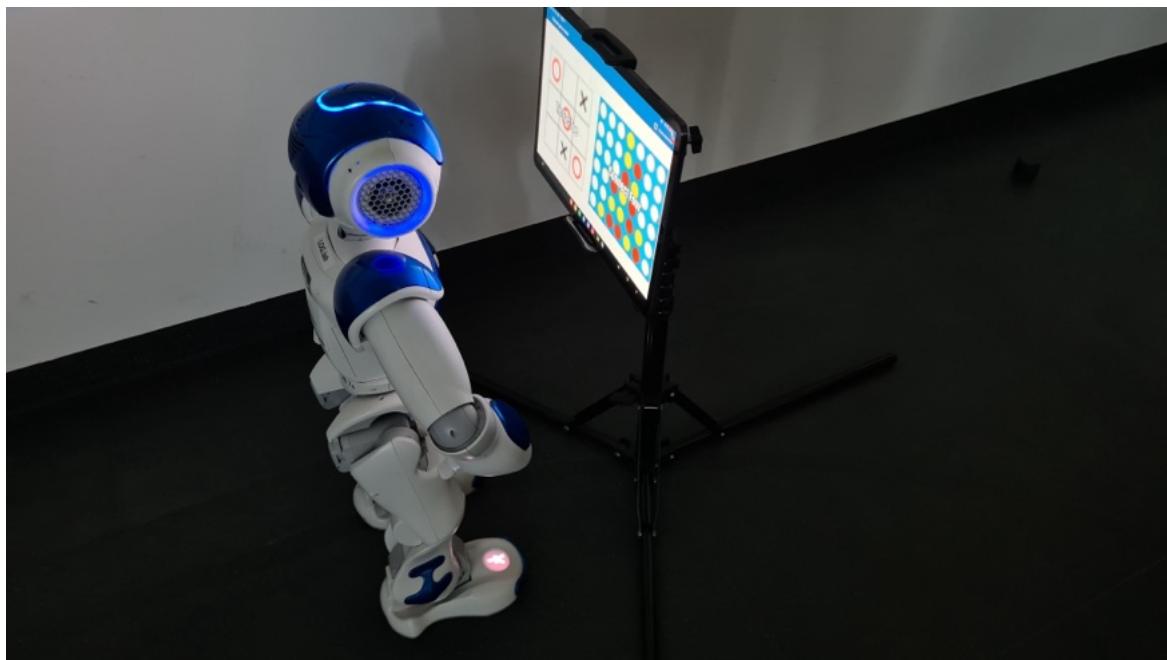


Abb. 4.1: NAO vor Tablet

## 4.1 Herausforderungen

Dabei gibt es mehrere Herausforderungen.

Die erste große Herausforderung ist, es dafür zu sorgen, dass die Finger des Roboters überhaupt auf dem Touchscreen erkannt werden. Da die Finger aus Plastik sind leiten sie keinen Strom und lösen somit in ihrem ursprünglichen Zustand keinen Klick auf dem Tablet aus.

Eine weitere Problemstellung ist die Positionierung des Roboters. Der Arm des Roboters hat nur eine beschränkte Reichweite und die Positionierung muss reproduzierbar sein.

Abschließend muss der eigentliche Klick, also das Berühren des Tablets an jeder möglichen Stelle umgesetzt werden. Dabei gilt es so präzise wie möglich zu sein, da ein Klick zu weit nach links, rechts, oben oder unten bereits ein anderes Feld sein könnte. Dazu kommt, dass in die richtige Tiefe geklickt werden muss, da bei einer zu kurzen Bewegung das Tablet gar nicht getroffen wird und bei einer zu weiten Bewegung das Tablet verschoben oder

## 4 Movement (NG)

beschädigt werden könnte. Eine Verschiebung würde dabei einen weiteren falschen Klick zur Folge haben.

### 4.2 Ideen und Lösungsansätze

#### 4.2.1 Touchscreenfähiger Finger

##### Tablet Stift

Eine einfache Lösung für dieses Problem wäre es den beim Tablet mitgelieferten Tablet Stift ?? auf Seite ?? zu nutzen. Dabei müsste man den Stift fest in die Hand des NAOs befestigen. Damit klickt der NAO aber nur indirekt aufs Tablet. Eine ähnliche Lösung wie diese, gibt es bereits, weshalb noch weitere Ideen in Betracht gezogen werden.



Abb. 4.2: Tabletstift

##### Mobile Gaming Finger Sleeve

Die nächste Idee war es Mobile Gaming Finger Sleeve (4.3) zu nutzen, die eigentlich dafür gedacht sind, bei einem Menschen die Touchscreenfähigkeit der menschlichen Finger noch weiter zu steigern. Sie sind in ersten Linie aus Stoff und enthalten verschiedene Metalle, die für eine bessere Leitfähigkeit sorgen sollen. Durch den Stoff passen sie sich perfekt an den Finger des NAO Roboters an. Leider reichen jedoch diese Fingerhütte alleine nicht aus, um den Plastikfingern die benötigte Leitfähigkeit zu geben.



Abb. 4.3: Mobile Gaming Finger Sleeve

## Leitfähiger Stoff

Der Ansatz der Mobile Gaming Finger Sleeve war dennoch grundsätzlich nicht falsch. Um einen Klick mit dem Finger des NAO Roboters am Tablet auszulösen, muss am Berührungs-punkt mit dem Tablet Strom fließen, denn das ist was am Tablet erkannt wird. Der menschliche Finger hat durch den hohen Wasseranteil im Körper diese Fähigkeit. Beim Plastikfinger des NAO Roboters muss nachgeholfen werden. Daraus resultiert, dass ein besonders leitfähig-er Stoff benötigt wird. Außerdem muss sich der Stoff eignen, um in irgendeiner Weise am NAO Finger befestigt zu werden. Aluminium Folie erfüllt beide Eigenschaften, da es zum einen sehr gut leitet und zum anderen biegsam ist.

Der Zweck der Aluminium Folie ist es an einem Kontaktpunkt viel Leitfähigkeit zu bieten und das auf eine größere Fläche des schlecht leitenden Plastiks zu verteilen.

Damit das Aluminium besser in Position bleibt, während der NAO beim Shutdown seine Hand schließt, wird zusätzlich die Mobile Gaming Finger Sleeve (siehe oben) genutzt. Dabei wird von der Sleeve die Fingerspitze abgeschnitten und über den hinteren Teil der Aluminium Folie gestülpt.

