FH Vorarlberg
University of Applied Sciences

Entwicklung einer Open-Source Programmierhilfe für Roboterzellen

Masterarbeit
zur Erlangung des akademischen Grades

Master of Science in Engineering (MSc)

Fachhochschule Vorarlberg
Mechatronics

Betreut von
Prof. (FH) Dipl.-Ing. Robert Amann

Vorgelegt von
Thomas Lerchbaumer, BSc

Durchgeführt bei
Hirschmann Automotive GmbH, Rankweil

Dornbirn, August 2021

Kurzreferat

Entwicklung einer Open-Source Programmierhilfe für Roboterzellen

Die Arbeit beschreibt die Entwicklung eines Open-Source Plugins für die 3D-Modellierungssoftware FreeCAD, mit welchem es möglich ist, Roboterpfade anhand von 3D-Modellen zu erstellen. Die Pfade sind in einem geeigneten Format exportierbar und können beispielsweise zur Steuerung eines Roboters durch eine speicherprogrammierbare Steuerung verwendet werden. Im ersten Teil der Arbeit wird der geplante Arbeitsablauf der Pfadgenerierung beschrieben und genauer erläutert, welche Vorteile und Pflichten die Erstellung von Open-Source-Software nach sich zieht. Anschließend wird anhand der Systeme „Robot Studio“ von ABB und „MotoSim EG VRC“ von Yaskawa analysiert, wie proprietäre Systeme die Programmierung von Roboterpfaden realisieren. Nach einem Überblick über den aktuellen Stand der Technik in der Roboterprogrammierung, wird die Implementierung des Plugins für FreeCAD beschrieben. Dazu wird anhand des Sourcecodes erläutert, wie neue Arbeitsbereiche erstellt werden können. Es werden verschiedene Funktionen implementiert, welche essenziell für die Erstellung von Roboterpfaden sind. Dazu zählen die Möglichkeit zur Definition von Koordinatensystemen, Roboterposen und das Beschreiben des Roboterpfades durch Pfadsegmente mit verschiedenen Parametern, wie Bewegungsart, Geschwindigkeit und Wegpunkte. Das Plugin wurde getestet indem eine einfache Pick & Place Anwendung erstellt wurde. Anschließend sind mögliche Erweiterungen des Plugins, wie zum Beispiel die Möglichkeit des Duplizierens von Pfadsegmenten am Ende der Arbeit beschrieben.

Abstract

Development of an Open Source Programming Tool for Robot Cells

The thesis describes the development of an open-source plugin for the 3D modeling software FreeCAD, with which it is possible to create robot paths based on 3D models. The paths are exportable in a suitable format and can be used, for example, to control a robot by a programmable logic controller. In the first part of the paper, the planned workflow of the path generation is described and it is explained in more detail what advantages and obligations the creation of open source software entails. Then, using the systems „Robot Studio“ from ABB and „MotoSim EG VRC“ from Yaskawa, it is analyzed how proprietary systems realize the programming of robot paths. After an overview of the current state of the art in robot programming, the implementation of the plugin for FreeCAD is described. For this purpose, the source code is used to explain how new workbenches can be created. Various functions are implemented, which are essential for the creation of robot paths. These include the ability to define coordinate systems, robot poses, and describe the robot path using path segments with various parameters such as motion type, velocity, and waypoints. The plugin was tested by creating a simple pick & place application. Subsequently, possible extensions of the plugin, such as the possibility of duplicating path segments, are described at the end of the thesis.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

SPS Speicherprogrammierbare Steuerung

TCP Tool Center Point

UX User-Experience

FEM Finite Elemente Methode

OSS Open-Source-Software

LGPL Lesser/Library General Puplic License

JSON JavaScript Object Notation

P2P Punkt-zu-Punkt

1 Einleitung

Im Bereich der industriellen Roboterprogrammierung sind proprietäre Entwicklungsumgebungen sehr weit verbreitet. So gut wie jeder Roboterhersteller stellt für die Programmierung der Steuerungen eigene Software zur Verfügung. Oft wird die Logik sogar in einer proprietären Programmiersprache implementiert. Es gibt auch Programmier- und Simulationssysteme die es ermöglichen, Steuerungen von unterschiedlichen Roboterherstellern zu programmieren.

Eines dieser Systeme dafür ist beispielsweise RoboDK. Dabei handelt es sich um eine Simulationsumgebung in der Roboterprogramme für verschiedene Steuerungen in einer 3D-Ansicht erstellt und simuliert werden können. [13]

Eine weitere Möglichkeit bieten standardisierte Kommunikationsschnittstellen für Robotersteuerungen. Die meisten Roboterhersteller stellen für ihre Steuerungen Schnittstellen zur Verfügung, durch die Fahrbefehle über verschiedene netzwerkbasierte Kommunikationsmittel von übergeordneten Steuerungssystemen, wie beispielsweise einer Speicherprogrammierbare Steuerung (SPS), an die Robotersteuerung übertragen werden können. Ein weit verbreiteter Standard, welcher den Grundaufbau der Schnittstelle definiert, wurde durch den Verband PLCOpen erstellt, in dem verschiedenste Unternehmen im Bereich der industriellen Automatisierung vertreten sind. In den Dokumenten „Function Blocks for motion control Part 1–6“ wird die Schnittstelle zwischen SPS und Robotersteuerungen standardisiert. Als Grundlage dient dabei der weltweit genutzte Standard IEC 61131-3, welcher die Programmiersprachen für SPSen behandelt. Dieser Ansatz wird in der vorliegenden Arbeit verwendet, um die Erstellung eines Roboterpfades unabhängig vom konkreten Robotersystem zu ermöglichen. [9]

1.1 Ziele

Ziel der folgenden Arbeit ist es, ein Plugin für die Open-Source Modellierungssoftware FreeCAD zu erstellen, welches die Erstellung eines Roboterpfades ermöglicht. FreeCAD kann durch zahlreiche, teilweise von der Community erstellten Werkzeuge für die unterschiedlichsten Anwendungen im Ingenieursbereich eingesetzt werden. Zu den Hauptanwendungen zählen Produktdesign, Maschinenbau, Finite Elemente Methode (FEM) und Architektur. Dieses Plugin soll

als Programmierhilfe für Roboterzellen dienen. Dazu soll es dem Benutzer möglich sein, eine „.step“-Datei in FreeCAD zu importieren. In der 3D-Ansicht sollen dann Punkte definiert werden und diese mit verschiedenen Bewegungsbefehlen (linear, zirkulär, Punkt-zu-Punkt (P2P)) verknüpft werden können. Diese Pfadinformationen sollen in der 3D-Ansicht visualisiert sein und in einem geeigneten Format exportierbar sein. Der generierte Pfad kann somit weiterverwendet werden, um beispielsweise einen Roboter über eine SPS zu steuern. Dabei soll der erstellte Pfad als Vorlage für die genauere Anpassung am realen System dienen. Durch das erstellte Plugin soll es möglich sein, Roboterpfade für Roboter zu erstellen, ohne die jeweilige proprietäre Programmierumgebung der einzelnen Roboterhersteller zu verwenden.

Anhand eines Beispiels wird zum Schluss der Arbeit überprüft, welche Teile des Plugins noch überarbeitet werden müssen. Außerdem wird analysiert, welche weiteren Funktionen hilfreich wären und der Workbench hinzugefügt werden könnten.
Zur Bearbeitung der Aufgabestellungen in dieser Masterarbeit werden die im Folgenden beschriebenen Ressourcen verwendet.

FreeCAD 0.19

Als Grundprogramm für die Erstellung des Plugins wird FreeCAD in der Version 0.19 verwendet.[6]
Dabei handelt es sich um ein Open-Source 3D-Modellierungstool für welches Plugins und Skripte zur Individualisierung oder Automatisierung von Arbeitsschritten erstellt werden können. In FreeCAD enthalten ist außerdem ein Python 3.8.6 Interpreter. Workbenches können in FreeCAD entweder in Python oder C++ implementiert werden. In dieser Arbeit wird das Plugin in Python erstellt.

Qt Designer 5.11.1

Zur Erstellung der benötigten Dialoge für die Interaktion des Benutzers mit dem Plugin wird die Software Qt Designer in der Version 5.11.1 verwendet.[10] Qt Designer ist eine grafischer Editor zur Erstellung von User Interfaces für unterschiedliche Anwendungen.

1.2 Arbeitsablauf

Für die Umsetzung der Arbeit wird vorausgesetzt, dass die „.step“-Datei der Roboterzelle vorhanden ist und dass die Maße der 3D-Zeichnung mit der Rea-

lität übereinstimmen.
Der geplante Arbeitsablauf für die Erstellung eines Roboterpfades ist in Abbildung 1.1 dargestellt. Das 3D-Modell der Roboterzelle wird in die Modellierungssoftware importiert. Anschließend werden die benötigten Koordinatensysteme definiert. Nun werden die für die Pfadsegmente verwendeten Roboterposen erstellt. Mit diesen Informationen können die einzelnen Bewegungssegmente für den Roboter erstellt und in der 3D-Ansicht hinsichtlich Kollisionen und Richtigkeit überprüft werden. Die Pfadinformationen können als JavaScript Object Notation (JSON) Struktur exportiert werden und sind für die weitere Verwendung verfügbar.

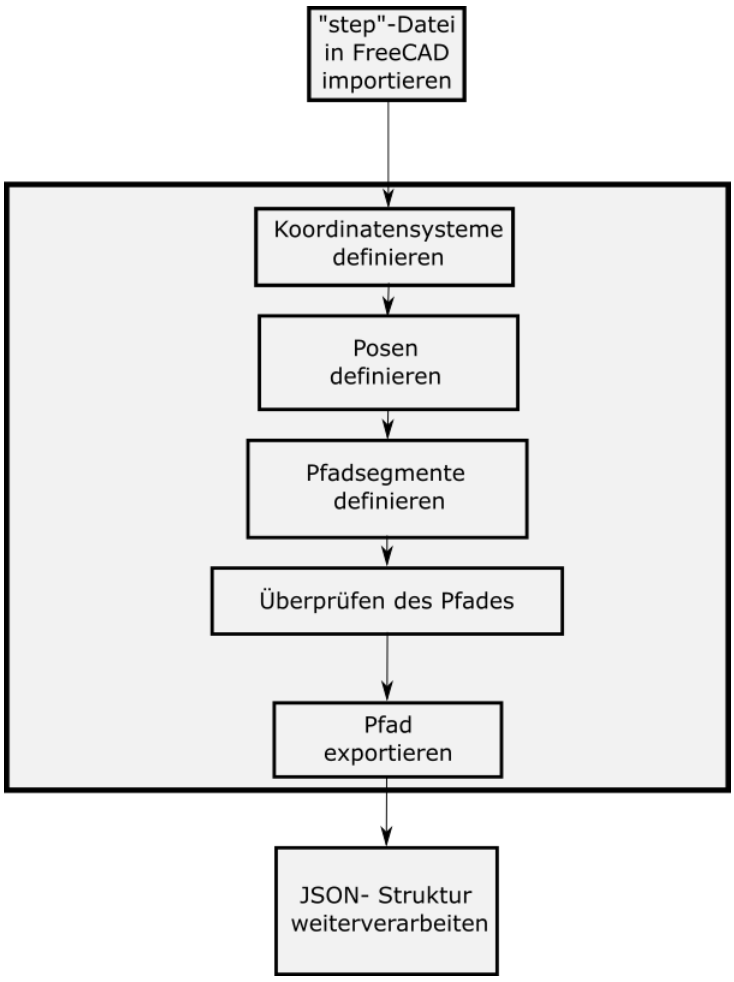

Abbildung 1.1: Ablauf der Pfadgenerierung.
Quelle: eigene Ausarbeitung

1.3 Open-Source-Software

Als Open-Source-Software (OSS) wird Programmcode bezeichnet, der dem Endbenutzer zur freien Verfügung gestellt wird. Das heißt, dass die Software dem Benutzer ohne zusätzliche Kosten in Form des Quellcodes zur Anzeige und beliebigen Veränderung zugänglich ist. Oft ist OSS günstiger als proprietäre Software beziehungsweise teilweise sogar gratis, weil diese oft nicht von einzelnen Unternehmen sondern von einer breiten Community entwickelt wird. Im Zusammenhang mit OSS stößt man oft auf die Begriffe „freie Software“ und „Freeware“. „Freie Software“ kommt dabei der Idee von OSS sehr nahe. Dabei geht es vor allem um die Freiheit der Endbenutzer, die Software anzupassen und zu analysieren. Für „freie Software“ kann, wie bei OSS auch, eine Bezahlung zur Nutzung verlangt werden. Oft werden deshalb die Begriffe OSS und „freie Software“ als Synonyme betrachtet. Im Gegensatz dazu wird Software als „Freeware“ bezeichnet, wenn diese ohne Bezahlung zur Verfügung gestellt wird. Der Sourcecode muss dem Endnutzer im Gegensatz zu OSS nicht zur Verfügung gestellt werden. [3]
Im Folgenden werden einige Werte der Open-Source Community beschrieben: [12]

- Peer-Review:
  Der Code ist frei zugänglich und wird meist von der Community aktiv geprüft und verbessert

- Transparenz:
  Änderungen am Code können selbst überprüft und nachvollzogen werden

- Zuverlässigkeit:
  Open-Source Programmcode wird von vielen verschiedenen Nutzern kontinuierlich aktualisiert, verbessert und getestet

- Flexibilität:
  Der Code kann auf die eigenen Anforderungen angepasst werden

- Offene Zusammenarbeit:
  Aktive Communities stecken Hilfe, Ressourcen und Perspektiven zur Verfügung, die über einzelne Interessensgruppen und Unternehmen hinausgehen

[11]

Lizenzmodelle bei Open-Source Projekten

Es gibt verschiedene Lizenzmodelle, welche die Regeln zur Nutzung und Verwendung von zur Verfügung gestellter Software definieren. Eine Open-Source Lizenz muss dabei dem Endnutzer erlauben, die Software frei zu nutzen, zu modifizieren und weiterzugeben. Oft liegt bei OSS-Lizenzen der größte Unterschied in den Vorgaben für die Weitergabe der Software.
Dabei gibt es zwei grundlegende Typen an Lizenzen:

- non-permissive Lizenzen:
  beispielsweise GNU General Public License
  Lizenzen mit strengeren Vorgaben (Copyleft-Lizenzen)

- permissive Lizenzen:
  beispielsweise MIT, BSD
  weniger Einschränkungen für Verwendung der Software(-komponente)

Copyleft-Lizenzen

Bei Copyleft-Lizenzen müssen alle Weiterentwicklungen der Software ebenfalls unter den ursprünglichen Bedingungen lizenziert werden. Dies gilt für alle Softwareprodukte in die der verwendete Code einfließt, sowie alle Veränderungen am Sourcecode.
Non-Copyleft-Lizenzen können Programmcode in neue Projekte einfließen lassen und diese sogar proprietär lizenzieren. [4]

Die Lizenz, unter der die 3D-Modellierungssoftware FreeCAD veröffentlicht wird, ist eine GNU Lesser/Library General Puplic License (LGPL) Lizenz. Sie ist eine Non-Copyleft-Lizenz. Somit wird dem Endnutzer die Freiheit eingeräumt, die Software mit weniger strengen Vorgaben in neue Projekte einfließen zu lassen. Die einzelnen Module des in dieser Arbeit erstellten Plugins werden ebenfalls unter der GNU LGPL Lizenz veröffentlicht.

2 State of the Art

2.1 Begriffserklärungen

Für das bessere Verständnis und die eindeutige Bezeichnung werden in diesem Kapitel die üblicherweise in der Robotik verwendeten Begriffe näher erklärt. Als Anhaltspunkte wurden die Werke „Modern Robotics“ von Kevin M. Lynch und Frank C. Park [7, S. 325-326] sowie „Introduction to Robotics“ von John J. Craig [2] herangezogen.

- Position:
  Als Position wird die translatorische Verschiebung eines Objektes relativ zu einem gegebenen Ursprungs(-koordinatensystem) bezeichnet. In dieser Arbeit beschreibt sie die kartesischen Koordinaten des Roboter Tool Center Point (TCP) in Bezug auf ein vorher definiertes Koordinatensystem.

- Orientierung:
  Die Orientierung beschreibt, ergänzend zur Position, die Rotation eines Objektes relativ zur Rotation des Ursprungs(-koordinatensystem). In dieser Arbeit werden dafür jeweils die ZYX-Eulerwinkel des Roboter-TCP spezifiziert. In FreeCAD wird die Pose eines Objekts ebenfalls in ZYX-Eulerwinkeln definiert. In der folgenden Arbeit werden die Rotationen um die jeweiligen Achsen wie folgt bezeichnet:

  Yaw (Gierwinkel) für die Rotation um die Z-Achse R_z

  Pitch (Nickwinkel) für die Rotation um die Y-Achse R_y

  Roll (Rollwinkel) für die Rotation um die X-Achse R_x

- Pose/Konfiguration:
  Als Pose beziehungsweise Konfiguration wird die Zusammenfassung der Position und Orientierung eines Objektes zu einem Informationspaar bezeichnet. Durch die Pose ist also Position und Orientierung eines Objektes vollständig im dreidimensionalen Raum definiert. In der Robotik wird die Angabe der einzelnen Achstellungen eines Roboter oft als Konfiguration bezeichnet. Durch die Winkelstellungen aller Roboterachsen werden Position und Orientierung des Roboter-TCP ebenfalls eindeutig definiert.

- Pfad:
  Als (Roboter-)Pfad wird die geordnete Aneinanderreihung verschiedener Posen bezeichnet, die weitere Angaben zu den Bewegungen zwischen den einzelnen Posen enthält. Zusätzliche Informationen können dabei die Bewegungsart (Linear, P2P, Kreisförmig), Geschwindigkeit, etc. sein.

