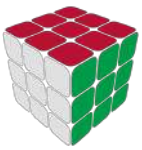
OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG



Fakultät: Elektro- und Informationstechnik
Studiengang: Elektro- und Informationstechnik

Robot-Rubi

Projektarbeit VMCB

Entwicklung und Implementierung eines mechanischen
Rubik's Cube solvers

vorgelegt von

Johannes Hofmann
Matrikelnr: [redacted]

Betreuer: Prof. Dr. Stefan Krämer

Erklärung

1. Mir ist bekannt, dass dieses Exemplar der Projektarbeit als Prüfungsleistung in das Eigentum der Ostbayerischen Technischen Hochschule Regensburg übergeht.

2. Ich erkläre hiermit, dass ich diese Projektarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

JOHANNES HOFMANN

Inhaltsverzeichnis

1 Einleitung

"What I cannot create, I do not understand."
—Richard Feynman

Diese Projektarbeit beschreibt die Entwicklung eines Rubik's Cube Solver Roboters. Zu Beginn werden im Kapitel Grundlagen die Geschichte, Notation und relevanten Algorithmen des Rubik's Cubes behandelt, um ein umfassendes Verständnis der theoretischen Basis zu schaffen. Im Kapitel Konzeption und Design wird der mechanische und elektrische Aufbau des Roboters sowie die zugehörige Desktop-Applikation dargestellt. Anschließend wird im Kapitel Implementierung die praktische Umsetzung dieser erläutert. Abschließend bietet die Zusammenfassung und Ausblick einen Rückblick auf das Projekt und mögliche Weiterentwicklungen.

Motivation

Die Motivation für dieses Projekt entspringt der Faszination für den Guinness-Weltrekord vom 23. Januar 2016, bei dem ein Roboter den Rubik's Cube in weniger als 0,9 Sekunden lösen konnte [1]. Dieser Rekord wurde 2018 von einem Team am Massachusetts Institute of Technology (MIT) mit einer Zeit von 0,38 Sekunden übertroffen und stellt den aktuellen Rekord dar [2]. Obwohl ein solcher Roboter keinen praktischen Nutzen hat, ist die Herausforderung aus ingenieurtechnischer Sicht äußerst interessant. Wie in Kapitel 2 dargelegt, ist das Finden einer möglichst schnellen oder kurzen Lösungssequenz keine triviale Aufgabe und erfordert die Anwendung interessanter Algorithmen für eine optimale Lösung. Darüber hinaus stellt die Entwicklung eines Roboters, der das Einlesen und mechanische Lösen des Cubes übernimmt, eine spannende Aufgabe dar, die viele im Ingenieurstudium erworbene Fähigkeiten vereint.

Angespornt von der Faszination, den kompletten Aufbau eines solchen Roboters zu verstehen, und der Vielfalt der zu bewältigenden Aufgaben – von Mechanik über Elektronik bis hin zu Software und Algorithmen – habe ich mich, ganz im Sinne des berühmten Physikers Richard Feynman, dazu entschlossen, selbst einen solchen Roboter in seiner Gesamtheit zu entwickeln, um diesen letztendlich vollständig zu verstehen.

Projektübersicht

Das Projekt lässt sich, wie in Abbildung 1.1 dargestellt, in folgende Teilbereiche unterteilen, die im späteren Kapitel 3 im Entwurf und Design und in der nachfolgenden Implementierung in Kapitel 4 genauer beschrieben werden:

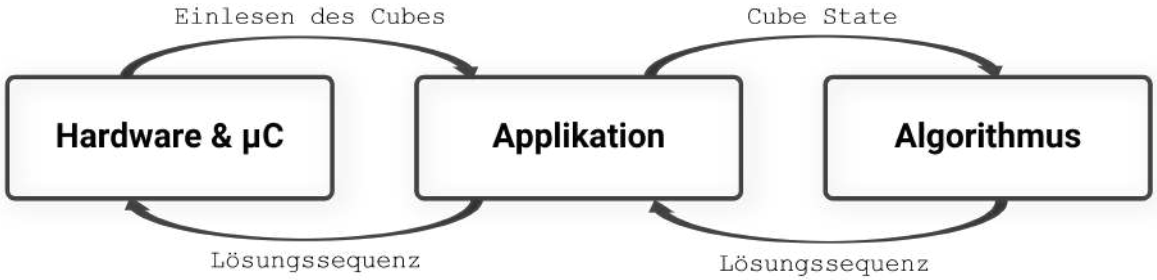


Abbildung 1.1: Übersicht der einzelnen Bestandteile des Projektes und deren Interaktionen untereinander

- **Hardware & μC:**
  - **Mechanisch:** Die mechanische Komponente des Roboters ist dafür zuständig, den Rubik's Cube in einer geeigneten Vorrichtung aufzunehmen, die es zudem erlaubt, die Seiten des Cubes zu drehen und somit den aktuellen Zustand zu manipulieren.
  - **Elektronisch:** Die Elektronik des Roboters soll es ermöglichen, mittels geeigneter Sensorik den aktuellen Zustand des Cubes einzulesen und die Ansteuerung zur Manipulation des Cubes zu ermöglichen.
  - **μC:** Der Microcontroller soll eine Schnittstelle für die Applikation bieten, mittels der die elektronische Hardware angesteuert werden kann.
- **Applikation:** Die Desktop-Applikation soll mit der Elektronik des Roboters kommunizieren und als Interface zu dessen Ansteuerung dienen. Sie stellt zudem die Schnittstelle zum Algorithmus dar und ist somit die zentrale Komponente des Systems.
- **Algorithmus:** Der Algorithmus soll zu einem gegebenen Ist-Zustand des Cubes als Eingabe eine möglichst effiziente Lösungssequenz finden und diese an die Applikation zurückgeben.

2 Grundlagen

In diesem Kapitel werden die grundlegenden Aspekte beleuchtet, die für das Verständnis der Entwicklung eines Rubik's Cube Solver Roboters von Bedeutung sind. Zunächst wird die Geschichte des Rubik's Cubes betrachtet, um die Grundlagen dieses komplexen Puzzles zu verstehen. Anschließend wird das Lösen des Rubik's Cubes thematisiert, wobei verschiedene Lösungsstrategien und -methoden vorgestellt werden. Ein weiterer wichtiger Punkt ist die Notation des Rubik's Cubes, die eine standardisierte Sprache zur Manipulation des Würfels darstellt. Danach folgen die Computermodelle des Rubik's Cubes, die die Grundlagen der Algorithmen zur Lösung des Puzzles darstellen. Abschließend wird der IDA-Algorithmus* beschrieben, der eine zentrale Rolle in der effizienten Lösung des Rubik's Cubes spielt und daher für die Funktionsweise des entwickelten Roboters von entscheidender Bedeutung ist.

2.1 Geschichte des Rubik's Cube

Der Rubik's Cube wurde 1974 von Ernő Rubik erfunden, um seinen Studenten dreidimensionale Bewegung zu lehren. Ursprünglich 'Bűvös kocka' (Zauberwürfel) genannt, wurde er 1980 als Rubik's Cube weltweit populär. Der Würfel erlebte eine rasante Verbreitung und wurde zum meistverkauften Puzzle-Spielzeug mit über 350 Millionen verkauften Exemplaren bis 2018. Er inspirierte zahlreiche Kunstwerke, Filme und die Sportart Speedcubing, bei der Teilnehmer den Würfel in kürzester Zeit lösen. Rubik selbst benötigte einen Monat, um seine Erfindung erstmals zu lösen [3].

2.2 Aufbau und Ziel des Rubik's Cube

Ein Rubik's Cube besteht aus einem mechanischen 3x3x3-Würfel, der aus 26 kleineren Würfeln (Cubies) besteht, die um einen festen Kern rotieren. Jede Seite des Würfels hat neun Sticker in einer der sechs Farben: Weiß, Gelb, Rot, Blau, Grün und Orange.

Das Ziel ist es, den Würfel so zu drehen, dass jede Seite des Würfels vollständig eine Farbe zeigt. Dies wird durch eine Reihe von Rotationen und Algorithmen erreicht, die die Position und Ausrichtung der einzelnen Cubies korrigieren.

2.3 Lösen des Rubik's Cube

Das Lösen des Rubik's Cube kann sowohl händisch als auch mittels Algorithmen durchgeführt werden. Eine weit verbreitete Methode für das händische Lösen ist die CFOP-Methode (Cross, F2L, OLL, PLL), die auch als Fridrich-Methode bekannt ist. Diese Methode besteht aus vier Hauptschritten (Abbildung 2.1): Zunächst wird ein Kreuz auf einer Seite des Würfels gebildet (Cross), gefolgt von der Lösung der ersten beiden Ebenen (First Two Layers, F2L). Danach werden die restlichen Ecken und Kanten der obersten Ebene orientiert (Orientation of the Last Layer, OLL) und schließlich in die korrekte Position permutiert (Permutation of the Last Layer, PLL). Diese Methode ist besonders bei Speedcubing-Wettbewerben beliebt, da sie eine systematische Herangehensweise bietet und eine effiziente Lösung ermöglicht. [4]

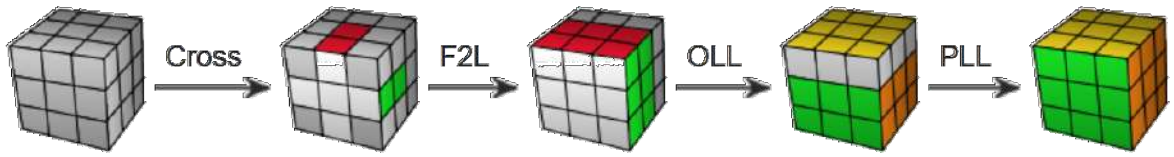


Abbildung 2.1: CFOP Methode zum lösen eines Rubik's Cubes [4]

Computergestütztes Lösen des Rubik's Cube

Im Gegensatz dazu verwenden Computer zur Lösung des Rubik's Cube komplexe Algorithmen, die auf mathematischen Optimierungen basieren. Einer der bekanntesten Algorithmen ist der von Richard Korf entwickelte **Korf-Algorithmus**. Dieser Algorithmus verwendet eine iterative Tiefensuche, die optimal ist, da sie garantiert die kürzest mögliche Lösung findet. Korf's Algorithmus durchsucht den Rubik's Cube Graphen auf der Suche nach dem gelösten Zustand, wobei er eine heuristische Funktion verwendet, um die Lösungssuche zu beschleunigen. [5]

Die maximale Anzahl an Zügen, die benötigt werden, um den Würfel aus jeder möglichen Position zu lösen wird auch als *God's Number* bezeichnet. Nach umfangreichen Berechnungen und Forschungen, unterstützt durch Google, wurde bewiesen, dass diese Zahl 20 beträgt. Das bedeutet, dass jede mögliche Konfiguration des Rubik's Cube in

maximal 20 Zügen[1] gelöst werden kann. Dies wurde durch die Analyse aller 43 Quintillionen möglichen Positionen des Würfels über einen Zeitraum von 35 CPU-Jahren erreicht. [6]

Ein weiterer bedeutender Algorithmus ist der **Thistlethwaite 52-Move Algorithmus**, der den Würfel in vier Phasen löst, wobei jede Phase eine bestimmte Gruppe von Würfelzuständen reduziert. In der ersten Phase wird der Würfel in eine Position gebracht, in der keine Kantenstücke umgekehrt sind. Danach werden in der zweiten Phase alle Kanten in die korrekte Ebene gebracht. In der dritten Phase werden die Kanten und Ecken orientiert und schließlich wird in der vierten Phase der Würfel vollständig gelöst. Dieser Algorithmus garantiert, dass der Würfel in maximal 52 Zügen gelöst werden kann, was ihn zu einem effizienten Ansatz macht. [7]

2.4 Notation des Rubik's Cube

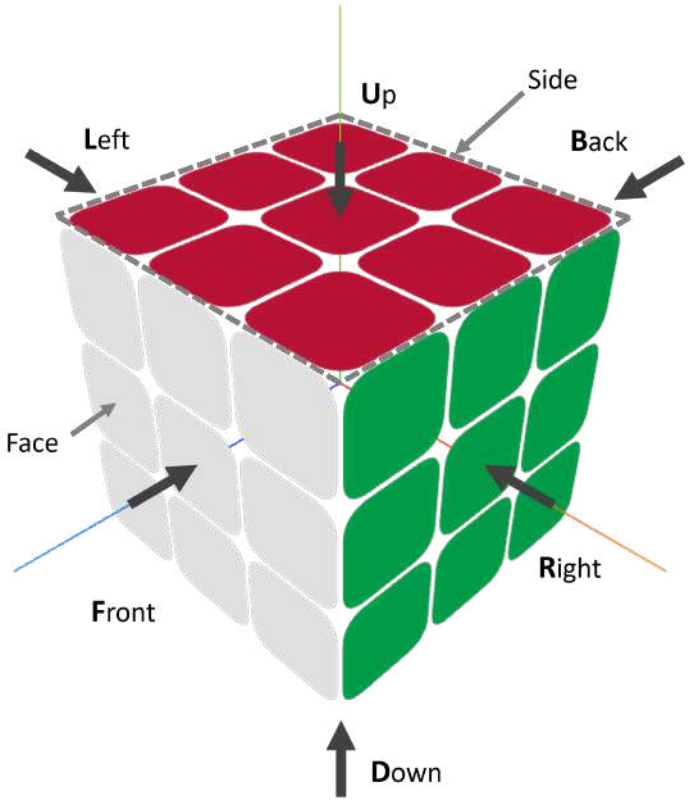


Abbildung 2.2: Übersicht der verwendeten Notation des Rubik's Cubes

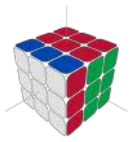
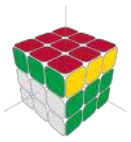
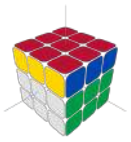
[1] Unter Annahme der *Half-turn metric*, welche sowohl Vierteldrehungen (90 Grad) als auch Halbdrehungen (180 Grad) jeweils als einen Zug zählt

Abbildung 2.2 zeigt die verwendete Notation des Rubik's Cube. Jede der sechs Seiten (engl. *Side*) des Cubes besteht aus neun Flächen (engl. *Face*). Wenn man die Mittelflächen jeder Seite räumlich fixiert und den gesamten Cube in einer isometrischen Ansicht betrachtet, werden die Seiten wie in Abbildung 2.2 in Front, Right, Back, Left, Up und Down unterteilt.

Tabelle 2.1 zeigt die verwendete Notation zur Manipulation des Rubik's Cube. Dabei wird angenommen, dass die Mittelflächen jeder Seite fixiert sind und nur die jeweiligen Seiten gedreht werden können. Das theoretisch mögliche Drehen der Mittelstücke wird nicht gesondert betrachtet, da es äquivalent zum Drehen der beiden angrenzenden Seitenflächen ist. Diese aufgeführten Manipulationsmöglichkeiten des Cubes werden als *Half Turn Metric* bezeichnet, wobei sowohl Vierteldrehungen (90 Grad) als auch Halbdrehungen (180 Grad) jeweils als ein Zug gezählt werden.

Die Konvention für die Abkürzung der Drehvorschriften erfolgt durch die Angabe der zu drehenden Seite (z.B. **F** für *Front*) und der Drehrichtung: 90 Grad im Uhrzeigersinn (**F**), 90 Grad gegen den Uhrzeigersinn (**F'**) oder 180 Grad (**F2**).

Tabelle 2.1: Übersicht der verwendeten Notation zur Manipulation des Rubik's Cubes

F	**R**	**B**	**L**	**U**	**D**
					
F'	**R'**	**B'**	**L'**	**U'**	**D'**
					
F2	**R2**	**B2**	**L2**	**U2**	**D2**
					

2.5 Computermodelle des Rubik's Cube

Um den Rubik's Cube mittels eines Computeralgorithmus zu lösen, ist es zunächst notwendig, ein geeignetes Modell zur Darstellung des Cubes auf dem Computer zu entwerfen. Im Folgenden werden zwei Modelle vorgestellt, die es ermöglichen, den Zustand eines Cubes zu erfassen. Das erste Modell (Array-Modell) ist zwar leicht zu implementieren, jedoch für den später eingesetzten IDA*-Algorithmus ungeeignet. Das zweite Modell (Index-Modell) ist weniger intuitiv, jedoch für die Umsetzung des Korf-Algorithmus geeignet.

Das Array-Modell dient aufgrund seiner schlichten Implementierung als Referenz für die Umsetzung des Index-Modells, da die Manipulationen des Cubes im Array-Modell sehr geradlinig umzusetzen sind und somit für Unit-Tests beim Index-Modell herangezogen werden können.

2.5.1 Array Modell

Das Array-Modell stellt den aktuellen Zustand des Cubes durch ein zweidimensionales Array dar. Der erste Index repräsentiert die Side (Front, Right, Back, Left, Up, Down), während der zweite Index das jeweilige Face angibt. Die Indexierung der Faces ist in Abbildung 2.3 dargestellt und die zugehörige Klasse wird in Listing 2.1 aufgeführt. In dem entsprechend indizierten Array wird dann die jeweilige Farbe des Faces gespeichert.

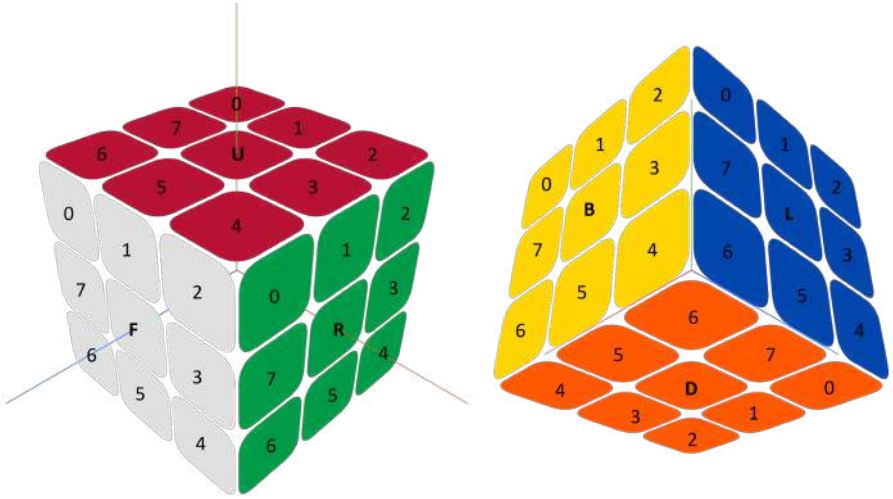


Abbildung 2.3: Indexierung von Faces im Array Modell des Rubik's Cubes

```
/*
 * RubiksCube representation with a two-dimensional array 6x8
 * indixed via cube[FACE][INDEX]:
 *
 *          0 1 2
 *          7 U 3
 *          6 5 4
 *
 *  0 1 2   0 1 2   0 1 2   0 1 2
 *  7 L 3   7 F 3   7 R 3   7 B 3
 *  6 5 4   6 5 4   6 5 4   6 5 4
 *
 *          0 1 2
 *          7 D 3
 *          6 5 4
 */
class RubiksCubeArrayModel : public RubiksCube
{
private:
    Color cube[6][8];
}
```

Listing 2.1: Rubik's Cube Array Model

2.5.2 Index Modell

Das Index-Modell stellt den Cube durch zwei Arrays dar, wobei eines den Zustand der Corner-Cubies (3-Faces) und das andere den der Edge-Cubies (2-Faces) beschreibt.

Im Corner-Cubie-Array werden die Zustände der acht Corner-Cubies festgehalten. Hierbei wird zwischen dem Postion-Index und dem Cubie-Index unterschieden. Der Postion-Index gibt an, welcher Corner-Cubie im Cube betrachtet wird, wie in Abbildung 2.4 dargestellt. Am entsprechenden Postion-Index im Corner-Cubie-Array wird dann ein Corner-Cubie-Struct gespeichert, welche den aktuellen Corner-Cubie an dieser Stelle mittels des Cubie-Index kennzeichnet und dessen Orientierung angibt.

Im Solved-State stimmen die Cubie-Indizes mit den Postion-Indizes überein. Das bedeutet, dass beispielsweise am Postion-Index 3 zu Beginn das Corner-Cubie-Struct eines Corner-Cubies abgelegt ist, welches den Cubie-Index 3 besitzt. Der Cubie-Index 3 beschreibt hierbei den Corner-Cubie mit den Farben **RGW** (Red, Green, White) (**URF** für Up, Right, Front, falls auf die Farben der Mittelflächen normiert wird). Falls bei einem manipulierten Cube der Postion-Index 3 einen Corner-Cubie mit dem Cubie-Index 0 beinhaltet, bedeutet dies, dass an Position 3 der Corner-Cubie mit den Farben **RWB** (Red, White, Blue) (**UFL** für Up, Front, Left, falls auf die Farben der Mittelflächen normiert wird) positioniert ist.

Die Orientierung in dem Corner-Cubie-Struct gibt an, in welcher Reihenfolge die Farben des jeweiligen Cubies angeordnet sind – im gelösten Zustand sind diese jeweils auf 0 gesetzt. Abbildung 2.4 zeigt beispielsweise, wie die Orientierung des Corner-Cubies mit dem Cubie-Index 3 an Position 3 zu interpretieren ist.

Für die Edge-Cubies wird ähnlich vorgegangen, wobei es hier insgesamt 12 Positionen gibt und die Edge-Cubies anstelle von drei nur zwei Orientierungen besitzen können. Die Postion-Indexes und ein Beispiel der Orientierung des Edge-Cubies mit dem Cubie-Index 0 an Position-Index 0 sind in Abbildung 2.5 dargestellt.

In Listing 2.2 ist die Klasse für das Index-Modell und die entsprechenden Structs für die Edge- und Corner-Cubies aufgelistet.