### 4.2.2 Arm Movement

Für die Bewegung des Arms bzw. Fingers auf das Tablet gibt es zwei Optionen: Zum einen kann der Finger des NAOs auf dem Weg zum Bildschirm getrackt werden und während der Bewegung live überprüft werden, in welche Richtung der Arm weiter bewegt werden muss, um das Ziel zu erreichen. Zum anderen kann die Armbewegung für fixe Punkte auf dem Bildschirm, welcher im gleichen Abstand bleibt, bestimmt werden und anhand von diesen auf alle dazwischen interpoliert werden.

#### Nachteile Tracken:

Es ist schwierig zu erkennen, wann der Bildschirm berührt wurde, da der Finger bzw. Arm genau im Sichtfeld des NAOs ist. Außerdem muss die Fingerbewegung und das Tracken perfekt koordiniert sein, damit das richtige Feld angeklickt wird. Wobei auch das Tracken des Finger sich als sehr kompliziert herausstellen kann. Dazu kommt, dass die Bewegung sehr langsam ablaufen muss, da die Berechnung langsam ist und der Finger nicht zu sehr in eine falsche Richtung daneben gehen sollte, da hierbei das Tablet umgeworfen, beschädigt oder der Finger des NAOs beschädigt werden könnte.

#### Nachteile fixe Punkte:

Bei fixen Punkten muss die Positionierung des Tablets zum Roboter immer gleich sein, damit das beabsichtigte Feld angeklickt wird. Außerdem besteht der Nachteil, dass bei verschiedenen Größen des Tablets die fixen Punkte falsch sind.

### 4.2.3 Positionierung NAO und Tablet

Bei der Positionierung des Tablets gibt es mehrere Möglichkeiten.

**Möglichkeit 1: Tablet an Wand befestigen.** Der große Nachteil hierbei ist jedoch, dass man sehr unflexibel ist und ggf. Probleme hat eine passende Wand zu finden. Dabei kann auch die Sonneneinstrahlung eine wichtige Rolle spielen, da sie für Spiegelungen auf dem Tablet sorgt, welche bei zu starken Spiegelungen zu Fehlern in der Vision führen können.

**Möglichkeit 2: Tablet auf Tisch bzw. Karton legen.** Dabei hat man aber einen schlechten Winkel für eine natürliche Bewegung der Hand auf das Tablet und für die Kamera. Jedoch wäre diese Position für die Verwendung eines Tablet Stifts praktikabel.

**Möglichkeit 3: Tablet auf Tripod stellen.** Bei Möglichkeit 3 hat man maximale Flexibilität bei der Positionierung des Tablets. Diese Flexibilität führt dazu, dass es mehr Variablen beim Tablet gibt, die eingestellt werden müssen. (Bspw. Höhe, Tiefe, Kippung)

**2 Tablets vs 1 Tablet:** Da bei den gewählten Spielen meist ein Roboter gegen einen anderen Roboter oder einen menschlichen Spieler spielt, kann entweder lokal auf einem Tablet oder online auf 2 Tablets gespielt werden.

Ein Tablet hat den Nachteil, dass beim Spiel Roboter gegen Roboter der eine dem anderen immer wieder Platz machen muss und sich danach wieder genau gleich vor dem Tablet positionieren muss. Dafür reicht die Präzision der Bewegungs- und Kameragenauigkeit der Roboter nicht aus. Die Problematik hängt mit der unten beschriebenen Idee Roboter positioniert sich selbst zusammen, denn bei der Neupositionierung muss der Roboter sehr präzise wieder zur gleichen Stelle laufen. Ein Spiel gegen einen menschlichen Spieler ist bei einem Tablet möglich, birgt aber das Risiko, dass Mensch und Maschine kollidieren.

Der einzige Nachteil der Lösung mit zwei Tablets ist, dass die Spieleapp einen lokalen online Multiplayer benötigt.

**Roboter positioniert sich selbst vs manuelle Positionierung des Tablets** Wichtig ist es, dass der Roboter in eine immer ähnliche bzw. gleiche (bei fixen) Position vor dem Tablet hat. Dabei gibt es zwei Herangehensweise: Variante 1: Der Roboter erkennt selbst, wie er sich positionieren muss und richtet sich automatisch so vor dem Tablet auf, wie es benötigt wird. Variante 2: Der Roboter wird stehen gelassen und das Tablet wird so vor dem Roboter platziert, wie es benötigt wird.

Bei Variante 1 könnte man mit der Vision verschiedene Orientierungspunkte festlegen, wie z. B. die Kanten des Tablets, Markierungen auf dem Boden vor dem Tablet oder man verwendet die NAO zugehörigen Marks mit ALLandMarkDetection. Problematisch dabei ist aber, dass sowohl die Vision als auch das Movement sehr präzise sein müssten, um sich genau in der richtigen Position vor dem Tablet zu platzieren. Gerade der Abstand, also die Tiefe stellt ein großes Problem dar, da der einzige Anhaltspunkt die Größe des Tablets ist.

Bei Variante 2 kann man zur Abstandsmessung die Arme benutzen. Hierbei muss sowohl die z-Rotation, die x-Rotation, y-Rotation und der Abstand mit der erwarteten Endposition übereinstimmen.

## 4.3 Lösung:

Benötigte Ausstattung 4.4:

- Nao Roboter
- Tripod
- Tablet
- Aluminium Folie
- Schere
- Klebemittel
- Optional: Maßband, Mobile Gaming Finger Sleeve (4.3 auf Seite 32)



Abb. 4.4: Benötigte Ausstattung

### 4.3.1 Touchscreenfähiger Finger

Die Touchscreenfähigkeit des Fingers wird durch Aluminium Folie erreicht. Diese wird in



Abb. 4.5: Aluminium Folie, Schere und Klebemittel

zwei Schritten um den Finger gewickelt. Bei beiden Schritten gilt es so viele Berührungs-punkte zum Finger, wie möglich herzustellen.

## 4 Movement (NG)

1. Fingerkuppe: Aus Aluminium Kreis oder Quadrat schneiden und hier mehrfach von außen Richtung Mitte schneiden. Dabei die Folie als ein Stück lassen. Dann die Mitte auf die Fingerspitze und den restlichen Finger wickeln und festkleben mit z.B. Tesa
2. Um den restlichen grob zylinderförmigen Teil des Fingers wird ein rechteckiges größeres Stück Aluminium gewickelt und ggf. geklebt.

Bei einem Neustart des Roboters kann es passieren, dass sich die Aluminium Folie wieder ein bisschen löst und nicht mehr genug Kontaktpunkte mit dem Roboter vorhanden sind. Meistens reicht es die Aluminium Folie wieder fester an den Finger zu drücken.