2.2 Proprietäre Umsetzung

Da es für die Ausarbeitung der eigenen Umsetzung interessant ist, wie die Aufgabenstellung in proprietärer Software gelöst wird, wird im folgenden Kapitel die Umsetzung der geplanten Grundfunktionen in den Systemen „Robot Studio“ von ABB und „MotoSim EG-VRC“ von Yaskawa betrachtet. Aus den gewonnenen Erkenntnissen werden Ideen für die eigene Lösung abgeleitet. Es wird versucht, dass sich gute Ausarbeitungen der proprietären Systeme im eigenen Plugin wieder finden, beziehungsweise für die gegeben Aufgabenstellung optimiert werden.

2.2.1 Koordinatensysteme

In Abbildung 2.1 ist die Definition von Koordinatensystemen in der Simulationssoftware MotoSim von Yaskawa zu sehen. Einstellbare Parameter sind dabei der zugehörige Roboter, für den das Koordinatensystem erstellt wird, ein übergeordnetes Basiskoordinatensystem sowie jeweils die Verschiebung in X-, Y- und Z-Richtung und die Rotation um die jeweiligen Achsen.

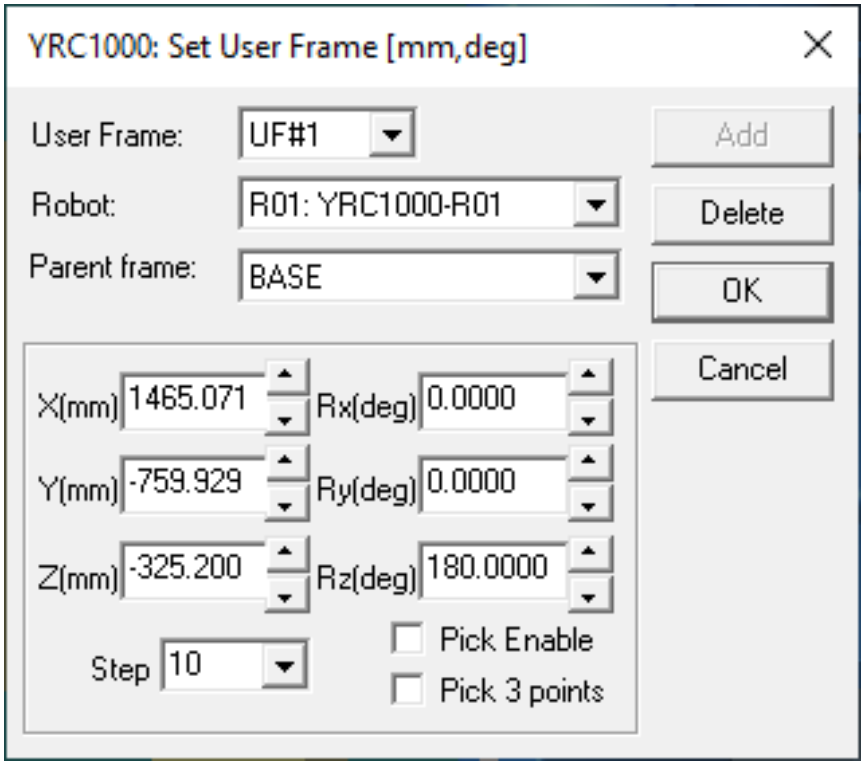

Abbildung 2.1: Koordinatensystem erstellen (Yaskawa)
Quelle: eigene Ausarbeitung

Abbildung 2.2 zeigt die Erstellung von Koordinatensystemen in Robot Studio. Die benötigten Daten sind ebenfalls die Position und Orientierung des Koordinatensystems, bezogen auf ein gegebenes Referenzkoordinatensystem. An dieser Stelle ist es möglich, die Position des Koordinatensystems einzustellen, indem man in eines der drei Eingabefenster für die Position klickt und anschließend einen markanten Punkt in der 3D-Ansicht von Robot Studio auswählt. Die X-, Y- und Z-Koordinaten des Punktes werden in die Eingabefelder übernommen. Für die Orientierung ist dies nicht möglich.

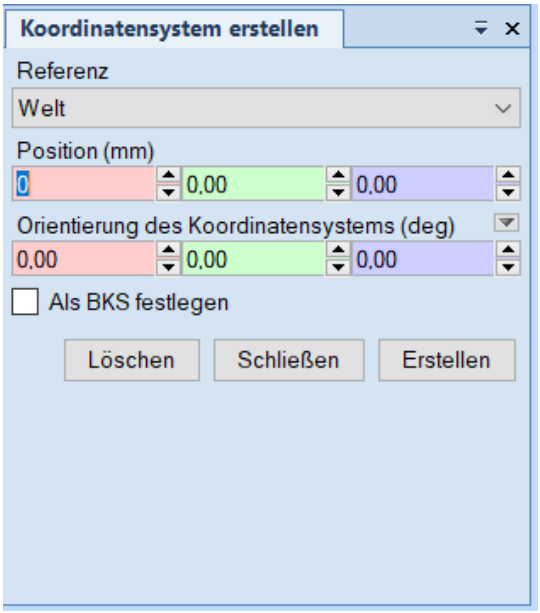

Abbildung 2.2: Koordinatensystem ertellen (Robot Studio)
Quelle: eigene Ausarbeitung

2.2.2 Posendefinition

Die Eingabemaske zur Definition von Roboterposen ist in Abbildung 2.3 zu sehen. Relevante Parameter für die Pose sind das Referenzkoordinatensystem, sowie die Position und Orientierung der Zielpose in diesem. Außerdem wird für jede Pose das verwendete Werkzeug definiert. Wie in Abbildung 2.5 ersichtlich können Posen im dreidimensionalen Raum für Roboter nicht eindeutig durch kartesische Koordinaten definiert werden. Teilweise kann eine Zielpose, die in kartesischen Koordinaten gegeben ist, durch mehrere unterschiedliche Kombinationen von Achsstellungen des Roboters erreicht werden. Deshalb kann bei Yaskawa zusätzlich angegeben werden, wie die definierte Position erreicht werden soll.

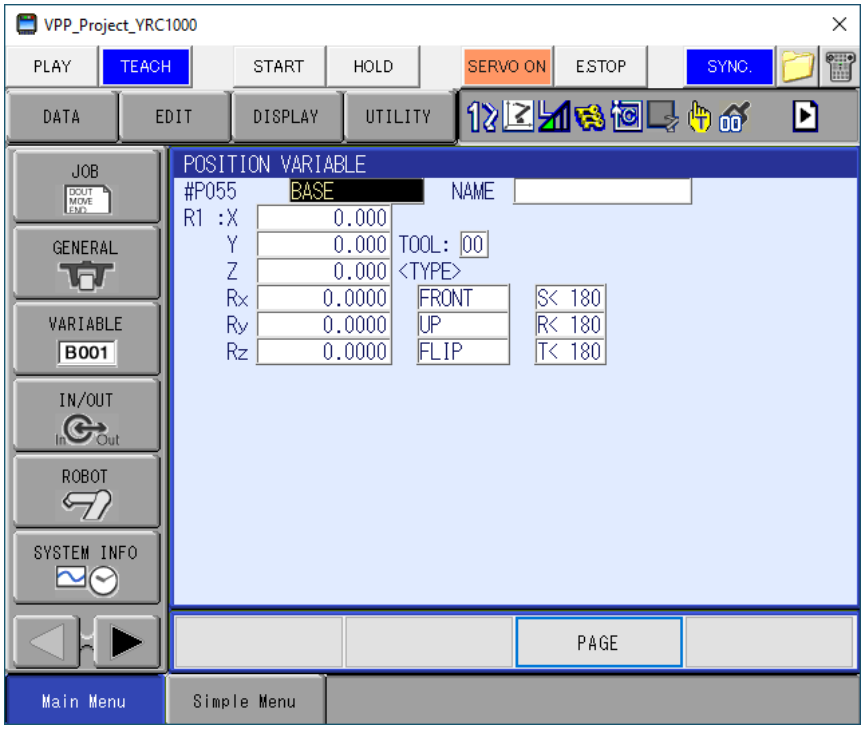

Abbildung 2.3: Eingabemaske Posendefinition (Yaskawa)
Quelle: eigene Ausarbeitung

Robot Studio hat für Posen-Definition die in Abbildung 2.4 gezeigte Eingabemaske. Die Position wird durch ein Koordinatensystem, Position und Orientierung definiert. An dieser Stelle ist es, wie bei der Erstellung eines Koordinatensystems, ebenfalls möglich die X-, Y- und Z-Koordinaten des Punktes in die Eingabefelder zu übernommen. Bei Roboterposen ist dies auch für die Orientierung möglich, indem eine der dafür vorgesehene Eingabefelder ausgewählt und beispielsweise ein Koordinatensystem in der 3D-Ansicht angeklickt wird. Die Orientierung des Koordinatensystems wird für die Orientierung der Pose übernommen.

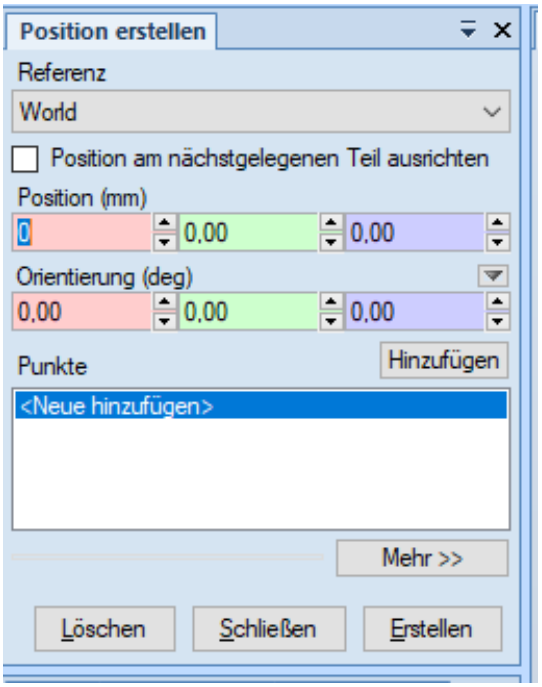

Abbildung 2.4: Eingabemaske Posendefinition (Robot Studio)
Quelle: eigene Ausarbeitung

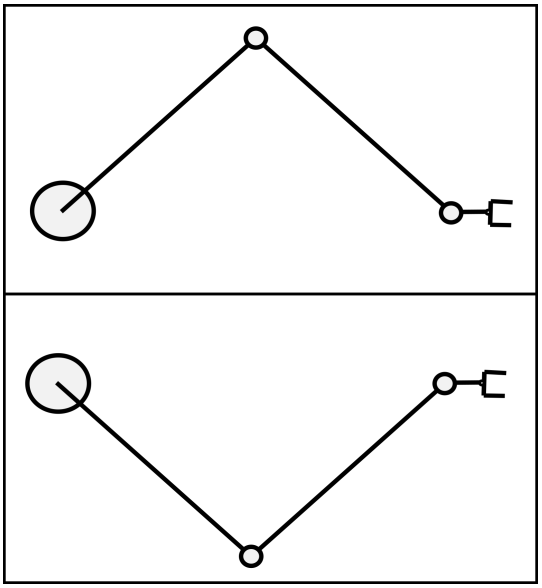

Abbildung 2.5: Uneindeutigkeit von Roboterposen
Quelle: eigene Ausarbeitung

2.2.3 Pfaddefinition

Beim System von Yaskawa können Pfade nicht grafisch programmiert werden. Die Pfade werden, wie in Abbildung 2.6 ersichtlich, durch die proprietäre Programmiersprache INFORM definiert. Dabei gibt es grundsätzlich die folgenden drei Arten von Fahrbefehlen:

- MOVL für Linearbewegung
- MOVJ für Punkt-zu-Punkt-Bewegung
- MOVC für Kreisbewegung

Benötigte Parameter für die jeweiligen Fahrbefehle sind die Zielpose, Geschwindigkeit, Beschleunigung/Verzögerung, sowie Informationen zum Überschleifen. MOVC benötigt noch eine weitere Pose für die Kreisinterpolation. Als Überschleifen wird das Abkürzen von Fahrwegen durch Starten des nächsten Fahrbefehls, bevor die aktuelle Zielposition erreicht wurde, bezeichnet. In Abbildung 2.9 ist zu sehen, dass beim Überschleifen Posen teilweise nicht mehr genau angefahren werden. Wie stark die Bewegungen abgekürzt werden, kann konfiguriert und für jede Bewegung separat definiert werden.

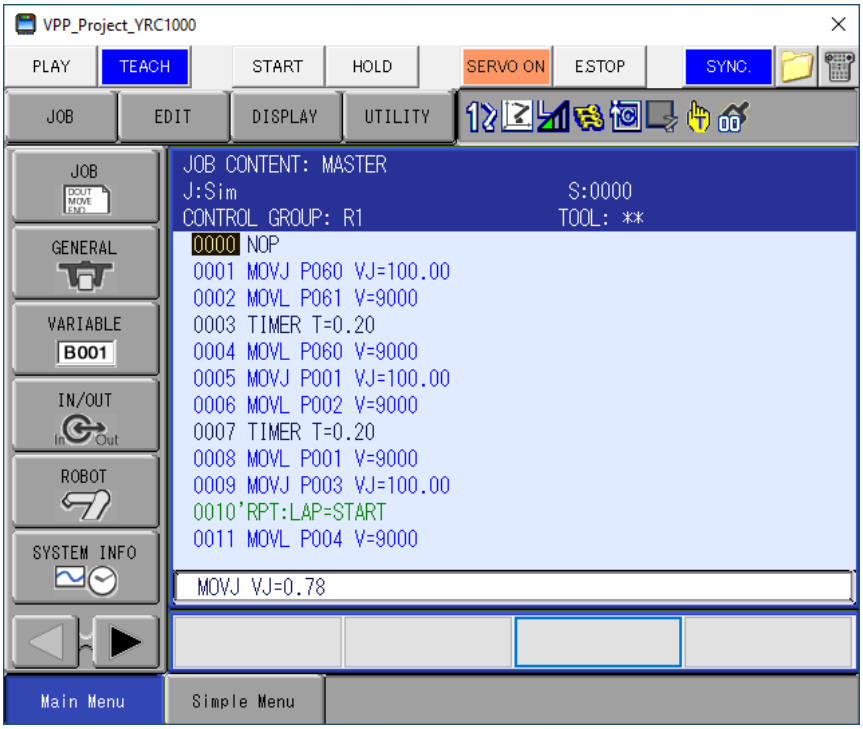

Abbildung 2.6: Roboterpfad programmieren (Yaskawa)
Quelle: eigene Ausarbeitung

Bei Robot Studio von ABB gibt es zwei unterschiedliche Möglichkeiten Pfade für die Roboterbewegung zu definieren. Zum einen können Pfade automatisch aus vorhandenen Kurven und Kanten erzeugt werden. In Abbildung 2.7 ist das Eingabemenü für die Auto-Pfadgenerierung abgebildet. Es können dabei Offsets, Toleranzen und die Bewegungsart definiert werden. Eine zweite Möglichkeit ist in Abbildung 2.8 zu sehen. Die Definition von manuellen Pfaden kann erfolgen, indem zuvor erstellte Wegpunkte aneinandergereiht und die Bewegungsparameter (Bewegungsart, Überschleifen, etc.) spezifiziert werden.

