Mit selbstkonfigurierbarem Greifarm Puzzle lösen

Maja Wantke mwantke@stud.hs-bremen.de
Lara Miritz lmiritz@stud.hs-bremen.de
Nikias Scharnke nscharnke@stud.hs-bremen.de
Sara-Ann Wong swong@stud.hs-bremen.de

Angewandte Mathematik für Medieninformatik
Hochschule Bremen

29. Juli 2024

Zusammenfassung

Diese Dokumentation beschreibt die Entwicklung und Implementierung einer 2D Roboterarm Simulation, die Konzepte der Kinematik praxisnah veranschaulicht. Die Simulation ermöglicht es, einen Roboterarm interaktiv zu steuern und so Puzzleteile zu bewegen. Der Nutzer kann den Roboterarm selbst konfigurieren und so die Komplexität der Steuerung kennenlernen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcode Verzeichnis

1 Motivation

Roboterarmsimulationen sind ein wesentlicher Bestandteil der Robotikforschung. Sie ermöglichen es, komplexe Bewegungsabläufe zu visualisieren und zu verstehen. Unser Projekt soll eine benutzerfreundliche und interaktive Anwendung bereitstellen, die es dem Benutzer ermöglicht, die Funktionsweise eines Roboterarms durch direkte Interaktion zu erforschen. Der Benutzer kann dabei selbst experimentieren und forschen, um eigene Erkenntnisse über die Bewegungssteuerung und Kinematik des Roboterarms zu gewinnen. Dies fördert das Verständnis und die Fähigkeit, theoretisches Wissen in die Praxis umzusetzen.

Roboterarm-Simulationen sind ein wesentlicher Bestandteil der Robotik-Forschung. Sie ermöglichen es, komplexe Bewegungsabläufe zu visualisieren und zu verstehen. Unser Projekt soll eine benutzerfreundliche und interaktive Anwendung bereitstellen, die es dem Benutzer ermöglicht, die Funktionsweise eines Roboterarms interaktiv zu erforschen. Der Benutzer kann dabei selbst experimentieren, um eigene Erkenntnisse über die Bewegungssteuerung und Kinematik des Roboterarms zu gewinnen.

2 Verwandte Arbeiten

Unser Projekt wurde durch verschiedene Arbeiten im Bereich der Roboterarm-Simulation inspiriert. Die Bachelorarbeit zur „Steuerung eines 5-DOF Handhabungsroboters in Arbeitsraumkoordinaten“[1] von Julia Dubcova erklärt die Grundlagen der Robotersteuerung und der inversen Kinematik, die wir für unseren Roboterarm nutzen. Eine weitere wichtige Quelle war die Veröffentlichung „Tactile Robotic Assembly“[2] vom BBSR. Diese half uns, die Interaktion zwischen einem Roboterarm und einem Objekt besser zu verstehen. Beide Arbeiten haben uns wertvolle Ideen geliefert und waren Inspiration für unser Projekt. Ergänzend haben wir uns auch an der neuesten Auflage des Werkes „Computer Animation: Algorithms and Techniques“[3] von Rick Parent, die unter anderem die Grundlagen der Animation Programmierung vermittelt, orientiert. Es behandelt die Grundlagen zu Themen wie Bewegung, Deformation und physikalische Simulationen, um realistische Animationen zu erstellen.

Thematisch ist das Projekt von den oben genannten Arbeiten inspiriert, während die mathematische Berechnung von uns selbstständig hergeleitet wurde. Sie orientiert sich an einem numerischen Lösungsverfahren, bei welchem iterativ mit Näherungsverfahren versucht wird, den Gelenk Vektor zu finden.

3 Projektübersicht

In unserem Projekt wurde ein 2D-Puzzle-Spiel entwickelt, das durch einen Robotergreifarm gelöst werden soll. Der Greifarm wird durch Drag Drop interaktiv gesteuert und verwendet inverse Kinematik, um die Bewegungen präzise auszuführen. Die Steuerung ermöglicht es dem Benutzer, die Komplexität der Roboterbewegungen zu verstehen. Zusätzlich können am Arm verschiedene Konfigurationen vorgenommen werden, so sind die Armlänge und die Anzahl der Gelenke frei wählbar. Die Entscheidungen in der Konfiguration beeinflussen die Komplexität des Roboterarms und können die Steuerung deutlich anspruchsvoller machen.

Der Benutzer kann Ziele per Mausinteraktion festlegen, die der Roboterarm erreichen soll, und Puzzleteile greifen, um sie an eine Position im Raster zu verschieben. Dabei wird durch Kollisionserkennung überprüft, ob eine Überschneidung zwischen einem Puzzleteil und dem Raster auftritt. Ist dies der Fall, so rastet das Puzzleteil im Raster ein. Durch diese interaktive Anwendung erhält der Benutzer die Möglichkeit, die Funktionsweise des Roboterarms praktisch zu erforschen und dabei eigene Experimente durchzuführen. Dies fördert ein tieferes Verständnis der Bewegungsabläufe und der Kinematik von Roboterarmen.

4 Entwicklungsprozess

Der Entwicklungsprozess begann mit der Ideenfindung und Planung, bei der verschiedene Konzepte für die Entwicklung eines Spiels besprochen wurden. Das Ziel war es, eine benutzerfreundliche Anwendung zu schaffen, die es Nutzern ermöglicht, die Funktionsweise eines Roboterarms durch direkte Interaktion zu erforschen und dabei gleichzeitig noch Spielspaß bietet.

Das erste grundlegende Konzept wurde am 10. Juni 2024 in Form eines Exposés abgegeben. Im Anschluss wurde eine Grundstruktur der Anwendung erstellt, die das Layout und die grundlegenden Funktionen umfasste. Diese Struktur bildete die Grundlage für die weitere Entwicklung.

Die erste Besprechung fand am 26. Juni 2024 statt, sie ermöglichte eine umfassende Überprüfung des bisherigen Fortschritts. Dabei entstand die Idee, die Anwendung um eine Konfiguration zu erweitern, um den Nutzern eine detailliertere und individuelle Untersuchung der Kinematik des Roboterarms zu ermöglichen. Diese Erweiterung war nicht von Anfang an geplant, sondern entstand aus der Erkenntnis heraus, den Benutzern mehr Flexibilität bei der Erkundung der Arm Kinematik zu bieten und das Projekt insgesamt etwas komplexer zu gestalten.