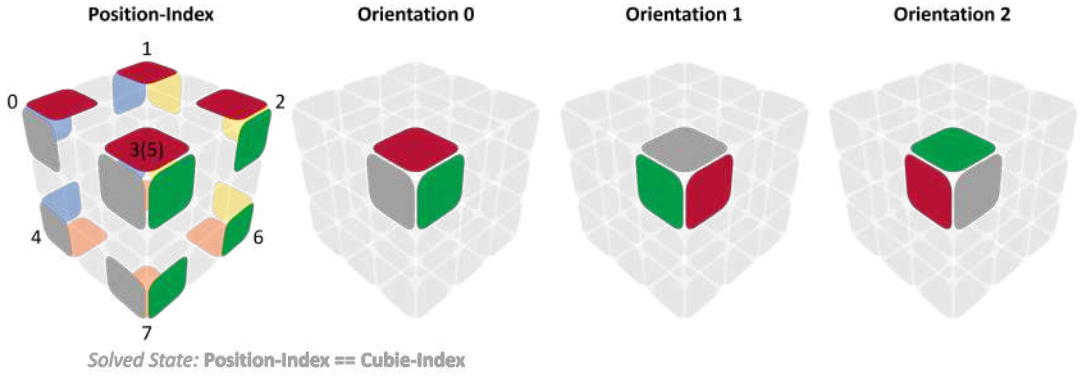


Abbildung 2.4: Übersicht der Corner-Cubie Indexierung und Orientierung im Index Modell. Das Orientierungs-Beispiel zeigt am Position-Index 3 den Corner-Cube mit dem Cubie-Index 3 ⇒ Corner-Cubie mit den Farben **RGW** *Red, Green, White* (**URF** *Up, Right, Front*, falls auf die Farben der Center Faces normiert wird).

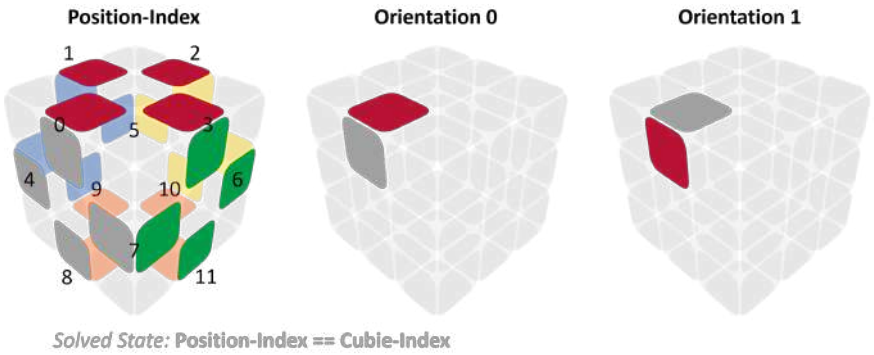


Abbildung 2.5: Übersicht der Edge-Cubie Indexierung und Orientierung im Index Modell. Das Orientierungs-Beispiel zeigt am Position-Index 0 den Edge-Cube mit dem Cubie-Index 0 ⇒ Edge-Cubie mit den Farben **RW** *Red, White* (**UF** *Up, Front*, falls auf die Farben der Center Faces normiert wird).

```
/*
 * RubiksCube Index representation with
 * - 8 corners with 3 orientations
 * - 12 edges with 2 orientations
 */
class RubiksCubeIndexModel : public RubiksCube
{
    struct cornerCubie
    {
        uint8_t index; // [0,1,2,3,4,5,6,7]
        uint8_t orientation; // [0,1,2]
    };

    struct edgeCubie
    {
        uint8_t index; // [0,1,2,3,4,5,6,7,8,9,10,11]
        uint8_t orientation; // [0,1]
    };

    /*  corners[CORNER_POSITION_INDEX]
        CORNER_POSITION_INDEX (== CORNER_CUBIE_INDEX if solved state):
        UFL=0, ULB=1, UBR=2, URF=3, DLF=4, DBL=5, DRB=6, DFR=7 */
    cornerCubie corners[8];

    /*  edges[EDGE_POSITION_INDEX]
        EDGE_POSITION_INDEX (== EDGE_CUBIE_INDEX if solved state):
        UF=0,   UL=1,   UB=2,   UR=3,   FL=4,   LB=5,
        BR=6,   RF=7,   DF=8,   DL=9,   DB=10,  DR=11 */
    edgeCubie   edges[12];
};
```

Listing 2.2: Rubik's Cube Index Model

Beim genaueren Betrachten der beiden Darstellungsweisen wird deutlich, dass beide auf die Darstellung des Zustands der Mittelflächen verzichten. Dies liegt daran, dass die in Kapitel 2.4 eingeführte Notation keine Drehung der Mittelflächen berücksichtigt (was äquivalent zur gleichzeitigen Drehung der angrenzenden Außenflächen ist), sodass sich die Farben der Mittelflächen nicht ändern können. Daher kann letztlich vollständig auf die Notation der Farben verzichtet werden, wie im Listing 2.2 gezeigt, indem die einzelnen Seiten anstelle der Farbe durch die Buchstaben (**F**ront, **R**ight, **B**ack, **L**eft, **U**p, **D**own) im Solved-State beschrieben werden (bzw. normiert auf die Mittelflächen).

Im Vergleich zum Array-Modell erscheint das Index-Modell willkürlich und nicht sehr intuitiv. Die Vorteile dieser Darstellungsweise werden jedoch erst im späteren Verlauf des Kapitels 4.4 bei der Implementierung des Korf-Algorithmus deutlich.

2.6 IDA* Algorithmus

Der in Kapitel 2.3 erwähnte Korf-Algorithmus zur Lösung des Rubik's Cubes basiert auf dem IDA*-Algorithmus (Iterative Deepening A*). Um die Funktionsweise dieses Algorithmus zu verstehen, ist es hilfreich, seine Grundlagen und Vorläuferalgorithmen zu betrachten: Graphen, Dijkstra's Algorithmus, A* und Iterative Deepening Search.

Graphen

Ein Graph besteht aus einer Menge von Knoten (Vertices) und einer Menge von Kanten (Edges), die Paare von Knoten verbinden. Graphen können gerichtet oder ungerichtet sein und mit Gewichten auf den Kanten versehen werden, die die Kosten für den Übergang von einem Knoten zu einem anderen repräsentieren. Graphen sind eine fundamentale Struktur in der Informatik und werden häufig zur Darstellung von Netzwerken, Straßenkarten und vielen anderen Beziehungen verwendet.

Der Rubik's Cube kann, wie in Abbildung 2.6 dargestellt, ebenfalls als Graph betrachtet werden, wobei die Knoten die möglichen Permutationen des Rubik's Cubes darstellen und jede Permutation durch 18 Kanten (entsprechend der möglichen Manipulationen) mit den benachbarten Zuständen verbunden ist. Dieser Graph ist ungerichtet, endlich, zusammenhängend (zwischen jedem Knotenpaar existiert ein Pfad) und gewichtet. Die Gewichte der Kanten betragen jeweils $+1$, da jede Kante eine der 18 möglichen Manipulationen des Cubes darstellt.

Die Problemstellung, für einen beliebig manipulierten Rubik's Cube die schnellstmögliche Lösungssequenz zu finden, lässt sich somit auf die Aufgabe reduzieren, den kürzesten Pfad in einem gewichteten Graphen zu ermitteln.

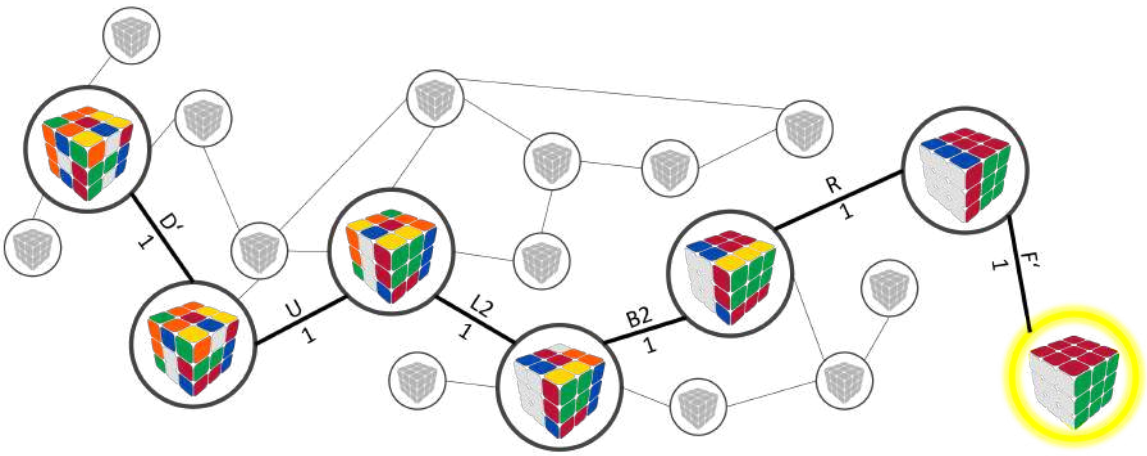


Abbildung 2.6: Darstellung eines Teilbereiches des Rubik's Cube Graphen mit einen visualisierten optimalen Pfad von einem Startpunkt zum Zielpunkt (gelöster Cube)

Dijkstra's Algorithmus

Dijkstra's Algorithmus ist ein bekannter Algorithmus zur Bestimmung der kürzesten Pfade in einem Graphen mit nicht-negativen Kantengewichten. Der Algorithmus arbeitet wie folgt:

1. Initialisiere die Distanzen zu allen Knoten mit Unendlich, außer dem Startknoten, der die Distanz 0 erhält.
2. Setze alle Knoten als unbesucht. Markiere den Startknoten als aktuellen Knoten.
3. Für den aktuellen Knoten:
    - Aktualisiere die Distanzen aller Nachbarknoten. Für jeden Nachbarknoten wird überprüft, ob der Weg über den aktuellen Knoten kürzer ist als der bisher bekannte Weg. Wenn ja, wird die kürzere Distanz gespeichert.
    - Markiere den aktuellen Knoten als besucht.
    - Wähle den unbesuchten Nachbarknoten mit der geringsten Distanz als neuen aktuellen Knoten.
4. Wiederhole Schritt 3, bis alle Knoten besucht wurden oder Zielknoten erreicht ist.

Aufgrund der enormen Komplexität des Zustandsraums des Rubik's Cube, der über 43 Quintillionen Zustände umfasst, ist der vorliegende Algorithmus ungeeignet. Der Dijkstra-Algorithmus würde versuchen, alle möglichen Zustände zu durchlaufen und zu bewerten, was in diesem Fall extrem ineffizient und speicherintensiv ist.

A* Algorithmus

Der A* (A-Star) Algorithmus ist eine erweiterte Version des Dijkstra-Algorithmus, die speziell für Pfadfindungsprobleme entwickelt wurde. Er kombiniert die Eigenschaften des Dijkstra-Algorithmus mit einer heuristischen Schätzfunktion, um effizientere Pfade zu finden.

Die Gesamtkostenfunktion an einem Knoten n definiert sich wie folgt:

$$f(n) = g(n) + h(n)$$

- $g(n)$: die Kosten vom Startknoten zum aktuellen Knoten n (Dijkstra-Algorithmus)
- $h(n)$: die geschätzten Kosten vom aktuellen Knoten n zum Zielknoten (Heuristik)

Der Hauptunterschied zwischen dem A* und dem Dijkstra-Algorithmus liegt in der Verwendung der heuristischen Funktion $h(n)$. Während Dijkstra nur die tatsächlich zurückgelegten Kosten $g(n)$ berücksichtigt, integriert A* auch eine Schätzung der verbleibenden Kosten $h(n)$.

A* Algorithmus - Heuristik

Um mit dem A* Algorithmus eine optimale Lösung zu finden, ist es entscheidend, dass die Heuristik die tatsächlichen Kosten zum Zielknoten niemals unterschätzt. Andernfalls könnte der Algorithmus einen Pfad wählen, der in Wirklichkeit teurer ist als ein anderer, der nicht gewählt wurde.

Ein anschauliches Beispiel für eine gute Heuristikfunktion ist die euklidische Distanz in der Routenplanung. Wie in Abbildung 2.7a dargestellt, kann ein Verkehrsnetz als gewichteter Graph dargestellt werden, bei dem die Knoten Kreuzungen und die Kanten Straßenverbindungen mit deren Länge als Gewicht repräsentieren. Die Heuristik ist in diesem Fall die euklidische Distanz bzw. Luftlinie zwischen Start- und Zielpunkt und wird wie folgt berechnet: $h(n) = \sqrt{(x_{\text{goal}} - x_{\text{current}})^2 + (y_{\text{goal}} - y_{\text{current}})^2}$

Durch die angepasste Gewichtung des Graphen werden die Gewichte der Kanten, die schneller zum Ziel führen, weniger stark erhöht als die der Kanten, die vom Zielknoten wegführen. Dies wird in Abbildung 2.7b verdeutlicht, wo die Heuristik an jedem Knoten auf der z-Achse aufgetragen wurde. Dadurch wird das zweidimensionale Straßennetz dreidimensional und es entsteht ein Gradient in Richtung des Zielknotens.

Wendet man nun den Dijkstra-Algorithmus mit den angepassten Gewichten an, durchsucht dieser den Graphen zielgerichteter und effizienter.

(a) Darstellung eines Verkehrsnetz als gewichteter Graph, wobei die Knoten Kreuzungen darstellen und die Kanten die Verbindungen darstellen. Das Gewicht der Kanten ist hierbei die Länge der Strecke.

(b) Visualisierung der Heuristik Euklidische Distanz (Luftlinie) am Beispiel des Verkehrsnetz. Hierbei wird der Anteil der Heuristik an jeder Kante auf der z-Achse aufgetragen.

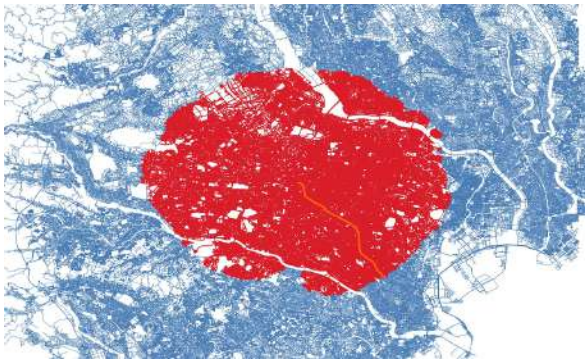
Abbildung 2.7: Visualisierung der Euklidische Distanz als Heuristikfunktion [8]

Vergleich der Dijkstra- und A*-Suchalgorithmen

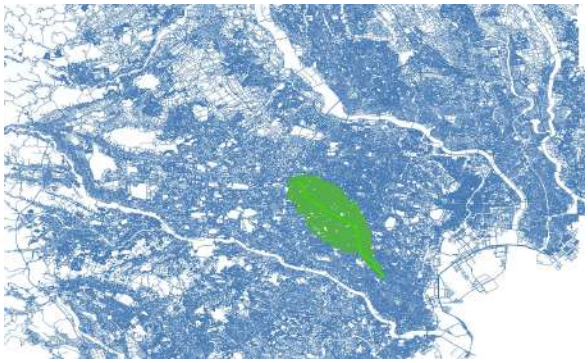
Abbildung 2.8 zeigt einen visuellen Vergleich zwischen dem Dijkstra-Algorithmus und dem A*-Algorithmus bei der Suche nach dem kürzesten Pfad in einem Straßennetz. In Abbildung 2.8a wird gezeigt, wie der Dijkstra-Algorithmus vom Startpunkt in der Mitte des Bildes ausgehend radial in alle Richtungen das Straßennetz durchsucht. Dabei werden viele Verbindungen berücksichtigt, die offensichtlich weiter vom Ziel entfernt liegen.

Im Gegensatz dazu verwendet der A*-Algorithmus die euklidische Distanz als Heuristik und durchsucht das Straßennetz wesentlich zielgerichteter und somit effizienter. Dies wird deutlich durch den grün markierten Bereich in Abbildung 2.8b, der den erforschten Teil des Graphen darstellt. Im Vergleich dazu ist der erforschte Bereich (rot) bei Dijkstra in Abbildung 2.8a wesentlich größer.

Beide Algorithmen haben den optimalen Weg gefunden, jedoch musste Dijkstra hierfür 209.744 Kanten im Graphen besuchen, während A* lediglich 28.254 Kanten untersuchen musste [9].



(a) Unentdeckte Knoten und Kanten in Blau. Der von Dijkstras Algorithmus erforschte Teil des Graphen ist in Rot dargestellt und beginnt in der Mitte des Bildes. Der kürzeste Pfad ist orange eingefärbt.



(b) Unentdeckte Knoten und Kanten in Blau. Der vom A*-Suchalgorithmus erforschte Teil des Graphen ist in Grün dargestellt und beginnt in der Mitte des Bildes. Der kürzeste Pfad ist hellgrün eingefärbt.

Abbildung 2.8: Vergleich von Dijkstras (a) und A*-Suchalgorithmus (b) [9]

Iterative Deepening Search (IDS)

Iterative Deepening Search kombiniert die Tiefensuche (Depth-First Search, DFS) mit der Breitensuche (Breadth-First Search, BFS). Der Algorithmus arbeitet in wiederholten Tiefensuche-Durchläufen, wobei jedes Mal die maximale Tiefe (Depth Limit) erhöht wird. Dies hat den Vorteil des geringen Speicherverbrauchs der Tiefensuche und garantiert wie die Breitensuche die vollständige Abdeckung des Suchraums.

3 Konzeption und Design

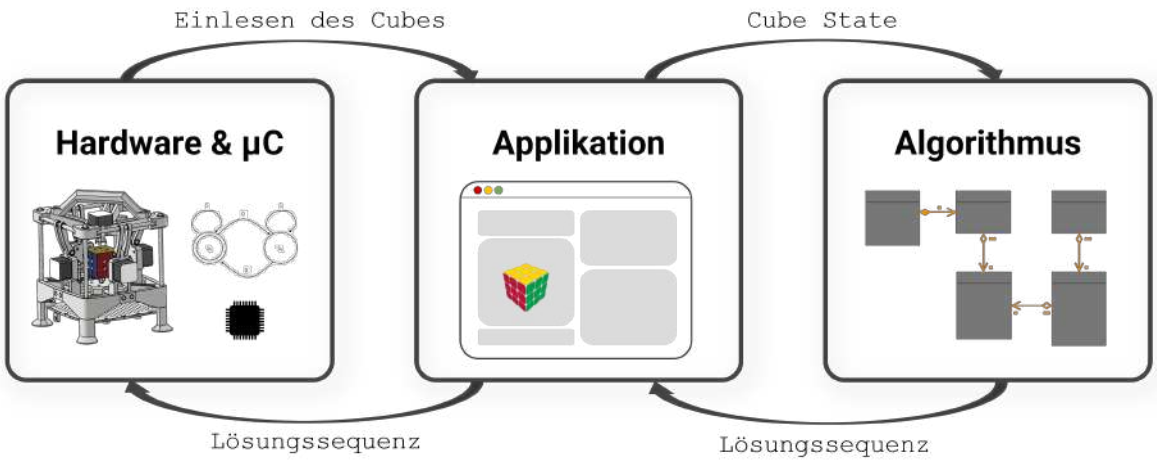


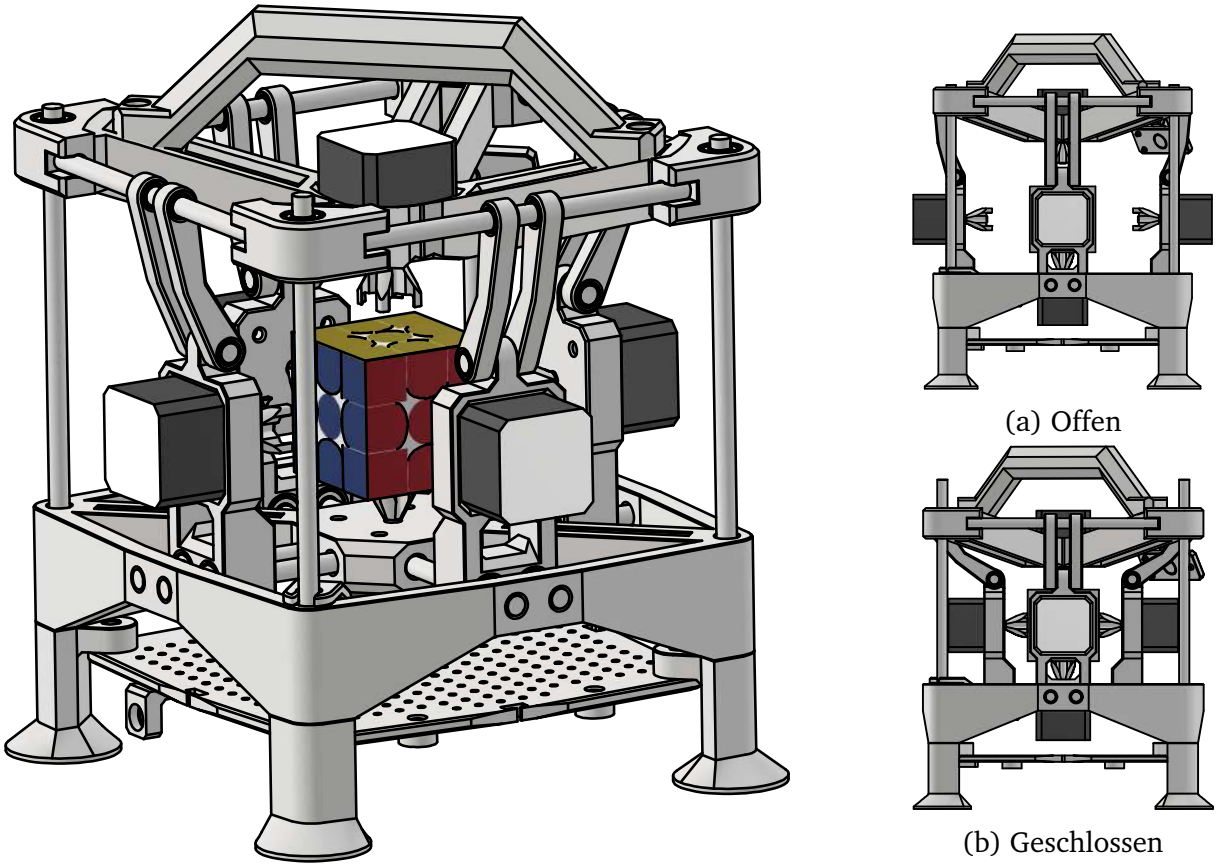
Abbildung 3.1: Übersicht der Projektbestandteile und Ergebnisse aus der Konzeptions- und Designphase

Abbildung 3.1 zeigt die verschiedenen Bestandteile des Projekts und visualisiert die Ergebnisse der Konzeptions- und Designphase. In den kommenden Unterkapiteln werden die folgenden wesentlichen Aspekte behandelt:

- **Hardware & μC:**
  - **Mechanisch:** Entwurf des mechanischen Rubik's Cube Solvers im CAD.
  - **Elektronisch:** Ausarbeitung der benötigten Komponenten für die Elektronik des Roboters.
  - **μC:** Entwurf einer Zustandsmaschine (FSM), die als Kommunikationsschnittstelle zur Applikationssoftware dient und Festlegung des Protokolls der seriellen Schnittstelle.
- **Applikation:** Entwurf eines Schemas für die Benutzeroberfläche der Applikationssoftware und deren Komponenten.
- **Algorithmus:** Klassenentwurf für den Cube Solver.