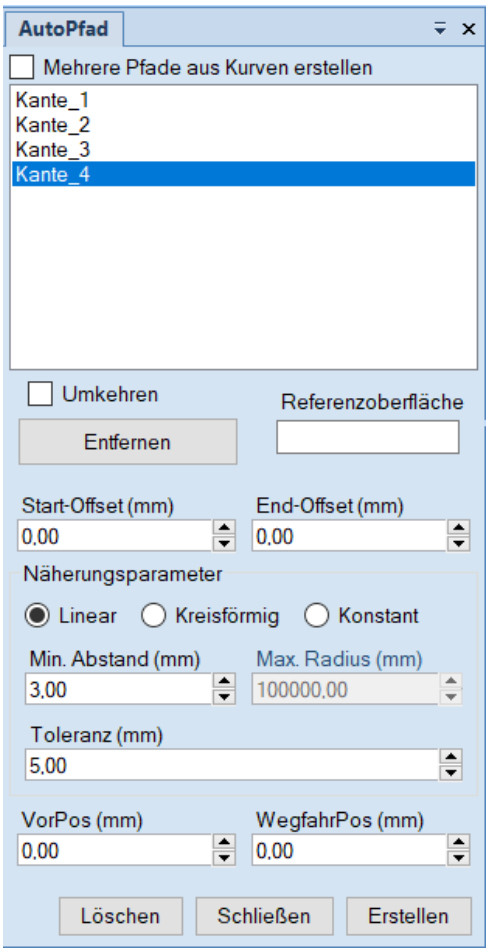

Abbildung 2.7: Auto-Roboterpfad (Robot Studio)
Quelle: eigene Ausarbeitung

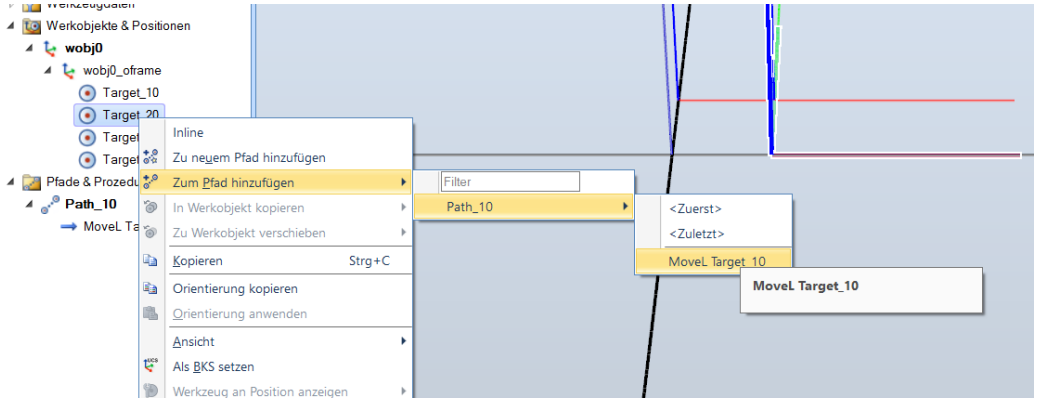

Abbildung 2.8: manueller Roboterpfad (Robot Studio)
Quelle: eigene Ausarbeitung

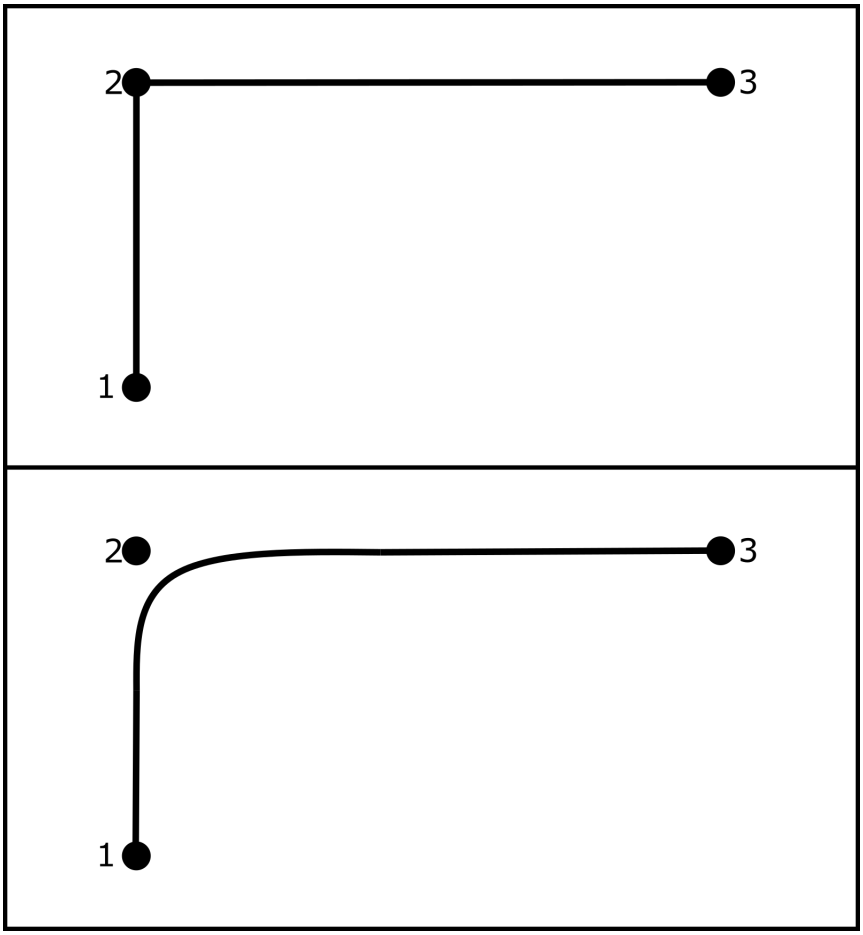

Abbildung 2.9: Überschleifen von Fahrbefehlen
Quelle: eigene Ausarbeitung

2.2.4 Visualisierung

In MotoSim werden die Roboterpfade durch Simulation des Roboterprogramms dargestellt. Dabei kann, wie in Abbildung 2.10 zu sehen ist, ein Teil des Roboterpfades als rote Linie visualisiert werden. Diese Linie zeigt dabei aber, je nach Länge des Roboterpfades, nicht den Pfad für das gesamte Roboterprogramm an.

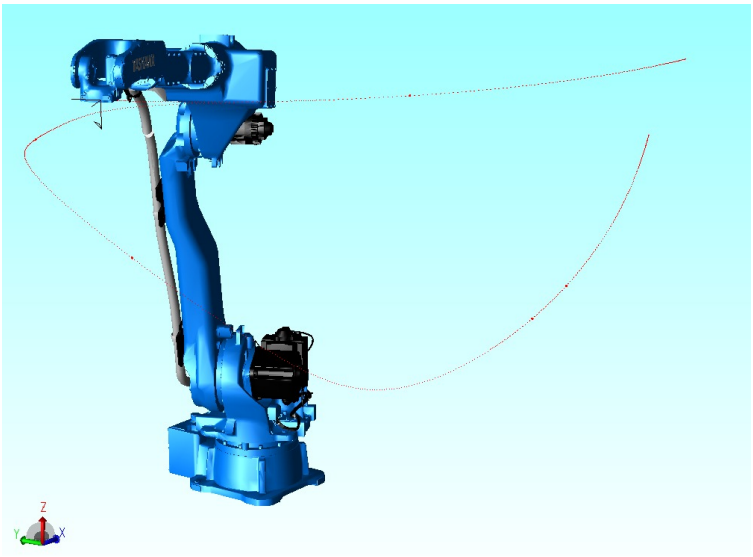

Abbildung 2.10: Visualisierung des Roboterpfades (Yaskawa)
Quelle: eigene Ausarbeitung

Die erstellten Roboterpfade werden in Robot Studio, wie in Abbildung 2.11 gezeigt, als gelbe Linien visualisiert. Ein Pfeil in der Mitte des jeweiligen Pfadsegments stellt dabei die Richtung der Bewegung dar.

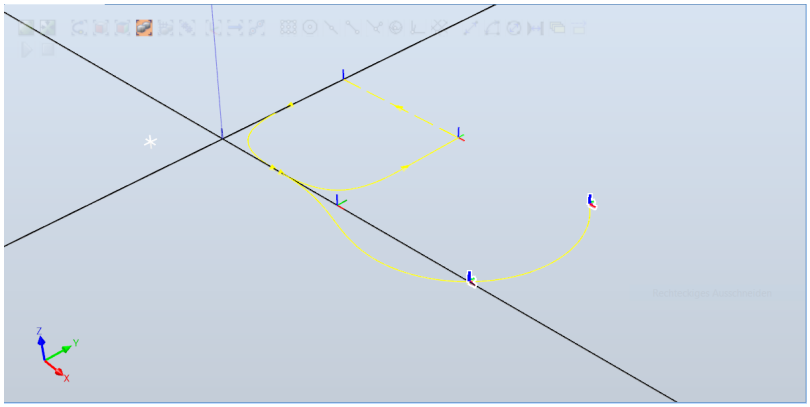

Abbildung 2.11: Visualisierung des Roboterpfades (Robot Studio)
Quelle: eigene Ausarbeitung

2.2.5 Wiederverwendbarkeit

In Abbildung 2.12 ist zu sehen, wie eine Wiederverwendbarkeit von Roboterpfaden in MotoSim von Yaskawa realisiert werden kann. Es wurde ein Roboter-

programm (Job: Test2) erstellt, welches den Roboter-TCP einen Roboterpfad durchfahren lässt. Zusätzlich wurde ein weiteres Roboterprogramm (Job: Test) erstellt. In diesem Programm fährt der Roboter-TCP vordefinierte Positionen an. In Zeile 0004 wird der Job Test2 aufgerufen. Es werden also die in diesem Job definierten Bewegungen ebenfalls ausgeführt. In den Zeilen 0008–0010 wird zuerst eine Verschiebung der folgenden Roboterposen aktiviert. Diese Verschiebung wird in der Variable P003 genauer definiert. Dabei können alle folgenden Positionen in X-,Y- und Z-Richtung verschoben, und um die jeweiligen Achsen rotiert werden. Anschließend wird der Job Test2 erneut aufgerufen. Die Bewegungen werden nun, verschoben um die Werte in der Variable P003, erneut ausgeführt. Anschließend muss die Verschiebung wieder deaktiviert werden.

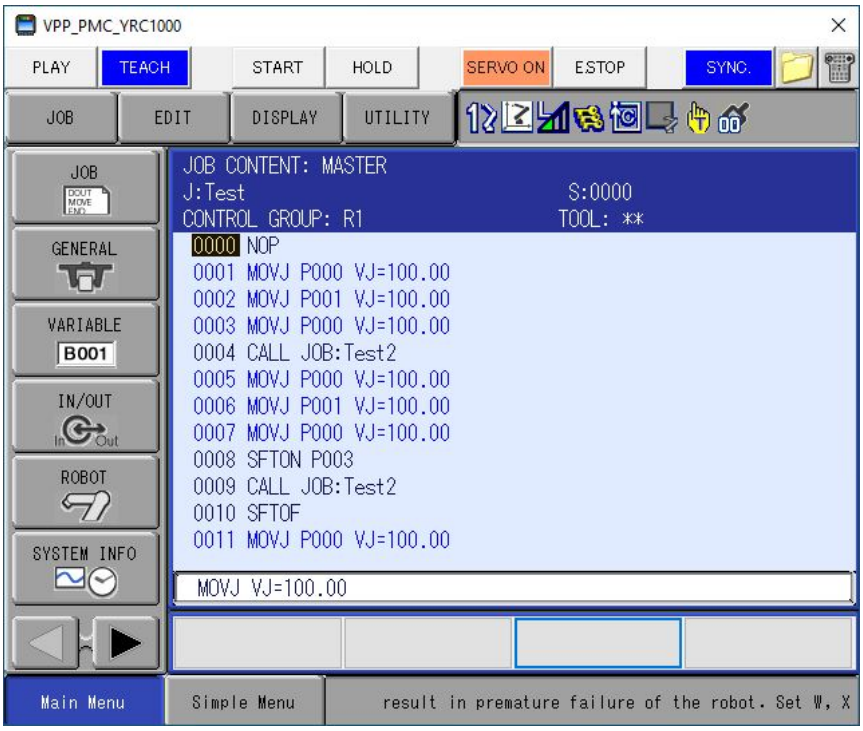

Abbildung 2.12: Roboterpfad wiederverwenden (Yaskawa)
Quelle: eigene Ausarbeitung

Die Abbildungen 2.13, 2.14 & 2.15 zeigen, wie eine Wiederverwendung von Roboterpfaden in Robot Studio realisiert werden kann. Abbildung 2.13 zeigt dabei, dass Roboterpfade von anderen für die Bewegungsinstruktionen verwendet/aufgerufen werden können. Um einen Pfad im dreidimensionalen Raum neu zu positionieren gibt es in Robot Studio unterschiedliche Möglichkeiten, welche in Abbildung 2.15 dargestellt sind. So kann beispielsweise ein Pfad erstellt

werden und dieser kopiert und verschoben werden, um die selben Bewegungsabläufe an anderer Stelle zu wiederholen. Abbildung 2.14 zeigt dabei, wie die Eingabemaske für die translatorische Verschiebung eines Pfades in Robot Studio realisiert ist.

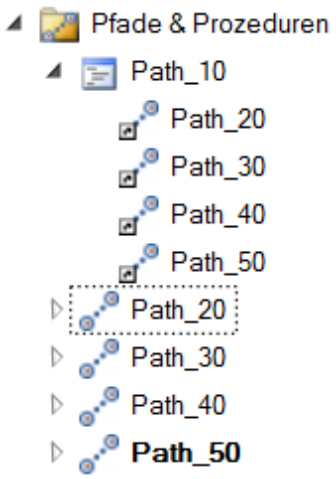

Abbildung 2.13: Roboterpfad wiederverwenden (Robot Studio) Quelle: eigene Ausarbeitung

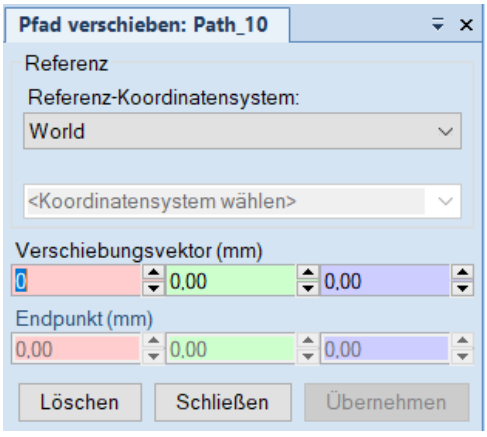

Abbildung 2.14: Verschieben eines Roboterpfades (Robot Studio) Quelle: eigene Ausarbeitung

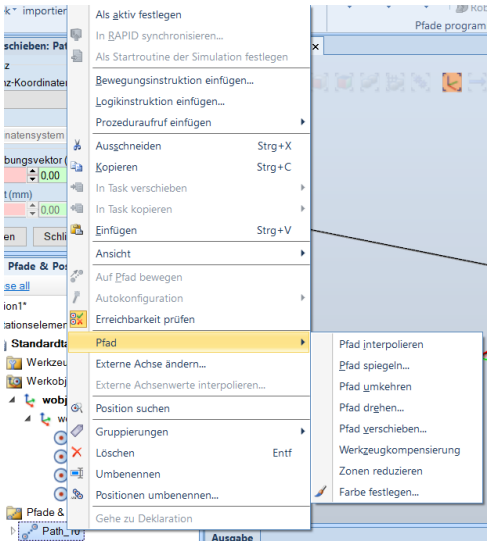

Abbildung 2.15: Möglichkeiten zur Pfadbearbeitung (Robot Studio) Quelle: eigene Ausarbeitung

3 Eigene Umsetzung

Im folgenden Kapitel wird beschrieben, wie die Programmierung des Plugins erfolgt ist. Dazu wird beispielhaft erläutert, wie einzelne Teile der Implementierung realisiert wurden und anhand des Sourcecodes auf die einzelnen Konzepte eingegangen. Außerdem wird beschrieben, wie das erstellte Plugin die Programmierung von Roboterpfaden gestaltet und ein Einblick in die Bedienung gegeben. Bei den Eingabemasken wurde das User-Experience (UX) Design nicht beachtet. Ziel war es, dem Benutzer die Interaktion mit dem Plugin zu ermöglichen und alle Funktionen zur Verfügung zu stellen. Außerdem werden in der aktuellen Version keine Multi-Roboter-Systeme unterstützt. Somit muss für jeden Roboter, auch wenn es sich um dieselbe Produktionsanlage handelt, ein separates Projekt erstellt werden. Die Verwendung des Plugins gestaltet sich wie folgt:

Beim Starten wird der Benutzer aufgefordert eine bereits vorhandene Konfigurationsdatei auszuwählen oder eine neue zu erstellen. Diese beinhaltet die Information, wo sich weitere benötigte Dateien befinden. Diese sind die FreeCAD-Projektdatei und jeweils eine Datei für die Daten zu den Koordinatensystemen, definierten Punkten und bereits erstellten Roboterpfaden. Für den Fall, dass diese Dateien noch nicht existieren, müssen diese jetzt erstellt werden. Die einzelnen weiteren Arbeitsschritte sind in diesem Kapitel genauer beschrieben.

3.1 Implementierung

FreeCAD bietet die Möglichkeit, Arbeitsschritte zu automatisieren beziehungsweise neue Funktionalitäten zu implementieren. Dies kann durch das Erstellen von Python-Skripten, welche als Makros verwendet werden können, realisiert werden. Wenn die Integration von mehreren Kommandos, die für eine bestimmte Aufgabenstellung gruppiert werden, geplant ist, kann es sinnvoll sein, einen neuen Arbeitsbereich zu erstellen. Arbeitsbereiche können entweder in C++ oder Python programmiert werden. Das in dieser Arbeit erstellte Plugin wurde als Python-Arbeitsbereich realisiert.

3.1.1 Makro-/Arbeitsbereicherstellung

Makros können in FreeCAD über das Dialogfeld, welches in Abbildung 3.1 zu sehen ist, erstellt werden. Mit einem Klick auf die Schaltfläche „Erstellen“ öffnet sich ein Editor in welchem das Python-Skript erstellt werden kann. Dabei können verschiedene FreeCAD-Module für die Interaktion mit der Modellierungssoftware genutzt werden, um beispielsweise Objekte zu erzeugen oder Informationen über vorhandene Körpern zu erhalten. [8]

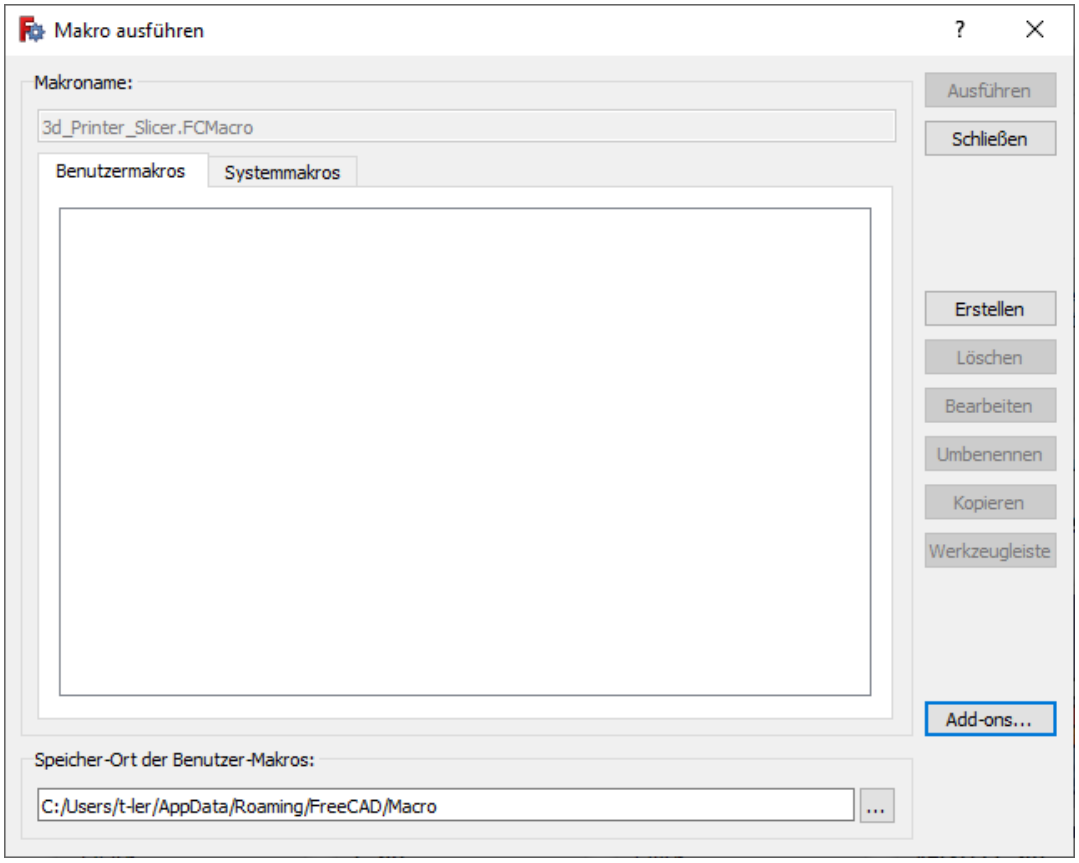

Abbildung 3.1: Erstellen von Makros
Quelle: eigene Ausarbeitung

Falls mehrere Befehle zur Erfüllung einer Aufgabenstellung benötigt werden, kann ein Arbeitsbereich erstellt werden. Für das Plugin, welches in dieser Arbeit entwickelt wird, werden mehrere unterschiedliche Befehle zur Erstellung von Roboterpfaden benötigt. Eine Auflistung aller implementierten Kommandos und sonstigen benötigten Klassen ist in Anhang A.1 zu finden. Im Folgenden wird die Vorgehensweise zur Erstellung eines Arbeitsbereichs für FreeCAD anhand des in dieser Arbeit entwickelten Plugins beschrieben. [5]