Am 12. Juli 2024 wurde der offiziell letzte Stand der Entwicklung vorgestellt. In dieser Präsentation lag der Schwerpunkt auf der neuen Konfiguration, die es den Nutzern ermöglicht, die Kinematik des Roboterarms selbst zu erforschen. Die Demonstration zeigte, wie Benutzer verschiedene Parameter anpassen und die Auswirkungen dieser Anpassungen in Echtzeit beobachten konnten. Diese Funktion ermöglichte eine tiefere Interaktion mit der Anwendung und stellte einen wesentlichen Fortschritt im Vergleich zur ursprünglichen Planung dar.

Vor der Präsentation wurden umfassende Tests durchgeführt, um die Funktionsfähigkeit der Implementierung sicherzustellen. Dazu gehörten Unittests zur Überprüfung der verschiedenen Komponenten der Anwendung, einschließlich der Kollisionserkennung und der Drag-and-Drop-Funktionalität. Diese Tests stellten sicher, dass alle Funktionen stabil und fehlerfrei arbeiteten.

Nach der Präsentation wurde noch eine neue Variante, den Roboterarm mit 3 Gelenken zu steuern, hinzugefügt, sodass die Maus mit dem Endeffektor überlappt und man die volle Kontrolle bei der Steuerung hat. Daneben lag der Fokus auf der Dokumentation und dem Abschlussbericht, in dem die Implementierung, die Ergebnisse und das Feedback aus der Präsentation zusammengefasst wurden. Die abschließende Bewertung reflektierte die Umsetzung des Projekts und bot Raum für mögliche Verbesserungsvorschläge.

5 Implementierung

5.1 Gegenstand der Entwicklung

In einer zweidimensionalen Umgebung soll ein Puzzlespiel entstehen, das mit Hilfe eines Roboterarms gesteuert wird. Dieser Arm kann per Drag-and-Drop gesteuert werden und so die einzelnen Puzzleteile erreichen und aufheben. Die Anzahl der Gelenke und die Länge der einzelnen Armteile sind dabei über ein Menü variabel einstellbar. Sobald das Puzzle gelöst ist, kann es mit Hilfe einer Korrekturfunktion überprüft werden.

5.2 Roboterarm

Um den Roboterarm später steuern zu können, muss dieser zunächst in einer eigenen Klasse konfiguriert werden. Dabei wird er mit Startwerten versehen und erhält Methoden, die die Positionen des Roboterarms bei der Bewegung neu berechnen.

5.2.1 Initialisierung

Um später beim Starten des Programms einen Roboterarm sehen zu können, muss dieser initialisiert werden. Dabei werden die Anzahl der Gelenke auf zwei sowie die Winkel der drei möglichen Gelenke auf 0 Grad festgelegt. Die Länge der einzelnen Armteile wurde zuvor als Konstante festgelegt auf jeweils 150 eingestellt.

Listing 1: Initialisierung der Klassenvariablen für den Roboterarm

```
class RoboticArm:
    # Initialisiert den Roboter Arm mit Startwerten
    def __init__(self, num_joints=2):

    # Anzahl der Gelenke
    self.num_joints = num_joints

    # Winkel vom Roboterarm
    self.shoulder_angle = 0
    self.elbow_angle = 0
    self.wrist_angle = 0

    # Positionen vom Roboterarm
    self.shoulder_pos = np.array([400, SCREEN_HEIGHT / 2])
    self.elbow_pos = 0
    self.wrist_pos = 0

    # Hilfsvariabeln für die 3 Gelenke
    self.initial_guess = [0, 0, 0]
    self.result = [0, 0, 0]

    # Abstände zur Zielposition
    self.dx = 0
    self.dy = 0

    # Armlängen
    self.arm_lengths = [ARM_LENGTH_1, ARM_LENGTH_2, ARM_LENGTH_3]

    # Endeffektor berechnen
    self.update_end_effector()
```

5.2.2 Berechnung Winkel

Während der Arm bewegt wird, werden dauerhaft die Winkel in den Gelenken neu berechnet, um den Endeffektor erreichen zu können. Benötigt werden dafür die Differenzen der x- und y-Koordinaten zwischen dem Endeffektor und dem Schultergelenk beziehungsweise dem Ursprung. Mit diesen kann durch den Satz des Pythagoras die Distanz bestimmt werden, die zur Berechnung der Winkel notwendig ist.

$$c = \sqrt{a^2 + b^2}$$
$$c = \sqrt{80^2 + 60^2} = 100$$

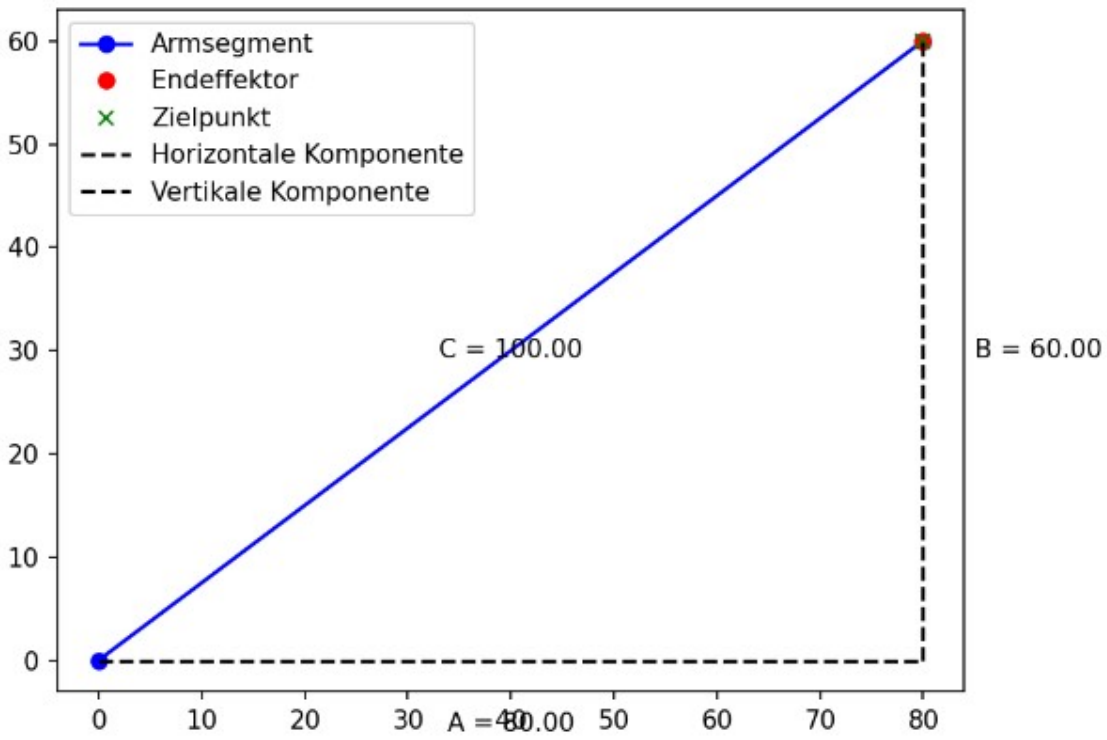

Abbildung 1: Berechnung Distanz

Listing 2: Neuberechnung der Winkel beim Drag

```
self.dx, self.dy = target[0] - self.shoulder_pos[0], target[1] - self.
    shoulder_pos[1]
distance = np.hypot(self.dx, self.dy)
```

Für die Berechnung bei 1, 2 und 3 Gelenken ist unterschiedlicher Code erforderlich. Dieser wird je nach Bedarf durch eine if-else-Anweisung aufgerufen.