### 4.3.2 Positionierung des Tablets zum NAO

Die Positionierung des NAOs geschieht mithilfe eines Tripods (4.6). Bei diesem kann sowohl x-, y-, z-Achse als auch der Abstand vom Roboter manuell eingestellt werden.



Abb. 4.6: Tripod

Die Höhe des Tripods kann dabei entweder in den folgenden Schritten parallel eingestellt werden oder bei Bedarf mithilfe eines Maßbands die Höhe bereits voreingestellt werden. Die Höhe des obersten Punkts des Tripods beträgt dabei ungefähr 48,5 cm (4.7).



Abb. 4.7: Höhe Tablet

Als nächstes muss die z-Achse des Tablets mithilfe einer Wasserwaage App (4.8) eingestellt und parallel überprüft. Danach gibt es mehrere Positionen des NAOs zur Einstellung



Abb. 4.8: Z-Achse falsch eingestellt



Abb. 4.9: Z-Achse richtig eingestellt

der korrekten Achsen und Positionen. Für die x-Achse streckt der NAO beide Arme nach vorne. Es sind jeweils Ellbogen und Hand der beiden Arme die Markierungen, um zu sehen, ob die x-Achse korrekt ist (4.11 auf der nächsten Seite; 4.12 auf der nächsten Seite). Dabei sollte das Tablet genügend Abstand zum Roboter haben und die z-Achse korrekt bleiben.

```
def tabletPreparationXAngle(robotIP, port):
    armMovement(robotIP, port, arm="L",
    position=armPosition.positionLTabletPreparation, go_back=False)

    armMovement(robotIP, port, arm="R",
    position=armPosition.positionLTabletPreparation, go_back=False)
```

Abschließend kann die letzte Position eingenommen werden, dabei begibt der NAO seinen linken Arm an die Position an der er ganz unten links klicken würde und seinen rechten Arm an die Position ganz oben rechts (4.15 auf Seite 39). Damit kann sowohl die richtige Höhe, y-Achse, als auch der richtige Abstand gewählt werden.

```
def tabletPosition(robotIP, port):
    armMovement(robotIP, port, arm="L",
    position=getInterpolatedPosition(left=4.4, up=0), go_back=False)
```

#### 4 Movement (NG)

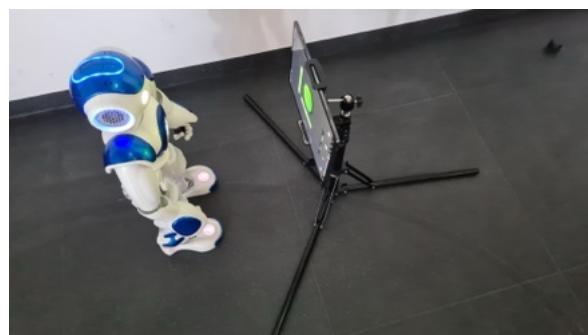


Abb. 4.10: X-Achse Startposition



Abb. 4.11: X-Achse korrekt (seitliche Ansicht)



Abb. 4.12: X-Achse korrekt (frontale Ansicht)

### 4.3 Lösung:

```
armMovement(robotIP, port, arm="R",
position=getInterpolatedPosition(left=4.4, up=8), go_back=False)

openHand(robotIP, port, arm="L")
openHand(robotIP, port, arm="R")
```



Abb. 4.13: Y-Achse Startposition



Abb. 4.14: Y-Achse falsch eingestellt



Abb. 4.15: Y-Achse korrekt eingestellt

Nachdem der Roboter in der letzten Einstellungsposition ist, darf sowohl der Roboter als auch das Tablet nicht mehr bewegt werden. Falls dieser jedoch trotzdem bewegt wurde,

## 4 Movement (NG)

nichts an den Einstellungen im Tripod verstellt wurde und der Untergrund eben ist, reicht die letzte Position (tabletPosition) zur erneuten Positionierung aus. Danach wird der Roboter in seine Grundposition (4.16) versetzt.



Abb. 4.16: Grundposition

### 4.3.3 Arm Movement über fixe Punkte

Die Armbewegung des Roboters funktioniert über ALMotion des ALProxy. Dafür muss zunächst der ALProxy, der von der NAOqi API bereitgestellt wird importiert werden:

```
from naoqi import ALProxy
```

Diese Klasse ALProxy erlaubt es Instanzen von verschiedenen Modulen, wie Motion Control, Speech Recognition, Text-To-Speech, Vision Processing, zu erstellen, die dann genutzt werden können, um Zugriff auf die unterschiedlichen Bereiche zu erhalten. Für den Zugriff benötigt man die Bezeichnung des Moduls, dass man benutzen will, die Roboter IP und den Port (meist 9559). In diesem Fall wird das Modul Motion Control benötigt:

```
motionProxy = ALProxy("ALMotion", robotIP, port)
```

Mithilfe des motionProxy können nun verschiedene Einstellungen der Motoren getroffen werden. Vor jeder Bewegung des Arms kann es sinnvoll sein, die Steifheit der Motoren der verschiedenen Bereiche aufs Maximum, also eins, zu stellen. Dadurch steht der Roboter stabiler und behält seine Grundposition. Bei einer niedrigeren Stufe lässt der Roboter seine Motoren lockerer, wodurch beispielsweise die Armposition durch dagegen drücken verändert werden kann. Das kann aber auch dazu führen, dass robotereigene Bewegungen des Arms ausreichen (bzw. je nach Steifheit die Schwerkraft ausreicht), um die Grundposition des Roboters zu verändern oder diesen zum Umkippen zu bringen. Deshalb wird hier die Steifheit überall aufs Maximum eingestellt:

```
motionProxy.setStiffnesses("LLeg", 1.0)
motionProxy.setStiffnesses("RLeg", 1.0)
motionProxy.setStiffnesses("Body", 1.0)
motionProxy.setStiffnesses("RArm", 1.0)
motionProxy.setStiffnesses("LArm", 1.0)
```

Nach diesen Schritten kann nun der Arm selbst bewegt werden. Dafür wird über die zuvor erstellte Instanz eine Methode aufgerufen. Diese erhält den Motor, der bewegt werden soll mit vorangestelltem R oder L für rechts oder links:

- Kopf: HeadYaw, HeadPitch
- Arm: ShoulderPitch, ShoulderRoll, ElbowYaw, ElbowRoll, WristYaw, Hand
- Bein: HipYawPitch, HipRoll, HipPitch, KneePitch, AnklePitch, AnkleRoll