Um einen neuen Arbeitsbereich in FreeCAD zu erstellen, muss im Mod-Verzeichnis der Benutzerdaten ein Ordner erstellt werden. Dieses Verzeichnis findet sich unter Windows üblicherweise an folgender Stelle:

„C:\Users\<username>\Appdata\Roaming\FreeCAD\Mod\“

Damit der Arbeitsbereich von FreeCAD geladen werden kann, müssen eine *„Init.py“*-Datei und eine *„InitGui.py“*-Datei erstellt werden. Die *„Init.py“*-Datei enthält dabei Funktionen, welche benötigt werden, wenn FreeCAD im Konsolenmodus arbeitet. Dazu gehören Datei-Importeure und -Exporteure. Diese Datei bleibt im Fall des in dieser Arbeit erstellten Plugins leer. In der *„InitGui.py“*-Datei wird der grafische Arbeitsbereich definiert. Dabei werden der Name des Arbeitsbereichs, das Icon und die vorhandenen Befehle spezifiziert. Dadurch ergibt sich folgende Ordnerstruktur:

```
FreeCAD\Mod\
 +- MeinArbeitsbereich
     +- Init.py
     +- InitGui.py
```

In Abbildung 3.2 ist ein Ausschnitt der *„InitGui.py“*-Datei zu sehen. In der „__init__“-Funktion wird der Name, des Arbeitsbereichs sowie ein Tooltip und Icon spezifiziert. Beim Icon handelt es sich hierbei um eine SVG-Datei. Die weiteren Icons für die einzelnen Befehle sind ebenfalls SVG-Dateien. In der „Initialize“-Funktion, welche für jeden Arbeitsbereich beim Starten von FreeCAD ausgeführt wird, wird die Struktur des Arbeitsbereichs definiert. Dafür werden Module importiert, welche verschiedene Befehle implementieren.Diese Befehle werden anschließend in Gruppen sortiert und der Toolbar von FreeCAD hinzugefügt. Eine weitere Möglichkeit ist das Hinzufügen von Befehlen zum Kontextmenü über die Funktion „ContextMenu“.

```
29      def __init__(self):
30          import RPWlib
31          self.__class__.MenuText = "Robot Path Workbench"
32          self.__class__.ToolTip = "A Workbench to create and export Paths for Robots"
33          self.__class__.Icon =  RPWlib.pathOfModule() + "/icons/WB_main_icon.svg"
34
35      def Initialize(self):
36          """This function is executed when FreeCAD starts"""
37          # import modules with commands
38          import CreateSeg, AddOrigin, AddPoints, EditPath, ReloadView, NewModule
39          # Group commands based on function
40          self.segmentCommands = ["Create_Linear_Segment","Create_P2P_Segment","Create_Circular_Segment"]
41          self.mainCommands = ["Add_Origin_Command", "Add_Points_Command"]
42          self.advancedCommands = ["Add_New_Module_Command"] #not workin in current Version
43          self.RPWMenuPath = ["Edit_Path_Command"]
44          self.RPWMenu = ["Reload_View_Command"]
45          # add commands to the FreeCAD toolbar
46          self.appendToolbar("Main Commands",self.mainCommands)
47          self.appendToolbar("Add Segment",self.segmentCommands)
48          self.appendToolbar("Advanced Commands",self.advancedCommands)
87      def ContextMenu(self,recipient):
88          self.appendContextMenu(["RPW Workbench", "Path"],self.RPWMenuPath)
89          self.appendContextMenu(["RPW Workbench"], self.RPWMenu)
```

Abbildung 3.2: Initialisierung des Arbeitsbereichs
Quelle: eigene Ausarbeitung

In Abbildung 3.3 ist die „Activated“-Funktion dargestellt. Diese Funktion wird ausgeführt, wenn der Arbeitsbereich aktiviert wird. Hier kann beispielsweise die Konfiguration des Arbeitsbereichs oder das Laden von benötigten Daten erfolgen. Zusätzlich kann noch eine Funktion beim Deaktivieren des Arbeitsbereichs definiert werden. Außerdem ist die Funktion „GetClassName“ zu sehen. Sie muss den String „Gui::PythonWorkbench“ zurückliefern, damit FreeCAD den Arbeitsbereich als solchen erkennt.

```
50      def Activated(self):
51          """This function is executed when the workbench is activated"""
52          import RPWClasses
53          import RPWlib
54          import json
55          from PySide import QtGui
56          RPWlib.MovementList.List = []
57          RPWlib.PointsList.List = []
58          RPWlib.CSList.List = []
59
60          RPWlib.config = RPWClasses.ProjectConfiguration()
61          RPWlib.config.readConfig()
62          RPWlib.config.writeConfig()
63
64          App.ActiveDocument.recompute()
65
66          try:
67              mw = Gui.getMainWindow()
68              c = mw.findChild(QtGui.QTextEdit, "Report view")
69              c.clear()
70          except Exception as e:
71              App.Console.PrintError(e)
72
73          try:
74              RPWlib.mainCMDs = mw.findChild(QtGui.QToolBar, "Main Commands")
75              RPWlib.addSegCMDs = mw.findChild(QtGui.QToolBar, "Add Segment")
76              RPWlib.newModCMDs = mw.findChild(QtGui.QToolBar, "Advanced Commands")
77              RPWlib.mainCMDs.show()
78              RPWlib.addSegCMDs.show()
79              RPWlib.newModCMDs.show()
80          except Exception as e:
81              App.Console.PrintError(e)
82          return
83
84      def Deactivated(self):···
87
88      def ContextMenu(self,recipient):···
91
92      def GetClassName(self):
93          # This function is mandatory if this is a full python workbench
94          # This is not a template, the returned string should be exactly "Gui::PythonWorkbench"
95          return "Gui::PythonWorkbench"
96
97  Gui.addWorkbench(RobotPathWorkbench)
```

Abbildung 3.3: Aktivierung des Arbeitsbereichs
Quelle: eigene Ausarbeitung

3.1.2 Befehlserstellung

Die in der Initialisierung des Arbeitsbereichs importierten Module dienen zum übersichtlicheren Aufbau des Plugins. Im Folgenden wird anhand des Befehls „Reload_View_Command“, welcher sich im Modul „ReloadView“ befindet, die Erstellung von Kommandos für einen Arbeitsbereich genauer erläutert. In Abbildung 3.4 ist die Struktur eines Befehls zu sehen. Bei einem Befehl handelt es sich um eine Klasse, welche die folgenden drei Funktion implementiert haben muss. Es muss eine Funktion „GetResources“ geben, die ein Dictionary zurückliefert, welches Informationen zum Icon des Befehls, dem Namen und einem Tooltip zur Funktion enthält. Zusätzlich kann noch ein Tastenkürzel definiert werden, mit dem der Befehl schnell ausgeführt werden kann. Das Tastenkürzel jedes Befehls kann vom Benutzer nachträglich in den Einstellungen angepasst werden.

```
24  import FreeCADGui as Gui
25  import FreeCAD as App
26  import RPWlib
27  import RPWClasses
28  class ReloadViewCmd():
29      """
30      Classname:
31      ReloadViewCmd
32
33      Description:
34      This Class is used to reload the 3D-View of FreeCAD and redraw
35      the coordinate systems, poses and path segments.
36      """
37      def GetResources(self):
38          return {'Pixmap'  : RPWlib.pathOfModule() + "/icons/WB_newModule_icon.svg",
39                  #'Accel' : "Shift+R",
40                  'MenuText': "Reload 3D View",
41                  'ToolTip' : "Reload 3D View"}
42
43  >   def Activated(self):···
70
71      def IsActive(self):
72          """Here you can define if the command must be active or not (greyed) if
73          certain conditionsare met or not. This function is optional."""
74          return True
75
76  Gui.addCommand('Reload_View_Command',ReloadViewCmd())
```

Abbildung 3.4: Erstellen eines Kommandos für den Arbeitsbereich
Quelle: eigene Ausarbeitung

In der Funktion „IsActive“ kann implementiert werden, in welchen Situatio-

nen der Befehl aktiviert werden kann, oder ausgegraut und somit nicht verfügbar ist. Die „Activated“-Funktion wird nach der Aktivierung des Befehls ausgeführt. Hier kann entweder direkt eine Funktionalität implementiert werden (siehe Abbildung 3.5), oder ein Dialogfeld aktiviert werden, welches dem Benutzer weitere Funktionalität über eine zusätzliche Klasse zur Verfügung stellt.

```
43      def Activated(self):
44          try:
45              doc = App.ActiveDocument
46              path = doc.getObject("Path")
47              pts = doc.getObject("Points")
48              cs = doc.getObject("Coordinate_Systems")
49              RPWClasses.delWithChildren(path)
50              RPWClasses.delWithChildren(pts)
51              RPWClasses.delWithChildren(cs)
52          except:
53              pass
54          try:
55              RPWlib.MovementList.pathGrp = doc.addObject("App::DocumentObjectGroup", "Path")
56              RPWlib.PointsList.pointsGrp = doc.addObject("App::DocumentObjectGroup", "Points")
57              RPWlib.CSList.csGrp = doc.addObject("App::DocumentObjectGroup", "Coordinate_Systems")
58          except Exception as e:
59              App.Console.PrintMessage(e)
60          try:
61              for cs in RPWlib.CSList.List:
62                  cs.selfDraw(cs.name,1)
63              for id,pt in enumerate(RPWlib.PointsList.List):
64                  pt.selfDraw(f"Point_{id}", 2)
65              for mv in RPWlib.MovementList.List:
66                  mv.selfdraw()
67          except Exception as e:
68              App.Console.PrintMessage(e)
69          return True
```

Abbildung 3.5: Befehl ohne Dialogfeld
Quelle: eigene Ausarbeitung

Das Aktivieren eines Dialogfeldes ist in Abbildung 3.6 zu sehen. Dafür wird eine weitere Klasse benötigt, welche sich um die Funktionalität des Dialogfeldes kümmert. In Abbildung 3.7 ist zu sehen, dass diese Klasse die UI-Datei lädt, welche mit QtDesigner erstellt wurde. Zusätzlich werden die Widgets, welche sich im Dialogfeld befinden, verwaltet und mit weiteren Funktionen verknüpft. Außerdem muss die Klasse für das Dialogfeld eine „accept“-Funktion implementieren, welche sich um den sauberen Abschluss des Befehls kümmert, nachdem dieser mit der von FreeCAD zur Verfügung gestellten Schaltfläche „OK“ beendet wurde.

```
232     def Activated(self):
233         panelOrig = AddPoints()
234         Gui.Control.showDialog(panelOrig)
235         return True
```

Abbildung 3.6: Aktivieren des Dialogfelds für einen Befehl
Quelle: eigene Ausarbeitung

```
36  path_to_ui = RPWlib.pathOfModule() + "/pointsView.ui"
37  class AddPoints():
38      def __init__(self):
39          self.current = 0
40          self.form = Gui.PySideUic.loadUi(path_to_ui)
41          self.reloadList()
42          for cs in RPWlib.CSList.List:
43              self.form.Box_Combo_CS.addItem(cs.name)
44          item = self.form.listWidget.item(0)
45          self.form.listWidget.setCurrentRow(0)
46          if (len(RPWlib.PointsList.List) != 0):
47              self.printItem(item)
48
49          self.form.listWidget.itemClicked.connect(self.printItem)
50          self.form.Button_Del.clicked.connect(lambda: self.deletePoint())
51          self.form.Button_Save.clicked.connect(lambda: self.savePoint())
52          self.form.Button_AddPoint.clicked.connect(lambda: self.addPoint())
53          self.form.Button_SetPos.clicked.connect(lambda: self.setPos())
54
55          self.form.Box_Point_X.valueChanged.connect(lambda: self.updateSphere())
56          self.form.Box_Point_Y.valueChanged.connect(lambda: self.updateSphere())
57          self.form.Box_Point_Z.valueChanged.connect(lambda: self.updateSphere())
58          self.form.Box_Point_Yaw.valueChanged.connect(lambda: self.updateSphere())
59          self.form.Box_Point_Pitch.valueChanged.connect(lambda: self.updateSphere())
60          self.form.Box_Point_Roll.valueChanged.connect(lambda: self.updateSphere())
61          self.form.Box_Combo_CS.currentIndexChanged.connect(lambda: self.updateSphere())
62
63      def accept(self):
64          if (len(RPWlib.PointsList.List) != 0):
65              self.savePoint()
66          doc = App.activeDocument()
67          user = getpass.getuser()
68          RPWlib.writePointsFile(RPWlib.PointsList.List, user)
69          return True
```

Abbildung 3.7: Klasse für das Dialogfeld
Quelle: eigene Ausarbeitung

3.1.3 FreeCAD-Module

Um über die eigens erstellten Befehle mit den Objekten und der 3D-Ansicht zu interagieren beziehungsweise neue Körper erstellen zu können, stellt FreeCAD unterschiedliche Module zur Verfügung. Diese können, wie in Abbildung 3.8 zu sehen ist, in die eigenen Module integriert werden. Es werden zwei Grundmodule bereitgestellt, welche die Schnittstelle zwischen dem erstellten Befehl und dem Grundprogramm und seiner 3D-Ansicht bereitstellen.

```
27  import FreeCAD as App
28  import FreeCADGui as Gui
29  import Part
```

Abbildung 3.8: Importieren von FreeCAD-Modulen
Quelle: eigene Ausarbeitung

In Abbildung 3.10 ist ein Teil der Funktion zu sehen, welche die Positionen von markanten Punkten ermittelt, um die X-, Y- und Z-Position von Roboterposen an diesen auszurichten. Anhand des Codeausschnitts ist zu sehen, wie das Module „FreeCADGui“ (importiert als „Gui“) verwendet werden kann, um Informationen aus Objekten in der 3D-Ansicht zu beschaffen. In den Zeilen 168 bis 172 wird die Selektion aus der 3D-Ansicht ermittelt. Die Zeilen 177 bis 194 dienen der Identifikation des ausgewählten Teiles. Dabei kann es sich entweder um ein Koordinatensystem oder eine Roboterpose (Zeile 178) oder um einen Eckpunkt (Zeile 183), eine Fläche (Zeile 186) oder Kante (Zeile 189) handeln. Je nach Ergebnis wird die Postion der Auswahl zwischengespeichert, um weiter verarbeitet zu werden.

Abbildung 3.9 zeigt, wie Module von anderen Arbeitsbereichen ebenfalls genutzt werden können, um die Interaktion mit FreeCAD zu vereinfachen. Es wird das Modul für den „Part“-Arbeitsbereich genutzt, um einen Kreisbogen zwischen drei Punkten zu zeichnen, damit ein kreisförmiges Pfadsegment in der 3D-Ansicht visualisiert werden kann.

```
174         try:
175             arc = Part.Arc(start,mid,end)
176             myLine = arc.toShape()
177             shape=App.ActiveDocument.addObject("Part::Feature", self.name)
178             shape.Shape=myLine
179             App.ActiveDocument.recompute()
180             shape.ViewObject.LineColor=RPWlib.lineColorCirc
181             RPWlib.MovementList.pathGrp.addObject(App.ActiveDocument.getObject(self.name))
182         except:
```

Abbildung 3.9: Verwendung importierter Arbeitsbereich-Module
Quelle: eigene Ausarbeitung

```
164     def setPos(self):
165         pos = [0,0,0]
166         ori = [0,0,0]
167 
168         sel = Gui.Selection.getSelection()
169         try:
170             object_Label = sel[0].Label
171             object_Name  = sel[0].Name
172             SubElement = Gui.Selection.getSelectionEx()[0]
173         except Exception:
174             object_Label = ""
175             object_Name  = ""
176             SubElement = None
177         try:
178             if SubElement.TypeName == "PartDesign::CoordinateSystem":
179                 element_ = App.ActiveDocument.getObject(SubElement.ObjectName)
180                 pos = element_.Placement.Base
181             else:
182                 element_ = SubElement.SubObjects[0]
183                 if (element_.ShapeType == "Vertex"):
184                     pos = element_.Point
185                     ori = [0,0,0]
186                 elif (element_.ShapeType == "Face"):
187                     pos = element_.BoundBox.Center
188                     ori = [0,0,0]
189                 elif (element_.ShapeType == "Edge"):
190                     startPoint = element_.Vertexes[0]
191                     endPoint = element_.Vertexes[1]
192                     direction = endPoint.Point - startPoint.Point
193                     pos = startPoint.Point + 0.5*direction
194                     ori = [0,0,0]
195         except Exception as e:
196             App.Console.PrintMessage(e)
```