3.1 Mechanischer Aufbau

Die Grundidee zur Manipulation des Rubik's Cubes besteht darin, die Seiten des Cubes durch das Ansteuern von Stepper-Motoren zu drehen. Daher ist es notwendig, an jeder der sechs Seiten des Cubes einen Motor zu platzieren. Die größte Herausforderung besteht darin, den Cube in den Roboter einlegen und herausnehmen zu können, ohne dass die Motoren im Weg sind. Das finale Design des Roboters, das in Abbildung 3.2 dargestellt ist, löst dieses Problem, indem die seitlichen vier Schrittmotoren auf Linearachsen montiert sind. Diese werden durch einen Hebelmechanismus beim Anheben des oberen Motors gleichzeitig nach außen gedrückt, wodurch ausreichend Platz zum Einlegen und Herausnehmen des Cubes geschaffen wird.



(a) Offen

(b) Geschlossen

Abbildung 3.2: Darstellung des im CAD entworfenen Roboters zum Lösen des Rubik's Cubes. Die rechten Bilder zeigen den Roboter in der Seitenansicht im geöffneten Zustand (a) und im geschlossenen Zustand (b).

Da die Möglichkeit der additiven Fertigung mittels eines FDM-3D-Druckers zur Verfügung stand, wurden die meisten Teile für den 3D-Druck entworfen. Es sind Lediglich zusätzliche mechanische Führungselemente wie Linearkugellager, Linearführungen und Gleitlager sowie Schrauben wurden verwendet. Eine detaillierte Ansicht der Mechanik

bietet die Explosionsdarstellung in Abbildung 3.3, die in Verbindung mit der Stückliste in Tabelle 3.1 einen Überblick über die verwendeten und entworfenen Teile des Roboters gibt.

Für die Modellierung im CAD und die Erstellung der Zeichnungen wurde die Software Autodesk Inventor verwendet.

Tabelle 3.1: Stückliste zu Abbildung 3.3

Anzahl	**Name**	**Beschreibung**
3D - Printed Parts		
1	Top Frame	
1	Bottom Frame	
1	Perforated Plate	Lochplatte für Elektronik Montage
4	Feet	
1	Socket Holder	für DC Buchse 5,5x2,1
4	Stepper Holder	
8	Lever	
6	Gripper	
1	Webcam Holder Top	
1	Webcam Holder Bottom	
1	Handle	
Norm Parts		
6	Stepper	Nema 17, 42Ncm 1.5A 42x42x39, 24V
12	Linear Ball Bearing	LM8UU ⌀8x⌀15x24
16	Bearing	⌀8x⌀10x10
4	Top Frame Linear Rail	⌀8x200
4	Top Frame Horizontal Linear Rail	⌀8x150
8	Stepper Linear Rail	⌀8x85
4	Lever Linear Rail	⌀8x30
X	Screws	nach Notwendigkeit

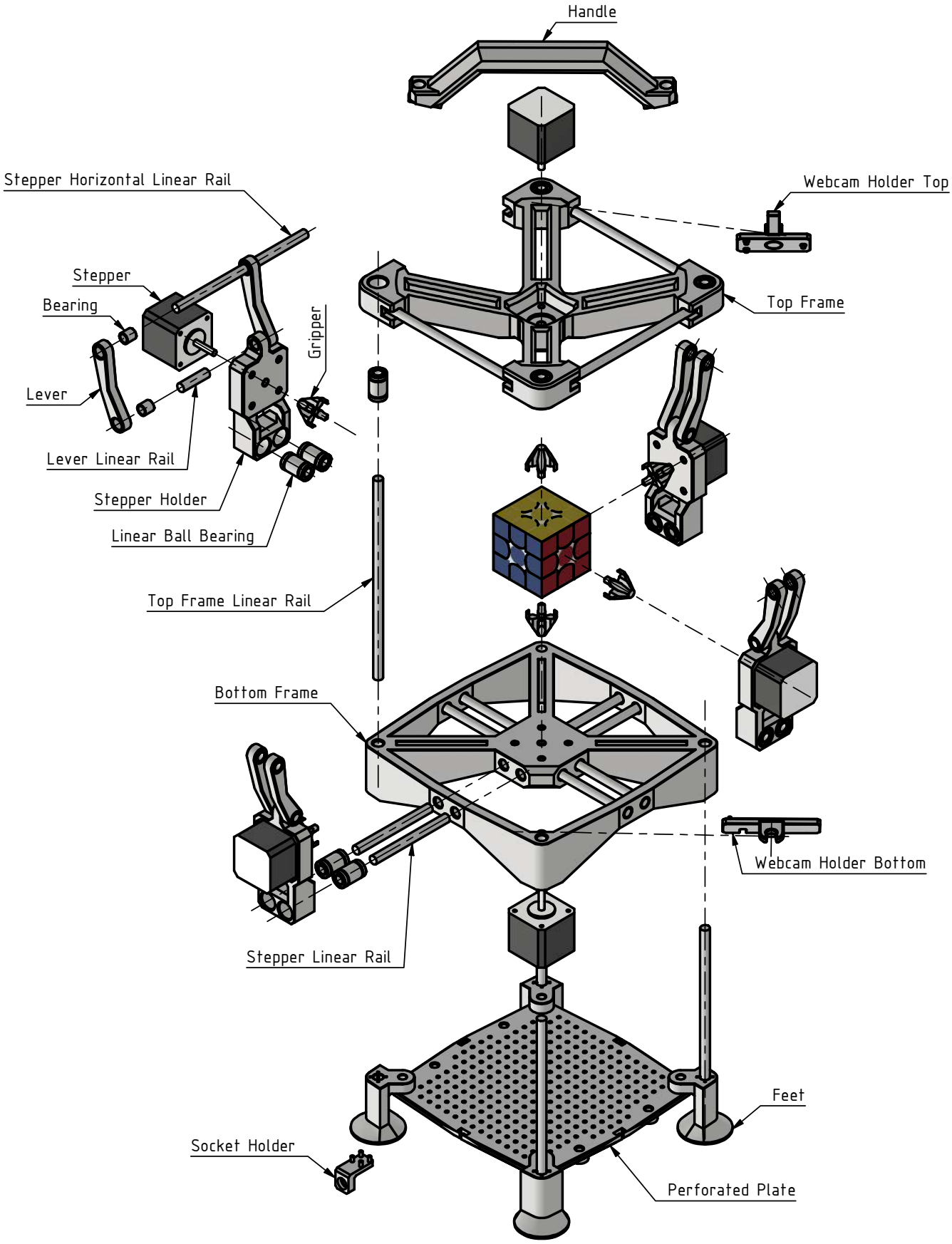


Abbildung 3.3: Explosionsansicht des CAD-Modells und Benennung der einzelnen Teile

3.2 Elektronik - Hardware

Die erforderliche elektronische Hardware sowie deren Schnittstellen sind in Abbildung 3.4 dargestellt. Die einzigen Schnittstellen zum Roboter beschränken sich auf die Versorgungsspannung und eine serielle USB-C-Verbindung. Innerhalb des Roboters kann die Hardware in drei Hauptkomponenten unterteilt werden: die Leiterplatte (PCB), die Schrittmotoren und die Webcams mit Beleuchtung zur besseren Erkennung des Cubes. Im Folgenden werden die einzelnen Komponenten aufgelistet und beschrieben:

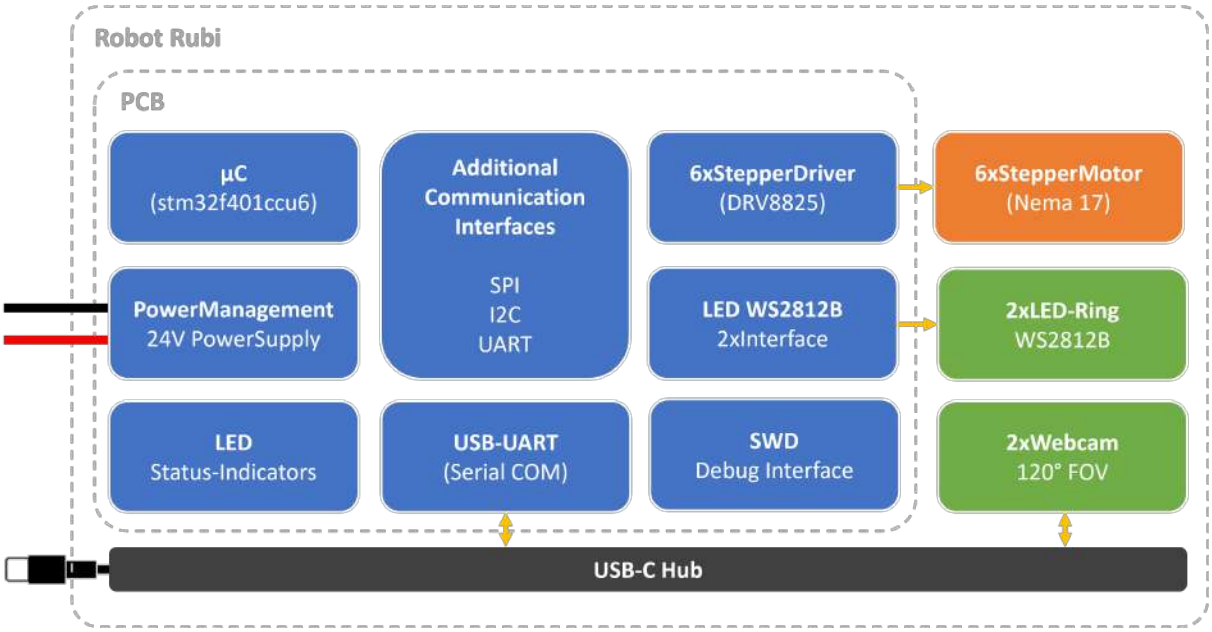


Abbildung 3.4: Blockschaltbild des Entwurfs der benötigten elektronischen Komponenten

- **PCB:** Leiterplatte, welche alle Komponenten zusammenfasst, die direkt vom μC angesteuert werden.
  - **μC:** STM32F401CCU6 - basierend auf dem Controller des weit verbreiteten *Black-Pill* Dev-Boards - 64 KB RAM, 256 KB ROM
  - **Power Management:** 24V Spannungsversorgung für die Schrittmotoren, zudem werden 5V für die Ring-LEDs an den Webcams und 3,3V für den μC benötigt.
  - **LED Status:** LEDs auf der Leiterplatte, die die Ansteuerung der Schrittmotor-Treiber visualisieren und die Spannungsversorgung signalisieren.
  - **USB-UART:** Serielle Schnittstelle zum μC, welche über den USB-C-Hub nach außen geführt wird.

  - **Additional Communication:** Zusätzliche Kommunikationsschnittstellen (UART, SPI, I2C), die bisher nicht berücksichtigte Erweiterungen ermöglichen
  - **Stepper Driver:** Treiberbausteine (DRV8825) für die Schrittmotoren mit $\frac{1}{32}$ Microstepping für einen gleichmäßigeren Lauf und geringere Lärmbelastung
  - **LED Interface:** Schnittstelle für die Ring-LEDs der Webcams
  - **SWD:** Debug/Flash-Schnittstelle des μC
- **Stepper Motoren:** Nema 17 Schrittmotoren für die Manipulation des Cubes.
- **LED Ring:** Lichtquelle zum Auslesen des Cubes mittels der Webcams.
- **Webcam:** Bildquelle zum Erfassen des aktuellen Cube-Zustands, welche über den USB-C-Hub nach außen geführt werden.
- **USB-C Hub:** Hub für das zusammengefasste Interface für die serielle Schnittstelle des μC und der beiden Webcams.

3.3 Elektronik - Software - μC

Der Mikrocontroller (μC) stellt über eine serielle USB-Schnittstelle ein Kommunikationsinterface zur Applikationssoftware bereit. Die Software für den Mikrocontroller soll mithilfe einer einfachen Zustandsmaschine (FSM) umgesetzt werden, deren Entwurf in Abbildung 3.5 dargestellt ist. Zu Beginn werden die Hardware und die Kommunikationsschnittstellen initialisiert. Schlägt diese Initialisierung fehl, wird in einen Fehlerzustand (ErrorState) gewechselt, in dem periodisch eine Fehlermeldung seriell ausgegeben wird. Ist die Initialisierung erfolgreich, wird im Kommunikationszustand (ComState) periodisch auf Nachrichten gehört, die durch ein Nullzeichen (`'\0'`) terminiert sind. Dies bietet eine einfache Methode, um festzustellen, wann eine vollständige Nachricht empfangen wurde. Es sollen zwei Arten von Nachrichten verarbeitet werden:

LED-Kommandos: Diese ermöglichen das Ein- und Ausschalten der Ring-LEDs an den Webcams sowie die Anpassung ihrer Helligkeit. Hier sind einige Beispiele:

- `"LED ON\0"` - Einschalten der LEDs mit 100% Helligkeit
- `"LED OFF\0"` - Ausschalten der LEDs
- `"LED 182\0"` - Einschalten der LEDs mit einem Helligkeitswert im Bereich von 0 bis 255

Bewegungskommandos: Diese spezifizieren eine Abfolge von Manipulationen des Cubes, die entsprechend ausgeführt werden sollen. Hierfür wird die in Kapitel 2.4 eingeführte Notation verwendet.

Beispiel: `"F U2 L'\0"` - Drehe den Stepper, der auf das **F**ront-Face gerichtet ist, um 90° im Uhrzeigersinn, den Stepper, der auf das **U**p-Face gerichtet ist, um 180° im Uhrzeigersinn, und den am **L**eft-Face um 90° gegen den Uhrzeigersinn.

Es ist wichtig, dass diese Bewegungen sequenziell abgearbeitet werden, wobei gegenüberliegende Faces parallel zueinander bewegt werden können.

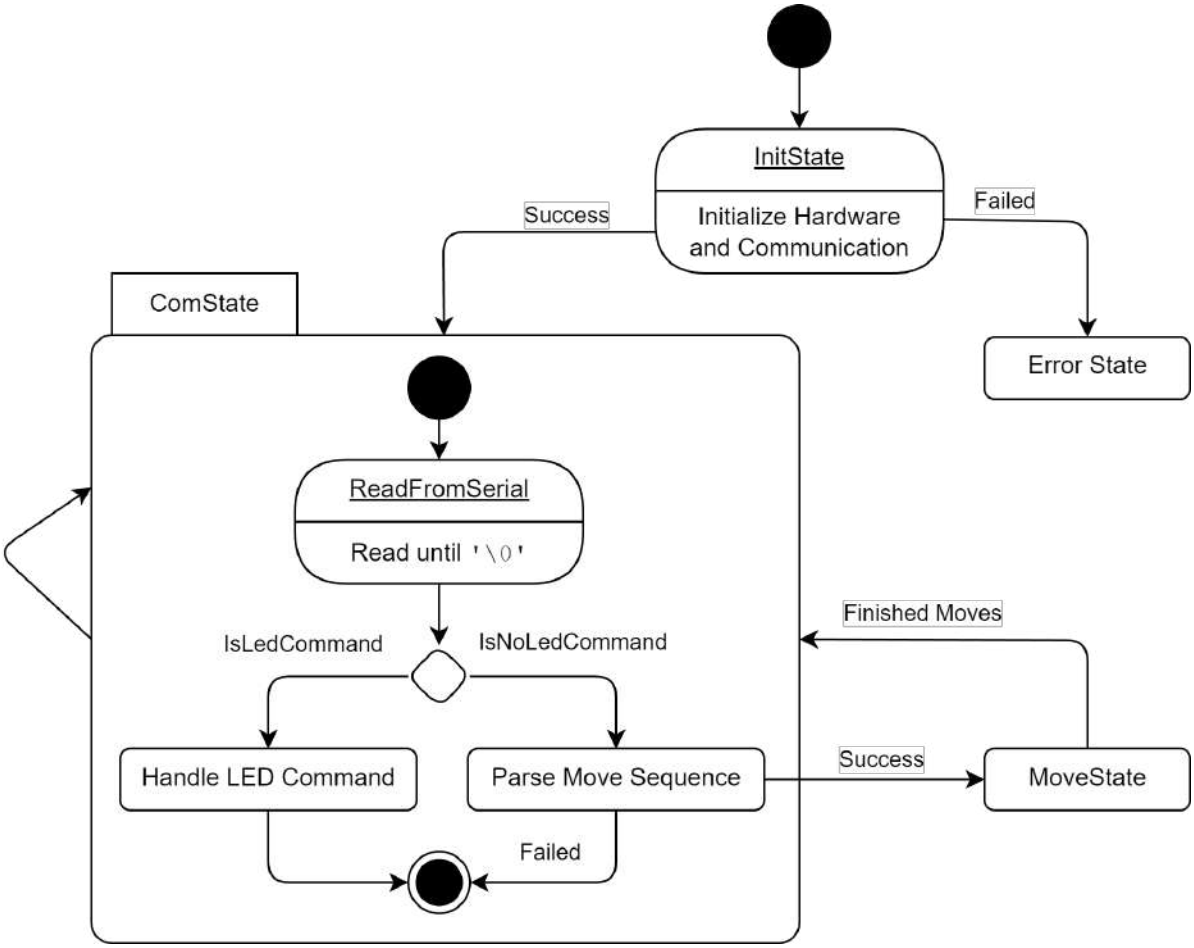


Abbildung 3.5: Darstellung des Entwurfs des Zustandsdiagramms der Mikrocontroller-Software

Zusätzlich sollen Zustandsänderungen beim Eintritt und Austritt eines Zustands über das serielle Interface kommuniziert werden. Außerdem sollen Debug-Informationen, wie beispielsweise ein Echo der geparsten Bewegungssequenz, ausgegeben werden. Dies liefert der Applikationssoftware zusätzliche Informationen und ermöglicht eine schnellere Fehlererkennung.

3.4 Algorithmus zur Lösung des Rubik's Cube

Um im späteren Verlauf den IDA*-Algorithmus zur Lösung des Rubik's Cubes implementieren zu können, wurde in der Entwurfsphase eine Klassenarchitektur für die in Kapitel 2.5 erwähnten Modelle des Cubes erarbeitet. Das Klassendiagramm in Abbildung 3.6 zeigt das Ergebnis dieses Entwurfs. Die abstrakte Klasse `RubiksCube` definiert grundlegende Funktionen, die jedes dieser Modelle unterstützen sollte, und spezifiziert entsprechende Enums für die Faces, Colors und Moves. Die beiden Modelle `RubiksCubeArrayModel` und `RubiksCubeIndexModel` leiten sich von dieser ab und implementieren die spezifischen Methoden zur Manipulation des Cubes entsprechend der gewählten Darstellungsweise.