Zusätzlich enthält sie die Einstellung für diesen in Bogenmaß und die Geschwindigkeit in der diese Bewegung ausgeführt werden soll.

```
motionProxy.angleInterpolationWithSpeed("RShoulderPitch",
                                         18.4 * almath.TO_RAD, 0.2)
motionProxy.waitUntilMoveIsFinished()
```

Da die Positionierung des Roboters zum Tablet fix ist und somit die gleiche Bewegung des Roboters immer zu einem Klick auf dem gleichen Feld auf dem Bildschirm führt, kann hier mit fixen Bildschirmkoordinaten gearbeitet werden. Dabei wurden 45 Positionen für den linken Arm auf dem Bildschirm definiert und die dazugehörigen Arm Movements bestimmt. Die Bildschirmkoordinaten sind von der Mitte mit 0 nach links mit 5 definiert (4 ist der linkste Punkt auf unserem Tablet) und von 0 unten bis 9 oben:

```
positionL = ["LShoulderPitch", "LShoulderRoll", "LElbowRoll",
             "LWristYaw", "LElbowYaw"]
positionLUp[4][8] = [-40, 17, -32, 0, 0]
```

Das heißt die oben genannten Einstellungen der Gradzahlen entsprechen der Position nach Links vier und nach oben acht. Somit hat der NAO für 45 Positionen auf dem Bildschirm feste Bewegungen. Diese können für den rechten Arm gespiegelt werden. Dies gelingt in dem die Arm Movements für den linken Arm hergenommen werden und für den Rechten übersetzt. Dabei müssen alle Bewegungen bis auf RShoulderPitch und RHand mit (-1) multipliziert werden. Somit wird nun der gesamte Bildschirm mit 81 Positionen (9x9: 9 für Höhe und 9 für Breite) abgedeckt. Das Spiegeln der Mitte erzeugt dabei keine neuen Punkte. Für genauere Klicks reichen jedoch auch die 81 Positionen nicht aus, deshalb werden zwischen diesen Positionen weitere Positionen durch Interpolieren erzeugt, sobald diese gebraucht werden. (Siehe Methode getInterpolatedPosition(left, up)) Dabei wird, wenn der NAO zwischen vier dieser 81 Punkte klicken will, aus diesen vier Punkten einer berechnet. Dafür berechnen wir zunächst aus den Gradzahlen der Motoren für die zwei linken Punkten die Gradzahlen für die dazwischen liegenden Punkt. Der Einfluss der Punkte hängt davon ab, wie nah die jeweiligen Punkte sind. Äquivalent wird das für die zwei rechten Punkte gemacht. Dann haben wir nur noch einen Punkt links und einen Punkt rechts von dem gewünschten Punkt. Diese rechnen wir auch wieder zusammen in Abhängigkeit von dem Abstand zum Zielpunkt. Wenn wir beispielsweise den Punkt (1,5/1,7) anklicken wollen, bedeutet das, dass wir die dazugehörigen Gradzahlen für die Punkte (1/1), (1/2), (2/1) und (2/2) betrachten müssen, also die Punkte, die das kleinstmögliche Rechteck, um die Armposition des gewählten Punkts spannen. Zunächst machen wir aus dem Punkt (1/1) (Gewichtet

## 4 Movement (NG)

mit  $1-0,7 = 0,3$  und  $(1/2)$  (Gewichtet mit  $0,7$ ) den Punkt  $(1/1,7)$  mit zugehöriger Armposition und aus  $(2/1)$  und  $(2/2)$  den Punkt  $(2/1,7)$  mit zugehöriger Armposition. Danach wird äquivalent dazu aus  $(1/1,7)$  und  $(2/1,7)$  der Punkt  $(1,5/1,7)$  mit zugehöriger Armposition gemacht. Ein Nachteil dieser Herangehensweise ist es, dass die interpolierte Armbewegung eine Tiefenverschiebung auslösen kann. Da die Positionen jedoch relativ nah aneinander sind weicht die Tiefe nur wenig ab und die Klicks bleiben präzise genug.

Bei der Bewegung des Arms des NAOs gilt außerdem zu beachten, dass es sehr entscheidend ist, in welcher Reihenfolge und aus welcher Position die einzelnen Motoren bewegt werden, da ansonsten der Roboterarm das Tablet an der falschen Stelle zu erst berührt bzw. umschmeißen kann. Deswegen startet der Roboter immer in einer vordefinierten Position (`armPosition.positionLStart`). Dabei werden zunächst die größeren Bewegungen durchgeführt, da diese mit den restlichen Positionen des Arms der Startpositionen nicht das Tablet erwischen können. Abschließend werden die kleineren Bewegungen durchgeführt wodurch der Klick ausgelöst wird. Mit kurzer Verzögerung werden dann die Bewegungen in umgekehrter Reihenfolge, also zuerst die kleinen Bewegungen dann die Großen, zurück in die Startposition durchgeführt.

Abschließend soll eine konkrete Verbindung zwischen der Armbewegung und dem korrekten Feld auf dem Tablet hergestellt werden. Dies geschieht in der Methode `clickTicTacToe`:

```
def clickTicTacToe(robotIP, port, positionName):
    if positionName == 0:
        armMovement(robotIP, port, arm="L",
                    position=getInterpolatedPosition(left=1.2, up=6), go_back=True)
```

`PositionName` ist dabei eine vordefinierte Position für jedes Feld bei Tic Tac Toe und für jede Reihe bei Vier Gewinnt. Bei Tic Tac Toe ist dabei die Position 0 gleichbedeutend mit dem Feld links oben. Die restlichen Felder werden von links nach rechts oben, dann links nach rechts mitte und links nach rechts unten durchnummert. Bei Vier Gewinnt werden die Spalten von links nach rechts 0 bis 6 durchnummert.