Abbildung 3.10: Verwendung importierter Grundmodule
Quelle: eigene Ausarbeitung

3.1.4 Informationsaustausch

Für die Erstellung von Roboterpfaden ist es notwendig, dass unterschiedliche Befehle auf die selben Informationen zugreifen können. Dafür wurde in diesem Plugin ein Modul implementiert, in dem Klassen und Funktionen für das Verwalten der Koordinatensysteme, Roboterposen und Pfadsegmente vorhanden sind. Abbildung 3.11 zeigt die definierten Klassen. Sie beinhalten die jeweilige Liste und Informationen, wann und von welchem Benutzer die Listen der Koordinatensysteme, Roboterposen und Pfadsegmente das letzte Mal bearbeitet wurden. Außerdem sind der Dateipfad zum Lesen und Abspeichern sowie die jeweilige Gruppe für die Ordnung in der Objektstruktur von FreeCAD hinterlegt. Diese Struktur wird benötigt, um jeweils alle Pfadsegmente, Koordinatensystem oder Roboterposen ausblenden zu können.

```
284 class MovementList:
285     pathGrp = None
286     pathToFile = None
287     lastEditedOn = None
288     lastEditor = None
289     List = []
290     currentId = 0
291
292 class PointsList:
293     pointsGrp = None
294     pathToFile = None
295     lastEditedOn = None
296     lastEditor = None
297     List = []
298
299
300 class CSList:
301     csGrp = None
302     pathToFile = None
303     lastEditedOn = None
304     lastEditor = None
305     List = []
```

Abbildung 3.11: Klassen zum Informationsaustausch
Quelle: eigene Ausarbeitung

In Abbildung 3.12 sind die erstellten Funktionen zu sehen, mit denen die Listen verwaltet werden können. Es gibt die drei folgenden Funktionen jeweils für die drei Listen:

reload<...>List :
Zum Einlesen der hinterlegten Datei und Laden der Informationen. Außerdem werden die Elemente in der 3D-Ansicht visualisiert.

write<...>File :
Zum Abspeichern der Liste als JSON-Struktur unter dem hinterlegten Dateipfad.

get<...>Dict :
Zum Umwandeln der Liste in ein Python-Dictionary, damit dieses als JSON-Struktur abgespeichert werden kann.

```
47  > def pathOfModule():···
49
50  > def reloadMovementList():···
101
102 > def reloadPointsList():···
133
134 > def reloadCSList():···
160
161 > def writePointsFile(points,editor):···
171
172 > def writeMovementsFile(movements, editor):···
182
183 > def writeCSFile(coords, editor):···
192
193 > def getMovementDict(movements):···
255
256 > def getPointsDict(points):···
267
268 > def getCSDict(coordinateSystems):···
```

Abbildung 3.12: Funktionen zur Informationsverarbeitung
Quelle: eigene Ausarbeitung

3.1.5 Koordinatentransformation

Koordinatensysteme und Roboterposen können, wie in Abbildung 3.13 zweidimensional dargestellt, verkettet und relativ zueinander ausgerichtet werden. In der Abbildung soll das Koordinatensystem {0} das Basiskoordinatensystem des Roboters darstellen. Koordinatensystem {1} stellt ein vom Benutzer definiertes Koordinatensystem dar und die Pose P1 eine Roboterpose, welche relativ zum Koordinatensystem {1} definiert wurde.

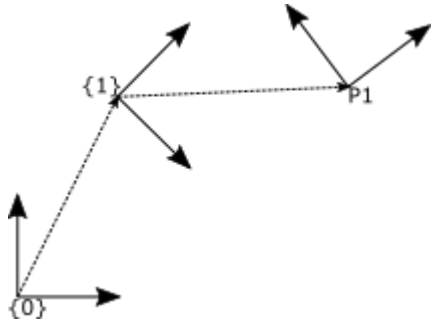

Abbildung 3.13: Transformation von Koordinatensystemen
Quelle: eigene Ausarbeitung

Die Information über die Lage des Punktes P1 im Koordinatensystem {1} kann als Transformationsmatrix dargestellt werden. Die folgende Matrix zeigt die Transformationsmatrix, wobei α die Rotation um die Z-Achse, β um die Y-Achse und γ um die X-Achse des Bezugssystems beschreibt. X, Y und Z sind die jeweiligen translatorischen Verschiebungen entlang der jeweiligen Koordinatenachsen des Bezugssystems. Analog dazu können auch die Position und Orientierung des Koordinatensystems {1} im Koordinatensystem {0} dargestellt werden.

$$\begin{bmatrix} c(\alpha)c(\beta) & c(\alpha)s(\beta)s(\gamma) - s(\alpha)c(\gamma) & c(\alpha)s(\beta)c(\gamma) + s(\alpha)s(\gamma) & X \\ s(\alpha)c(\beta) & s(\alpha)s(\beta)s(\gamma) + c(\alpha)c(\gamma) & s(\alpha)s(\beta)c(\gamma) - c(\alpha)s(\gamma) & Y \\ -s(\beta) & c(\beta)s(\gamma) & c(\beta)c(\gamma) & Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Um die Position und Orientierung des Punktes P1 im Koordinatensystem {0} zu erhalten, kann folgende Formel verwendet werden:

$$^{0}P = {}^{0}_{1}T \; {}^{1}P$$

mit:

^{0}P ... Roboterpose im Koordinatensystem {0}

^{1}P ... Roboterpose im Koordinatensystem {1}

${}^{0}_{1}T$ ... Transformationsmatrix von {0} nach {1}

Die im Plugin implementierte Funktion zum Errechnen der Gesamttransformation eines Punktes oder Koordinatensystems bis ins Basiskoordinatensystem des Roboters ist in Abbildung 3.14 zu sehen. Zuerst werden alle relevanten Koordinatensystem von der Roboterpose/dem Koordinatensystem bis hin zum

Basiskoordinatensystem des Roboters ermittelt. Anschließend wird, vom Basiskoordinatensystem ausgehend, die jeweilige Transformation zum nachfolgenden Koordinatensystem ermittelt bis die Gesamttransformation zur gegebenen Pose oder dem Koordinatensystem ermittelt wurde. Wie in der folgenden Formel ersichtlich, lässt sich die Transformation von einem Koordinatensystem {0} zu einem Koordinatensystem {3} als Produkt der Einzeltransformationen beschreiben.

$$ {}^{0}_{3}T = {}^{0}_{1}T \, {}^{1}_{2}T \, {}^{2}_{3}T $$

wobei:

${}^{A}_{B}T$ ... Transformationsmatrix von {A} nach {B}

```
95      def getTotalTransform(self):
96          totalTransform = App.Base.Matrix()
97          parentList = []
98          cs = self
99          while True:
100             parent = cs.coordinateSystem
101             if parent == None:
102                 break
103             parentList.append(parent)
104             cs = RPWlib.CSList.List[parent]
105         while len(parentList) != 0:
106             cs = RPWlib.CSList.List[parentList.pop()]
107             totalTransform = totalTransform.multiply(cs.getTransform())
108         totalTransform = totalTransform*self.getTransform()
109         return totalTransform
```

Abbildung 3.14: Verkettung von Transformationen
Quelle: eigene Ausarbeitung

3.2 Koordinatensysteme

Sind noch keine Koordinatensysteme in der dafür vorgesehenen Datei hinterlegt, wird ein Standard-Koordinatensystem erstellt, welches die Basis des Roboters darstellen soll. Dieses wird im Ursprung der 3D-Umgebung platziert und kann wie jedes andere Koordinatensystem bearbeitet werden. Falls schon Koordinatensysteme definiert wurden, scheinen diese in der Liste im unteren Teil der Eingabemaske auf und können ebenso bearbeitet oder gelöscht werden. Die Erstellung und Bearbeitung der Koordinatensysteme erfolgt durch die in Abbildung 3.15 dargestellte Eingabemaske. Relevante Parameter für die Pose der

Koordinatensysteme sind dabei die translatorische Verschiebung in X-, Y- und Z-Richtung, sowie die Rotation um die jeweiligen Koordinatenachsen. Die Angaben beziehen sich auf das definierte „Parent“-Koordinatensystem. Das einzige Koordinatensystem, welches kein Übergeordnetes benötigt, ist das standardmäßig erstellte, für die Basis des Roboters. Dessen Position und Orientierung bezieht sich auf den Ursprung der 3D-Umgebung von FreeCAD. Die Schaltfläche „Set Position“ kann verwendet werden, um die Position des Koordinatensystems an einem markanten Punkt eines Objektes zu definieren. Dazu zählen der Mittelpunkt einer Fläche, die Mitte einer Kante oder Eckpunkte. Außerdem kann auch die Position einer bereits definierten Pose beziehungsweise eines Koordinatensystems übernommen werden. Es können beliebig viele Koordinatensysteme hinzugefügt, gelöscht und bearbeitet werden.

Beim Speichern der Koordinatensysteme werden diese in einer JSON-Struktur abgelegt, in welcher für jedes Koordinatensystem eine eindeutige ID, der Name, die relative sowie absolute Position und Orientierung, sowie die ID des übergeordneten Koordinatensystems hinterlegt sind. Abbildung 3.16 zeigt dies am Beispiel der Roboterbasis.

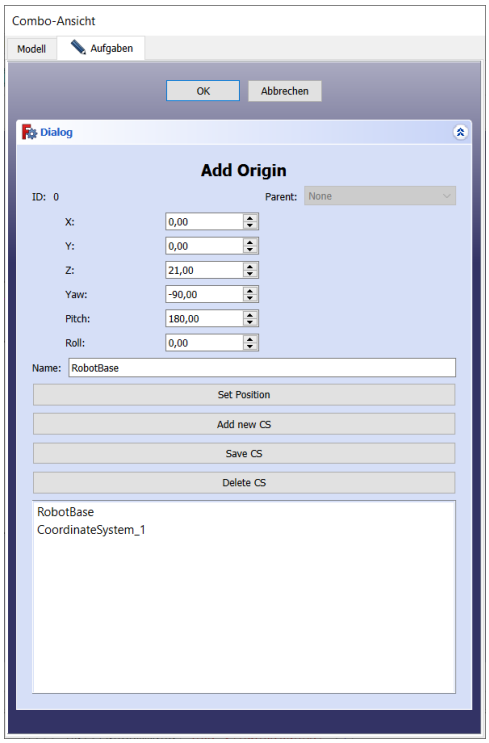

Abbildung 3.15: Erstellen von Koordinatensystemen (Eigene Umsetzung) Quelle: eigene Ausarbeitung


```
"CoordinateSystems": [
    {
        "id": 0,
        "name": "RobotBase",
        "position": {
            "X": 0.0,
            "Y": 0.0,
            "Z": 20.0
        },
        "orientation": {
            "yaw": 0.0,
            "pitch": 0.0,
            "roll": 0.0
        },
        "offsetPos": {
            "X": 0.0,
            "Y": 0.0,
            "Z": 20.0
        },
        "offsetRot": {
            "yaw": 0.0,
            "pitch": 0.0,
            "roll": 0.0
        },
        "coordinateSystem": null
    }
]
```


Abbildung 3.16: Gespeichertes Koordinatensystem als JSON-Struktur (Eigene Umsetzung) Quelle: eigene Ausarbeitung

3.3 Posendefinition

In Abbildung 3.17 ist die Eingabemaske zu sehen, mit welcher neue Posen erstellt beziehungsweise bestehende bearbeitet werden können. Sind bereits Roboterposen definiert und in der Datei gespeichert, werden diese in der Liste in der Eingabemaske angezeigt und können bearbeitet oder gelöscht werden. Es müssen ein Koordinatensystem, in welchem die Pose definiert wird, sowie die translatorische Verschiebung in X-, Y- und Z-Richtung und die Rotation um die jeweiligen Koordinatenachsen angegeben werden. Wie beim Bearbeiten der Koordinatensysteme, kann die Schaltfläche „Set Position“ verwendet werden, um die Roboterpose an einem markanten Punkt eines Objektes zu definieren. Dazu zählen der Mittelpunkt einer Fläche, die Mitte einer Kante, oder Eckpunkte. Außerdem kann auch die Position einer bereits definierten Pose beziehungsweise eines Koordinatensystems übernommen werden.

Beim Speichern der Roboterposen werden diese in einer JSON-Struktur abgelegt, in welcher für jede Pose die ID des Koordinatensystems, sowie die relative Position und Orientierung zu diesem hinterlegt sind. Abbildung 3.18 zeigt dies anhand eines Beispiels.

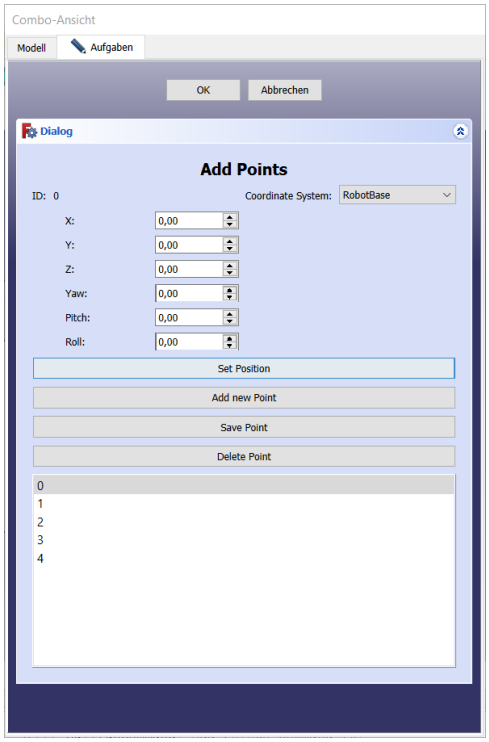

Abbildung 3.17: Definieren von Roboterposen (Eigene Umsetzung) Quelle: eigene Ausarbeitung


```
{
    "position": {
        "X": 10.0,
        "Y": 0.0,
        "Z": 0.0
    },
    "orientation": {
        "yaw": 0.0,
        "pitch": 0.0,
        "roll": 0.0
    },
    "coordinateSystem": 0
},
```


Abbildung 3.18: Gespeicherte Roboterpose als JSON-Struktur (Eigene Umsetzung) Quelle: eigene Ausarbeitung

3.4 Pfaddefinition

Wenn die benötigten Roboterposen definiert wurden, kann damit begonnen werden die unterschiedlichen Pfadsegmente zu erstellen. Es stehen dem Benutzer dabei drei unterschiedliche Bewegungstypen zur Verfügung. Wie bei den untersuchten proprietären Systemen (siehe Kapitel 2.2.3) handelt es sich auch um die Bewegungsarten linear, P2P und kreisförmig. Die Erstellung eines linearen (Abbildung 3.19) beziehungsweise P2P (Abbildung 3.20) Pfadsegments erfolgt dabei durch Angabe eines Namen für das Segment, definieren der Geschwindigkeit in % mit Hilfe des Schiebereglers, sowie durch Auswahl der Roboterposen für den Start- beziehungsweise Endpunkt der Bewegung. Zusätzlich kann für die Bewegungen eine Notiz hinzugefügt werden, um weitere Informationen zur Bewegung zu speichern. Durch Drücken der Schaltfläche „OK“ wird das Segment

dem Pfad hinzugefügt. In Abbildung 3.21 ist zu sehen, dass für kreisförmige Pfadsegmente ein weiterer Punkt angegeben werden muss. Dieser wird benötigt, um die Bewegung kreisförmig interpolieren zu können.

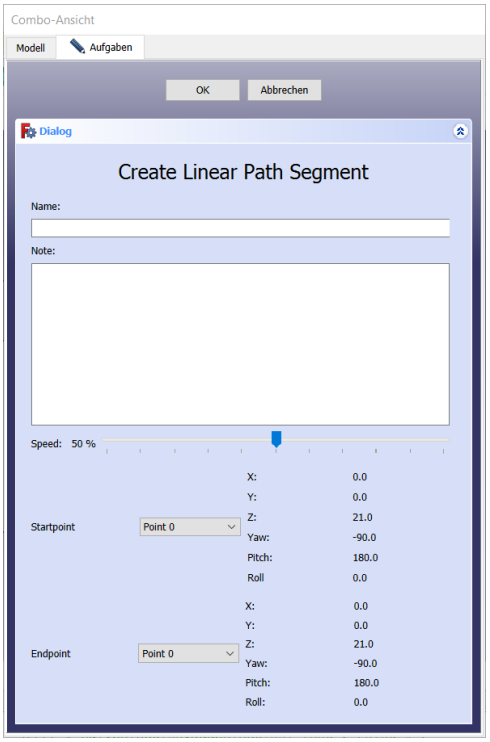

Abbildung 3.19: Erstellung lineare Pfadsegmente (Eigene Umsetzung) Quelle: eigene Ausarbeitung

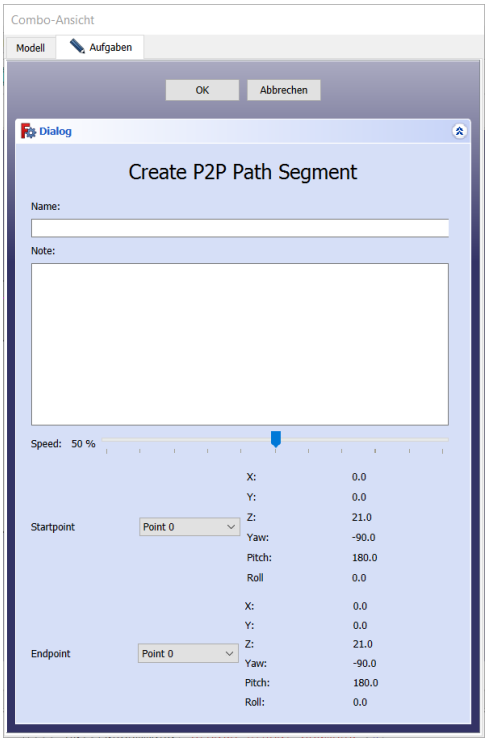

Abbildung 3.20: Erstellung P2P-Pfadsegmente (Eigene Umsetzung) Quelle: eigene Ausarbeitung

Abbildung 3.22 zeigt das Dialogfeld zum Bearbeiten des gesamten Roboterpfades. Die einzelnen Pfadsegmente können hier neu konfiguriert werden. Es können die Namen, sowie Wegpunkte und Geschwindigkeiten der einzelnen Bewegungen angepasst werden. Zusätzlich kann hier die Reihenfolge der Segmente mit den Schaltflächen „Move Item Up“ und „Move Item Down“ verändert werden. Auch das Entfernen von nicht benötigten Elementen ist möglich. Außerdem bietet die Schaltfläche „Add Action“ die Möglichkeit, zusätzliche Aktionen hinzuzufügen. So ist es beispielsweise möglich, zu definieren, an welchen Stellen des Roboterpfades ein Greifer geöffnet beziehungsweise geschlossen werden soll, oder ob ein Warten auf externe Ereignisse nötig ist, bevor die Bewegung

fortgesetzt werden darf.