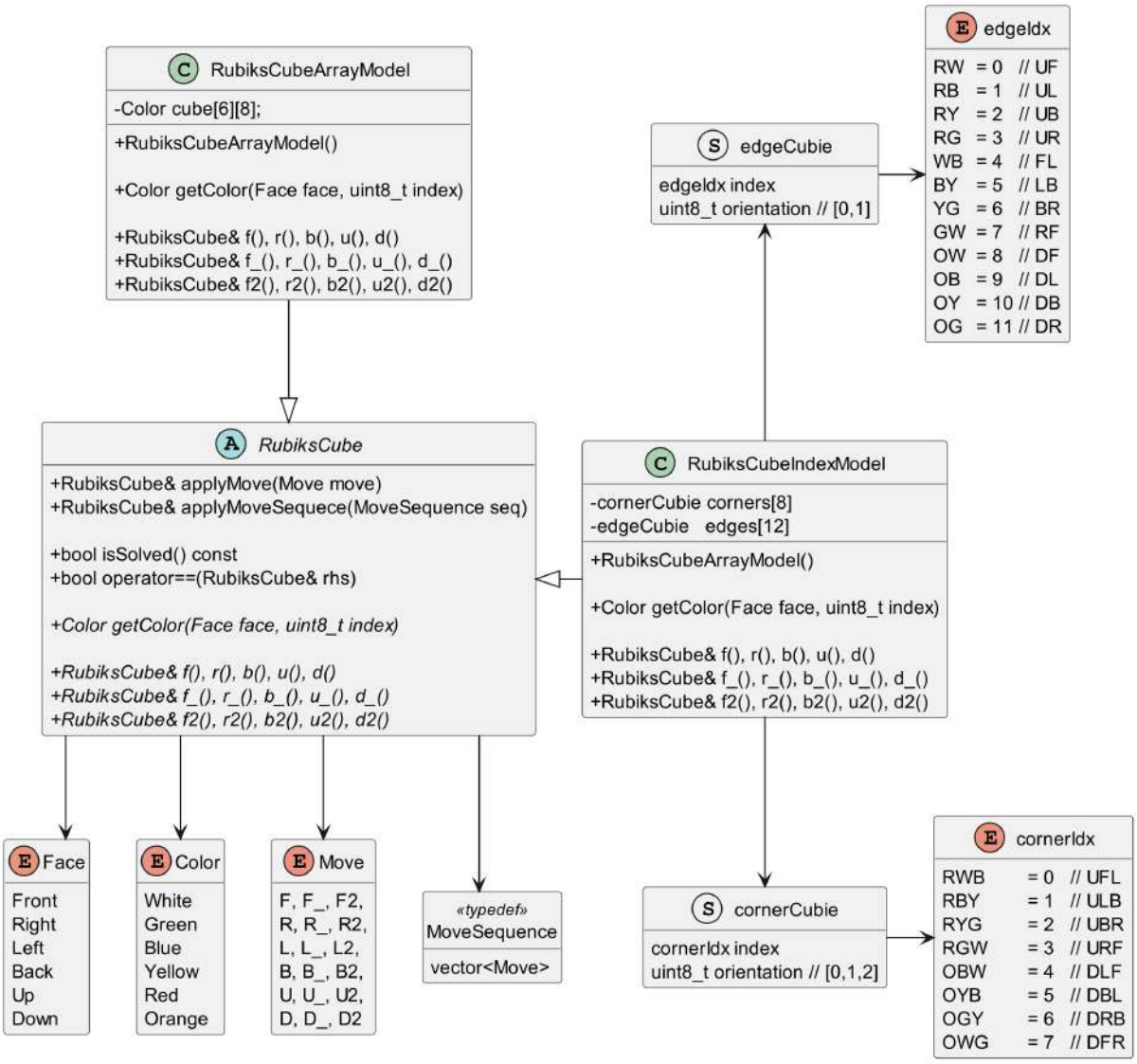


Abbildung 3.6: Klassendiagramm der Rubik's Cube Modelle

3.5 Applikationssoftware

Die Applikationssoftware verfügt, wie im Entwurf der Elektronik-Hardware in Kapitel 3.2 beschrieben, über eine serielle Verbindung zum Mikrocontroller, über die die LEDs angesteuert und Bewegungssequenzen gesendet werden können. Zusätzlich sind zwei Webcams angeschlossen, die das Auslesen des Cubes ermöglichen sollen. Das Ziel der Applikationssoftware ist es, ein möglichst simples und intuitives User Interface bereitzustellen, über das der Roboter gesteuert werden kann. Für den Entwurf wurde ein Wireframe der angestrebten Benutzeroberfläche erstellt, der in Abbildung 3.7 dargestellt ist. Die einzelnen Komponenten werden im Folgenden beschrieben:

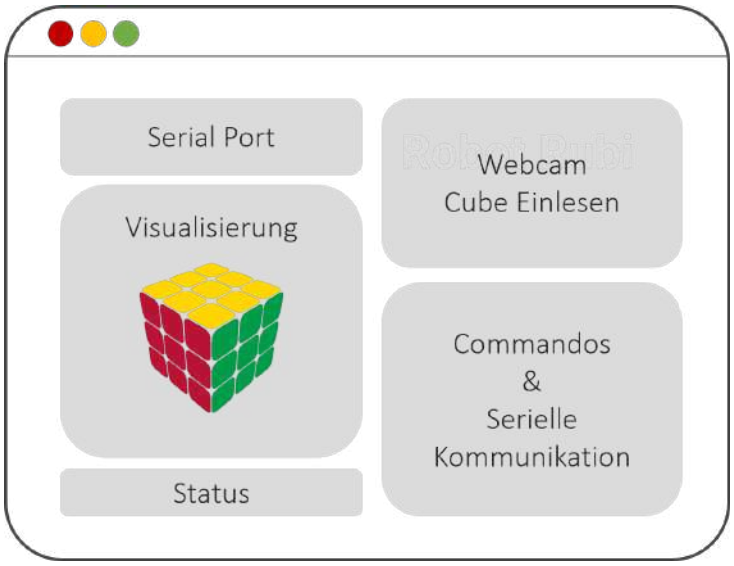


Abbildung 3.7: Konzeption der Applikationssoftware UI

- **Serial Port:** Auswahl des seriellen Ports zum Mikrocontroller und Verbinden des COM-Ports.
- **Visualisierung:** 3D-Darstellung des aktuellen Zustands des Cubes und Animation der Manipulationen des Cubes parallel zum Roboter.
- **Status:** Ausgabe sämtlicher Statusmeldungen, wie beispielsweise ungültige eingelesene Cubes oder das Fehlschlagen der seriellen Verbindung.
- **Webcam:** Videoanzeige der beiden Webcams sowie eine Benutzeroberfläche zum Einlesen des Cubes und manuelle Anpassung der eingelesenen Seitenfarben.
- **Commandos:** Vordefinierte Kommandos zum Ansteuern des Roboters sowie die Ausgabe der seriellen Kommunikation mit dem Mikrocontroller.

4 Implementierung

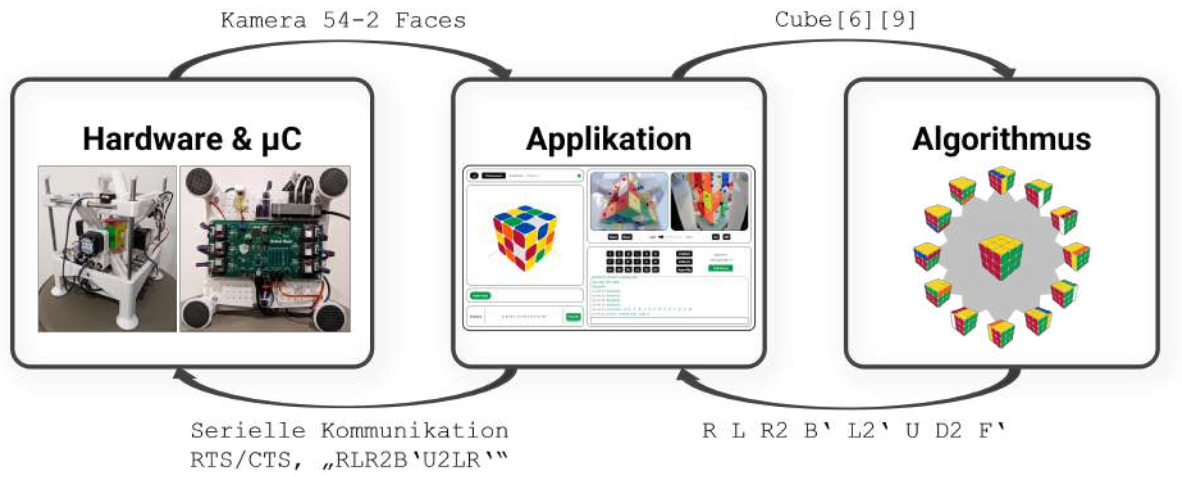


Abbildung 4.1: Finale Struktur der Implementierung

In diesem Kapitel wird die Implementierung des Projekts beschrieben. Die Abbildung 4.1 zeigt die verschiedenen Bestandteile des Projekts und veranschaulicht die Ergebnisse der Implementierung. Ziel dieses Kapitels ist es, einen Überblick über die durchgeführten Arbeiten zu geben und deren wesentlichen Komponenten zu erläutern.

- **Hardware & μC:**
  - **Mechanisch:** Fertigung der mechanischen Komponenten des Roboters mittels 3D-Druck und deren Montage
  - **Elektronisch:** Erstellung eines Prototyps für die Elektronik und anschließender Entwurf einer Leiterplatte (PCB)
  - **μC:** Einrichtung der Build-Tools für die Mikrocontroller-Software und deren Implementierung
- **Applikation:** Entwicklung der Anwendungssoftware mittels einer Electron-Applikation
- **Algorithmus:** Implementierung des IDA*-Lösungsalgorithmus

4.1 Mechanischer Aufbau

Nachdem, wie in Kapitel 3.1 behandelt, die Mechanik mittels CAD entworfen wurde und ein Großteil der Teile für die Fertigung mittels 3D-Druck konzipiert ist, besteht der erste Schritt nun in der Herstellung dieser Teile. Da ein geeigneter 3D-Drucker zur Verfügung stand und es selten vorkommt, dass bei komplexeren mechanischen Konstruktionen das erste Design aller Teile miteinander kompatibel ist (was auch hier der Fall war), wurde für die Fertigung der Teile das Konzept des Rapid Prototyping genutzt. Der Ablauf dieses Prozesses ist in Abbildung 4.2 dargestellt und wird im Folgenden erläutert:

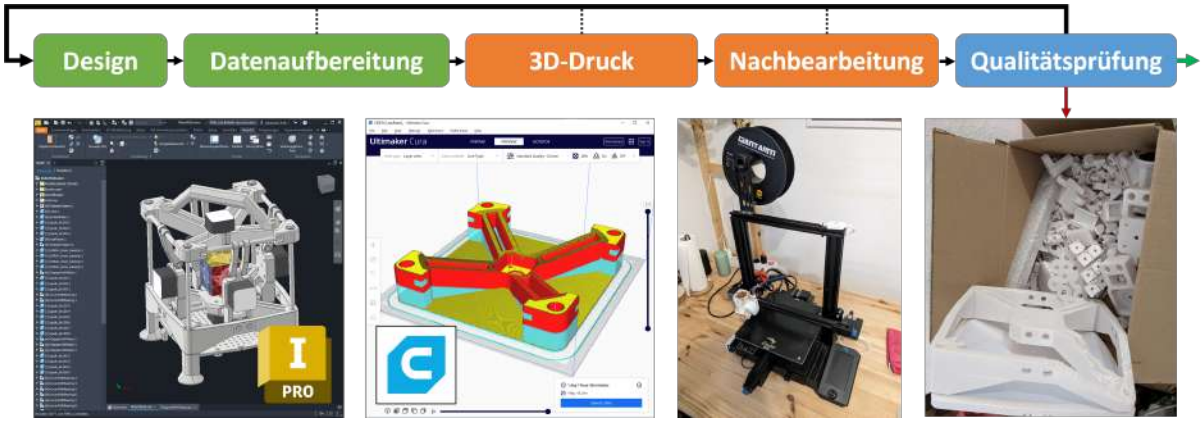


Abbildung 4.2: Darstellung des Rapid Prototyping Prozesses für die 3D gedruckten Mechanischen Teile des Roboters.

1. **Design:** Erstellung bzw. Anpassung der mechanischen Teile im CAD. Verwendete Software: Inventor von Autodesk.
2. **Datenaufbereitung:** Export der Modelldaten aus dem CAD im STL-Format und anschließendes Slicen. Beim Slicen wird ein 3D-Modell in horizontale Schichten unterteilt und die Druckparameter werden festgelegt. Das Ergebnis ist eine G-Code-Datei, die der 3D-Drucker für den schichtweisen Druck des Modells verwendet. Verwendete Software: Cura von Ultimaker.
3. **3D-Druck:** Die G-Code-Datei des Slicers wird dem 3D-Drucker übergeben. In diesem Fall wird das Fused Deposition Modeling (FDM) verwendet, bei dem ein Filamentstrang erhitzt und durch eine Düse extrudiert wird, um schichtweise das dreidimensionale Objekt zu erzeugen. Verwendeter Drucker: Ender3V2.
4. **Nachbearbeitung:** Entfernung von Stützstrukturen und Schleifen.
5. **Qualitätsprüfung:** Überprüfung der Maßhaltigkeit und Durchführung von Funktionstests.

Da alle Prozesselemente meist schnell durchlaufen werden können (kleinstes Teil: *Lever* ≈ 30 Minuten, größtes Teil: *BottomFrame* ≈ 1,5 Tage) und die Kosten dabei nicht sehr hoch sind (1kg Filament ≈ 15€, kleinstes Teil: *Lever* ≈ 6g, größtes Teil: *BottomFrame* ≈ 200g), ist es möglich, diesen Prozess mehrfach zu durchlaufen und somit Anpassungen einfach umzusetzen.

Dieser flexible Prozess hat sich besonders beim iterativen Entwurfsprozess der Greifer bewährt. Die dabei entstandenen Versionen sind in Abbildung 4.3 zusammengefasst. Die ersten Ansätze zielten darauf ab, die Mittelflächen des Würfels zu umschließen, da diese an den Kanten leicht abgerundet waren und die Greifelemente an den Ecken das Drehen der angrenzenden Seiten nicht behinderten. Allerdings konnte damit kein zuverlässiger Betrieb sichergestellt werden, da es dennoch zu Verkantungen beim Drehen der Seitenflächen kam oder die Greifer nicht starr genug waren, um eine ideale 90°-Drehung auszuführen. Ein weiterer Ansatz, der eine Klebefläche nutzte, um die Kraft lediglich durch Reibung zu übertragen, war ebenfalls erfolglos. Der finale Ansatz, die Mittelflächen des Würfels mit einer greifbaren Struktur (ähnlich einer Schraube) auszustatten und den Greifer dann mittels einer Feder vorzuspannen, um Toleranzen im System zu kompensieren, erwies sich schließlich als erfolgreich.

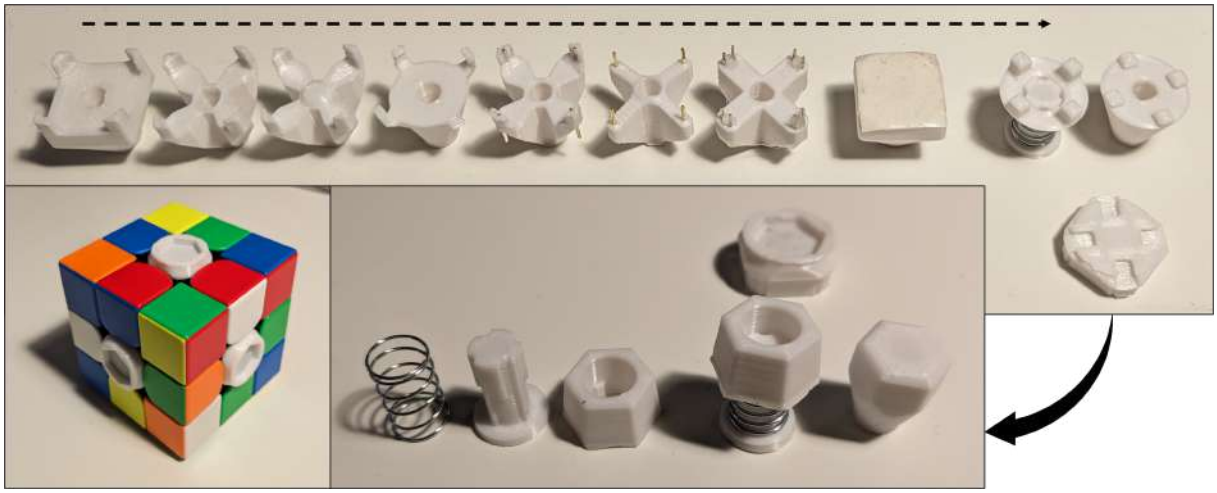


Abbildung 4.3: Iterative Anpassung der Roboter-Greifer

Die finale Version des Greifers ist in Abbildung 4.4 dargestellt. Der Greifer ist als Oktagon ausgeführt, da dieses sowohl in der x- als auch y-Achse symmetrisch ist und bei Drehungen von 90° bzw. 180° seine Form beibehält. Das entsprechende Gegenstück auf den Mittelflächen des Würfels wurde in den jeweiligen Farben der Seiten gedruckt, sodass die Farberkennung nicht beeinträchtigt wird.

Der finale Stand der montierten Mechanik ist in Abbildung 4.5 dargestellt.

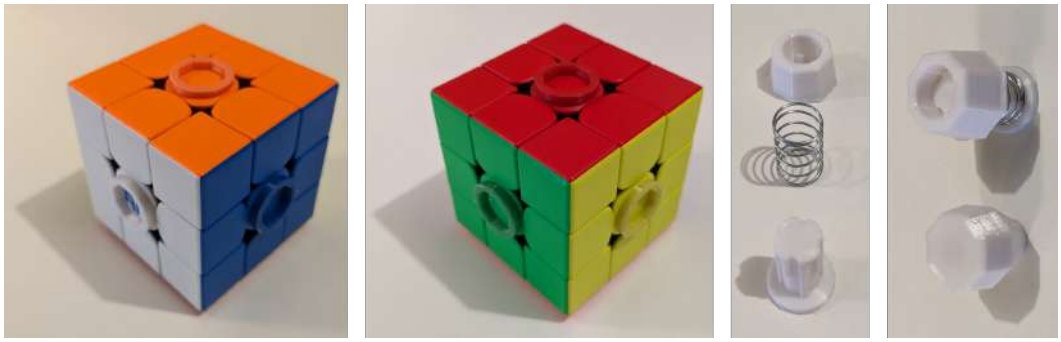


Abbildung 4.4: Endgültige Version der Roboter-Greifer nach der zehnten Iteration

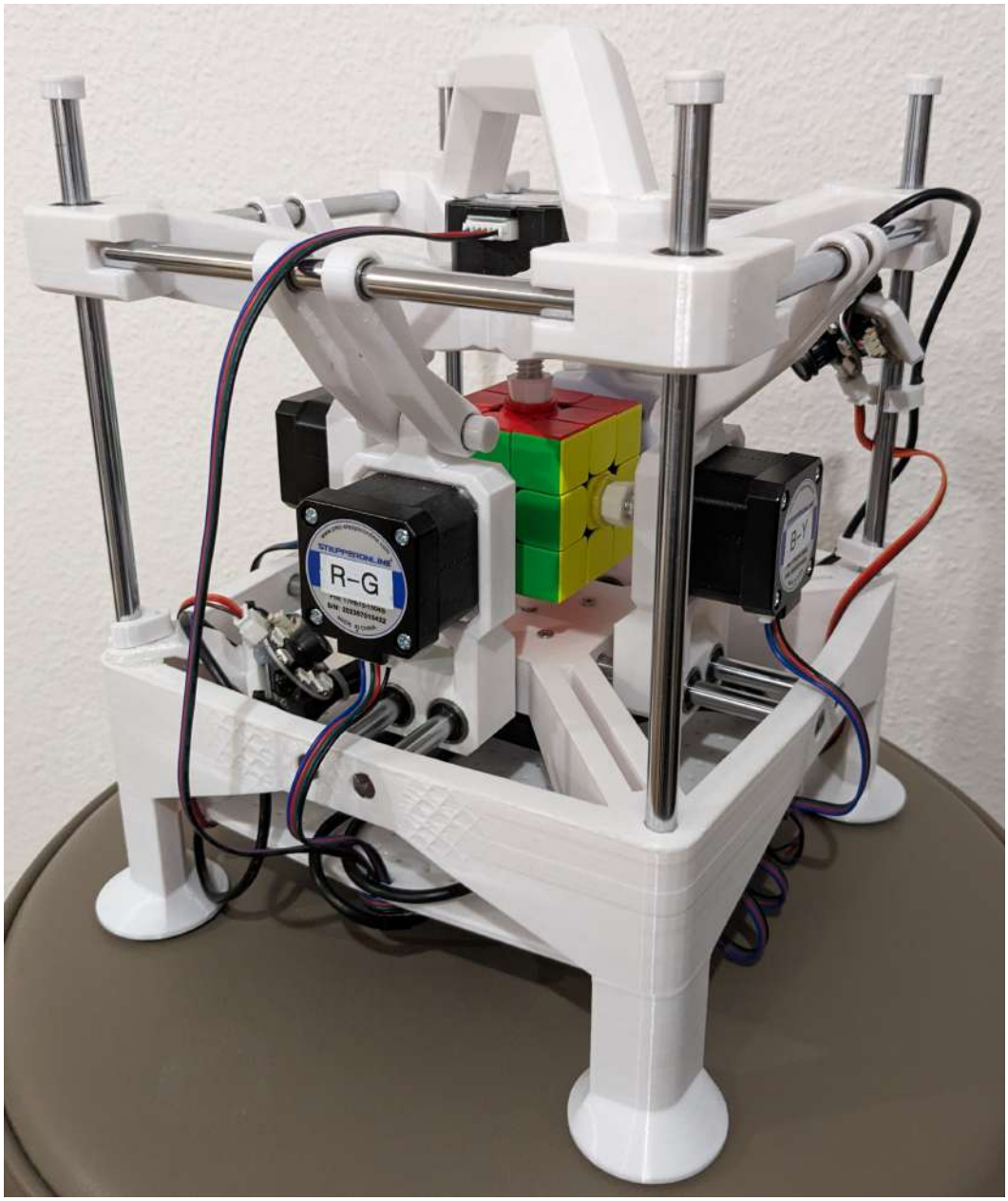


Abbildung 4.5: Finaler Stand der Mechanik

4.2 Elektronik - Hardware

Im Folgenden wird die Umsetzung der elektronischen Hardware erläutert, die aus einer Leiterplatte (PCB) besteht. Diese ermöglicht die Ansteuerung der sechs Schrittmotoren des Roboters und die Kommunikation mit der Applikation über eine serielle Schnittstelle. Hierbei wurde zuerst ein Prototyp entworfen, welcher die Funktionsweise der ausgewählten Komponenten validieren soll.

Prototyp

Vor der Erstellung einer eigenständigen Platine wurde ein Prototyp auf einer Steckplatine verkabelt (Abbildung 4.6), der dieselben externen Schnittstellen wie das geplante finale PCB besitzt. Dabei wurden die folgenden Komponenten verwendet:

- *BlackPill* Development Board mit dem μC STM32F401CCU6 (Arm Cortex-M4, 84 MHz, 256 Kbytes Flash, 64 Kbytes SRAM)
- 6x DRV8825 Stepper Motor Driver Module

Die Spannungsversorgung der Stepper Driver mit 24 V erfolgt durch ein DC-Labornetzteil. Das Development Board wird über USB-C mit Spannung versorgt und kann als COM-Port seriell kommunizieren. Das Flashen und Debuggen des μC erfolgt über die SWD-Schnittstelle mithilfe des Programmieradapters ST-Link V2.