# 5 Spielablauf (NG, JG)

## 5.1 Spielablauf

### 5.1.1 Menüführung und Spielauswahl

Die einfachste und verlässlichste Möglichkeit, mit dem Roboter zu interagieren, um ein Spiel zu starten, sprich das Spiel und den Schwierigkeitsgrad auszuwählen, besteht darin, die drei Knöpfe auf dessen Kopf zu verwenden. Mit diesen lässt sich unkompliziert und sprachgeführt durch die einzelnen Modi navigieren. Genutzt wurden hierfür die  *naoqi*  Module *TextToSpeech* und *Touch*[12].

```
tts = ALProxy("ALTextToSpeech", robotIP, PORT)
tts.say("Hallo, ich bin NAO")
```

Abb. 5.1: Allgemeines Beispiel für die *TextToSpeech*-Funktion

```
touch = ALProxy("ALTouch", robotIP, PORT)
status = touch.getStatus()
counter = 0
for e in status:
    if e[1]:
        print("Button "+str(counter)+" touched")
    counter += 1
```

Abb. 5.2: Allgemeines Beispiel für die *Touch*-Funktion

Die *getStatus*-Funktion gibt ein Array zurück, welches an zweiter Stelle ein *boolean* enthält, welcher *True* für Knopf gedrückt und *False* für Knopf nicht gedrückt anzeigt. Reihenfolge und Aufbau ist wie folgt:

```
0  ['Head', False, []],
1  ['LArm', False, []],
2  ['LLeg', False, []],
3  ['RLeg', False, []],
4  ['RArm', False, []],
5  ['LHand', False, []],
```

## 5 Spielablauf (NG, JG)

```
6  ['RHand', False, []],  
7  ['Head/Touch/Front', False, []],  
8  ['Head/Touch/Middle', False, []],  
9  ['Head/Touch/Rear', False, []],  
10 ['LFoot/Bumper/Left', False, []],  
11 ['LFoot/Bumper/Right', False, []],  
12 ['RFoot/Bumper/Left', False, []],  
13 ['RFoot/Bumper/Right', False, []],  
14 ['LHand/Touch/Left', False, []],  
15 ['LHand/Touch/Back', False, []],  
16 ['LHand/Touch/Right', False, []],  
17 ['RHand/Touch/Left', False, []],  
18 ['RHand/Touch/Back', False, []],  
19 ['RHand/Touch/Right', False, []]
```

"Horcht" man also wie in 5.2 darauf, dass dieser *boolean* auf wahr gesetzt wird, lässt sich einfach ein Menü konstruieren, durch welches man mit den Kopf-Tasten navigieren kann, indem man auf die Indizes 7 (Vorne), 8 (Mitte) und 9 (Hinten) reagiert.

1. Start Program;
2. Choose between *TicTacToe*, *Connect Four* and *End*;
3. Choose between *NAO* playing against opponent or against itself;  
**if NAO vs. Opponent then**
  - Choose, whether *NAO* or opponent makes the first move
  - Choose difficulty
4. Play game;
5. **Go to 2;**

### **Algorithmus 7 : Ablauf Spielauswahl per Knöpfe**

Der Ablauf ist sequentiell, gleich für die beiden implementierten Spiele und letztendlich eine Endlosschleife, sodass man, sollte man das Spielen nicht beenden, so oft spielen kann wie man möchte. Zunächst wird man gefragt, welches Spiel man spielen möchte, dann, ob der *NAO* oder der Gegner beginnen oder der Roboter gegen sich selbst spielen soll. Im Anschluss an ersteres wird noch nach der Schwierigkeitsstufe gefragt, im Falle von *NAO* gegen sich selbst spielt immer Schwierigkeitsgrad "ummöglich" gegen "schwer". Schematisch ist der Spielablauf gegen einen Menschen/anderen *NAO* aus Sicht des *NAO* wie folgt:

```

Initialize field_after_move;
while playing do
    Get game state from algorithm 6 auf Seite 17 or 5 auf Seite 16;
    while game state not detected correctly do
        Wait;
        Get game state again;
        if Not detected correctly five times then
            | End;
    end
    if field not equal to field_after_move then
        Calculate next move and expected field_after_move;
        if Winning move then
            | Celebrate;
            | End;
    end

```

**Algorithmus 8 :** Ablauf NAO vs. Gegner Spiel aus Sicht des NAO

Spielt ein NAO gegen einen anderen NAO oder einen Menschen, so wird zunächst *field\_after\_move* initialisiert, welches nachfolgend den erwarteten Spielstand nach einem Spielzug des NAOs repräsentiert. Danach wird das aktuelle Spielfeld ermittelt, sollte dieses nicht korrekt erkannt werden, versucht es der NAO bis zu fünfmal erneut, bevor er das Spiel abbricht. In diesem Fall liegt entweder ein Problem bei der Spielständerkennung vor, beispielsweise aufgrund von starker Spiegelung auf dem Bildschirm, oder da das Spiel ist zu Ende. Dass der Spielstand vereinzelt nicht auf Anhieb erkannt wird ist jedoch möglich, auch wenn sich beispielsweise noch eine Hand im Sichtfeld des Roboters bewegt hat. Im Anschluss an die erfolgreiche Erkennung des Spielstands wird der aktuelle Spielstand via Algorithmus 6 auf Seite 17 oder 5 auf Seite 16 ermittelt und verglichen. Wenn der Gegenspieler keinen Zug gemacht hat, so unterscheiden sich der ermittelte und errechnete Spielstand nicht, so wartet der NAO eine kurze Zeit und prüft erneut, ob ein Zug gemacht wurde. Falls der Gegenspieler einen Spielzug getätigt hat, unterscheiden sich die Spielstände und der Roboter ist an der Reihe, errechnet also seinen nächsten Zug und führt ihn aus. Sollte der Roboter erkennen, dass er mit dem Zug gewonnen hat, so führt er einen Jubel aus, beendet das aktuelle Spiel und geht zurück in die Spielauswahl.

```

while playing do
    Get game state from algorithm 6 auf Seite 17 or 5 auf Seite 16;
    while game state not detected correctly do
        Get game state again;
        if Not detected correctly five times then
            | End;
    end
    Calculate next move;
    if Winning move then
        | Celebrate;
        | End;
    end

```

**Algorithmus 9 :** Ablauf NAO vs. itself Spiel aus Sicht des NAO

## 5 Spielablauf (NG, JG)

Spielt ein *NAO* gegen sich selbst, wird das aktuelle Spielfeld nach gleichem Ablauf wie bei *NAO vs. opponent* ermittelt. Im Anschluss an die erfolgreiche Erkennung des Spielstands wird nächste Zug im Wechsel von gelb/rot bzw. Kreis/Kreuz errechnet und der Roboter führt ihn aus. Sollte der Roboter erkennen, dass er mit dem Zug gewonnen hat, so führt er einen Jubel aus, beendet das aktuelle Spiel und geht zurück in die Spieldauswahl.

# 6 Flutter app (MS, LK)

## 6.1 Einrichten von Flutter und Erstellen einer APK

### 6.1.1 Flutter Installation

Um Flutter zu installieren, befolge die folgenden Schritte:

1. Besuche die offizielle Flutter-Website: <https://flutter.dev/>.
2. Lade die Flutter-Installationsdatei für Ihr Betriebssystem herunter.
3. Extrahiere das Archiv und lege den Flutter-Binärpfad in Ihrem System-Pfad fest.