Berechnung für 1 Gelenk Handelt es sich um nur ein Gelenk, soll der Endeffektor des Arms in Richtung des Zielpunkts bewegt werden. Für die genauen Koordinaten wird die zuvor berechnete Distanz und der Winkel des Armsegments benötigt. Diesen erhält man durch Einsetzen der Koordinaten in die Arcustangens-Funktion.

$$\theta = arctan(\frac{B}{A})$$

$$\theta = arctan(\frac{60}{80}) = 0,64(36,87°)$$

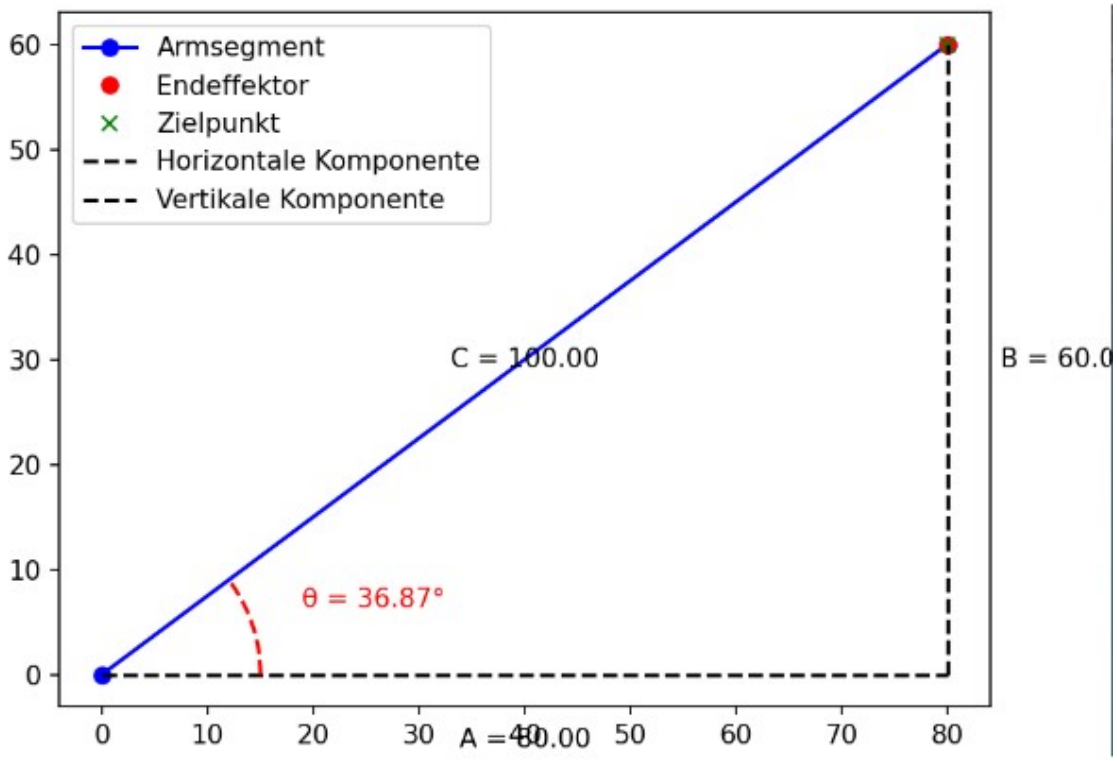

Abbildung 2: Berechnung Schulterwinkel für 1 Armsegment

Listing 3: Winkelberechnung für ein Gelenk

```
if self.num_joints == 1:
    distance = min(distance, self.arm_lengths[0])
    self.shoulder_angle = np.arctan2(self.dy, self.dx)
```

Multipliziert man nun die Armlänge mit dem Sinus beziehungsweise dem Kosinus des Winkels, erhält man den Endeffektor.

$$x = c * cos(\theta)$$

$$y = c * sin(\theta)$$

$$x = 100 * cos(0,64) = 80$$

$$y = 100 * sin(0,64) = 60$$

Listing 4: Berechnung des Endeffektors für ein Gelenk beim Drag

```
    if self.num_joints == 1:
        self.elbow_pos = self.shoulder_pos + np.array([self.arm_lengths[0]
    * np.cos(self.shoulder_angle), self.arm_lengths[0] * np.sin(self.
    shoulder_angle)])
        self.end_effector_pos = self.elbow_pos
```

Berechnung für 2 Gelenke Bei der Berechnung der Winkel mit zwei Armsegmenten wird wie bei einem Armsegment auch zuerst die Distanz zwischen dem Endeffektor und dem Ursprung berechnet. Zusätzlich sind die Längen der beiden Armsegmente gegeben, woraus ein Dreieck mit drei bekannten Seitenlängen entsteht.