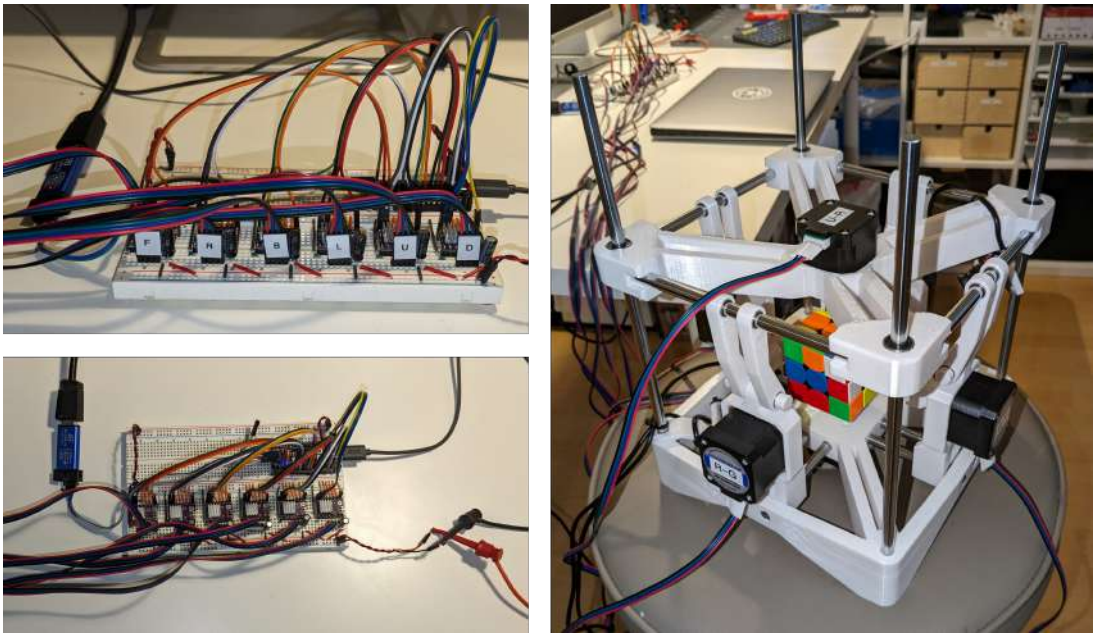


Abbildung 4.6: Darstellung des Prototypen-Setups für die Entwicklung der elektronischen Hardware.

PCB-Design

Nachdem der Prototyp auf dem Steckboard getestet und validiert wurde, ist im Folgenden die Schaltungslogik auf eine Platine gelayoutet worden. Als Software zur Entwicklung des PCB's wurde KiCad7 verwendet. Für die Produktion und Bestückung fiel die Entscheidung auf JLCPCB in China, da hier zeit- und ressourceneffizient PCB-Prototypen angefordert werden können. Zudem besitzt JLCPCB eine große Auswahl an SMD-Komponenten die zur direkten Bestückung vor Ort ausgewählt werden können.

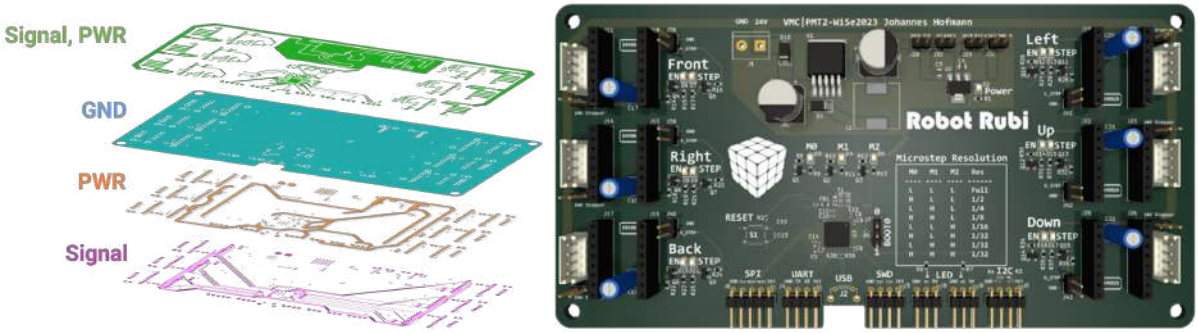


Abbildung 4.7: Darstellung der entworfenen 4-Layer-PCB mit den Kupferbahnen (links) und einer gerenderten Ansicht der Platine mithilfe der verwendeten Software KiCad7 (rechts).

Es wurde eine 4-Lagen-Leiterplatte (PCB) gewählt (Abbildung 4.7), wobei die Schichten wie folgt aufgeteilt sind:

1. **Signal, Power** Schnittstellen des μC, wie SPI, UART, USB, SWD, I2C und die LED-Ansteuerung. 24V Spannungsversorgung der Stepper-Motoren
2. **Ground**
3. **Power** 5V Spannungsversorgung für LEDs (extern und auf PCB) und 3.3V für μC
4. **Signal** Ansteuerung der Stepper Driver

Die verwendeten SMD-Bauteile wurden in Zusammenhang mit der Parts Library von JLCPCB ausgewählt, und die entsprechenden Teilenummern wurden in KiCad hinterlegt. Dadurch war es möglich, bei der Erstellung der Stückliste (BOM) bereits eindeutige Bauteile für die Bestückung festzulegen. Die verwendeten THT-Teile bestanden hauptsächlich aus Header-Pins und Elektrolytkondensatoren, die nachträglich manuell aufgelötet wurden.

PCB-Layout

Die Pin-Belegung wurde zu Beginn des Layoutens mithilfe der STM Cube IDE möglichst vorteilhaft gewählt (Abbildung 4.8), um Überschneidungen beim Verlegen der Traces zu minimieren. Beispielsweise wurden die Pins für USB und alle externen Kommunikationsschnittstellen auf einer Seite des Chips platziert, sodass die Traces direkt nach unten geführt werden können.

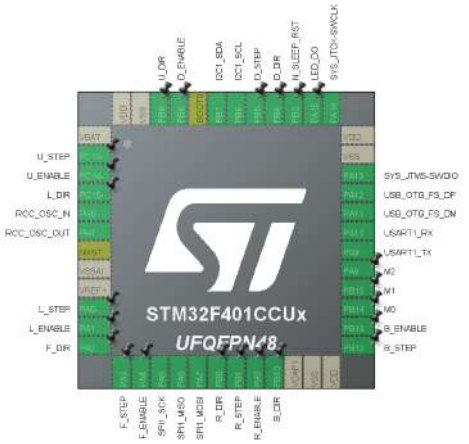


Abbildung 4.8: Pin-Belegung mittels der STM Cube IDE

Abbildung 4.10 zeigt das Schaltbild der erstellten Platine und lässt sich in folgende Bereiche unterteilen:

Power Supply

- Schraubklemme zum Anschluss von 24V DC für Stepper Driver
- Verpolungsschutz Diode
- Spannungsregler (24V - 5V - 3,3V)
  - LM2596-5.0 - 3A Step-Down Voltage (Switching) Regulator, Fixed Output Voltage: 5V, 7V < VIN < 40V, $\tau \approx 80\%$
  - AMS1117-3.3 - 1A Linear Voltage Regulator, Fixed Output Voltage: 3,3V

Die einzelnen Spannungslevel 24V, 5V und 3.3V sind durch Jumper Header Pins initial getrennt, sodass bei erster Inbetriebnahme die Spannungsregler auf fehlerfreie Funktion überprüft werden können.

μC

- Decoupling Kapazitäten für die VDD Eingänge des μC
- VDDA Filterschaltung für Analoge Spannungsversorgung
- HSE (High-Speed External Oscillator) 25MHz für CPU Clock
- Reset Button
- USB, SWD, UART und I2C Schnittstelle
- 2xLED Schnittstelle (Spannungsversorgung 5V und Data Out Pin) zum Ansteuern von WS2812 - Digital steuerbaren RGB-LEDs

Stepper Drivers

- 6x Female Header zur Aufnahme von DRV8825 Stepper Motor Driver Modulen
- Ansteuerung und Darstellen der Microstep Resolution mittel LEDs `M0`,`M1`,`M2`
- 6x100μF Kondensatoren für Kurzzeitige Lastspitzen am Treiber
- Ansteuerung und Darstellen des `nEnable` (invertiert) und `Step` Pins jedes Treiberbausteins mittels LEDs
- Ansteuerung der `nSleep` und `nReset` Pins der Treiber

Zum Ansteuern der LEDs werden NPN Transistoren (und PNP Transistoren für invertierte Logik) verwendet, um Signalleitungen des μC nicht mit zu hohen Strömen zu belasten.

Die Ansteuerung der `nSleep` und `nReset` Pins ist notwendig, da die GPIO Pins des μC zu Beginn ein logisches LOW aufweisen und somit den `nEnable` Pin aller Treiber aktivieren würden. Durch einschalten alle Stepper Treiber würde sich ein sehr hoher Laststrom ergeben.

PCB-Fertigung

Es wurden insgesamt 5 Platinen von JLCPCB (mit Versand) für $127.37 (116,70€) bestellt (3 unbestückt, 2 bestückt - siehe Abbildung 4.9). Bei der Bestückung gab es keine Mängel und auch die Inbetriebnahme der Platine gestaltete sich problemlos.

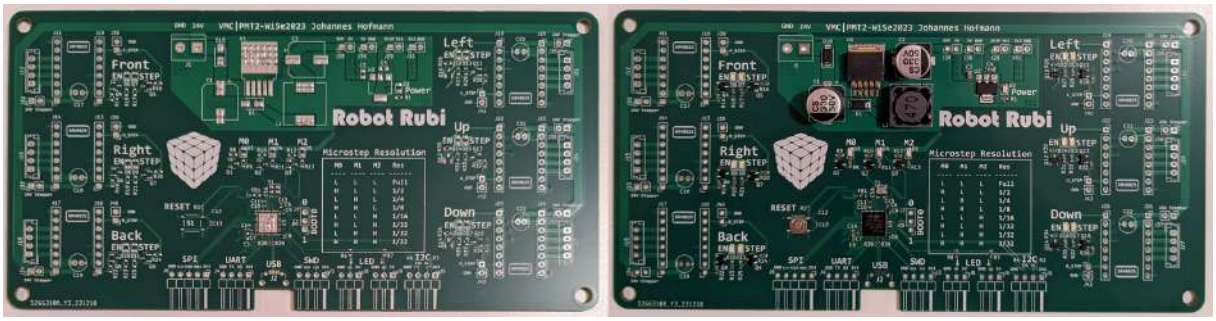


Abbildung 4.9: PCB Unbestückt (links) Bestückt (rechts)

PCB-Inbetriebnahme

Da die SMD-Bauteile bereits von JLCPCB bestückt wurden, mussten lediglich die fehlenden THT Komponenten, welche hauptsächlich aus Header Pins bestanden, auf die Platine gelötet werden. Bei der Inbetriebnahme der Platine wurde das in Tabelle 4.1 aufgeführte Testprotokoll abgearbeitet, um eventuelle Fehler systematisch ausfindig zu machen.

Tabelle 4.1: Testprotokoll zur Inbetriebnahme der Platine

Nr.	Beschreibung	Status
1	Auflöten der Schraubklemme zur 24V-Spannungsversorgung	✓
2	Testen, ob Kurzschluss zwischen +24V und GND besteht	✓
3	Spannungsversorgung (24V) anschließen und Funktionalität der Switching Regulator Schaltung (24V $\rightarrow$ 5V) überprüfen	✓
4	Reverse Polarity Protection Diode Funktionalität überprüfen durch Vertauschen der Spannungsversorgung Polung	✓
5	Auflöten und Verbinden des Jumpers (5VP-5V)	✓
6	Spannungsversorgung (24V) anschließen und Funktionalität des Linear Voltage Regulators (5V $\rightarrow$ 3.3V) überprüfen	✓
7	Auflöten und Verbinden des Jumpers (3.3VP-3.3V)	✓
8	Auflöten des Boot-Jumpers und `BOOT0` auf `GND` brücken (kennzeichnet, dass μC vom internen Flash booten soll)	✓
9	Auflöten der `SWD` Male Header Pins	✓
10	Spannungsversorgung (24V) anschließen und überprüfen, ob Kommunikation mit μC mittels der SWD Schnittstelle möglich ist (durch Auslesen des Flash-Speichers mittels des Programmieradapters ST-Link V2 und der Software ST-Link Utility)	✓
11	Auflöten der Pins und den Kondensator für den Front-Stepper Driver + einbringen eines Treiberbausteins	✓
12	Anschließen eines Stepper-Motors an den Front-Stepper Driver	✓
13	Aufspielen einer Test-Software, welche die Ansteuerung des Front-Stepper-Motors testet	✓
14	Überprüfen, ob LEDs am Board richtig angesteuert werden	✓
15	Auflöten aller anderen Stepper Driver Pins	✓
16	Auflöten der restlichen Header Pins und der USB-Schnittstelle	✓
17	Flashen der Finalen Software	✓
18	Überprüfe Ansteuerung aller Stepper Motoren	✓
19	1h Stress Test mit durchgängiger Ansteuerung der Motoren	✓
20	Finish	✓

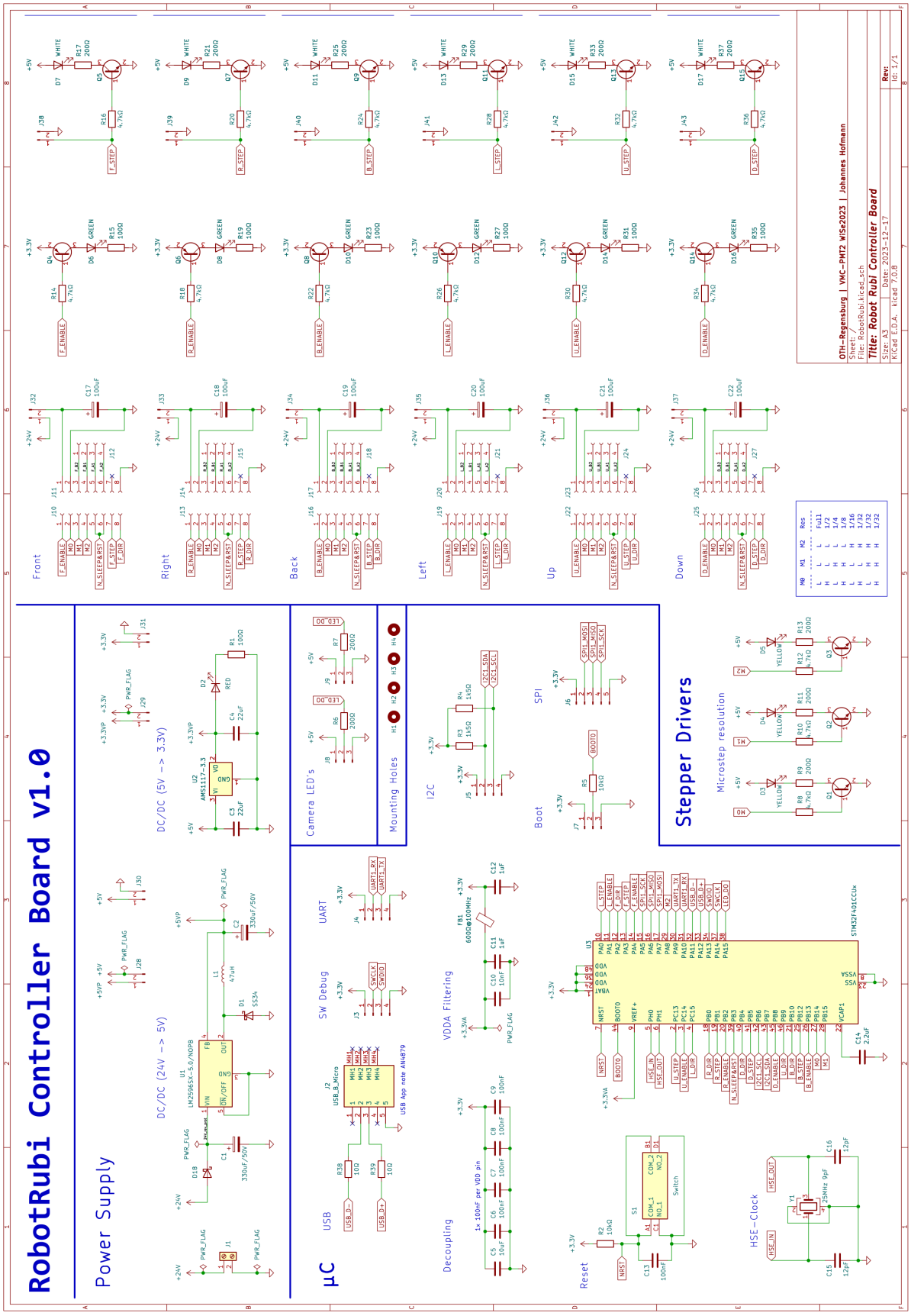


Abbildung 4.10: PCB Schematic

Elektronik - Montage

Nachdem die Platine fertig bestückt und auf Funktionalität überprüft wurde, ging es an die Verkabelung des Roboters. Hierfür wurden zuerst die Verbindungen zu den Stepper Motoren verlegt und auf der Lochplatte so platziert, dass sie möglichst nahe am jeweiligen Treiber der Platine positioniert sind. Danach wurde die Platine an der Unterseite der Lochplatte befestigt und die Stepper Motoren angeschlossen. Die USB-Verbindungen zum μC und den Kameras sind an einem USB-Hub, welcher fester Bestandteil des Roboters ist, angeschlossen. Hierdurch ist die einzige notwendige Schnittstelle von PC zu Roboter eine USB-C-Verbindung. Die Spannungsversorgung erfolgt über einen (5.5 x 2.1 mm) DC Steckverbinder, welcher mittels eines entsprechenden 24V (min. 24W) Netzteiles betrieben werden kann.

Die abschließende Integration der elektronischen Hardware in die Mechanik ist in Abbildung 4.11 dargestellt. Darüber hinaus ist die Integration der Kameras vergrößert dargestellt. Es handelt sich dabei um handelsübliche Webcams, deren Gehäuse entfernt wurde. Zur Verbesserung der Beleuchtung beim Auslesen des Würfels sind zudem Ring-LEDs an den Linsen angebracht worden.

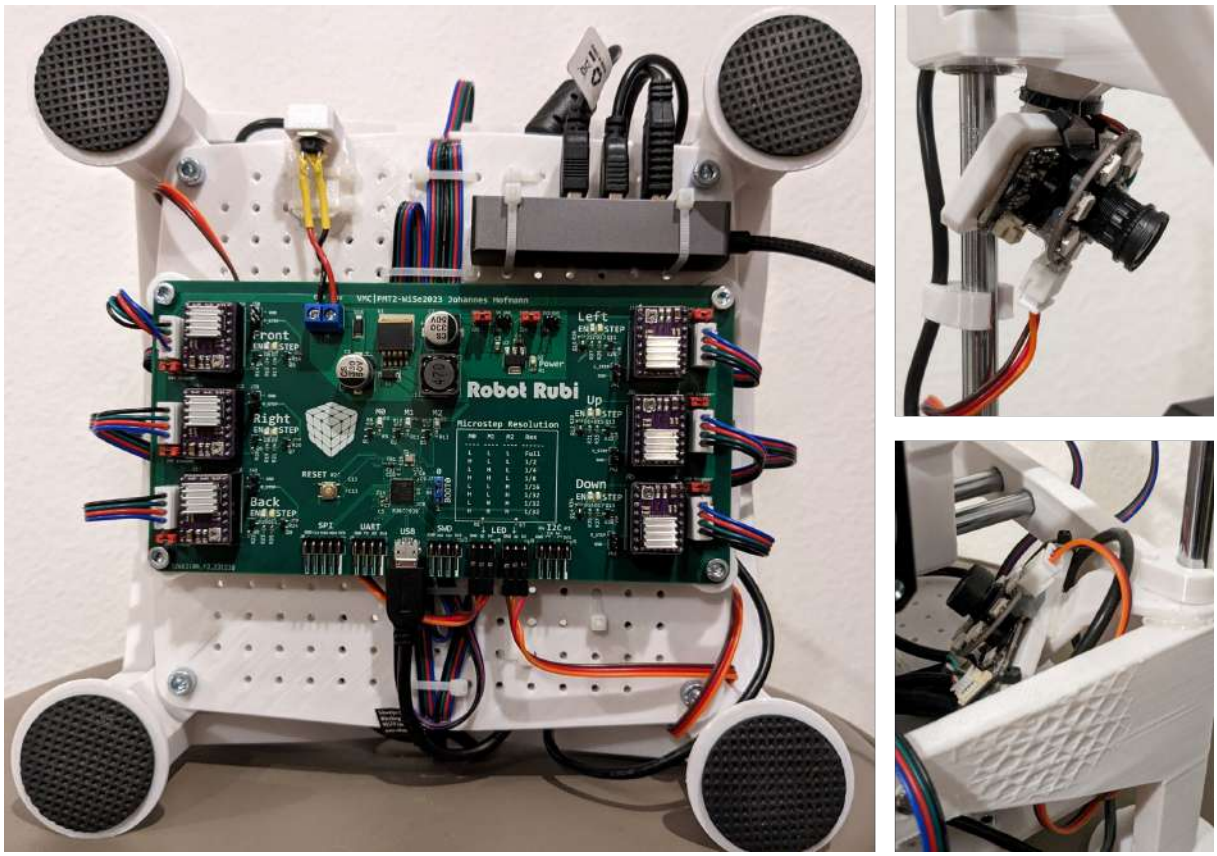


Abbildung 4.11: Finale Integration der elektronischen Hardware in die Mechanik

4.3 Elektronik - Software - μC

Die Softwarearchitektur auf dem μC zur Umsetzung der in 3.3 beschriebenen Zustandsmaschine (FSM) ist im Blockdiagramm der Abbildung 4.12 dargestellt. Für die Basis der Build-Tools und die unterste Schicht der Entwicklungsumgebung wurde PlatformIO als Plugin für Visual Studio Code verwendet. PlatformIO ist eine plattformübergreifende Entwicklungsumgebung für eingebettete Systeme, die eine Vielzahl von Mikrocontroller-Boards und Frameworks unterstützt. Sie bietet dabei eine einheitliche und erweiterbare Umgebung für die Entwicklung, das Testen und das Deployment von Firmware. [10]