### 6.1.2 Flutter Prüfung und Konfiguration

1. Öffne das Terminal und gebe den Befehl `flutter doctor` ein. Dieser Befehl überprüft Ihre Flutter-Installation und gibt Hinweise auf fehlende Komponenten oder Konfigurationsschritte.

2. Befolge die Anweisungen von `flutter doctor`, um fehlende Komponenten wie Android Studio, Xcode oder Android SDK zu installieren und zu konfigurieren.

### 6.1.3 APK erstellen

Navigiere zu dem Projektpfad der Naolympics App im Terminal. Dazu von der Projekt-Root-Ebene in den `naolympics_app` Ordner gehen.

Gebe den folgenden Befehl ein, um eine APK zu erstellen:

```
flutter build apk
```

Nach Abschluss des Build-Vorgangs findet man die generierte APK-Datei im Verzeichnis `build/app/outputs/flutter-apk/` innerhalb des Projektpfads.

## 6.2 Singleplayer

### 6.2.1 Tic Tac Toe

Die Tic Tac Toe App ist eine vergleichsweise einfache Umsetzung eines 3x3 Spielfelds, auf dem abwechselnd rote Kreise und schwarze Kreuze platziert werden. Rot startet immer zuerst, und die Farben wechseln sich in den folgenden Zügen ab, bis ein Gewinner feststeht oder das Spiel unentschieden endet.

Die Seite wird mithilfe eines *StatefulWidget* verwaltet. Je nachdem, ob sich der Nutzer im Multiplayer- oder Singleplayer-Modus befindet, wird der entsprechende Tic Tac Toe Service geladen.

## 6.2.2 Vier gewinnt

Das Vier gewinnt Spielfeld besteht aus mehreren Säulen, in denen die Spieler abwechselnd ihre Münzen durch Berühren des Bildschirms platzieren können. Der erste Zug wird immer von Spieler 1 in gelber Farbe ausgeführt. Eine Besonderheit des Vier gewinnt Singleplayers ist, dass nach jedem Zug eine einsekündige Verzögerung eintritt, die jegliche weiteren Züge in dieser Zeitspanne blockiert. Dies liegt an der Art und Weise, wie der Nao-Roboter den Bildschirm mit der Alufolie berührt. Anstelle einer einzelnen Berührung, die zu einem Zug führen würde, zittert der Finger des Nao's vor dem Bildschirm, und ohne die Verzögerung würde er fünf bis zehn Züge auf einmal ausführen. Die eingebaute Verzögerung umgeht dieses Problem, da der Nao seinen Finger vor Ablauf der Blockierung wieder vom Bildschirm entfernt.

Die Vier gewinnt App setzt sich aus mehreren Komponenten zusammen, die insgesamt das Spielfeld abbilden. Angefangen bei den individuellen Münzen in den Spalten bis hin zur Steuerung der Spiellogik. Jede Komponente ist ein *StatelessWidget*. State Management, Dependency Injection und Routing werden durch das Get-Framework geregelt. Dieses Framework erlaubt es, die verschiedenen Komponenten des Spielfelds bei Änderungen automatisch und performant zu aktualisieren, ohne manuell einen Refresh der verschiedenen Widgets auslösen zu müssen. Die Spiellogik wird von einem *GetXController* namens *GameController* geregelt, der bei Spielstart ein einziges mal initialisiert wird. Durch die *GetXController* Klasse können alle Widgets leicht auf den GameController und dessen Daten zugreifen und so das Spielfeld auf Änderungen überwachen.

## 6.3 Multplayer

### 6.3.1 Verbindungsauftbau und Datenübertragung

Für den Verbindungsauftbau und die Kommunikation innerhalb eines Netzwerks bietet Dart die *Socket* Klasse an. Diese ermöglicht den Aufbau einer TCP Verbindung zwischen zwei Geräten. Für den Verbindungsauftbau wird ein freier Port des lokalen Gerätes und die IP-Adresse des Verbindungspartners benötigt.

Sockets können Daten in Form von Bytes senden und empfangen. Allerdings kann der Byte-Stream jedes Socket Objekts nur von einem einzigen Consumer genutzt werden, was die Nutzung der empfangenen Daten auf eine einzige Stelle im Code oder ein einziges UI-Element beschränkt. Um dieses Problem zu lösen, wird jeder *Socket* in der App genutzt, um ein *SocketManager* Objekt zu erstellen. In einem *SocketManager* wird der Byte-Stream zu einem Broadcast-Stream umgewandelt. Dieser Stream kann von mehreren Consumern gleichzeitig auf neue Events abgehört werden.

Die Daten für den Verbindungsauftbau, die Navigation und den Spielfluss werden im JSON-Format übertragen. Dart bietet die Möglichkeit zur manuellen Serialisierung und Deserialisierung von Daten mithilfe der integrierten Funktion *dart:convert*. Doch diese Methode stößt in mittleren bis größeren Projekten schnell an ihre Grenzen, da für jeden zu kodierenden oder dekodierenden Datentyp die Dekodierungslogik manuell implementiert werden muss. Wir haben uns daher entschieden die *json\_serializable* Library zu nutzen. Diese Li-

brary generiert die notwendige JSON Logik automatisch für alle Datentypen, die mit der Annotation `@JsonSerializable()` versehen werden. Zum Start der Generierung muss nur der Befehl `dart run build_runner build` im Projektverzeichnis ausgeführt werden. So können beliebig viele und komplexe Datentypen mit relativ geringem Aufwand JSON-fähig gemacht werden.

Im Verzeichnis `network` befinden sich mehrere Dart-Klassen, die neben der JSON-Konvertierung das Finden verbindungsbereiter Geräte im lokalen Netzwerk regeln. Bevor der Verbindungsaubau begonnen werden kann, müssen die lokalen IP-Adressen dieser Geräte gefunden und auf der Seite für die Spielersuche des Clients angezeigt werden. Hierfür werden alle möglichen lokalen IP-Adressen im Netzwerk einmal gepingt, und die Adresse jedes antwortenden Geräts angezeigt. Beim anschließenden Verbindungsaubau wird zwischen beiden Geräten eine festgelegte Sequenz von vordefinierten Nachrichten ausgetauscht, um sicherzustellen, dass eine fehlerfreie Verbindung vorliegt.