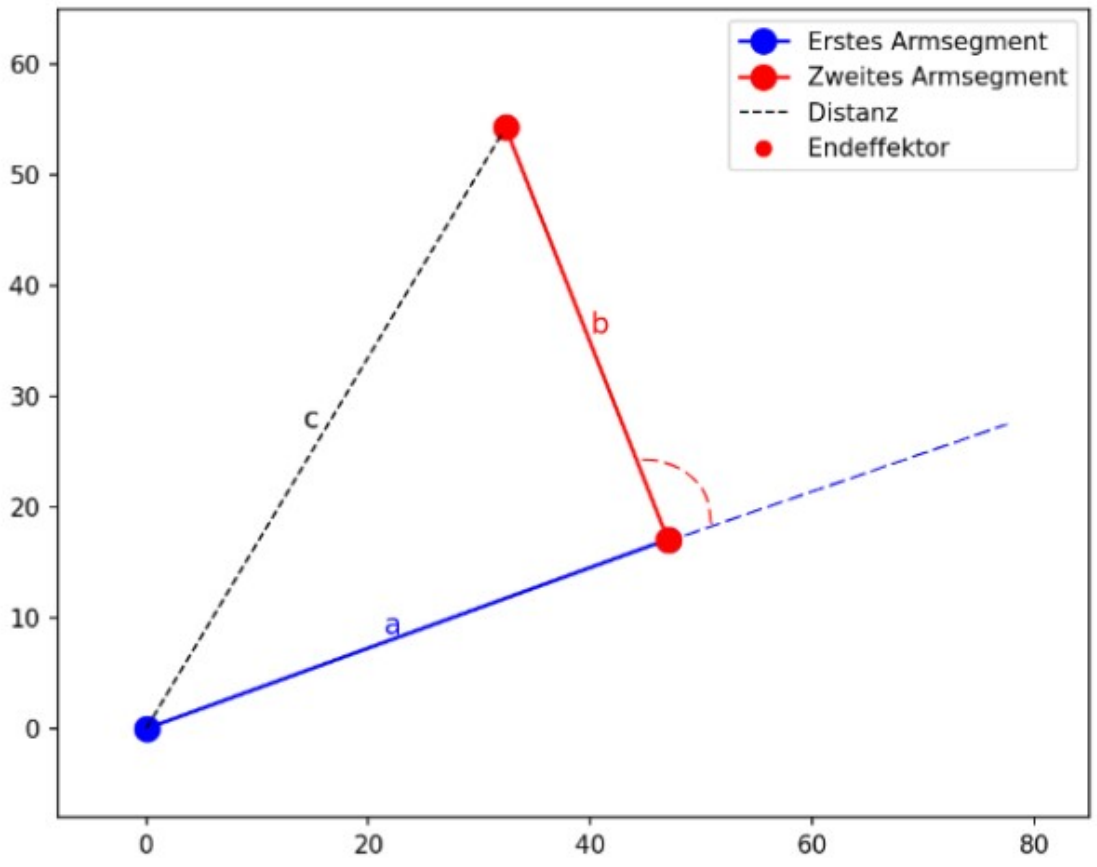

Abbildung 3: Berechnung Ellenbogenwinkel für 2 Armsegmente

$$a = 50$$

$$b = 40$$

$$c = 63,25$$

Für die Berechnung des Ellenbogenwinkels kann nun der Kosinussatz umgestellt nach dem Winkel angewendet werden.

$$cos(\theta) = \frac{a^2 + b^2 - c^2}{2 * a * b}$$
$$cos(\theta) = \frac{50^2 + 40^2 - 63,25^2}{2 * 20 * 40}$$
$$\theta = arccos(0,025) = 89,86°$$

Nachdem der Ellenbogenwinkel berechnet wurde, kann mit dessen Hilfe auch der Schulterwinkel berechnet werden, indem folgende Formel verwendet wird.

$$\alpha = arctan(\frac{y}{x}) - arctan(\frac{b * cos(\theta)}{a + b * sin(\theta)})$$
$$\alpha = arctan(\frac{20}{60}) - arctan(\frac{40 * cos(89,86°)}{50 + 40 * sin(89,86°)})$$
$$\alpha = 18,43° - 37,97° = -19,55°$$

Listing 5: Winkelberechnungen für zwei Gelenke

```
elif self.num_joints == 2:
    distance = min(distance, self.arm_lengths[0] + self.arm_lengths[1])
    cos_angle2 = (distance ** 2 - self.arm_lengths[0] ** 2 - self.
    arm_lengths[1] ** 2) / (2 * self.arm_lengths[0] * self.arm_lengths[1])
    cos_angle2 = np.clip(cos_angle2, -1, 1)
    self.elbow_angle = np.arccos(cos_angle2
    self.shoulder_angle = np.arctan2(self.dy, self.dx) - np.arctan2(self.
    arm_lengths[1] * np.sin(self.elbow_angle), self.arm_lengths[0] + self.
    arm_lengths[1] * np.cos(self.elbow_angle))
```

Nachdem beide Winkel berechnet wurden, kann mit Hilfe der Armlängen der Arm konstruiert werden. Die Positionen des Ellenbogengelenks und des Endeffektors werden durch Einsetzen der Winkel in Sinus- und Kosinusfunktionen multipliziert mit der Armlänge bestimmt.

Ellenbogengelenk:

$$x1 = a * cos(\theta)$$
$$y1 = a * sin(\theta)$$

$$x1 = 50 * cos(-19,55°) = 47,12$$
$$y1 = 50 * sin(.19,55°) = 16,73$$

Endeffektor:

$$x2 = x1 + b * cos(\theta + \alpha)$$
$$y2 = y1 + b * sin(\theta + \alpha)$$

$$x2 = 47,12 + 40 * cos(-19,55° + 89,86°) = 60,6$$
$$y2 = 16,73 + 40 * sin(-19,55° + 89,86°) = 54,4$$

Diese Punkte werden miteinander verbunden und werden dauerhaft aktualisiert, wodurch der Roboterarm dargestellt wird und sich den Bewegungen anpasst.

Listing 6: Berechnung des Endeffektors für zwei Gelenke beim Drag

```
1    elif self.num_joints == 2:
2        self.elbow_pos = self.shoulder_pos + np.array([self.arm_lengths[0]
   * np.cos(self.shoulder_angle), self.arm_lengths[0] * np.sin(self.
   shoulder_angle)])
3        self.end_effector_pos = self.elbow_pos + np.array([self.arm_lengths
   [1] * np.cos(self.shoulder_angle + self.elbow_angle), self.arm_lengths
   [1] * np.sin(self.shoulder_angle + self.elbow_angle)])
```

Berechnung für 3 Gelenke Die Winkel für die 3 Gelenke werden die Formeln genutzt, die man durch die gegebenen Werte aufstellen kann, um die fehlenden Winkel rauszubekommen. Das geschieht durch ein numerisches Optimierungsverfahren, das darauf abzielt, die Differenzen zwischen den berechneten Werten und den zuvor gegebenen Winkeln zu minimieren. Die beiden Formeln werden aus der folgenden Skizze ersichtlich:

$$dx = armlengths[0] * cos(\alpha) + armlengths[1] * cos(\beta) + armlength[2] * cos(\gamma)$$
$$dx = armlengths[0] * sin(\alpha) + armlengths[1] * sin(\beta) + armlength[2] * sin(\gamma)$$