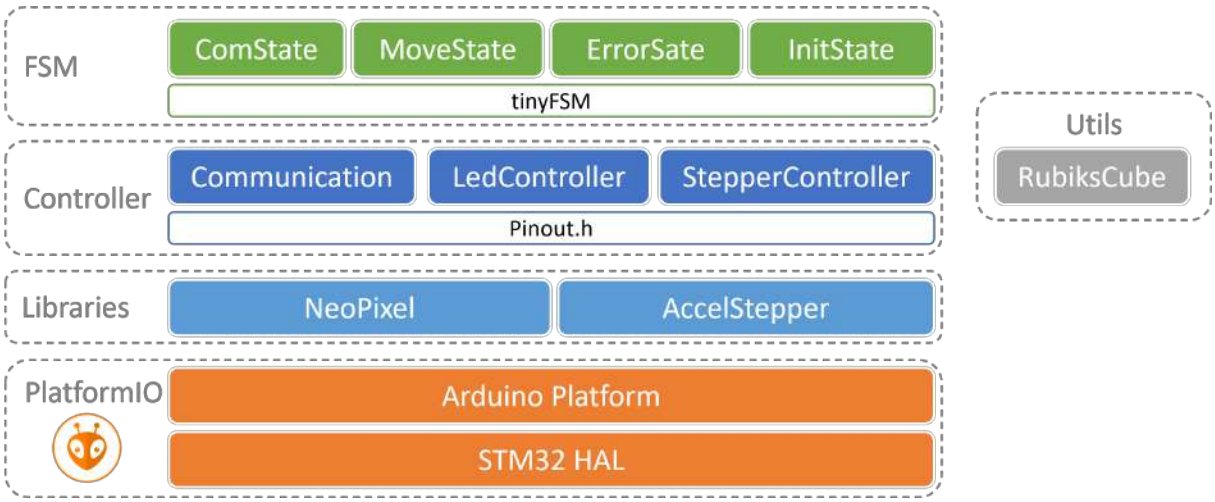


Abbildung 4.12: Architektur der Software für den μC

```
[env:genericSTM32F401CC]
platform = ststm32
board = genericSTM32F401CC
framework = arduino
board_build.f_cpu = 84000000L
build_flags =
    -D PIO_FRAMEWORK_ARDUINO_ENABLE_CDC
    -D USBCON
debug_tool = stlink
upload_protocol = stlink
lib_deps =
    waspinator/AccelStepper@^1.64
    adafruit/Adafruit NeoPixel@^1.12.0
```

Listing 4.1: PlatformIO Projekt Konfiguration

Die entsprechende Projektkonfiguration für PlatformIO ist im Listing 4.1 dargestellt. Als Framework wurde Arduino gewählt, da die Bibliotheken AccelStepper zur Ansteuerung der Schrittmotoren [11] und NeoPixel [12] zur Ansteuerung der LEDs angesehen

waren, welche auf dem Arduino-Framework basieren. Darüber hinaus ist dieses Framework sehr intuitiv und für den vorliegenden Anwendungsfall völlig ausreichend.

Für das Flashen und Debuggen der kompilierten Software wurde die SWD-Schnittstelle des μC mithilfe eines ST-Link-Programmers verwendet.

Der Zugriff auf die Hardware zur Kommunikation sowie zur Ansteuerung der LEDs und Schrittmotoren erfolgt mittels entsprechender Controller-Klassen, die über eine `Pinout.h` Datei von der eigentlichen Pinbelegung abstrahiert sind. Diese Klassen bauen auf den Bibliotheken AccelStepper und NeoPixel sowie dem Arduino-Framework auf.

Zur softwareseitigen Implementierung der FSM wurde die Library TinyFSM [13] verwendet, die eine Template C++-Klasse zur Umsetzung einer Zustandsmaschine bereitstellt. Diese Zustände nutzen wiederum die Controller-Klassen zur Ansteuerung der eigentlichen Hardware.

Das vollständige UML-Diagramm der implementierten Klassen ist in Abbildung 4.13 dargestellt. Jede der State-Klassen besitzt eine `entry()`, `exit()` und `periodic()` Methode. Die Methoden `entry()` und `exit()` werden jeweils beim Wechsel in und aus den jeweiligen Zustand aufgerufen, während die Methode `periodic()` in einer Endlosschleife ausgeführt wird.

Diese Klassen setzen die im Entwurf in Kapitel 3.3 beschriebene und in Abbildung 3.5 dargestellte Zustandsmaschine (FSM) um. Im `InitState` werden die Controller-Klassen initialisiert, falls dies fehlschlägt, wird in den `ErrorState` gewechselt. Im Normalbetrieb verweilt die FSM im `ComState`, in dem auf der seriellen Kommunikationsschnittstelle auf eingehende Kommandos gewartet wird. Falls ein Datenstrom eingeht, wird zuerst überprüft, ob es sich um ein LED-Kommando handelt, welches mittels der Methoden der `LedController`-Klasse abgearbeitet wird. Andernfalls wird davon ausgegangen, dass eine Move-Sequenz (Ansteuerung der Seiten des Rubik's Cubes) eingegangen ist, die anschließend mittels der `RubiksCube`-Klasse geparst wird. Falls das Parsen erfolgreich ist, werden die Moves in einer Queue abgespeichert und in den `MoveState` gewechselt. In diesem Zustand werden die eingereihten Moves sequentiell abgearbeitet und die Stepper-Motoren entsprechend angesteuert. Handelt es sich um gegenüberliegende Flächen des Cubes, werden die Motoransteuerungen parallel ausgeführt. Nach Beendigung der Ansteuerung wechselt die FSM wieder in den `ComState`.

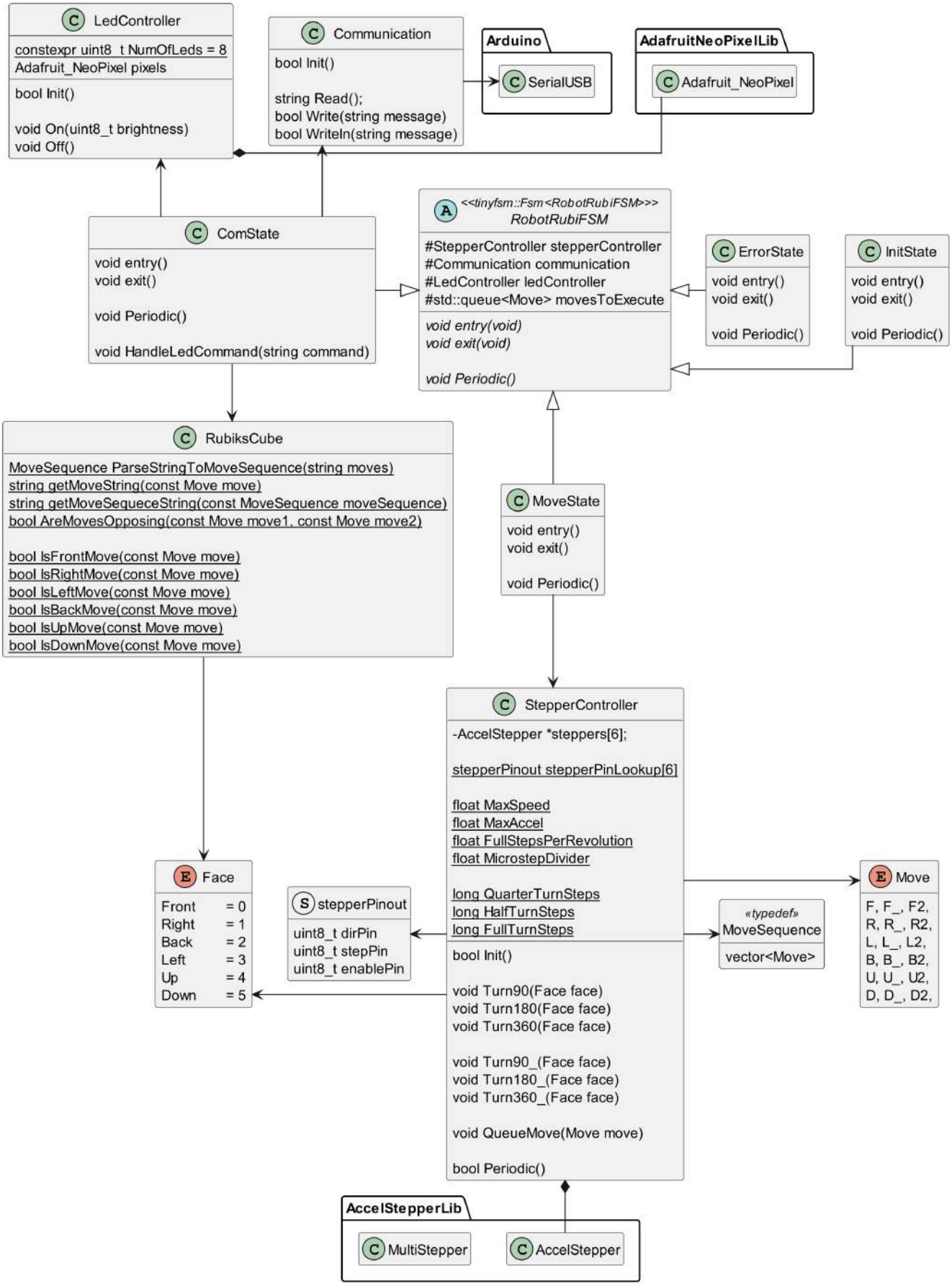


Abbildung 4.13: Klassendiagramm der Software für den μC

4.4 Algorithmus zur Lösung des Rubik's Cube

Im Folgenden werden die notwendigen Schritte zur Implementierung des Algorithmus zur Lösung des Rubik's Cubes beschrieben. Zu Beginn wurden die beiden Computermodelle des Rubik's Cubes, das Array-Modell und das Index-Modell, die bereits im Kapitel 3.4 erläutert wurden, umgesetzt. Das einfachere Array-Modell diente hauptsächlich als Gegenstelle zum Testen des komplexeren Index-Modells. Für die Erstellung des Index-Modells wurden entsprechende Unit-Tests mithilfe des Google Test Frameworks entwickelt. Als Programmiersprache kam C++ zum Einsatz, programmiert in der Entwicklungsumgebung Visual Studio.

Im nächsten Schritt war es notwendig, die für den IDA*-Algorithmus benötigte Heuristik zur effizienteren Durchsuchung des Rubik's Cube Graphen zu implementieren. Dies erfolgte mittels der von Richard Korf beschriebenen Pattern Databases [5], welche im Folgenden genauer beschrieben werden.

Heuristik - Pattern Databases

Wie bereits in den Grundlagen zum IDA*-Algorithmus in Kapitel 2.6 beschrieben, benötigt der A*-Algorithmus eine Heuristik, die den Weg zum Zielknoten nicht überschätzt. Diese Heuristik ermöglicht es, die Gewichte des Graphen entsprechend anzupassen und die Suche nach dem Zielknoten im Graphen effizienter zu gestalten.

Eine ideale Heuristik für den Rubik's Cube wäre eine Datenbank, die für jede Permutation des Cubes die exakte Anzahl der Züge angibt, die zur optimalen Lösung benötigt werden. Da es 43 Quintillionen mögliche Permutationen des Rubik's Cube gibt und maximal 20 Züge erforderlich sind, könnte für jede Permutation an einem eindeutigen Index in einer Datenbank ein 8-Bit-Wert stehen, welcher die maximal notwendigen Züge für die vorliegende Permutation beinhaltet. Diese Datenbank hätte dann eine Größe von $43 \cdot 10^{18}$ Bit $= 5.375 \cdot 10^{15}$ Terabyte.

Da die Größe dieser Datenbank praktisch unmöglich umzusetzen ist und deren Erstellung enorme Rechenressourcen und Zeit in Anspruch nehmen würde, wird beim Korf-Algorithmus auf eine nicht ideale, jedoch praktisch umsetzbare Heuristik gesetzt.

Für diese Heuristik werden nach Korf [5] Pattern Databases verwendet. Dabei wird nur eine Untermenge des Rubik's Cube betrachtet, und für jeden möglichen Zustand dieser Untermenge werden die minimal notwendigen Züge zum Lösen des Cubes in einer Datenbank gespeichert. Diese Datenbank dient dann im eigentlichen Lösungsalgorithmus als Heuristik.

Richard Korf hat hierfür drei Datenbanken vorgeschlagen [5]:

- **8-Corner-Cubies:** In dieser Datenbank werden nur die acht Corner-Cubies betrachtet. Diese können jeweils eine der 8 Positionen im Cube einnehmen, was insgesamt $8! = 40.320$ Permutationen ergibt. Betrachtet man zusätzlich die Orientierung der Corner-Cubies, können diese sich jeweils in einer von drei Orientierungen befinden (3^8). Da jedoch die Orientierung von 7 Corner-Cubies die des achten bestimmt [5], ergeben sich beim Betrachten der 8-Corner-Cubies insgesamt $8! \cdot 3^7 = 88.179.840$ mögliche Permutationen. Die maximale Länge der Lösungssequenz für dieses Subset beträgt 11 Züge, was jeweils 4 Bit Speicher benötigt und somit eine Datenbank der Größe von 42 MB ergibt.
- **6-Edge-Cubies-1:** Als nächste Datenbank werden 6 Edge-Cubies in ihrer Positionierung betrachtet. 6 Edge-Cubies können jeweils 12 Positionen einnehmen ($12\text{P}6 = 665.280$) und sich in einer von zwei Orientierungen befinden (2^6). Somit ergeben sich beim Betrachten der 6-Edge-Cubies insgesamt $12\text{P}6 \cdot 2^6 = 42.577.920$ mögliche Permutationen. Die maximale Länge der Lösungssequenz für dieses Subset beträgt 10 Züge, was jeweils 4 Bit benötigt und somit eine Datenbank der Größe von 20 MB ergibt.
- **6-Edge-Cubies-2:** Diese Datenbank ist identisch mit der 6-Edge-Cubies-1, jedoch werden nun die bisher nicht berücksichtigten Edge-Cubies betrachtet.

Die von Richard Korf vorgeschlagenen Datenbanken haben eine Gesamtgröße von $42\text{MB} + 2 \cdot 20\text{MB} = 82\text{MB}$. Da sich die Rechenleistung und Speichergröße seit der Veröffentlichung des Papers von Richard Korf [5] im Jahr 1997 verbessert haben, und größere Pattern-Datenbanken die Heuristik und somit die Effizienz des Algorithmus verbessern, wurden, wie in einem Artikel von Ben Botto [14] beschrieben, die Datenbanken der Edges von 6 auf 7 erhöht und zusätzlich eine Datenbank hinzugefügt, die lediglich die Permutationen der 12-Edges behandelt. Die verwendeten Datenbanken sind in Tabelle 4.3 dargestellt und haben nun eine Gesamtgröße von 758MB.

Tabelle 4.14 zeigt zudem, wie sich diese Datenbanken zusammensetzen, indem auf der x-Achse die Anzahl der benötigten Schritte zum Lösen eines Cubes angegeben ist und jeweils dazu auf der y-Achse die entsprechende Anzahl der Permutationen.

Ein weiteres Vergrößern der Datenbanken würde sehr schnell zu einem drastisch höheren Speicherbedarf führen. Würde man beispielsweise die 7-Edge-Datenbank auf acht erhöhen, wäre der benötigte Speicherbedarf $12P8 \cdot 2^8 = 5.109.350.400 \Rightarrow 2,55\text{GB}$.

Tabelle 4.3: Korf Rubik's Cube Pattern Databases

Database:	**8 Corner DB**	**7 Edges DB1**	**7 Edges DB2**	**12 Edges CB**
File:	`Corner.pdb`	`Edges1.pdb`	`Edges2.pdb`	`Edges3.pdb`
Cube:				
Consider:	Position & Orientation	Position & Orientation	Position & Orientation	Position
Size:	$8! \cdot 3^7$ $= 88.179.840$	$12P7 \cdot 2^7 = 510.935.040$		$12!$ $= 479.001.600$
max 11 Moves $\Rightarrow$ 4Bit Memory for each Permutation				
Memory:	**`42 MB`**	**`2 x 244MB`**		**`228 MB`**

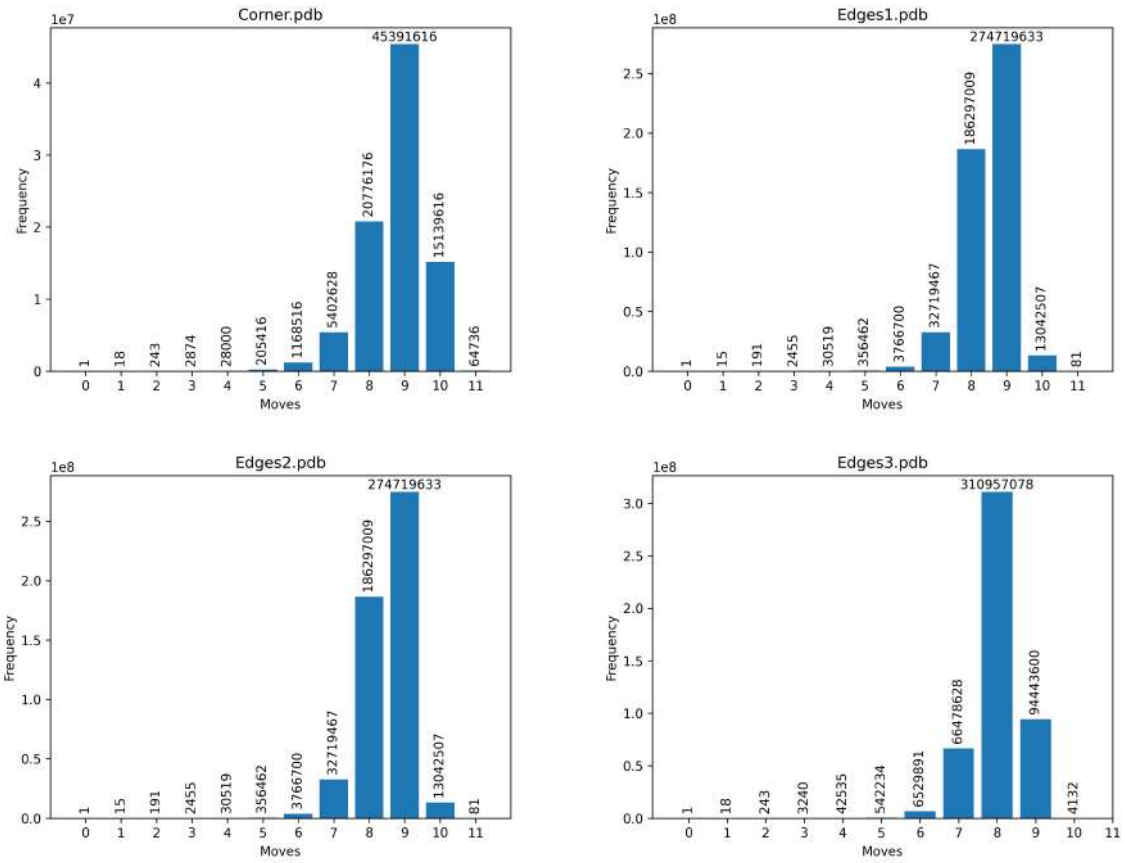


Abbildung 4.14: Analyse der Zusammensetzung der Datenbanken aus Tabelle 4.3

Pattern Database - Indexierung

Um die Pattern-Datenbank zu erstellen und darauf zuzugreifen, ist es notwendig, jeder Permutation einen eindeutigen Index zuzuweisen. Da die Implementierung der Indexierung keineswegs trivial ist, sei auf den Artikel *„Sequentially Indexing Permutations: A Linear Algorithm for Computing Lexicographic Rank “* von Ben Botto [15] verwiesen. Da diese Indexierung im Algorithmus für jede Permutation und somit sehr häufig ausgeführt wird, wurde eine bereits implementierte und optimierte templatisierte C++ Klasse (`PermutationIndexer` [16]) verwendet.

Das Konzept wird im Folgenden beschrieben:

```
Example Permutation Indexing for 8-Corner-DB

Position-                          Orientation-
Permutation          Rank          Permutation        Rank
(0 1 2 3 4 5 6 7)    0             (0 0 0 0 0 0 0)    0
(0 1 2 3 4 5 7 6)    1             (0 0 0 0 0 0 1)    1
(0 1 2 3 4 6 5 7)    2             (0 0 0 0 0 0 2)    2
(0 1 2 3 4 6 7 5)    3             (0 0 0 0 0 1 0)    3
...                    ...
(4 0 1 2 3 5 6 7)    20160         (1 0 0 0 0 0 0)    729
...                    ...
(7 6 5 4 3 2 1 0)    40319         (2 2 2 2 2 2 2)    2186
```

Listing 4.2: Beispiel für das Indexing der 8-Corner-DB

Die Permutationen der 8-Corner-DB können wie im Listing 4.2 dargestellt werden. Die Positions-Permutation wird dabei als ein Array mit 8 Elementen betrachtet, das in jeder möglichen Anordnung durch einen eindeutigen Permutations-Index (Rank) beschrieben werden kann. Gleichzeitig wird die Orientierungs-Permutation als ein Array mit 7 Elementen aufgefasst (die 8. Orientierung ergibt sich automatisch), wobei jedes Element den Zustand 0, 1 oder 2 annehmen kann. Auch hier wird jeder Anordnung ein eindeutiger Permutations-Index (Rank) zugeordnet. Dadurch kann jeder Corner-Permutation ein eindeutiger Index zugewiesen werden:

$$\text{Index}_{\text{Corners}} = \text{PositionRank}_{\text{Corners}} \cdot 3^7 + \text{OrientationRank}_{\text{Corners}}$$

Hier zeigt sich die Stärke des Index-Modells des Rubik's Cubes, da es dieselben Arrays für die Positions- und Orientierungs-Permutation verwendet und somit direkt zur Indexbildung herangezogen werden kann.