### 6.3.2 Aufbau der Host-Client Verbindung

Um die Multiplayerfunktion nutzen zu können, müssen sich beide Geräte im selben Netzwerk befinden. Auf der Startseite kann der *Multiplayer* Button ausgewählt werden, der den Nutzer auf die Spielersuche weitergeleitet. Hier muss einer der beiden Spieler den *Host*-Button im unteren rechten Teil des Bildschirms drücken, um für andere Spieler im selben Netzwerk sichtbar zu werden und den Verbindungsaubau zwischen Host und Client zu ermöglichen. Im Hintergrund wird dabei die IP des Gerätes bestimmt und im Falle, dass es einen Mobilen-Hotspot aktiviert hat, wird die IP dessen priorisiert. Mit der bestimmten IP-Adresse wird ein Dart *ServerSocket* erstellt welches sichtbar für andere Spieler ist.

Nach kurzer Zeit sollte die IP-Adresse des Hosts im selben Fester des Clients angezeigt werden und dieser kann sich per Knopfdruck daraufhin verbinden. Läuft der Verbindungsaubau erfolgreich ab, werden beide Spieler automatisch zur *Spieldatenwahl* navigiert.

Die Verbindung kann jederzeit von beiden Spieler durch den *Close connection*-Button in der App-Leiste im oberen Teil des Bildschirms beendet werden. Daraufhin werden Client und Host zurück zur Startseite navigiert.

### 6.3.3 Navigator

#### RouteAwareWidgets

Um den Client über den Wechsel zwischen Seiten in der App zu informieren, müssen diese Änderungen zuerst auf Seiten des Hosts wahrgenommen werden. Dies wird durch *RouteAwareWidgets* erreicht.

*RouteAwareWidgets* sind *StatefulWidget*, deren State das Mixin *RouteAware* verwendet und dessen Methoden überschreibt, um Änderungen des Flutter Navigators, wie `push()`, `pop()` oder `dispose()`, erfassen und in unserem Fall JSON-Objekte vom Typ *NavigationData* an den Client senden zu können.

## 6 Flutter app (MS, LK)

Um die Funktionalität der *RouteAwareWidgets* zu nutzen, muss an den Stellen im Code, an denen normalerweise auf die gewünschte Seite des Navigators verwiesen wird, ein *RouteAwareWidget* initialisiert werden, wobei die gewünschte Seite als *Child-Element* angegeben wird.

Dadurch werden die überschriebenen Methoden bei Seitenwechsel oder Rückkehr ausgeführt, und der Client wird jedes Mal über diese Änderungen informiert.

### **ClientRoutingService**

Nach erfolgreicher Verbindung zum Server wird auf der Client-Seite der *ClientRoutingService* initialisiert und dem *MultiplayerState* hinzugefügt. Nach seiner Initialisierung wartet der entsprechende Broadcast-Stream auf eingehende JSON-Objekte vom Typ *NavigationData*. Basierend auf diesen Daten entscheidet der *ClientRoutingService*, zu welcher Seite des Flutter Navigators gewechselt oder ob zur vorherigen Seite zurückgegangen werden soll.

Sobald jedoch ein Spiel beginnt, sollte der *ClientRoutingService* mit seiner *pause()*-Methode pausiert werden, um eingehende Spieldaten nicht als *NavigationData* zu interpretieren. Nach Beendigung des Spiels und wenn die Client-Navigation durch den Host wieder aktiviert werden soll, muss die *resume()*-Methode des Services aufgerufen werden, und der Client folgt erneut den Routing-Anweisungen des Hosts.

#### **6.3.4 Tic Tac Toe**

Sobald sich der Nutzer auf der Tic Tac Toe Seite befindet, startet der Host das Spiel und beginnt mit dem Setzen von Kreisen. Während des Zuges des einen Spielers muss der andere Spieler warten, bis das Symbol des ersten Spielers platziert wurde. Nach Abschluss des Spielzugs kann der andere Spieler sein Symbol platzieren.

Am Ende des Spiels wird bei beiden Spielern der *Winscreen* angezeigt, jedoch hat nur der Host die Befugnis zu entscheiden, ob eine weitere Runde gespielt wird oder ob zur Spieldauswahl zurückgekehrt wird. Der Host kann auch während eines laufenden Spiels zur Spieldauswahl zurückkehren, wodurch die aktuelle Runde Tic Tac Toe beendet wird und der Client ebenfalls zur Spieldauswahl navigiert wird.

#### **6.3.5 Vier gewinnt**

Ein Vier gewinnt Multiplayer-Spiel verläuft, abgesehen von der anderen Spiellogik, genau wie ein Tic Tac Toe Multiplayer-Spiel. Der Host startet immer als Spieler 1 (gelb), während der Client als Spieler 2 (rot) beginnt. Wenn einer der beiden Spieler gewinnt, wird der *Winscreen* angezeigt, und nur der Host hat die Möglichkeit zu entscheiden, ob das Spiel verlassen oder eine neue Runde gestartet werden soll. Der Client wird dabei automatisch entsprechend den Aktionen des Hosts durch die App weitergeleitet.

# 7 Zusammenfassung und Probleme

Die entwickelten Methoden haben gezeigt, dass es möglich ist, einerseits den *NAO* Roboter eigenständig Spielelogiken und Spielfeld und -standerkennung durchführen, andererseits die daraus resultierenden Spielzüge auch selbst ausführen zu lassen. Sowohl das Spielen von *TicTacToe* als auch *Vier gewinnt* sind keine neuen Ideen, beide wurden schon mehrfach und in verschiedener Ausführung für die Roboter implementiert[19, 22, 20, 21]. Die hier aufgeführten Ansätze basieren jedoch einerseits auf dem Szenario *NAO vs. Mensch*, andererseits benötigen sie auch immer Hilfsmittel wie Stifte oder Spielsteine, um den Spielzug zu tätigen. Der hier entwickelte Ansatz verzichtet auf all das, indem es durch das jederzeit verfügbare Material Aluminiumfolie die Roboter dazu befähigt, eigenständig Klicks auf einem Touchpad zu tätigen. Zudem läuft das entwickelte Python-Skript des Projektes auf dem Roboter selbst und kann auf Wunsch direkt beim Start des *NAO* ausgeführt werden, wobei die Kalibration des Roboters zum Tablet hin jedoch entscheidend für das Spielen ist und vorher ausgeführt werden sollte. Dies lässt sich bei Bedarf vor das eigentliche "Hauptmenü" schalten.

Ein im Rahmen des Projektes nicht mehr zu realisierendes Feature war das jederzeit mögliche Abbrechen des Spiels durch Knopfdruck. Hierzu müsste jedoch nebenläufig programmiert werden, welches sich im verbliebenen Zeitrahmen und den Einschränkungen durch die Python-Version als schwierig erwiesen hat.