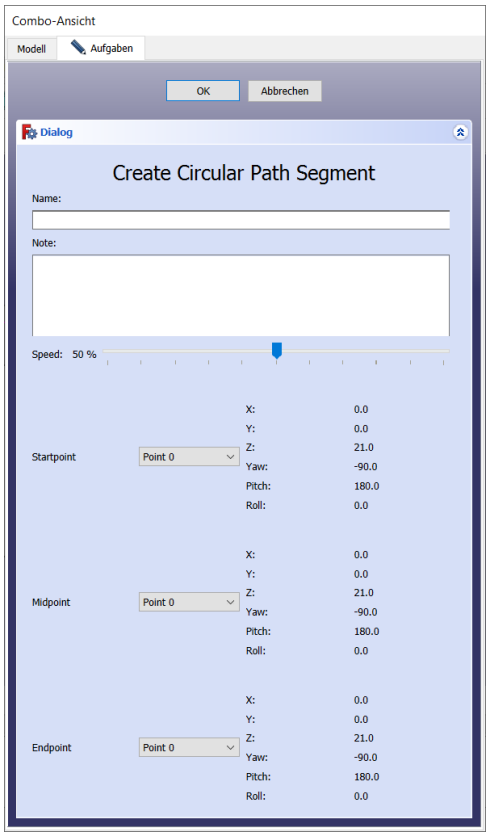

Abbildung 3.21: Erstellung kreisförmige Pfadsegmente (Eigene Umsetzung) Quelle: eigene Ausarbeitung

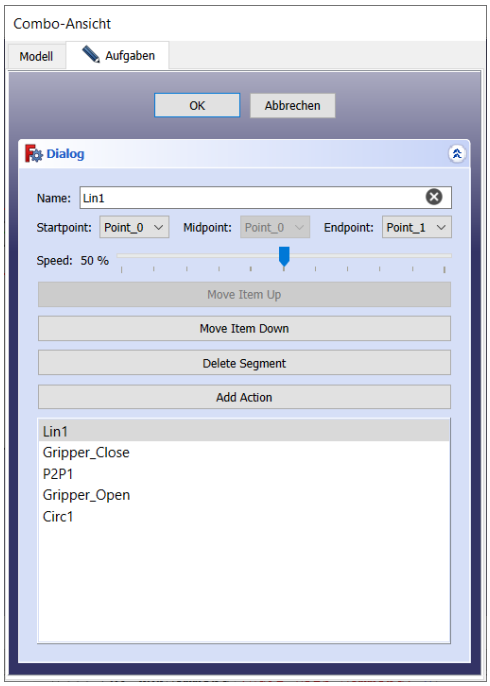

Abbildung 3.22: Pfadsegmente editieren (Eigene Umsetzung) Quelle: eigene Ausarbeitung

Beim Speichern des Roboterpfades wird dieser in einer JSON-Struktur abgelegt. Für Pfadsegmente werden, wie in Abbildung 3.23 zu sehen, eine eindeutige ID, der Typ, die Geschwindigkeit, die Notiz, der Name und die Wegpunkte hinterlegt. Diese werden dabei als Absolutposen relativ zur Roboterbasis gespeichert. Abbildung 3.24 zeigt, dass für Aktionen eine ID, der Typ, die Notiz, sowie ein Name gespeichert wird. Die Geschwindigkeit wird bei Aktionen standardmäßig mit dem Wert 0 abgespeichert.

```
{
    "type": "Circular",
    "speed": 50,
    "label": "",
    "name": "Circ1",
    "id": 4,
    "startPoint": {
        "id": 2,
        "position": {
            "X": 20.0,
            "Y": 20.0,
            "Z": 0.0
        },
        "orientation": {
            "yaw": 0.0,
            "pitch": 0.0,
            "roll": 0.0
        },
        "coordinateSystem": 0
    },
    "midPoint": {···
    },
    "endPoint": {···
    }
},
```


Abbildung 3.23: Gespeichertes Pfadsegment als JSON-Struktur (Eigene Umsetzung) Quelle: eigene Ausarbeitung


```
{
    "type": "Action",
    "speed": 0,
    "label": "",
    "name": "Gripper_Open",
    "id": 3
},
```


Abbildung 3.24: Gespeicherte Pfadaktion als JSON-Struktur (Eigene Umsetzung) Quelle: eigene Ausarbeitung

3.5 Visualisierung

Die einzelnen Koordinatensysteme und Roboterposen werden, wie in Abbildung 3.25 zu sehen, als Koordinatensysteme mit eingefärbten Achsen (X-Achse: rot, Y-Achse: grün, Z-Achse: blau) gezeichnet.

Die einzelnen Pfadsegmente werden nach dem Erstellen, wie in Abbildung 3.25 zu sehen, in der 3D-Ansicht in FreeCAD angezeigt. Lineare und P2P Pfadsegmente werden dabei als gerade Linien zwischen Start- und Endpunkt dargestellt. Kreisförmige Bewegungen werden visualisiert, indem ein Kreisbogen zwischen Start- und Endpunkt gezeichnet wird. Durch einen dritten Punkt kann der Kreisbogen interpoliert werden.

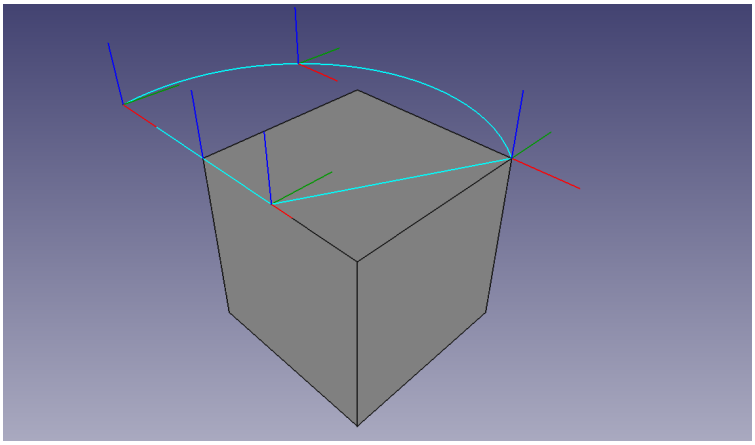

Abbildung 3.25: Visualisierung des Roboterpfades (Eigene Umsetzung)
Quelle: eigene Ausarbeitung

Falls Roboterposen oder Koordinatensysteme nachträglich verändert werden, kann die Visualisierung des Roboterpfades über das in Abbildung 3.26 gezeigte Kontextmenü aktualisiert werden. Diese Funktion zeichnet auch alle Koordinatensysteme oder Roboterposen neu und kann somit auch hilfreich sein, falls diese aus der 3D-Ansicht gelöscht wurden.

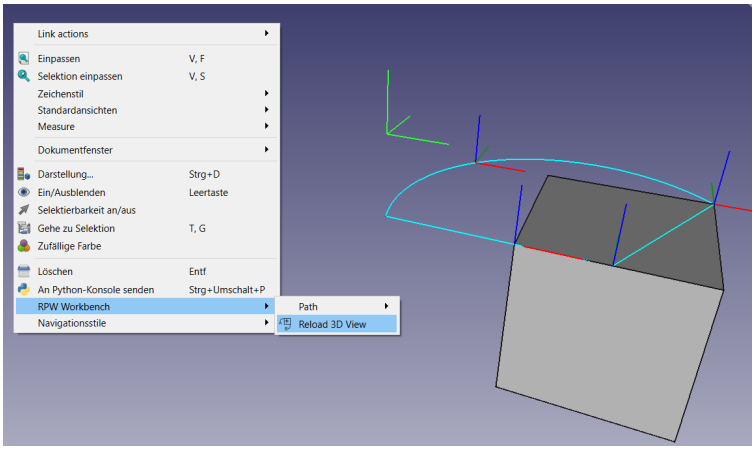

Abbildung 3.26: Visualisierung des Roboterpfades aktualisieren (Eigene Umsetzung)
Quelle: eigene Ausarbeitung

3.6 Wiederverwendbarkeit

Roboterpfade und Befehle für die Interaktion mit einem Kameramodul oder einer Presse könnten in Standardmodulen abgespeichert werden. Die Möglichkeit wie solche Standardmodule realisiert werden können, wurde konzipiert, aber steht in der aktuellen Version des Plugins nicht als Funktion für die Benutzer zur Verfügung. Eine Möglichkeit die Wiederverwendbarkeit zu realisieren, wäre beispielsweise, dass der Roboterpfad für ein Modul relativ zu einem Koordinatensystem definiert werden kann. Die Informationen zu den definierten Roboterposen, Fahrbefehlen und sonstigen Aktionen (Greifer öffnen/schließen, etc.) können gespeichert und beliebig oft wiederverwendet werden. Beim Importieren des Moduls muss nur die Position und Orientierung des Modul-Koordinatensystem definiert werden. Eventuell können auch weitere Einstellungen, wie beispielsweise das Überschreiben der Geschwindigkeiten für das Modul, erfolgen. Alle Pfadsegmente werden dann relativ zum Modul-Koordinatensystem platziert und dem Roboterpfad hinzugefügt. Eine schematische 2D-Darstellung der Idee findet sich in Abbildung 3.27. Es sind ein Roboter mit dessen Roboter-Koordinatensystem und zwei Instanzen eines Beispielmoduls zu sehen. Diese haben, bezogen auf das jeweilige Koordinatensystem einen identischen Roboterpfad, sind aber im Bezug auf das des Roboters an andere Positionen verschoben und rotiert.

Diese Art der Umsetzung ist dem Konzept der parametrischen Modellierung, wie es in der Konstruktion von Bauteilen verwendet wird, sehr ähnlich. Bei der parametrischen Modellierung werden dreidimensionale Objekte erstellt, indem verschiedene Eigenschaften hinzugefügt und parametriert werden. Oft wird dabei mit einer zweidimensionalen Skizze gestartet, welche um dreidimensionale Features erweitert wird. [1] Die Ähnlichkeit besteht darin, dass im geplanten Konzept ein Roboterpfad um Features (Module, beispielsweise Kameramodul) erweitert werden kann, die parametriert (Position, Orientierung, etc.) werden können.

Eine weitere Analogie zum geplanten Konzept stellt die objektorientierte Programmierung in der Informatik dar. Dabei werden Vorlagen erstellt, welche durch verschiedene Eigenschaften beschrieben sind. Aus diesen Vorlagen können dann Objekte erzeugt werden, welche alle denselben Grundaufbau haben, die sich aber in den Werten der Attribute unterscheiden können. In der Konzeptidee für die Umsetzung der Wiederverwendbarkeit von Roboterpfaden wäre die Definition des Roboterpfades also das Pendant zur Erstellung einer Klasse, welche als Vorlage für die Instanziierung von Objekten dient. Die veränderbaren Parameter des Pfades, wie Position, Orientierung und Geschwindigkeit, würden dabei als Attribute die einzelnen Objekte beschreiben.

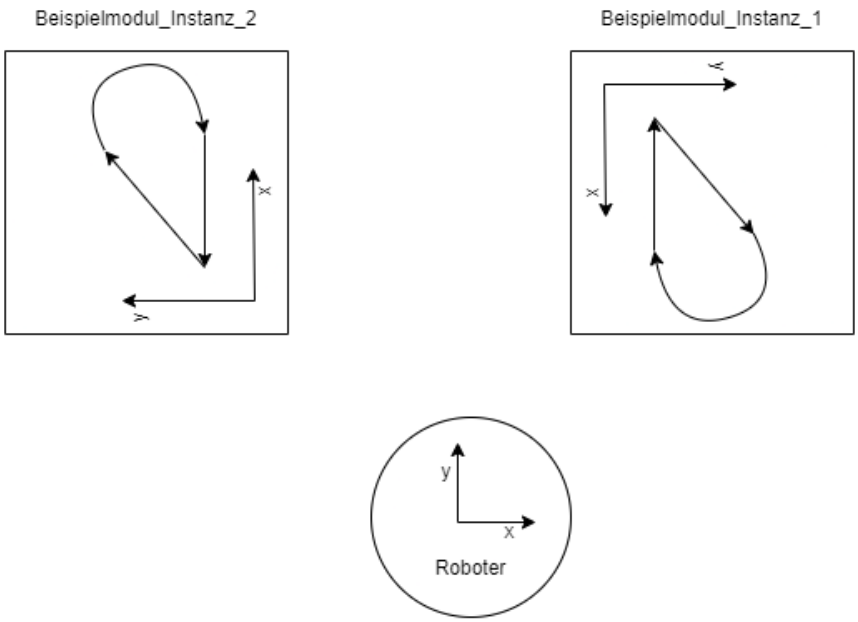

Abbildung 3.27: 2D-Darstellung Modularität und Wiederverwendbarkeit
Quelle: eigene Ausarbeitung

4 Anwendungsbeispiel

Im folgenden Kapitel wird das erstellte Plugin anhand einer einfachen Pick & Place Anwendung getestet und analysiert. Dabei wird das Plugin auf die Funktionalität überprüft. Es werden außerdem mögliche fehlende Funktionen ausfindig gemacht und eine Idee zur Implementierung dieser ausgearbeitet.

In Abbildung 4.1 ist eine Übersicht des geplanten Anwendungsbeispiels zu sehen. Die Anwendung besteht darin, einen Zylinder, welcher immer an der selben Stelle (Bereich 1) abzuholen ist, in die Ablagepositionen (Bereich 2) zu transportieren. Zur besseren Übersichtlichkeit werden nur Roboterposen für die vier Eckpositionen definiert und die beiden markierten Ablagepositionen für die Pfadgenerierung berücksichtigt. Der importierte Roboter ist ein Yaskawa Motoman SG400 [14]. Dieser dient nur zur besseren Visualisierung und zur Bestimmung der Position für das Roboter-Koordinatensystem.

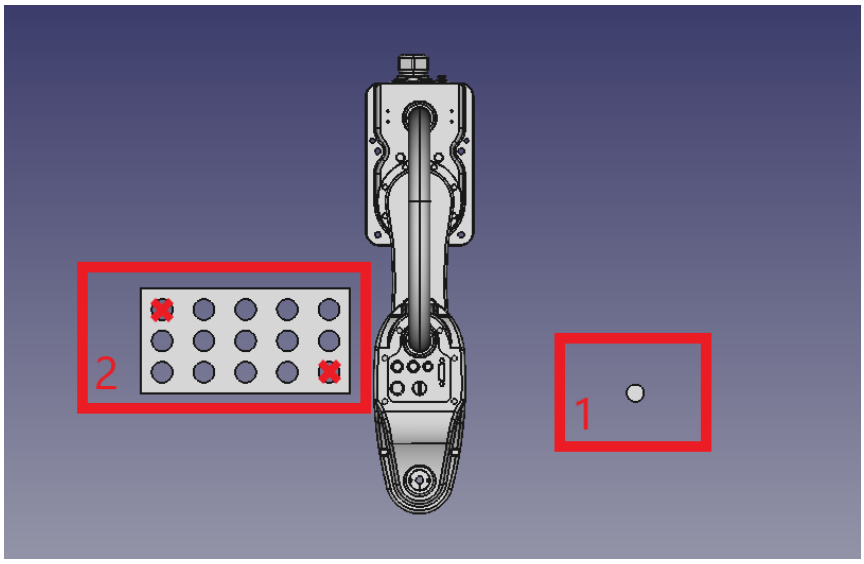

Abbildung 4.1: Übersicht des Anwendungsbeispiels
Quelle: eigene Ausarbeitung

4.1 Koordinatensysteme

Zuerst werden die benötigten Koordinatensysteme definiert. Für das Roboter-Koordinatensystem wird die ausgewählte Fläche, wie sie in Abbildung 4.2 in der Farbe grün zu sehen ist, selektiert. Anschließend wird die Position des Koordinatensystems mit der Schaltfläche „Set Position“ auf den Mittelpunkt der ausgewählten Fläche gesetzt. Um die finale Stelle zu erreichen, wird das Roboter-Koordinatensystem nun noch in positive Y- beziehungsweise negative Z-Richtung verschoben, bis es die in Abbildung 4.3 gezeigte Position erreicht. Anschließend wird noch jeweils ein Koordinatensystem für die Aufnahmeposition des Zylinders und für die Ablagepositionen erstellt. In Abbildung 4.4 sind alle Koordinatensysteme zu sehen. Bei der Erstellung dieser wäre es noch nützlich, wenn nicht nur die Position von markanten Punkten übernommen werden kann, sondern auch die Orientierung über eine Schnelleinstellung geändert werden könnte. Beispielsweise wäre dies durch Auswählen der Richtung der Koordinatenachsen realisierbar.