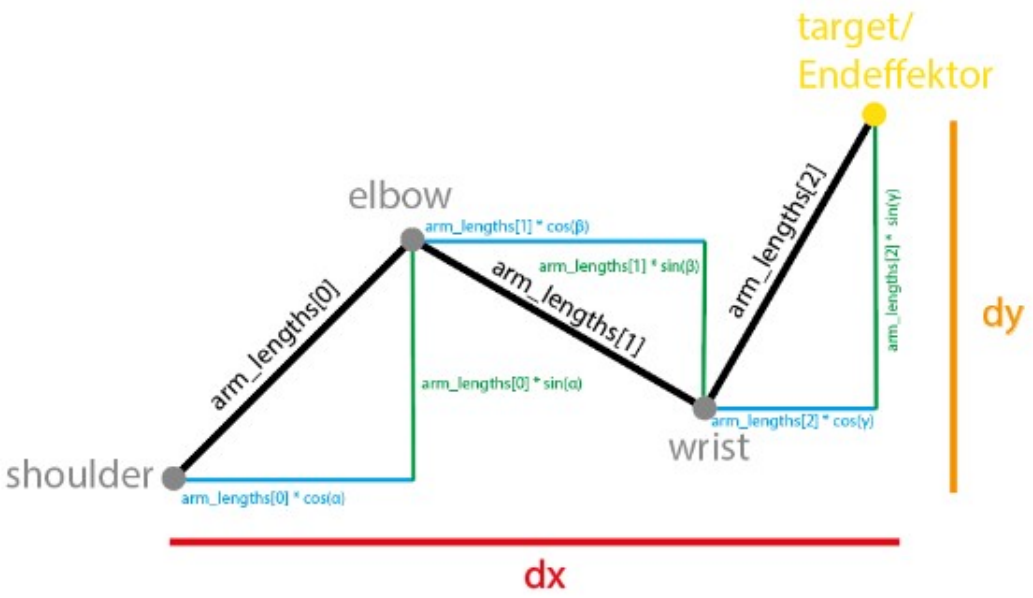

Abbildung 4: Bestimmung Endeffektor für 3 Armsegmente

Listing 7: Berechnung der Positions-Differenzen

```
def equations(self, angles):
    alpha, beta, gamma = angles
    # Kumulierte Winkel
    total_alpha = alpha
    total_beta = alpha + beta
    total_gamma = alpha + beta + gamma
    return [self.arm_lengths[0] * np.cos(total_alpha) + self.arm_lengths[1]
     * np.cos(total_beta) + self.arm_lengths[2] * np.cos(total_gamma) -
    self.dx, self.arm_lengths[0] * np.sin(total_alpha) + self.arm_lengths
    [1] * np.sin(total_beta) + self.arm_lengths[2] * np.sin(total_gamma) -
    self.dy]
```

In dem Beispiel ist die Ausgangsposition vom Roboterarm auf der linken Skizze abgebildet und die Winkel und Armlängen sind:

$$\alpha = 0,5 * \pi$$

$$\beta = 1,5 * \pi$$

$$\gamma = 2,5 * \pi$$

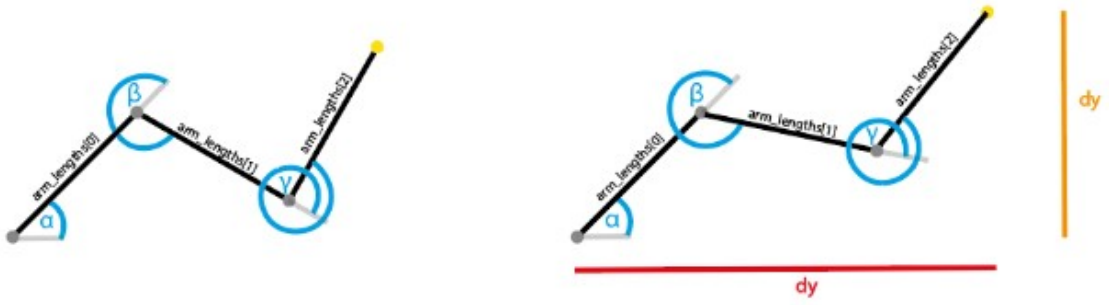

Abbildung 5: Winkelberechnung bei 3 Armsegmenten

$$armlengths[0] = 100$$
$$armlengths[1] = 105$$
$$armlengths[2] = 110$$

$$dx = 250$$
$$dy = 150$$

Diese Werte werden zum Initial Guess, das sind die Winkel, an denen sich der Algorithmus orientiert.

Listing 8: Initial Guess

```
1    self.initial_guess = [self.shoulder_angle, self.elbow_angle, self.
    wrist_angle]
```

Die Formeln sähen mit den gegebenen Werten folgendermaßen aus:

$$0 = 100 * cos(\alpha) + 105 * cos(\beta) + 110 * cos(\gamma) - 250$$
$$0 = 100 * sin(\alpha) + 105 * sin(\beta) + 110 * sin(\gamma) - 150$$

Nun wird mit den obigen aufgestellten Formeln und den Orientierungswerten der least squares- Algorithmus aufgerufen. In dem Ergebnis sind dann die optimalen Winkel für den Roboterarm.

Listing 9: Berechnung des optimalsten Winkel

```
1    self.result = least_squares(self.equations, self.initial_guess)
2    self.shoulder_angle, self.elbow_angle, self.wrist_angle = self.result.x
```

Das Ergebnis der Rechnung lautet somit:

$$self.shoulderangle = 0,27 * \pi$$

$$self.elbowangle = 1,72 * \pi$$

$$self.wristangle = 2,25 * \pi$$

Schließlich müssen auch noch die Positionen aller Gelenke und des Endeffektors berechnet werden. Die Position der Schulter, welche sich beim Bewegen des Roboterarmes nicht mitbewegt, liegt bei [200, 200].

$$elbow = [200, 200] + [100 * cos(0,27 * \pi), 100 * sin(0,27 * \pi)] = [266, 275]$$

$$wrist = [266, 275] + [105 * cos(0,27 * \pi + 1,72 * \pi), 105 * sin(0,27 * \pi + 1,72 * \pi)] = [371, 272]$$

$$endeffector = [371, 272] + [110 * cos(0,27 * \pi + 1,72 * \pi + 2,25 * \pi),$$

$$110 * sin(0,27 * \pi + 1,72 * \pi + 2,25 * \pi)] = [451, 347]$$

Listing 10: Berechnung aller Positionen

```
1        self.elbow_pos = self.shoulder_pos + np.array([self.arm_lengths[0]
   * np.cos(self.shoulder_angle), self.arm_lengths[0] * np.sin(self.
   shoulder_angle)])
2        self.wrist_pos = self.elbow_pos + np.array([self.arm_lengths[1] *
   np.cos(self.shoulder_angle + self.elbow_angle), self.arm_lengths[1] *
   np.sin(self.shoulder_angle + self.elbow_angle)])
3        self.end_effector_pos = self.wrist_pos + np.array([self.arm_lengths
   [2] * np.cos(self.shoulder_angle + self.elbow_angle + self.wrist_angle)
   , self.arm_lengths[2] * np.sin(self.shoulder_angle + self.elbow_angle +
    self.wrist_angle)])
```

5.3 GUI-Klasse

Die GUI-Klasse initialisiert die Hauptkomponenten der Benutzeroberfläche. Sie erstellt ein Canvas-Widget, auf dem der Roboterarm und die Puzzleteile angezeigt werden, sowie verschiedene Konfigurationsoptionen für die Steuerung des Roboterarms.