Problematisch haben sich bei der Arbeit einerseits die veraltete Python-Version (2.7.18 vs. Stand September 2023 3.11.5), andererseits viele mechanische Probleme mit den Robotern. Von sechs vorhandenen *NAOs* waren nur zwei in der Lage, den rechten Arm korrekt zu bewegen. Somit blieben nur zwei Roboter für das Projekt übrig, einen mit Version 5 (*naoqi* Version 2.1.4, IP: 10.30.4.13), und einen mit Version 6 (*naoqi* Version 2.8.6, IP: 10.30.4.31).

Beim dem Roboter mit der neueren *naoqi* Version (2.8.6) gab es außerdem ein weiteres Problem. Wie sich zeigte stellte der Roboter seinen rechten Arm oft falsch ein, obwohl die übermittelten Werte korrekt waren. In der Abbildung unten sieht man die Differenz zwischen den übergebenen Werten des linken und rechten Arms und den Werten, die tatsächlich von der Sensorik des Roboters erfasst wurden. Dabei fällt bei dem Motor *ShoulderPitch* des

```
Eingestellte Winkel:  
['0.0121758630311', '56.6544764361', '-8.03292113307e-44', '-80.8589663475', '8.03292113307e-44', '57.0726106121']  
[-40.0202908334, '-19.4098379961', '9.21435149858e-31', '31.6160313643', '-8.03292113307e-44', '57.118471261']  
Tatsächliche Winkel:  
['3.33916913172', '55.9250898422', '0.876962758933', '-81.2507465387', '0.789027305166', '57.0726106121']  
[-36.5790986711, '-20.4915711834', '-1.14558001233', '32.6267731418', '-0.442082714883', '57.118471261']
```

rechten Arms auf, dass die Einstellung 40 Grad beträgt, aber der Nao seinen Arm nur 36 Grad nach oben bewegt. Dieser Fehler tritt bei dem besagten Roboter immer wieder auf. Diese Einstellung ist wichtig, um das Tablet zum Roboter korrekt aufzustellen. Ähnliche Ungenauigkeiten treten auch bei den anderen Nao Robotern auf sind hier jedoch nicht so gravierend, wodurch dies nur zu weniger präzisen Klicks führt.



# Literaturverzeichnis

- [1] Adrian Bradski. *Learning OpenCV, [Computer Vision with OpenCV Library ; software that sees]*. O'Reilly Media, 1. ed. edition, 2008. Gary Bradski and Adrian Kaehler.
- [2] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.
- [3] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8:679 – 698, 12 1986.
- [4] kang & atul. Suzuki contour algorithm opencv, 2019. <https://theailearner.com/tag/suzuki-contour-algorithm-opencv/> [Online; accessed 2023-09-02].
- [5] Reinhard Klette. *Concise Computer Vision - An Introduction into Theory and Algorithms*. Undergraduate Topics in Computer Science. Springer, 2014.
- [6] S.K. Nayar. Boundary Detection. In *Monograph FPCV-2-2, First Principles of Computer Vision*, Columbia University, New York, Jun 2022.
- [7] S.K. Nayar. Edge Detection. In *Monograph FPCV-2-1, First Principles of Computer Vision*, Columbia University, New York, May 2022.
- [8] S.K. Nayar. Image Processing i. In *Monograph FPCV-1-4, First Principles of Computer Vision*, Columbia University, New York, Mar 2022.
- [9] OpenCV. Opencv: Cascade classifier, 2023. [https://docs.opencv.org/3.4/db/d28/tutorial\\_cascade\\_classifier.html](https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html) [Online; accessed 2023-09-08].
- [10] OpenCV. Opencv: Template matching, 2023. [https://docs.opencv.org/3.4/de/da9/tutorial\\_template\\_matching.html](https://docs.opencv.org/3.4/de/da9/tutorial_template_matching.html) [Online; accessed 2023-09-08].
- [11] Yann Philippczyk. Implementing deep learning object recognition on nao, 2016.
- [12] Aldebaran Robotics. Nao software 1.14.5 documentation. <http://doc.aldebaran.com/1-14/dev/python/index.html> [Online; accessed 2023-09-01].
- [13] Aldebaran Robotics. Choregraphe installation guide for 2.1, 2023. [http://doc.aldebaran.com/2-1/getting\\_started/installing.html](http://doc.aldebaran.com/2-1/getting_started/installing.html) [Online; accessed 2023-09-13].

## Literaturverzeichnis

- [14] Aldebaran Robotics. Choregraphe installation guide for 2.8, 2023. <http://doc.aldebaran.com/2-8/software/choregraphe/installing.html#desktop-installation-cho-installing> [Online; accessed 2023-09-13].
- [15] Aldebaran Robotics. Python sdk installation guide for 2.1, 2023. [http://doc.aldebaran.com/2-1/dev/python/install\\_guide.html#python-install-guide](http://doc.aldebaran.com/2-1/dev/python/install_guide.html#python-install-guide) [Online; accessed 2023-09-13].
- [16] Aldebaran Robotics. Python sdk installation guide for 2.8, 2023. [http://doc.aldebaran.com/2-8/dev/python/install\\_guide.html#python-install-guide](http://doc.aldebaran.com/2-8/dev/python/install_guide.html#python-install-guide) [Online; accessed 2023-09-13].
- [17] Aldebaran Robotics. Running python code on startup, 2023. [http://doc.aldebaran.com/1-14/dev/python/running\\_python\\_code\\_on\\_the\\_robot.html](http://doc.aldebaran.com/1-14/dev/python/running_python_code_on_the_robot.html) [Online; accessed 2023-09-12].
- [18] Satoshi Suzuki and Keiichi Abe. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32–46, 1985.
- [19] Youtube. Nao robot-playing tic tac toe, 2011. <https://www.youtube.com/watch?app=desktop&v=WLemr3WspXE> [Online; accessed 2023-09-12].
- [20] Youtube. Nao connect 4, 2012. <https://www.youtube.com/watch?v=1xyRxHNwvXM> [Online; accessed 2023-09-12].
- [21] Youtube. Connect4nao: Nao robot plays connect 4, 2016. <https://www.youtube.com/watch?v=UyRx1V4rx-M> [Online; accessed 2023-09-12].
- [22] Youtube. Nao playing tic tac toe, 2016. [https://www.youtube.com/watch?v=zW\\_q048Q16I](https://www.youtube.com/watch?v=zW_q048Q16I) [Online; accessed 2023-09-12].