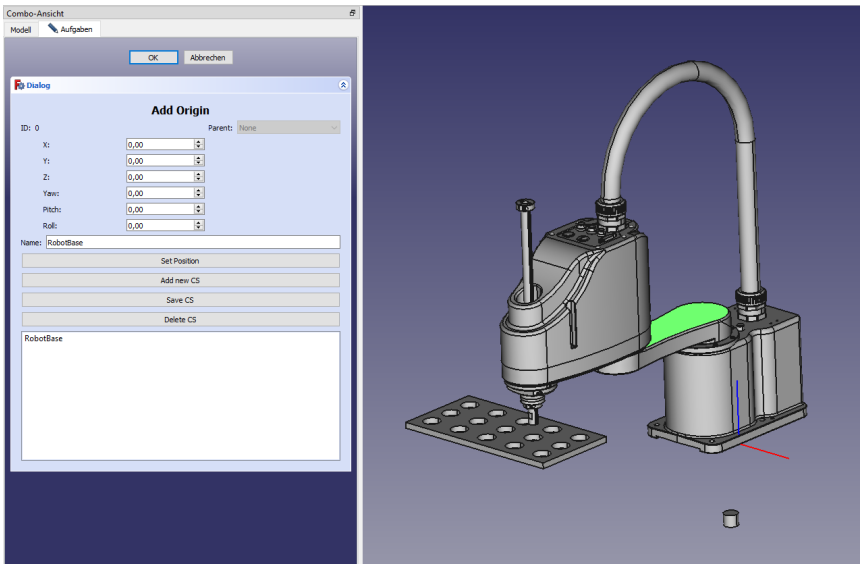

Abbildung 4.2: Erstellung des Roboter-Koordinatensystem
Quelle: eigene Ausarbeitung

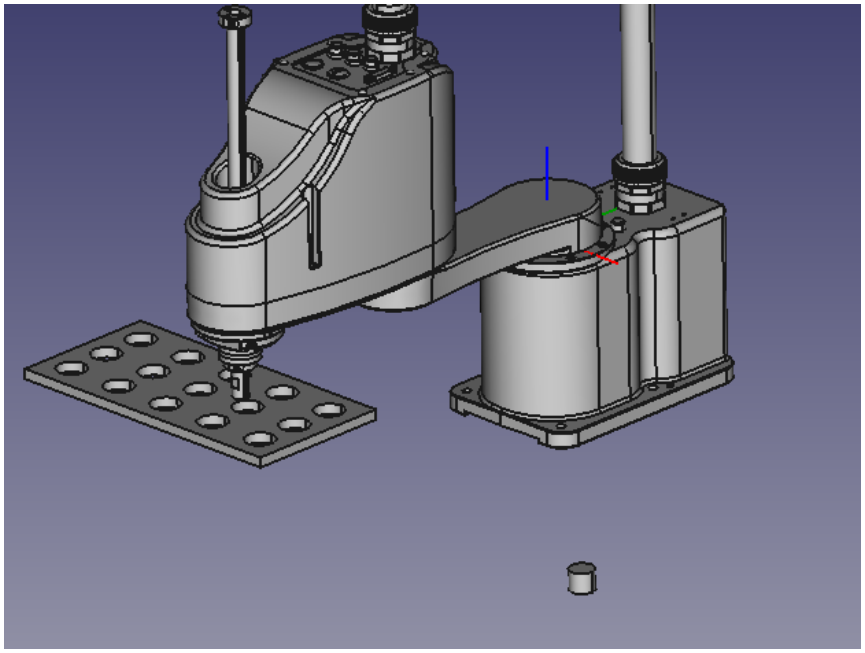

Abbildung 4.3: Position des Roboter-Koordinatensystem
Quelle: eigene Ausarbeitung

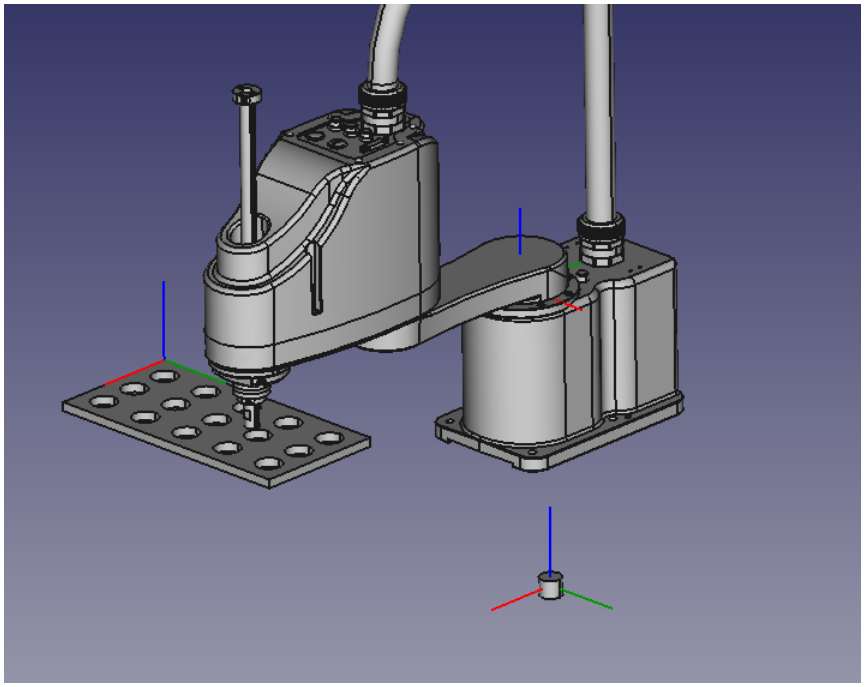

Abbildung 4.4: Alle Koordinatensysteme im Anwendungsbeispiel
Quelle: eigene Ausarbeitung

4.2 Posendefinition

Im nächsten Schritt werden die benötigten Roboterposen definiert. Die Homepose, wie sie in Abbildung 4.5 zu sehen ist, dient als Verbindungspose zwischen der Pose für die Aufnahme des Werkstücks und dem Bereich, in welchem es wieder abzulegen ist. Die Homepose wird in diesem Fall im Roboter-Koordinatensystem definiert.

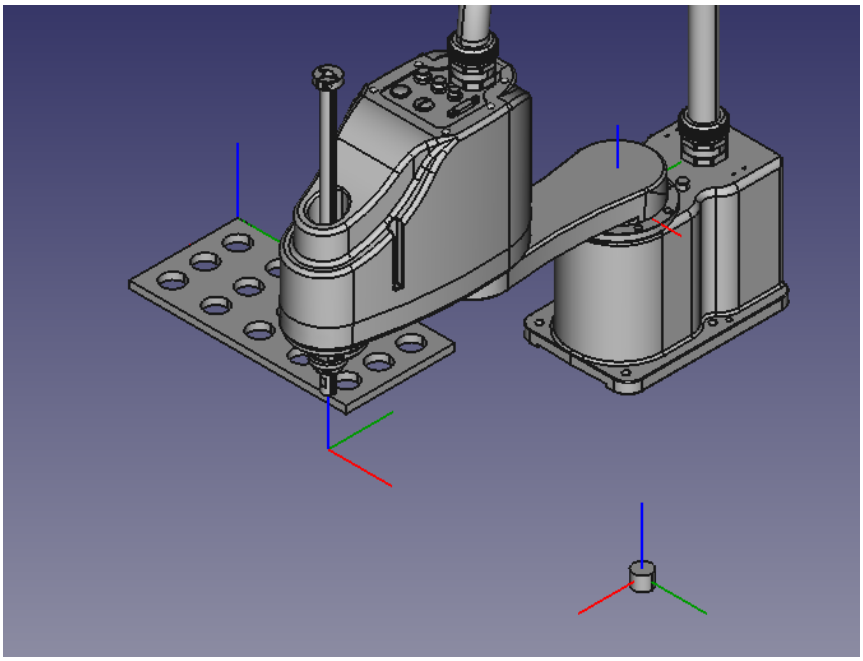

Abbildung 4.5: Homepose des Roboters im Anwendungsbeispiel
Quelle: eigene Ausarbeitung

Abbildung 4.6 zeigt die Posen, welche für die Ablage des Werkstücks definiert wurden. Zur Vereinfachung und besseren Übersicht werden nur die vier Eckpositionen des Warenträgers berücksichtigt. Für diese Roboterposen wäre es noch eine Erleichterung gewesen, wenn die Mittelpunkte von den Löchern, in welche der Zylinder eingesetzt werden soll, ebenfalls über die Schaltfläche „Set Position“ zur Schnelleinstellung der Positionen verwendet werden könnten. Eine zusätzliche Erweiterungsmöglichkeit wäre es also, diese Funktion noch weiter auszubauen und weitere markante Stellen zu unterstützen.

Zusätzlich erfolgt noch die Erstellung von Anfahrtspositionen. Diese Posen wurden erstellt, indem die Positionen für die Ablage übernommen, und danach um jeweils 50mm in Z-Richtung verschoben wurden. In Abbildung 4.7 sind alle Roboterposen zu sehen, die für das Einsetzen der Zylinder in den Warenträger definiert wurden.

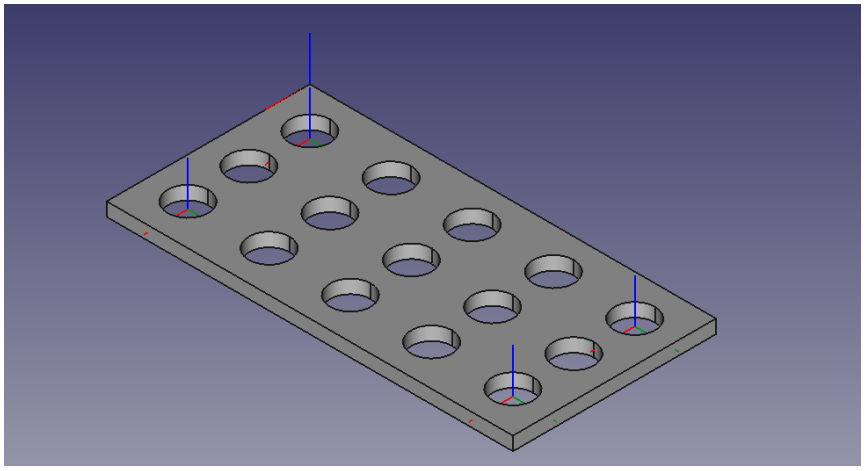

Abbildung 4.6: Ablageposen an den Eckpositionen
Quelle: eigene Ausarbeitung

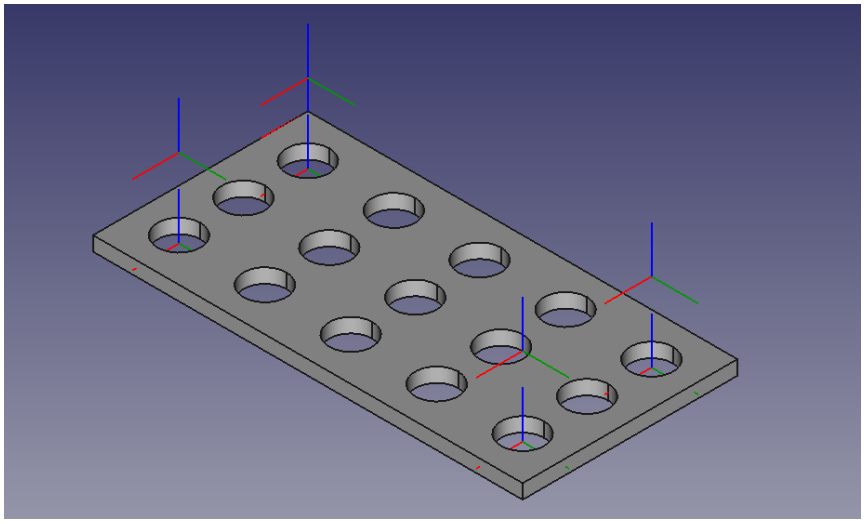

Abbildung 4.7: Anfahrtspunkte für Ablage an den Eckpositionen
Quelle: eigene Ausarbeitung

Die letzten Roboterposen, die jetzt noch definiert werden, sind die Position an welcher der Zylinder abgeholt wird und die Anfahrtsposition dafür. In Abbildung 4.8 sind alle Roboterposen inklusive der Koordinatensysteme zu sehen, welche für die Erstellung des gesamten Roboterpfades benötigt werden.

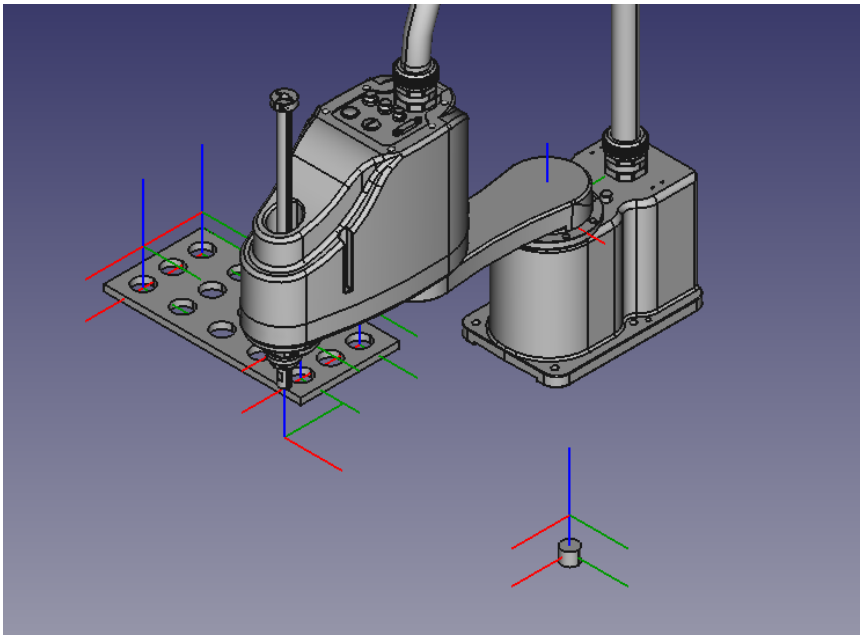

Abbildung 4.8: Alle benötigten Roboterposen
Quelle: eigene Ausarbeitung

4.3 Pfaddefinition

Im nächsten Schritt müssen noch die einzelnen Pfadsegmente definiert werden. Zur besseren Übersicht in den folgenden Abbildungen werden nur Pfade für die in Abbildung 4.1 markierten Ablagepositionen erstellt.

Der Roboterpfad des Roboters beginnt in der Homepose. Das erste Pfadsegment soll denn Roboter-TCP an die Anfahrtsposition für die Aufnahme des Zylinders führen. In Abbildung 4.9 ist das Segment in der 3D-Ansicht zu sehen. Abbildung 4.10 zeigt die darauf folgende Linearbewegung, die den Roboter-TCP zur Aufnahme des Zylinders bewegen soll. Nachdem eine weitere lineare Bewegung zurück zur Anfahrtsposition definiert wurde, kann nun eine kreisförmige Bewegung über die Homepose zur Anfahrt an die erste Ablageposition definiert werden. Die bisher erstellten Pfadsegmente sind in Abbildung 4.11 zu sehen. Anschließend führen weitere Linearbewegungen zur Ablageposition und

zurück. Eine P2P-Bewegung führt den Roboter wieder in Ausgangslage zur Homepose zurück. Der selbe Bewegungsablauf wird jetzt noch für die zweite Ablageposition erstellt. Bei der weiteren Pfaddefinition wäre es eine Erleichterung gewesen, wenn einzelne Pfadsegmente dupliziert werden könnten. Man könnte Bewegungsabläufe kopieren und, wenn nötig, die Roboterposen, die für das jeweilige Segment benötigt werden, anpassen. Die Abbildungen 4.13 und 4.14 zeigen den gesamten Roboterpfad aus unterschiedlichen Perspektiven.

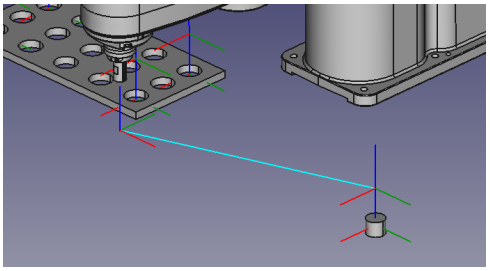

Abbildung 4.9: Pfadsegment P2P-Fahrbefehl zur Anfahrtsposition Quelle: eigene Ausarbeitung

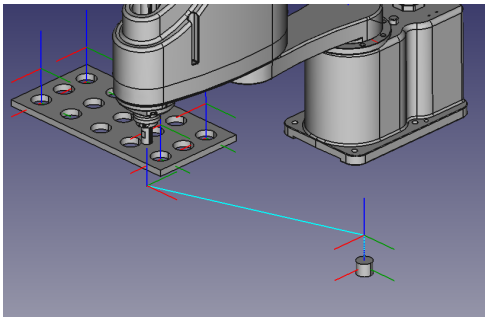

Abbildung 4.10: Pfadsegment linearer Fahrbefehl zur Pick-Position Quelle: eigene Ausarbeitung

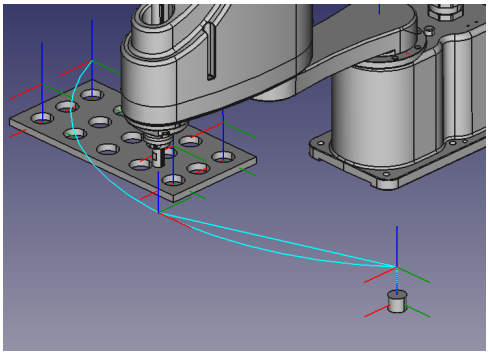

Abbildung 4.11: Pfadsegment zirkulär über Homeposition Richtung Ablageposition Quelle: eigene Ausarbeitung

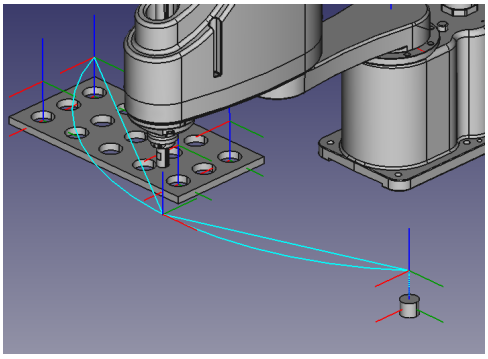

Abbildung 4.12: Pfadsegment P2P zur Homeposition Quelle: eigene Ausarbeitung

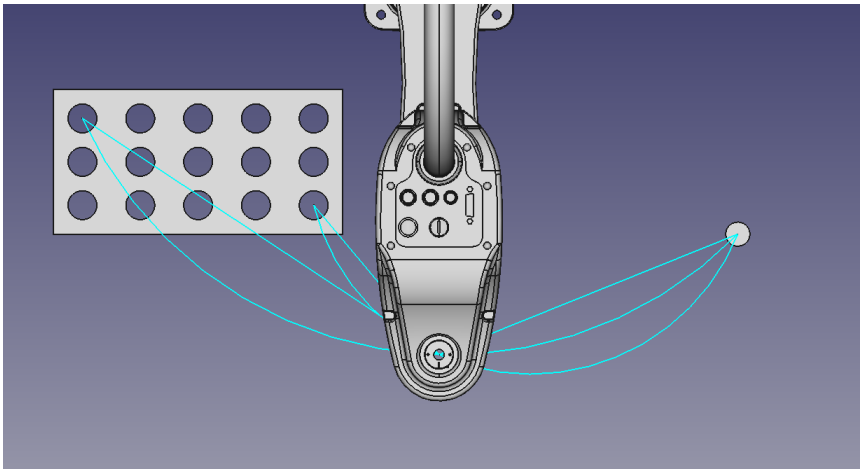

Abbildung 4.13: Übersicht über den gesamten Roboterpfad von oben
Quelle: eigene Ausarbeitung