5.3.1 Konfigurationspanel

Das Konfigurationspanel ermöglicht es dem Benutzer, die Anzahl der Gelenke sowie die Längen der Arme des Roboterarms anzupassen. Diese Einstellungen werden dabei dynamisch aktualisiert, sodass Änderungen in Echtzeit umgesetzt und visuell dargestellt werden.

Listing 11: Panel für das Konfigurationsmenü

```
# Panel für das Menü
def create_config_panel(self):
    config_frame = tk.Frame(self.master, bd=2, relief=tk.RIDGE)
config_frame.place(x=SCREEN_WIDTH - 250, y=10, width=220, height=320)

tk.Label(config_frame, text="Konfiguration").pack(pady=10)
tk.Label(config_frame, text="Anzahl der Gelenke:").pack()

self.num_joints_var = tk.IntVar(value=2)
joints_option_menu = tk.OptionMenu(config_frame, self.num_joints_var, 1, 2,
    3, command=self.update_num_joints)
joints_option_menu.pack(pady=5)

tk.Label(config_frame, text="Armlänge 1:").pack()
self.arm_length1_slider = tk.Scale(config_frame, from_=50, to_=300, orient=
    tk.HORIZONTAL, command=self.update_arm_length1)
self.arm_length1_slider.set(ARM_LENGTH_1)
self.arm_length1_slider.pack()

self.arm_length2_label = tk.Label(config_frame, text="Armlänge 2:")
self.arm_length2_label.pack()
self.arm_length2_slider = tk.Scale(config_frame, from_=50, to_=300, orient=
    tk.HORIZONTAL, command=self.update_arm_length2)
self.arm_length2_slider.set(ARM_LENGTH_2)
self.arm_length2_slider.pack()

self.arm_length3_label = tk.Label(config_frame, text="Armlänge 3:")
self.arm_length3_label.pack()
self.arm_length3_slider = tk.Scale(config_frame, from_=50, to_=300, orient=
    tk.HORIZONTAL, command=self.update_arm_length3)
self.arm_length3_slider.set(ARM_LENGTH_3)
self.arm_length3_slider.pack()
```

Die Methoden *update_num_joints()*, *update_arm_length1()*, *upda-*

te_arm_length2() und *update_arm_length3()* dienen dazu, die Parameter des Roboterarms zu aktualisieren und ihn anschließend neu zu zeichnen. Die Methode *update_num_joints()* aktualisiert die Anpassungsmöglichkeiten des Roboterarms. Je nach der gewählten Anzahl der Gelenke werden die entsprechenden Konfigurationsoptionen des Roboterarms angezeigt oder ausgeblendet. Nach der Aktualisierung wird der Roboterarm neu gezeichnet, um die Änderungen visuell darzustellen.

Listing 12: Aktualisieren der Parameter des Roboterarms

```
# Updatet Anzahl der Gelenke
def update_num_joints(self, value):
    value = int(value)
    if value >= 2:
        self.arm_length2_label.pack()
        self.arm_length2_slider.pack()
    else:
        self.arm_length2_label.pack_forget()
        self.arm_length2_slider.pack_forget()
    if value == 3:
        self.arm_length3_label.pack()
        self.arm_length3_slider.pack()
    else:
        self.arm_length3_label.pack_forget()
        self.arm_length3_slider.pack_forget()

    self.robotic_arm.set_num_joints(int(value))
    self.draw_robotic_arm()

# Aktualisiert den Roboterarm nach dem Ändern des ersten Arms
def update_arm_length1(self, value):
    self.robotic_arm.set_arm_length(0, int(value))
    self.draw_robotic_arm()

# Aktualisiert den Roboterarm nach dem Ändern des zweiten Arms
def update_arm_length2(self, value):
    self.robotic_arm.set_arm_length(1, int(value))
    self.draw_robotic_arm()

# Aktualisiert den Roboterarm nach dem Ändern des dritten Arms
def update_arm_length3(self, value):
    self.robotic_arm.set_arm_length(2, int(value))
    self.draw_robotic_arm()
```

5.3.2 Puzzleteile und Grid (Zielfläche)

Die Puzzleteile werden erstellt und zufällig auf dem Canvas platziert. Ein 2x2-Grid dient als Zielbereich für die Puzzleteile. Ein Bild ist hinterlegt, aus welchem die Puzzleteile erstellt werden. Das Bild ist flexibel austauschbar, solange der Dateiname gleich bezeichnet und im gleichen Verzeichnis wie die Spiel-Datei gelegt wird. Vorerst bildet ein 2x2-Grid bestehend aus Quadraten das Zielfeld, in welches die Puzzleteile hineingelegt werden sollen. Auf dieser Basis wird das Bild auf die benötigte Größe skaliert, sodass das Zielfeld vom Bild abgedeckt wird. Anschließend werden die Puzzleteile mit der vorher definierten Feldgröße erstellt, indem entsprechend große Teile aus dem Bild ausgeschnitten werden. Jedes Puzzleteil wird mit einem Eventlistener verbunden und in eine Liste von Puzzleteilen hinzugefügt.

Listing 13: Erstellung der Puzzleteile

```
# Erstellt die Puzzleteile
def create_puzzle_pieces(self):
    script_dir = os.path.dirname(os.path.abspath(__file__))
    image_path = os.path.join(script_dir, "math-puzzles-image.jpg")
    self.puzzle_image = Image.open(image_path)
    self.puzzle_image = self.puzzle_image.resize((2 * GRID_SIZE, 2 *
    GRID_SIZE), Image.Resampling.LANCZOS)
    self.puzzle_pieces = []
    positions = [(0, 0), (0, 1), (1, 0), (1, 1)] # 2x2 Grid
    random.shuffle(positions) # Zufällige Anordnung der Puzzleteile
    for i in range(2):
        for j in range(2):
            box_image = self.puzzle_image.crop((j * GRID_SIZE, i *
    GRID_SIZE, (j + 1) * GRID_SIZE, (i + 1) * GRID_SIZE))
            box_image = ImageTk.PhotoImage(box_image)
            pos = positions.pop()
            piece = self.canvas.create_image(450 + pos[0] * (BOX_SIZE + 10)
    , SCREEN_HEIGHT / 2 - 100 + pos[1] * (BOX_SIZE + 10), image=box_image,
    tags="puzzle_piece")
            self.canvas.tag_bind(piece, "<Button-3>", self.on_right_click)
            self.puzzle_pieces.append((piece, (i, j), box_image))
            self.canvas.tag_raise(piece)
```

Das Grid wird auf ähnliche Weise erstellt wie die Puzzleteile. Hierbei werden Quadrate in der zuvor definierten Feldgröße ohne Lücken aneinandergesetzt. Abschließend

wird jedem Quadrat eine Position zugewiesen.