Pattern Database - Erstellung

Da es nun möglich ist, jeder Permutation eindeutig einen Index zuzuordnen, kann mit der Erstellung der Datenbanken begonnen werden. Zunächst werden hierfür alle Einträge der Pattern-Datenbank mit `0xFF` gefüllt. Danach wird mittels eines Iterative Deepening Depth-First Search (IDDFS) vom gelösten Zustand aus für jede Permutation die Länge bis zum gelösten Zustand in der Datenbank gespeichert. Hierbei wird für jede Position des Würfels im Graphen der Index berechnet und in der Datenbank überprüft, ob bereits ein Eintrag vorhanden ist (`entry != 0xFF`). Falls nicht, wird die aktuelle Tiefe eingetragen. Zusätzlich wird bei jedem Eintrag eine Zählervariable erhöht, um zu überprüfen, ob alle Permutationen einen Eintrag erhalten haben. Durch den Einsatz von IDDFS wird sichergestellt, dass immer der kürzeste Weg zur jeweiligen Permutation gefunden wird, indem in der Datenbank stets auf die geringste Tiefe aktualisiert wird.

```
Debug output for Corner.pdb

Building Pattern Database for Corners, Database Size: 44089920 Bytes

Starting Time:                              2023-09-03  12:46:29
Bound:  1, Indexed       19/88179840 =   0.00%, Time: +00:00:00:001
Bound:  2, Indexed      262/88179840 =   0.00%, Time: +00:00:00:001
Bound:  3, Indexed     3136/88179840 =   0.00%, Time: +00:00:00:002
Bound:  4, Indexed    31136/88179840 =   0.04%, Time: +00:00:00:009
Bound:  5, Indexed   236552/88179840 =   0.27%, Time: +00:00:00:086
Bound:  6, Indexed  1405068/88179840 =   1.59%, Time: +00:00:01:120
Bound:  7, Indexed  6807696/88179840 =   7.72%, Time: +00:00:09:473
Bound:  8, Indexed 27583872/88179840 =  31.28%, Time: +00:01:28:309
Bound:  9, Indexed 72975488/88179840 =  82.76%, Time: +00:13:35:677
Bound: 10, Indexed 88115104/88179840 =  99.93%, Time: +01:44:21:448
Bound: 11, Indexed 88179840/88179840 = 100.00%, Time: +03:32:21:465
End Time:                                   2023-09-03  16:18:51
```

Listing 4.3: Debug Output für die Erstellung der 8-Corner-DB

Im Listing 4.3 ist der Debug-Output für die Erstellung der 8-Corner-DB aufgeführt, wobei die Zeiten und die Anzahl der indizierten Permutationen aufgelistet sind. Die Erstellung dieser Datenbank hat circa 3,5 Stunden in Anspruch genommen und endete bei Tiefe 11, wobei 99,93 % der Datenbank bereits nach 1,75 Stunden bei Tiefe 10 fertiggestellt waren. Dies verdeutlicht, wie der Suchaufwand mit zunehmender Tiefe im Graphen deutlich anspruchsvoller wird und warum es nicht möglich ist, den gesamten Würfel mit einer maximalen Tiefe von 20 zu durchsuchen.

Abbildung 4.15 zeigt das UML-Diagramm der erstellten Klassen für die `PatternDatabase` und die Erweiterung der `RubiksCubeIndexModel`-Klasse um die `Permuta-`

`tionIndexer`-Klasse sowie die ergänzten Methoden, die zum Indexieren der Pattern-Datenbank benötigt werden.

Die `PatternDatabase` ermöglicht es, mittels eines entsprechenden `DatabaseType`-Enums den Typ der Datenbank zu bestimmen. Sie kann sowohl zum Erstellen der Datenbank als auch als Schnittstelle zum Einlesen einer bereits erstellten Datenbank verwendet werden. Die Klasse erlaubt es dem `Solver`, indexbasiert auf die 4-Bit-Einträge in der Datenbank zuzugreifen.

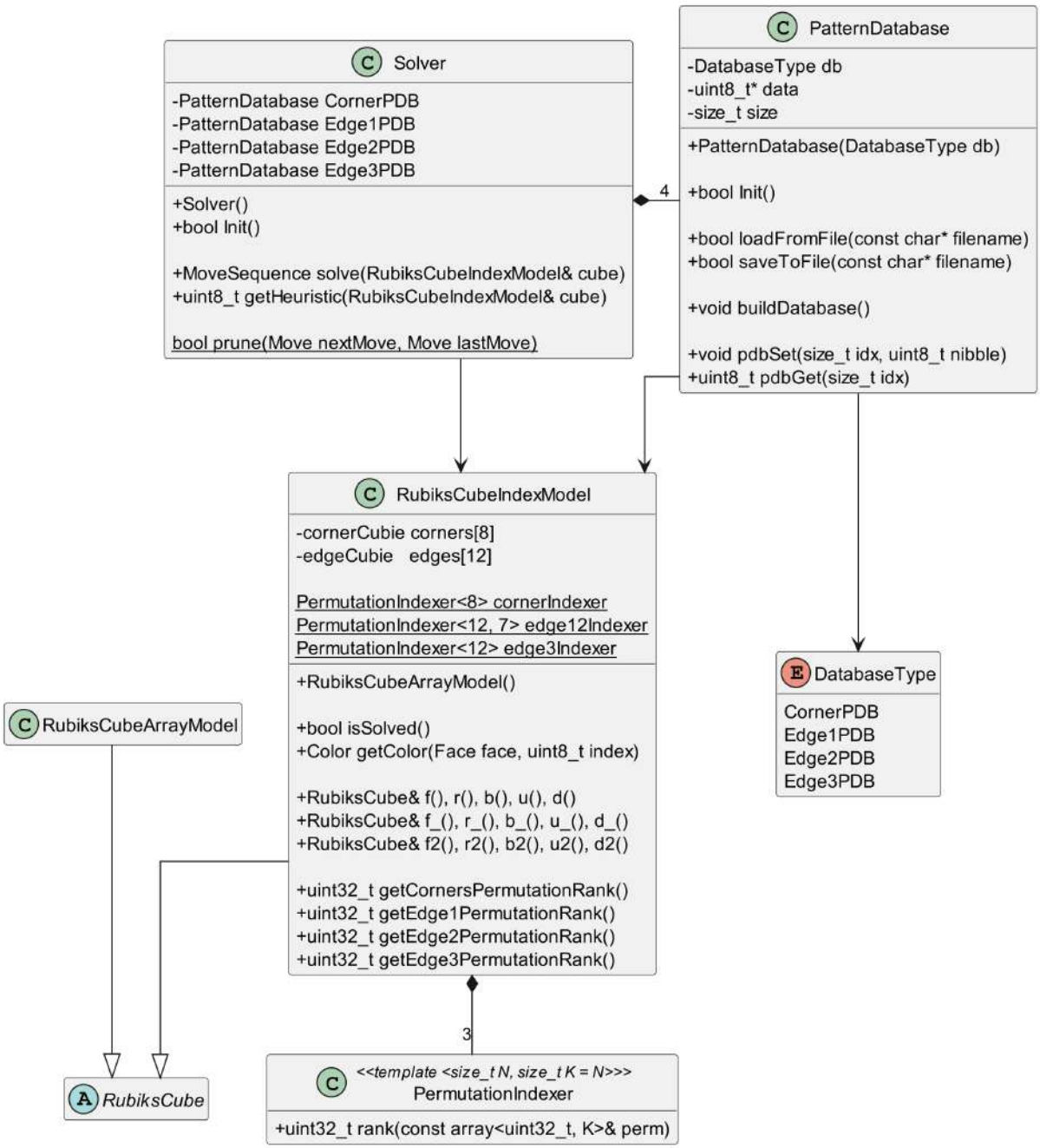


Abbildung 4.15: UML Diagramm der Klassen für den Korf-IDA* Lösungsalgorithmus

IDA* mit Pattern Databases

Durch die Verwendung der erstellten Pattern Databases ist es nun möglich, die Heuristik für den IDA* zu implementieren. Diese Heuristik dient als Schätzung der benötigten Züge bis zum gelösten Zustand und wird wie folgt umgesetzt:

$$\text{Heuristik} = \max[\text{pdb}_1(\text{cube}), \text{pdb}_2(\text{cube}), \text{pdb}_3(\text{cube}), \text{pdb}_4(\text{cube})]$$

Der Lösungsalgorithmus der Klasse `Solver` aus dem UML-Diagramm in Abbildung 4.15 für einen nicht-rekursiven IDA* kann wie in Listing 4.4 implementiert werden. Diese Implementierung orientiert sich an der von Ben Botto [14] und ist an die vorhandene Klassenstruktur angepasst. Im Folgenden wird der Ablauf und Aufbau des Algorithmus beschrieben:

1. **Initialisierung** *(Zeile 1-12)*
    - Ein Stack (`nodeStack`) wird genutzt, um die Knoten (`Node`) zu speichern, die noch untersucht werden müssen. Jeder Knoten speichert den jeweiligen Rubik's Cube State (`cube`), den `move` der vom Vorgänger-Node genutzt wurde um an den vorliegenden zu kommen und die mittels der Heuristik geschätzten Tiefe, die zum Lösen des aktuellen Nodes notwendig ist.
    - Ein boolescher Wert `solved` wird verwendet, um anzuzeigen, ob der Würfel gelöst ist.
    - `bound` repräsentiert die aktuelle Schranke für die Iteration, während nextBound die Schranke für die nächste Iteration speichert. nextBound wird initial mit dem heuristischen Wert des Anfangszustands des Würfels gesetzt.
2. **Iterativer Ablauf** *Zeile 14-45*
    - Wenn der `nodeStack` leer ist, bedeutet dies, dass eine neue Iteration begonnen wird:
        - Der Anfangszustand des Würfels wird auf den Stack gelegt, wobei kein Zug (`Move::Undefined`) und eine Tiefe von 0 gesetzt werden.
        - Die Schranke `bound` wird auf den Wert von `nextBound` gesetzt, und `nextBound` wird auf 255 (`0xFF`) gesetzt.
    - Der oberste Knoten (`curNode`) wird vom Stack genommen.
    - Falls Tiefe des aktuellen Knotens der aktuellen Schranke (`bound`) entspricht:

- Wird überprüft, ob der Würfel in diesem Zustand gelöst ist. Falls ja, wird `solved` auf `true` gesetzt.
- Andernfalls werden alle nicht redundanten (`prune`) möglichen Züge des Würfels durchlaufen:
  * Eine Kopie des aktuellen Würfels wird erstellt und der Zug auf diese Kopie angewendet.
  * Die geschätzte Anzahl der verbleibenden Züge (`estSuccMoves`) wird als Summe aus der aktuellen Tiefe, 1 (für den aktuellen Zug), und dem heuristischen Wert des neuen Zustands berechnet.
  * Wenn `estSuccMoves` kleiner oder gleich der aktuellen Schranke (`bound`) ist, wird der neue Zustand auf den Stack gelegt.
  * Wenn `estSuccMoves` kleiner als `nextBound` ist, wird `nextBound` aktualisiert.

3. **Beendigung**
   - Der Algorithmus beendet die Schleife, sobald der Würfel gelöst ist.

Im folgenden werden die **Schlüsselkonzepte** des Algorithmus zusammengefasst:

- **Heuristik:** Die Heuristikfunktion schätzt die verbleibenden Züge, um den Würfel zu lösen.
- **Iterative Deepening:** Der Algorithmus arbeitet iterativ mit zunehmenden Schranken (`bound`), um den Zustand zu durchsuchen.
- **Stack-basierte Tiefensuche:** Anstelle einer rekursiven Implementierung wird ein Stack verwendet, um die Knoten zu speichern, was den Algorithmus nicht-rekursiv macht.
- **Pruning:** Redundante Züge, die lediglich den vorherigen rückgängig machen werden nicht berücksichtigt. Somit verbessert sich der Branching-Faktor im Graphen von 18 (Alle Züge) auf 13 (Pruned).
- **Bound Adjustment:** `nextBound` wird verwendet, um die Schranke für die nächste Iteration anzupassen, basierend auf den minimalen geschätzten Zügen, die über die aktuelle Schranke hinausgehen.

```
struct Node
{
  RubiksCube cube;
  uint8_t move;
  uint8_t depth;
};
stack<Node> nodeStack;
Node curNode;
bool solved = false;

uint8_t bound;
uint8_t nextBound = heuristic(cube);

while (!solved)
{
  if (nodeStack.empty())
  {
    nodeStack.push({cube, Move::Undefined, 0});
    bound = nextBound;
    nextBound = 0xFF;
  }
  curNode = nodeStack.top();
  nodeStack.pop();

  if (curNode.depth == bound)
    if (curNode.cube.isSolved())
      solved = true;
    else
    {
      for (Move &move : RubiksCube::AllMoves)
      {
        if (false == prune(move, curNode.move))
        {
          RubiksCube cubeCopy(curNode.cube);
          cubeCopy.applyMove(move);
          uint8_t estSuccMoves = curNode.depth + 1 + heuristic(cubeCopy);

          if (estSuccMoves <= bound)
            nodeStack.push({cubeCopy, move, estSuccMoves});
          else if (estSuccMoves < nextBound)
            nextBound = estSuccMoves;
        }
      }
    }
}
```

Listing 4.4: IDA* Algorithmus zum Lösen eines Rubik's Cube

Korf-IDA* - Performance

Der IDA*-Algorithmus findet immer die optimale (kürzeste) Lösungssequenz für den Rubik's Cube. Abhängig von der Länge dieser Lösung muss der Algorithmus entsprechend tief im Graphen suchen. Je tiefer die Suche dabei geht, desto aufwändiger wird sie, da selbst bei der Eliminierung redundanter Züge ein durchschnittlicher Verzweigungsfaktor von etwa 13 bleibt.

Abbildung 4.16 zeigt den Zeitaufwand des Algorithmus (y-Achse) für verschiedene Tiefen im Graphen (x-Achse). Es ist deutlich zu erkennen, dass ab einer Tiefe von 9-10 der Zeitaufwand erheblich zunimmt. Dabei ist zu beachten, dass die y-Achse logarithmisch skaliert ist.

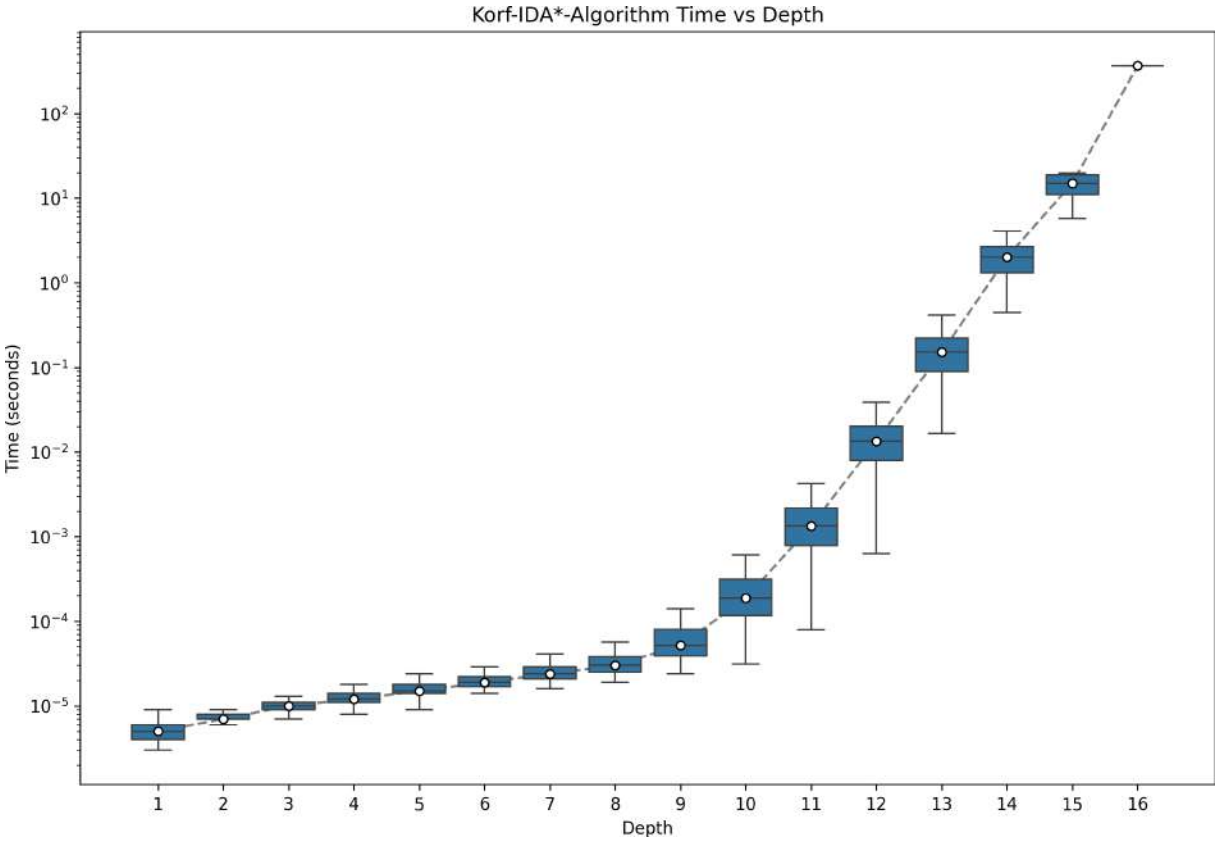


Abbildung 4.16: Boxplot zur Analyse des Zeitaufwands des Korf-IDA*-Algorithmus bei verschiedenen Suchtiefen im Graphen

Die Laufzeitkomplexität von IDA* ergibt sich wie folgt: $\mathcal{O}(b^d)$ [17]

b : Verzweigungsfaktor (ca. 13)

d : Tiefe der Lösung im Graphen

Daraus ergibt sich ein exponentieller Anstieg des Zeitaufwands des Algorithmus, was in einem logarithmischen Plot als Gerade dargestellt wird. Die Tatsache, dass der Plot in Abbildung 4.16 sich grob mittels zweier Geraden annähern lässt, liegt daran, dass bis zu einer Tiefe von 9-10 andere Faktoren (wie beispielsweise Initialisierungen im Code) den Zeitaufwand dominieren. Ab einer Tiefe von 10 überwiegt jedoch der Zeitaufwand der eigentlichen Suche im Graphen.

Tabelle 4.4 zeigt die ermittelten Medianwerte für die Tiefen 1-16 aus dem Plot 4.16, zusätzlich sind die extrapolierten Zeiten für die Tiefen 17-20 angegeben, die durch einen exponentiellen Fit berechnet wurden (basierend auf Medianwerten der Tiefen 10-16). Es wird deutlich, dass der Algorithmus ab einer Tiefe von 15-16 für den Einsatz in einem Roboter zu viel Zeit benötigt. Daher wird im Folgenden ein alternativer Algorithmus vorgestellt, der zwar nicht die optimale Lösung findet, jedoch stets innerhalb von Sekunden eine Lösungssequenz liefert.

Tabelle 4.4: Ermittelte Medianzeiten des Korf-IDA*-Algorithmus für Suchtiefen 1-16 sowie extrapolierte Zeiten für Tiefen 17-20 anhand eines exponentiellen Fits (basierend auf Medianwerten der Tiefen 10-16)

Depth	**Median Value / s**	**Extrapolation / s**	**Time**
1-9	$\leq$ 5,20000000e-05	-	52 μs
10	1,87000000e-04	-	0,2 ms
11	1,33850000e-03	-	1,3 ms
12	1,34650000e-02	-	13,5 ms
13	1,51091000e-01	-	151 ms
14	2,00803800e+00	-	2 s
15	1,50140570e+01	-	15 s
16	3,67237007e+02	-	6 min
17	-	2,599100e+03	43 min
18	-	2,857516e+04	8 h
19	-	3,141625e+05	3,6 Days
20	-	3,453982e+06	5,7 Weeks

Thistlethwaite-Algorithmus

Da der Korf-IDA*-Algorithmus bei Suchtiefen $\geq$ 16 sehr lange für das Finden der optimalen Lösung benötigt, wurde zusätzlich eine bereits vorhandene Implementierung herangezogen [18], die mittels des Thistlethwaite-Algorithmus den Cube zuverlässig innerhalb weniger Sekunden lösen kann.

Der Thistlethwaite-Algorithmus teilt den Lösungsprozess in vier Phasen auf, wobei jede Phase die Komplexität des Problems schrittweise reduziert. Ziel ist es, den Würfel in maximal 52 Zügen zu lösen [19].

Phasen des Thistlethwaite's Algorithm

1. **Reduktion auf Gruppe G1**
   Zulässige Züge: <L, R, F, B, U, D>
   In der ersten Phase wird der Würfel so manipuliert, dass alle Kantenstücke korrekt orientiert sind. Das bedeutet, dass alle Kanten so gedreht werden, dass ihre Farben entweder zur Ober- oder Unterseite des Würfels zeigen.

2. **Reduktion auf Gruppe G2**
   Zulässige Züge: <L, R, F, B, U2,D2>
   Nachdem die Kanten richtig orientiert sind, konzentriert sich die zweite Phase darauf, die Ecken zu orientieren. Das Ziel ist es, die Ecken so zu drehen, dass auch sie korrekt ausgerichtet sind.

3. **Reduktion auf Gruppe G3**
   Zulässige Züge: <L, R, F2,B2,U2,D2>
   In der dritten Phase werden die Kantenstücke in ihre richtigen Positionen gebracht. Obwohl die Kanten in Phase 1 orientiert wurden, müssen sie nun an den richtigen Stellen im Würfel platziert werden.