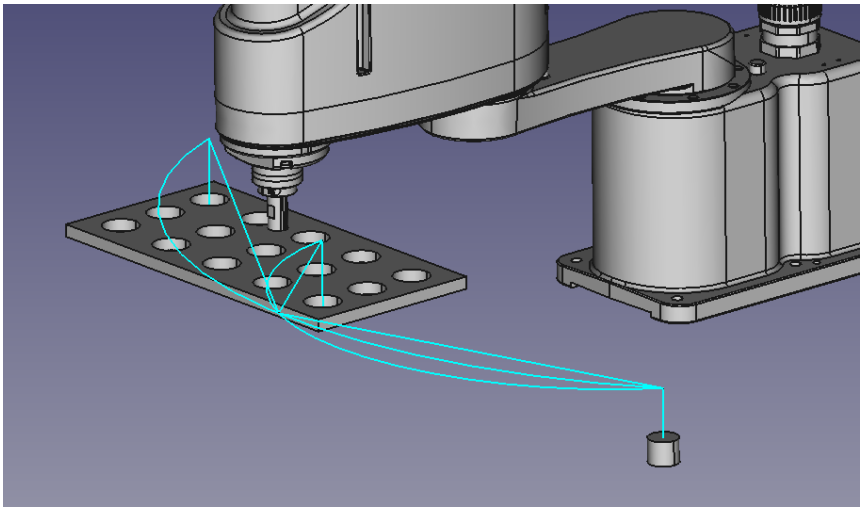

Abbildung 4.14: Seitliche Übersicht über den gesamten Roboterpfad
Quelle: eigene Ausarbeitung

4.4 Pfadfinalisierung

Zum Schluss wird der Roboterpfad noch um weitere Schritte ergänzt, welche keine Bewegung sondern zusätzliche Aktionen repräsentieren. An den jeweiligen Stellen werden Aktionen zum Greifer öffnen beziehungsweise schließen eingefügt. Außerdem erfolgt noch eine Kontrolle der einzelnen Pfadsegmente, bei der die Geschwindigkeiten dieser noch einmal angepasst werden. Beispielsweise sollen die Linearbewegungen zu den Aufnahme-/ beziehungsweise Ablagepositionen mit verminderter Geschwindigkeit erfolgen. In Abbildung 4.15 ist das Fenster zur Bearbeitung des Pfades zu sehen. Hier wäre es möglich, die Funktion zum Duplizieren von Pfadsegmenten, welche in Kapitel 4.3 beschrieben wurde, zu integrieren. Außerdem wäre es wichtig, die Notizen, welche den Pfadsegmenten hinzugefügt wurden, ebenfalls bearbeiten zu können. Anschließend kann der exportierte Pfad zur weiteren Verwendung genutzt werden. Die JSON-Struktur beinhaltet die definierten Pfadsegmente und Aktionen in richtiger Reihenfolge. Abbildung 4.16 zeigt die Struktur mit dem ersten Pfadsegment und der ersten Aktion, welche den Greifer des Roboters öffnet.

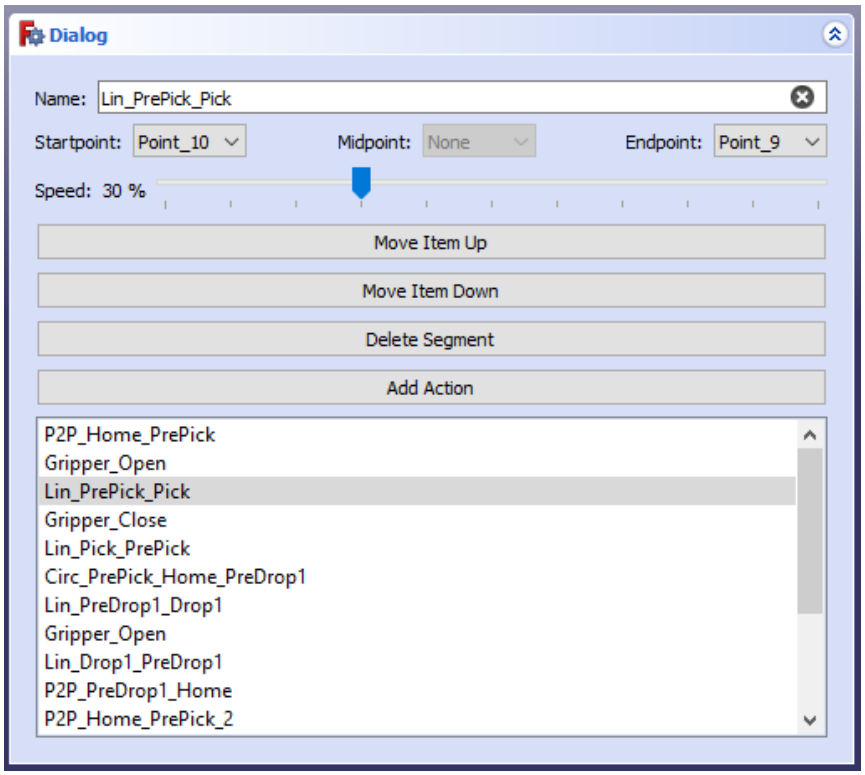

Abbildung 4.15: Bearbeiten des Roboterpfades
Quelle: eigene Ausarbeitung

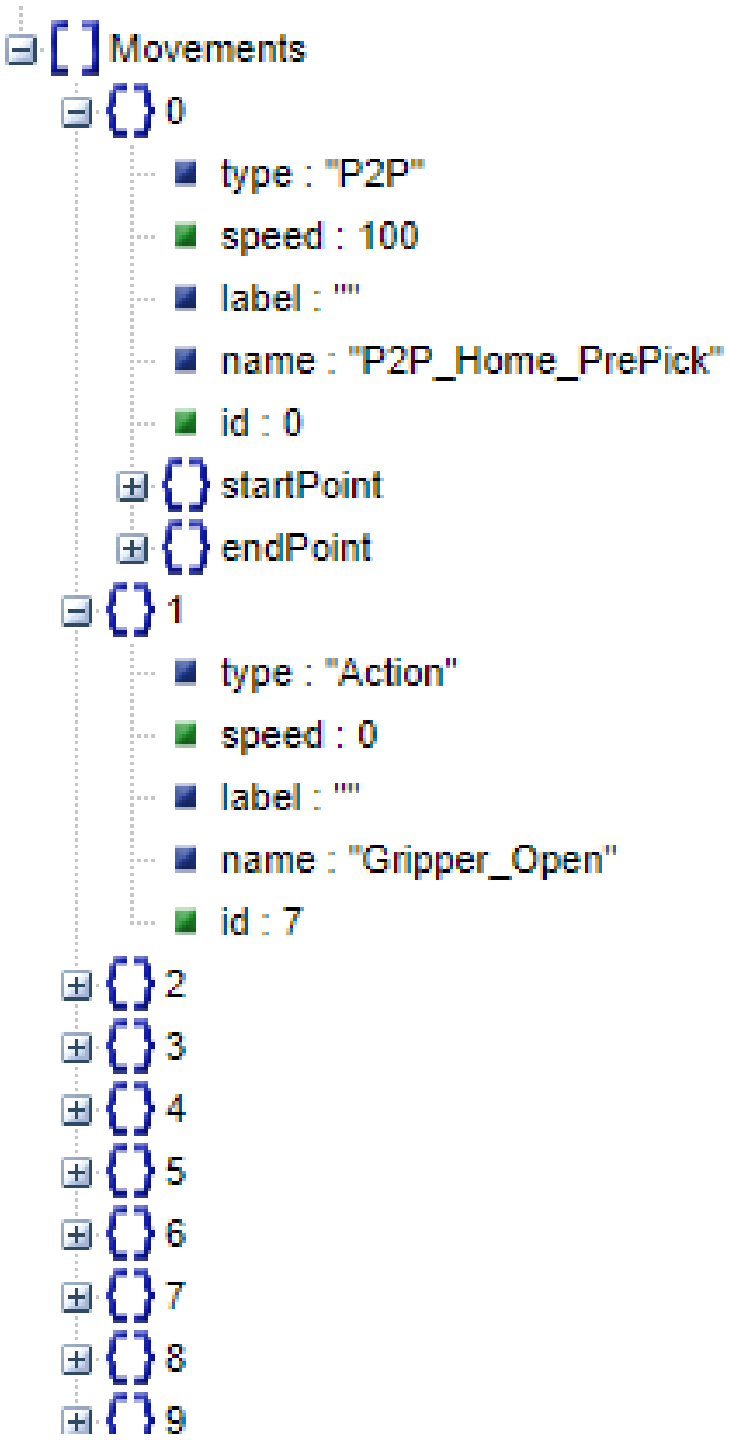

Abbildung 4.16: Ansicht der JSON-Struktur des gesamten Roboterpfades
Quelle: eigene Ausarbeitung

5 Fazit und Ausblick

Ziel dieser Masterarbeit war es, eine Programmierhilfe für Roboterzellen zu entwickeln, mit der die Erstellung von Roboterpfaden ermöglicht werden soll. Im ersten Teil der Arbeit wird dafür der geplante Arbeitsablauf zum Erstellen eines Roboterpfades erläutert. Anschließend wird auf die Vor- und Nachteile von Open-Source-Software eingegangen und ein kurzer Einblick in verschiedene Lizenzmodelle im Bereich von Open-Source gegeben. Im weiteren Verlauf werden grundlegende Begriffe der Robotik erklärt, um die Eindeutigkeit der in dieser Arbeit verwendeten Bezeichnungen zu gewährleisten. Weiters wird analysiert, wie unterschiedliche proprietäre Systeme die Programmierung von Roboterpfaden umsetzen. Nach dem Überblick über den aktuellen Stand der Technik in diesem Bereich, wird die Implementierung des erstellten Plugins für FreeCAD näher beschrieben. Dafür wird zuerst anhand von Beispielen des Programmcodes erklärt, wie die grundsätzliche Erstellung eines neuen Arbeitsbereiches für FreeCAD erfolgen kann. Im Anschluss daran wird die Verwendung der implementierten Funktionen beschrieben. Zum Schluss wird anhand eines Anwendungsbeispiels die Funktionalität der Plugins beurteilt und analysiert, welche weiteren Verbesserungen und Erweiterungen noch möglich sind. Abschließend lässt sich sagen, dass die bisherige Version des Plugins die Anforderungen für einfachere Anwendungen, wie beispielsweise die in Kapitel 4 erstellte Pick & Place Anwendung, größtenteils erfüllt. Allerdings kamen während der Bearbeitung der Aufgabenstellung einige mögliche Erweiterungen beziehungsweise Verbesserungen auf, welche abschließend noch einmal zusammengefasst werden. Ein Teil der erweiterbaren Funktionalitäten bezieht sich auf die Erstellung von Koordinatensystemen und Roboterposen. Bisher gibt es für die schnelle Platzierung dieser die Möglichkeit, Positionen von markanten Punkten zu übernehmen. Dazu zählen derzeit die Mittelpunkte von Flächen, die Mitte von Kanten, Eckpunkte, sowie die Position von bereits definierten Koordinatensystemen beziehungsweise Roboterposen. Hilfreich wäre hierbei die Unterstützung weiterer markanter Punkte, wie beispielsweise der Mittelpunkt von Löchern und Bohrungen. Außerdem wäre es hilfreich, wenn neben der Position auch die Orientierung über eine Schnelleinstellung angepasst werden könnte. Eine mögliche Lösung dafür wäre die Ausrichtung des Koordinatensystems oder der Roboterpose an vorhandenen Kanten, beziehungsweise das Kopieren der Orientierung von bereits definierten Koordinatensystemen und Posen. Eine zusätzliche

mögliche Erweiterung wäre es, das Duplizieren von Pfadsegmenten zu ermöglichen. Dadurch könnte das Erstellen des Roboterpfades beschleunigt werden. Es müssten nur noch die jeweiligen Wegpunkte des Segments und die Bewegungsgeschwindigkeit angepasst werden. Außerdem können in der aktuellen Version des Plugins die Notizen zu den Pfadsegmenten nach der erstmaligen Erstellung nicht mehr bearbeitet werden. Dies wäre auch noch ein wichtiger Punkt, der umgesetzt werden müsste. Da das Plugin in Zukunft auch für komplexere Anwendung zum Einsatz kommen soll, wäre es noch wichtig, die Modularität, wie sie in Kapitel 3.6 beschrieben wurde, zu implementieren. Eine weitere interessante Erweiterung, wenn es um komplexere Aufgabenstellungen geht, wäre das Ermöglichen von Multi-Roboter-Systemen. Dadurch könnten größere Roboterzellen, welche mehrere Roboter für die Anwendung benötigen, in einem einzigen Projekt bearbeitet und aufeinander abgestimmt werden. Ein Konzept für die Implementierung muss dafür noch erstellt werden. Zu berücksichtigen wäre dabei, dass sich die Möglichkeit mehrere Roboter zu verwenden auf alle Bereiche des Plugins (Koordinatensysteme, Posendefinition, Roboterpfade, etc.) auswirkt und diese Erweiterung wahrscheinlich die komplexeste der hier erwähnten ist. Um die Interaktion des Benutzers mit dem Plugin intuitiver zu gestalten, wäre es noch sinnvoll das UX-Design und somit die visuelle Darstellung der Eingabedialoge zu überarbeiten.

Der Programmcode des Plugins wird der Öffentlichkeit in einem Github Repository zur Verfügung gestellt. Das Repository kann unter der Adresse „https://github.com/lerchi0/RobotPathWorkbench“ gefunden und heruntergeladen werden.

Literatur

[1] Michael Alba. *Worin unterscheiden sich die parametrische und die direkte Modellierung?* Mai 2018. URL: https://www.engineering.com/deutsch/cadcae/worin-unterscheiden-sich-die-parametrische-und-die-direkte-modellierung/ (besucht am 14. 08. 2021).

[2] John J. Craig. *Introduction to Robotics: Mechanics and Control.* 3rd Edition. Pearson/Prentice Hall, 2005.

[3] Dr. Andreas Bergler. *Was ist Freeware?* Apr. 2019. URL: https://www.it-business.de/was-ist-freeware-a-830656/ (besucht am 16. 06. 2021).

[4] Elena Di Gianvittorio. *Open-Source-Lizenzen – Grundlagen und Entscheidungshilfen.* URL: https://www.bitfactory.io/de/blog/open-source-lizenzen/ (besucht am 21. 05. 2021).

[5] *Erstellung von Arbeitsbereichen.* URL: https://wiki.freecadweb.org/Workbench_creation/de (besucht am 19. 08. 2021).

[6] *FreeCAD 0.19.1.* März 2021. URL: https://github.com/FreeCAD/FreeCAD/releases/tag/0.19.1 (besucht am 28. 06. 2021).

[7] Kevin M. Lynch und Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control.* Cambridge University Press, 2017. ISBN: 978-1-107-15630-2.

[8] *Makros.* URL: https://wiki.freecadweb.org/Macros/de (besucht am 19. 08. 2021).

[9] PLCopen Technical Committee 2 – Task Force. *Function Blocks for motion control: Part 4 –Coordinated Motion V1.0.* Dez. 2008.

[10] *QtDesigner 5.11.1.* Juni 2018. URL: https://download.qt.io/new_archive/qt/5.11/5.11.1/ (besucht am 28. 06. 2021).

[11] Red Hat Limited. *Was ist Open Source?* URL: https://www.redhat.com/de/topics/open-source/what-is-open-source (besucht am 02. 05. 2021).

[12] Red Hat Limited. *Was ist Open Source-Software?* URL: https://www.redhat.com/de/topics/open-source/what-is-open-source-software (besucht am 02. 05. 2021).

[13] RoboDK Inc. *RoboDK - Simulation & Programming.* URL: https://robodk.com/simulation (besucht am 28. 07. 2021).

[14] Yaskawa Europe GmbH. *Yaskawa Motoman SG400.* 2021. URL: https://www.yaskawa.eu.com/products/robots/pick-place/productdetail/product/sg400_6637 (besucht am 12. 08. 2021).

Anhang

A.1 Befehls-/Klassenübersicht des Plugins

Name	**Funktion**
AddOrigin	Hinzufügen/Bearbeiten von Koordinatensystemen
AddPoints	Hinzufügen/Bearbeiten von Roboterposen
CreateCircSeg	Hinzufügen eines neuen kreisförmigen Pfadsegments
CreateP2PSeg	Hinzufügen eines neuen P2P Pfadsegments
CreateLinSeg	Hinzufügen eines neuen linearen Pfadsegments
CreateSeg	Modul in dem die Befehle zu den einzelnen Pfadsegmenten hinterlegt sind
EditPath	Bearbeiten des Roboterpfades und der einzelnen Pfadsegmente bzw. Hinzufügen von sonstigen Aktionen (Greifer öffnen/schließen)
ReloadView	Aktualisieren der 3D-Ansicht (Koordinatensysteme, Posen und Pfadsegmente neu zeichnen)
Movements	Modul in dem die unterschiedlichen Pfadsegmente definiert sind
RPWClasses	Modul in dem Roboterposen, Koordinatensysteme und die Projektkonfiguration definiert sind
RPWlib	Modul in den die Klassen und Funktionen zum Informationsaustausch definiert sind

Abbildung A.1: Auflistung der erstellten Befehle
Quelle: eigene Ausarbeitung

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Dornbirn, am 27. August 2021 Thomas Lerchbaumer