Listing 14: Erstellung des Grids

```
# Erstellt Grid für das Puzzle
def create_grid(self):
    offset_x = 600
    offset_y = SCREEN_HEIGHT / 2 - 100
    self.grid_positions = {}
    for i in range(2):
        for j in range(2):
            x0, y0 = offset_x + j * GRID_SIZE, offset_y + i * GRID_SIZE
            x1, y1 = offset_x + GRID_SIZE + j * GRID_SIZE, offset_y +
    GRID_SIZE + i * GRID_SIZE
            square = self.canvas.create_rectangle(x0, y0, x1, y1, fill='
    white', outline='black', tags="grid")
            self.grid_positions[square] = (i, j)
```

5.3.3 Puzzleteile bewegen

Um den Roboterarm zu bewegen, muss dieser mit einem Linksklick ausgewählt werden. Dabei wird zunächst überprüft, ob sich der Abstand zwischen dem Mauszeiger und dem Endeffektor innerhalb eines Radius von 10 befindet.

Listing 15: Linksklick auf den Endeffektor

```
# Endeffektor auswählen
def on_press(self, event):
    if self.is_near_end_effector(event.x, event.y):
        self.dragging = True
```

Listing 16: Feld-Definition in dem der Endeffektor ausgewählt werden kann

```
# Definiert das Feld in dem der Endeffektor ausgewählt werden kann
def is_near_end_effector(self, x, y):
    ex, ey = self.robotic_arm.end_effector_pos
    return np.hypot(ex - x, ey - y) < 10
```

Während der Bewegung des Arms wird die Methode *on_ drag()* kontinuierlich ausgeführt. In diesem Prozess werden fortlaufend die Position des Endeffektors sowie die des ausgewählten Puzzleteils an die Position des Mauszeigers angepasst.

Listing 17: Fortlaufende Anpassung der Endeffektor Position

```
# Arm bewegen
def on_drag(self, event):
    if self.dragging:
        target = np.array([event.x, event.y])
        self.robotic_arm.move_to(target)
        self.draw_robotic_arm()
        if self.selected_piece:
            piece, pos, box_image = self.selected_piece
            ex, ey = self.robotic_arm.end_effector_pos
            self.canvas.coords(piece, ex, ey)
```

5.3.4 Puzzleteile aufnehmen und ablegen

Durch einen Rechtsklick soll der Endeffektor ein Puzzleteil „greifen“ können. Falls der Greifarm noch kein Puzzleteil aufgenommen hat, wird ihm das Puzzleteil zugewiesen, dessen Koordinaten mit der Position des Endeffektors übereinstimmen. Danach wird der Endeffektor zentriert auf das Puzzleteil positioniert.

Wenn der Greifarm bereits ein Puzzleteil aufgenommen hat, werden die Koordinaten des Puzzleteils mit den Koordinaten der Quadrate im Zielfeld verglichen. Ein Puzzleteil wird auf einem Quadrat platziert, wenn der Abstand zwischen den Koordinaten kleiner als 3 ist. Diese Bedingung wird durch den logischen Ausdruck in der Methode *is_ inside()* überprüft. Ist dies zutreffend, wird das Puzzleteil auf das entsprechende Quadrat im Grid gesetzt. Diese Vorgehensweise trägt zur Verbesserung der Benutzerfreundlichkeit bei.

Listing 18: Aufnehmen und ablegen von Puzzleteilen

```
# Puzzleteil wenn möglich aufheben
def on_right_click(self, event):
    if self.selected_piece:
        piece, pos, box_image = self.selected_piece
        piece_coords = self.canvas.coords(piece)
        for square in self.canvas.find_withtag("grid"):
            square_coords = self.canvas.coords(square)
            if self.is_inside(square_coords, piece_coords):
                self.canvas.coords(piece, square_coords[0] + (BOX_SIZE / 2)
    , square_coords[1] + (BOX_SIZE / 2))
                break
        self.selected_piece = None
    else:
```

```
        items = self.canvas.find_overlapping(event.x, event.y, event.x,
    event.y)
        for item in items:
            if item in [piece for piece, pos, box_image in self.
    puzzle_pieces]:
                self.selected_piece = [(piece, pos, box_image) for piece,
    pos, box_image in self.puzzle_pieces if piece == item][0]
                piece, pos, box_image = self.selected_piece
                ex, ey = self.robotic_arm.end_effector_pos
                self.canvas.coords(piece, ex, ey)
```

Listing 19: Überprüfung, ob das Puzzleteil mit einer Toleranz im Grid liegt

```
# Überprüft ob das Puzzleteil im Grid liegt
def is_inside(self, square_coords, piece_coords):
    tolerance = 3
    piece_x0 = piece_coords[0] - BOX_SIZE / 2
    piece_y0 = piece_coords[1] - BOX_SIZE / 2
    piece_x1 = piece_coords[0] + BOX_SIZE / 2
    piece_y1 = piece_coords[1] + BOX_SIZE / 2
    return (square_coords[0] - tolerance <= piece_x0 and
            square_coords[1] - tolerance <= piece_y0 and
            square_coords[2] + tolerance >= piece_x1 and
            square_coords[3] + tolerance >= piece_y1)
```

5.3.5 Überprüfung der Position

Die Überprüfung beginnt erst nach Betätigung des Knopfes „Check Positions“. Bei diesem Schritt werden die hinterlegten Positionen der Puzzleteile mit den aktuellen Positionen verglichen, um sicherzustellen, dass alle Teile korrekt platziert sind. Ein Zähler wird nur dann erhöht, wenn die Positionen der Puzzleteile mit den vorgegebenen Positionen übereinstimmen. Dieser Zähler gibt nach der Überprüfung an, wie viele Puzzleteile korrekt positioniert wurden.

Listing 20: Überprüfung der Position

```
# Überprüft ob die Puzzleteile richtig liegen
def check_positions(self):
    correct_positions = 0
    for piece, pos, box_image in self.puzzle_pieces:
        piece_coords = self.canvas.coords(piece)
        for square in self.canvas.find_withtag("grid"):
```

```
            grid_pos = self.grid_positions[square]
            square_coords = self.canvas.coords(square)
            if self.is_inside(square_coords, piece_coords) and pos ==
    grid_pos:
                correct_positions += 1
    if correct_positions == 4:
        print("Herzlichen Glückwunsch! Alle Puzzleteile sind korrekt
    platziert!")
    else:
        print(f"{correct_positions} von 4 Puzzleteilen sind korrekt
    platziert.")
```

5.4 Kollisionserkennung

Diese Mechanismen tragen dazu bei, eine präzise und benutzerfreundliche Interaktion mit dem Roboterarm und den Puzzleteilen zu gewährleisten. Die Kollisionserkennung ist entscheidend für die korrekte Funktionalität der Anwendung und die Genauigkeit der Interaktionen.

5.4.1 Kollisionserkennung beim Drag & Drop von Puzzleteilen

Ein zentraler Aspekt der Kollisionserkennung betrifft das Drag Drop von Puzzleteilen. Hierbei wird die Methode *on_ right_ click(self, event)* verwendet, um den Greifarm Puzzleteile mit der rechten Maustaste aufnehmen oder ablegen zu lassen. Um sicherzustellen, dass das Puzzleteil korrekt auf einem Quadrat positioniert wird, wird die Methode *is_ inside(self, square_ coords, piece_ coords)* verwendet. Diese Methode überprüft, ob die Randkoordinaten des Puzzleteils innerhalb der Grenzen eines Grid-Quadrats liegen, wobei eine Toleranz von 3 berücksichtigt wird. Diese Toleranz hilft, kleine Abweichungen in der Positionierung auszugleichen und gewährleistet eine präzise Platzierung der Puzzleteile, was zu einer höheren Benutzerfreundlichkeit beiträgt.

5.4.2 Kollisionserkennung bei der Positionierung des Endeffektors

Zusätzlich zur Platzierung von Puzzleteilen ist die Kollisionserkennung auch bei der Positionierung des Endeffektors von Bedeutung. Die Methode *on_ press(self, event)* wird aktiviert, wenn der Benutzer die Maus in der Nähe des Endeffektors klickt, um den Drag-Modus zu starten. Um festzustellen, ob sich der Mauszeiger in der Nähe

des Endeffektors befindet, verwendet die Methode *is_ near_ end_ effector(self, x, y)* die euklidische Distanz zwischen der Position des Mauszeigers und der Position des Endeffektors. Ein Radius von 10 wird hierbei als Kriterium verwendet, um die Nähe zu überprüfen.

5.4.3 Kollisionserkennung während der Armbewegung

Während der Armbewegung wird die Methode *on_ drag(self, event)* aufgerufen, die es ermöglicht, dass das ausgewählte Puzzleteil der Bewegung des Roboterarms folgt. Hierbei wird sichergestellt, dass das Puzzleteil immer entsprechend der Position des Endeffektors aktualisiert wird.

6 Erweiterungsmöglichkeiten

Das Projekt bietet mehrere Ansatzpunkte für zukünftige Erweiterungen. Eine wesentliche Möglichkeit besteht in der Erweiterung der Konfigurationsmöglichkeiten des Roboterarms. Derzeit können Benutzer grundlegende Parameter anpassen, um die Kinematik des Arms zu erforschen. Weitere Entwicklungen könnten es ermöglichen, die Konfiguration des Roboterarms noch detaillierter vorzunehmen. Dazu könnten erweiterte Optionen zur Feinabstimmung der Gelenkwinkel und zur Anpassung der Geschwindigkeiten gehören sowie zusätzliche Konfigurationsparameter, die eine noch präzisere Anpassung und Untersuchung der Bewegungsabläufe ermöglichen.

Neben den funktionalen Erweiterungen besteht auch die Möglichkeit zur Verbesserung des Designs der Benutzeroberfläche. Die aktuelle Benutzeroberfläche bietet eine funktionale Grundstruktur, jedoch besteht Potenzial für eine ansprechendere und intuitivere Gestaltung. Eine Überarbeitung könnte detailreiche Grafiken und eine Anleitung zur Interaktiven Nutzung, beinhalten. Die Benutzerfreundlichkeit würde durch diese Änderungen deutlich verbessert werden, denn sie ermöglichen eine klarere und intuitivere Navigation durch die Simulation.

7 Schlussfolgerung

Das entwickelte 2D-Puzzle-Spiel mit Roboterarm Steuerung bietet eine Grundlage für die interaktive Simulation der Roboterkinematik. Durch die Möglichkeit, verschiedene Konfigurationen des Roboterarms zu testen, können Benutzer ein tieferes

Verständnis für die Funktionsweise und Bewegungsabläufe von Roboterarmen entwickeln. Dabei haben sich die flexible Konfigurationsmöglichkeit und die Drag-and-Drop-Funktionalität als zentrale Merkmale der Anwendung herausgestellt.

Während der Umsetzung des Projekts haben wir wertvolle Erkenntnisse gewonnen. Besonders hervorzuheben ist, dass die Implementierung eines Roboterarms mit drei Gelenken deutlich komplexer und fehleranfälliger war als ursprünglich erwartet. Die Herausforderungen bei der mathematischen Modellierung und der kinematischen Steuerung haben gezeigt, wie anspruchsvoll die Entwicklung eines solchen Systems sein kann. Insbesondere bei der exakten Positionierung der Endeffektoren, traten immer wieder Schwierigkeiten auf. Es benötigte mehrere Anläufe und unterschiedliche Herangehensweisen, um die präzise Berechnung der Gelenkwinkel und deren Koordination erfolgreich umzusetzen. Der Prozess umfasste einige Anpassungen und häufige Fehlersuche, doch zum Schluss gelang die genaue Positionierung des Endeffektors.

Trotz der sorgfältigen Planung und Implementierung sind weiterhin Verbesserungen und Erweiterungen möglich, um die Funktionalität und Benutzerfreundlichkeit weiter zu optimieren. Die gesammelten Erfahrungen haben unsere Kenntnisse in der Roboterkinematik und der praktischen Anwendung von Simulationstechniken erheblich erweitert.

Literatur

[1] Julia Dubcova: *Steuerung eines 5-DOF-Handhabungsroboters in Arbeitsraumkoordinaten*
URL: https://reposit.haw-hamburg.de/handle/20.500.12738/5263
(Stand: 24.07.2024 15:30 Uhr)

[2] Prof. Dr.-Ing. Oliver Tessmann et al.: *Tactile Robotic Assembly*
URL: https://www.bbsr.bund.de/BBSR/DE/veroeffentlichungen/bbsr-online/2024/bbsr-online-05-2024-dl.pdf
(Stand: 24.07.2024 15:25 Uhr)

[3] Rick Parent, *Computer Animation: Algorithms and Techniques*
2. Auflage, Morgan Kaufmann, ISBN 9780124158429