4. **Endgültiges Lösen (Gruppe G4)**
   Zulässige Züge: <L2,R2,F2,B2,U2,D2>
   In der letzten Phase wird der Würfel vollständig gelöst. Die Ecken und Kanten sind bereits orientiert und positioniert; es geht nun darum, sie durch eine beschränkte Menge von Zügen in ihre endgültigen Positionen zu bringen.

4.5 Applikationssoftware

Die Benutzeroberfläche (UI) der Applikationssoftware wurde gemäß des Wireframe-Entwurfs aus Kapitel 3.5 umgesetzt und ist in ihrer finalen Version in Abbildung 4.17 dargestellt. Für die Entwicklung der Desktop-Anwendung kam das Electron-Framework zum Einsatz, das es ermöglicht, plattformübergreifende Desktop-Applikationen mit Webtechnologien wie HTML, CSS und JavaScript zu erstellen. Das JavaScript-Framework ReactJS wurde verwendet, um die UI-Elemente komponentenweise zu verschachteln und wiederverwendbare Komponenten zu entwickeln. Zur Gestaltung der Benutzeroberfläche wurde das CSS-Framework TailwindCSS genutzt, das eine Vielzahl von vordefinierten Klassen bereitstellt.

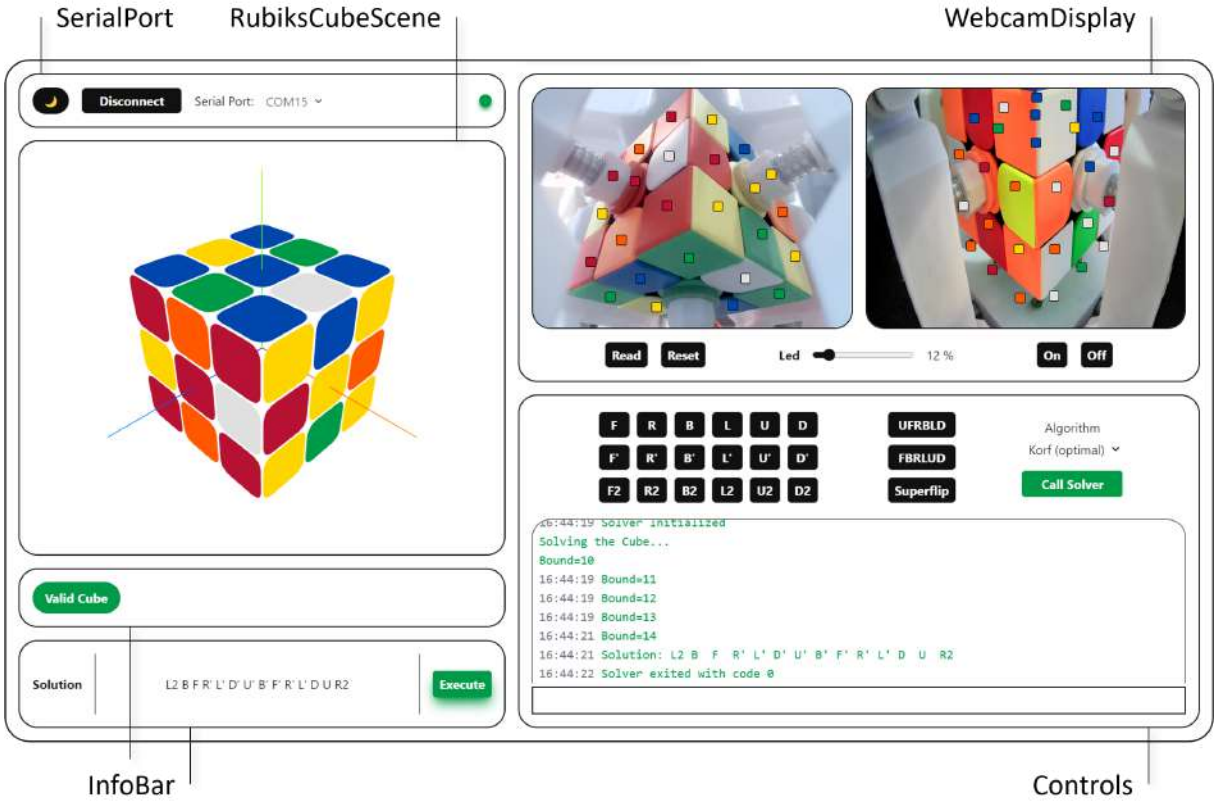


Abbildung 4.17: Darstellung der finalen Applikationssoftware und Bezeichnung der einzelnen UI-Komponenten

Im Folgenden werden die einzelnen UI-Komponenten der finalen Applikations-Software aus Abbildung 4.17 genauer beschrieben.

- **SerialPort:** Diese Komponente ermöglicht die Auswahl des seriellen Ports zur Kommunikation mit dem μC und die Verbindung mittels des *Connect*-Buttons. Bei einer erfolgreichen Verbindung wird der Status-Indikator rechts grün. Falls die

Verbindung fehlschlägt, wird dies durch einen roten Status-Indikator angezeigt und zusätzliche Fehler-Informationen werden in der InfoBar ausgegeben. Für die serielle Kommunikation wird das npm-Package serialport verwendet. Der linke Button ermöglicht zudem den Wechsel zwischen Dark- und Light-Mode der UI.

- **RubiksCubeScene:** Hier wird der aktuelle Zustand des Rubik's Cubes dreidimensional dargestellt. Diese Darstellung ist interaktiv und erlaubt es, den Cube frei im Raum zu drehen. Die drei Farbachsen helfen bei der Orientierung, wobei die linke Seite des Cubes bei der Betrachtung, wie in Abbildung 4.17 dargestellt, als *Front* gekennzeichnet ist. Bei der Manipulation des Cubes wird das Drehen der Seiten animiert und zeitlich mit der Ansteuerung des Roboters abgestimmt. Für die 3D-Darstellung wurde das npm-Package three.js verwendet und für die Animationen tween.js.

- **WebcamDisplay:** Diese Komponente zeigt den Video-Stream der beiden Webcams in Echtzeit an. Zudem besteht hier die Möglichkeit, den aktuellen Zustand des Rubik's Cubes auszulesen. Die kleinen Quadrate, die dem Webcam-Stream überlagert sind, kennzeichnen die Stellen, an denen der Cube beim Drücken des *Read*-Buttons ausgelesen wird. Diese Positionen können mittels Drag & Drop beliebig verschoben werden. Beim Auslesen des Cubes ändern sich die Flächen der Quadrate zu der jeweiligen interpretierten Farbe. Falls eine falsche Farbe eingelesen wird und somit ein invalider Cube erkannt wird, wird dies in der Info-Bar gekennzeichnet. Danach kann durch Drücken der Quadrate die Farbe manuell angepasst werden. Beim Drücken des *Read*-Buttons wird zudem die Beleuchtung der Webcams eingeschaltet, wobei die Helligkeit über einen Schieberegler einstellbar ist. Die LEDs bleiben an, bis ein valider Cube-State eingelesen worden ist. Das verwendete npm-Package für die Webcam-Funktionalität ist react-webcam und für die interaktiven verschiebbaren Auslese-Quadrate wurde react-draggable verwendet.

- **Controls:** Diese Komponente ermöglicht das Senden vordefinierter Kommandos an den μC zur Ansteuerung der Schrittmotoren. Zudem kann, bei Vorhandensein eines validen Cube-States, einer der beiden in dieser Arbeit behandelten Algorithmen zum Lösen ausgewählt werden. Beim Ausführen eines Algorithmus wird dessen Output in grün mit Timestamps im Textfeld der Controls-Komponente dargestellt, während die serielle Kommunikation in Grau dort mit aufgezeichnet wird. Das Eingabefeld ermöglicht es zudem, manuelle Kommandos an den μC zu senden. Das Ausführen des Algorithmus wird mittels eines asynchronen nodejs childprocess gehandhabt, wobei der aktuelle Cube-State als Übergabeargument mitgegeben wird. Falls der Solver eine Lösung findet, wird diese über eine Suche nach

dem String *'Solution:'* im Output des Solvers erkannt und anschließend die Lösungssequenz in der Applikation registriert. Danach ist es möglich, in der InfoBar die interpretierte Lösungssequenz zu betrachten und mittels des *Execute*-Buttons an den μC zu senden, um den Cube zu lösen.

- **Infobar:** Wie bereits bei der Beschreibung der anderen Komponenten erklärt, wird in der InfoBar der Fehler-Output für die serielle Verbindung und fehlerhaftes Auslesen des Cubes angezeigt. Zudem kann hier, bei Vorhandensein einer Lösungssequenz, diese betrachtet und ausgeführt werden.

Auslesen des Cubes

Für das Auslesen des Cubes mittels der beiden Webcams werden, wie bereits bei der Beschreibung der Webcam-Display-Komponente erwähnt, bestimmte Bereiche durch kleine Quadrate markiert. Beim Drücken des *Read*-Buttons wird eine Momentaufnahme der Webcam-Streams als Bild gespeichert. Die Farben dieser Bilder werden an den markierten Positionen der Quadrate gemittelt und ausgelesen. Das Ergebnis ist jeweils ein 8-Bit-Wert für die Farben RGB (Rot, Grün, Blau) für jede eingelesene Fläche des Cubes.

Zur Klassifizierung dieser ausgelesenen Farbwerte in die Rubik's Cube Farben (Red, Green, Blue, White, Yellow, Orange) wird ermittelt, zu welcher dieser Klassifizierungsfarben die eingelesene Farbe die geringste Distanz im RGB-Farbraum aufweist. Dabei wird folgendermaßen vorgegangen:

Ausgelesene Farbe: $C = [R, G, B]$
Vordefinierte Farben: $C_i = [R_i, G_i, B_i]$ für $i =$ Red, Green, Blue, White, Yellow, Orange

Der euklidische Abstand zwischen der ausgelesenen Farbe C und einer vordefinierten Farbe C_i wird berechnet als:

$$d(C, C_i) = \sqrt{(R - R_i)^2 + (G - G_i)^2 + (B - B_i)^2}$$

Die klassifizierte Farbe ist dann diejenige, die den geringsten Abstand aufweist:

$$\text{klassifizierte Farbe} = C_{i^*} \quad \text{mit} \quad i^* = \arg\min_i \sqrt{(R - R_i)^2 + (G - G_i)^2 + (B - B_i)^2}$$

Wie in Abbildung 4.19 zu erkennen ist, besitzen die beiden Webcams eine sehr unterschiedliche Farbdarstellung, wodurch für die Klassifizierungsfarben zwei separate Datensätze für jede Webcam erstellt werden mussten. Zur Erstellung der Klassifizierungsfarben wurde der gelöste Rubik's Cube mit jeder Webcam auf jeder der sechs Seiten

unter verschiedenen Lichtbedingungen eingelesen, und die Farbwerte wurden jeweils gemittelt. Abbildung 4.18 stellt diese ermittelten Farben im 3D-RGB-Farbraum dar.

Für eine ideale Klassifizierung sollten die Farben im RGB-Farbraum möglichst weit auseinanderliegen. Wie jedoch in Abbildung 4.18 zu erkennen ist, sind diese nicht optimal verteilt und liegen vor allem bei Cam1 (linke Webcam in der Applikationssoftware) sehr nahe beieinander, was beim Auslesen häufig zu falschen Klassifizierungen führt.

Diese Klassifizierungsfehler treten vor allem bei Farben auf, die nahe beieinander liegen und somit bei unterschiedlichen Lichtbedingungen dazu führen können, dass eine eingelesene Farbe einen geringeren Abstand zu einer falschen als zur eigentlich richtigen Klassifizierungsfarbe hat. Eine mögliche Lösung zur Minimierung dieses Problems wäre der Austausch von Cam1 durch eine Webcam vom gleichen Typ wie Cam2, da hier die Farben weiter auseinander liegen und es weniger häufig zu Fehlklassifizierungen kommt. Zudem könnten Farbkalibrierungsflächen in Sichtweite der Kameras am Roboter befestigt werden, wodurch vor jedem Einlesen die Klassifizierungsfarben neu kalibriert werden könnten.

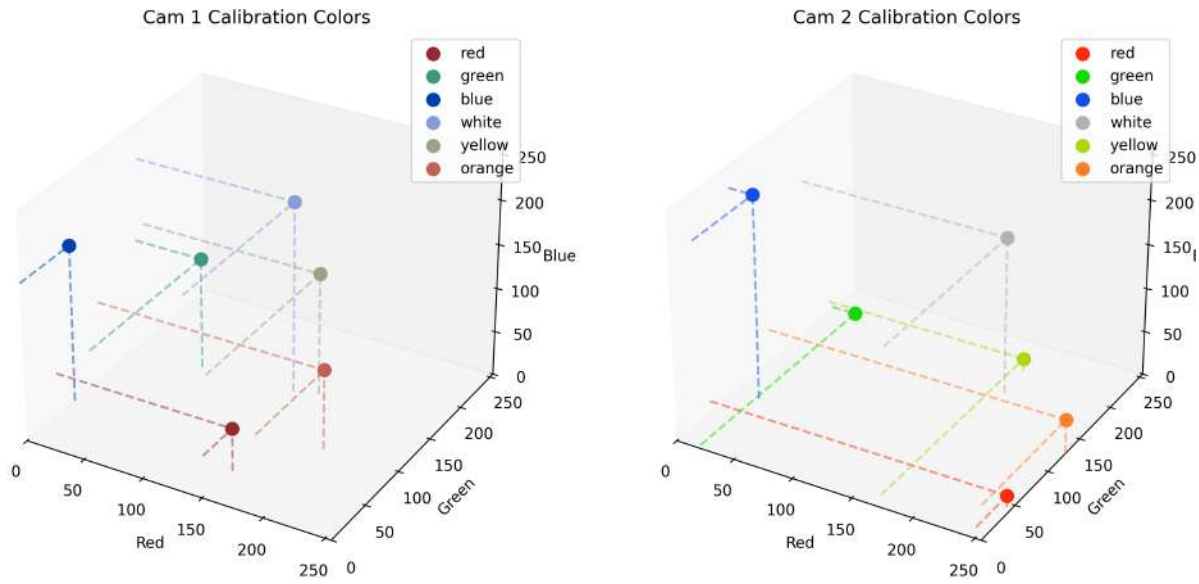


Abbildung 4.18: Darstellung der Klassifizierungsfarben der beiden Webcams: Cam1 (linke Webcam in der Applikationssoftware) und Cam2 (rechte Webcam in der Applikationssoftware)

Interpolation und Validierung des eingelesenen Cubes

Wie in Abbildung 4.19 zu erkennen ist, ist es aufgrund der Konstruktion des Roboters nicht möglich, alle Flächen des Rubik's Cubes einzulesen. Besonders bei der linken Kamera an der Unterseite ist eine Eckfläche komplett durch den unteren Greifer verdeckt und die untere Mittelfläche nur suboptimal lesbar und daher vom Auslesen ausgenommen.

Für die Interpolation der fehlenden Flächen und zur Validierung des Cubes wird folgendermaßen vorgegangen:

1. **Mittelflächen:** Entferne aus einem Array, das alle sechs Farben des Cubes enthält, die eingelesenen Mittelflächen. Wenn eine Farbe zweimal entfernt wird, sind die eingelesenen Mittelflächen an mindestens einer Stelle falsch. Bleibt am Ende nur noch eine Farbe im Array übrig, ist dies die fehlende untere Mittelfläche.

2. **Ecken:** Erstelle ein Array, das alle Eckstücke mittels dreier Buchstaben (z.B. 'bry' für Blue, Red, Yellow) enthält. Entferne daraus die eingelesenen Eckstücke (die Buchstaben werden dabei alphabetisch sortiert, sodass die Reihenfolge keine Rolle spielt). Wenn ein Eckstück im Array zweimal entfernt wird oder gar nicht erst im Array vorhanden ist, wurde mindestens eine Ecke falsch eingelesen. Bleibt am Ende nur noch eine Ecke im Array übrig, gibt der verbleibende Buchstabe nach Entfernen der beiden eingelesenen Farben die fehlende Eckfarbe an.

3. **Kanten:** Vorgehen wie bei den Eckstücken.

Das Vorgehen bei einen Invaliden eingelesenen Cube in der Applikationssoftware ist in Abbildung 4.19 dargestellt. Hier wurde beim Einlesen zweier Roter Flächen diese als Orange Interpretiert und eine Orange Fläche als Gelb. Die Falsch eingelesenen Flächen sind hierbei Magenta markiert. Die UI gibt in der InfoBar dann eine Fehlmeldung *'Invalid Cube'* aus und ist zudem in der Lage die Fehlstelle auf Eck- Kanten- oder Mittelstücke einzugrenzen. Danach ist es möglich diese Farben durch klicken der Lese-Quadrate manuell anzupassen bis die Applikationssoftware in der InfoBar den Cube als valide kennzeichnet und daraufhin die Beleuchtung zum Auslesen des Cubes ausgeschaltet wir und es nun möglich ist den Lösungsalgrithmus aufzurufen.

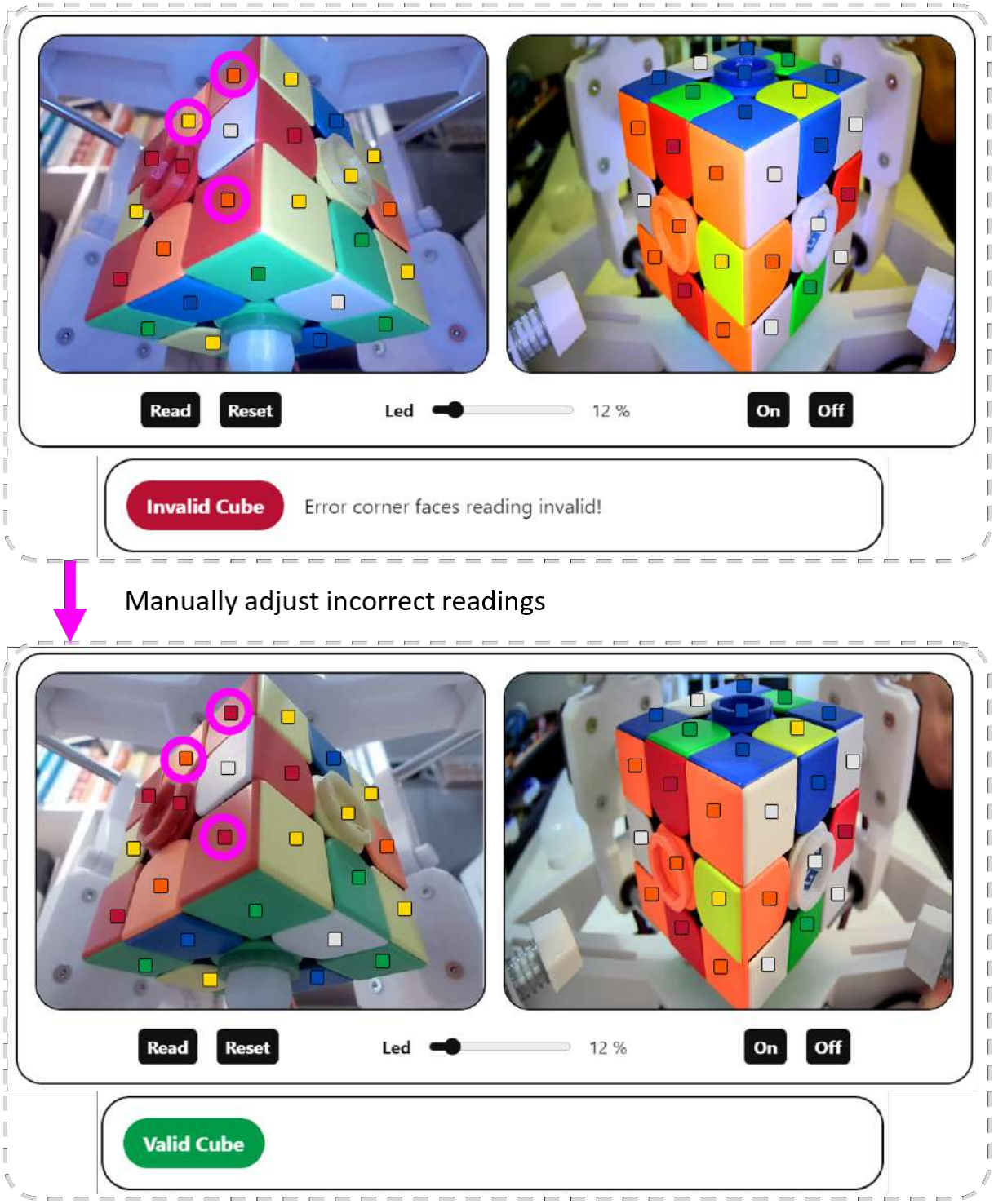


Abbildung 4.19: Darstellung der Handhabung eines fehlerhaft eingelesenen Cubes in der Anwendungssoftware. Falsch interpretierte Flächen sind magentafarben markiert.

5 Zusammenfassung und Ausblick

Die Implementierung des Robot-Rubi wurde mit der Fertigstellung der Applikationssoftware als zentrale Steuereinheit zwischen Roboter, Algorithmus und Benutzer erfolgreich abgeschlossen. Die Applikation ermöglicht es dem Anwender, einen Rubik's Cube in den Roboter einzulegen, den Würfel auszulesen, diesen an den Lösungsalgorithmus zu übergeben und schließlich die Lösungssequenz an die μC-Steuereinheit des Roboters weiterzugeben. Diese steuert dann die Schrittmotoren, um die Lösungssequenz auszuführen und den Würfel in seinen gelösten Zustand zurückzuversetzen.

Obwohl ein solcher Roboter, wie bereits in der Einleitung erörtert, keinen direkten praktischen Nutzen hat, war die Erstellung durch die vielfältigen Herausforderungen und weitreichenden Themengebieten von Linearkugellagern über die Auslegung einer Spannungsreglerschaltung bis hin zum sequenziellen Indexieren eines permutierten Arrays sehr lehrreich. Auf diesem Weg konnte somit viel Wissen gesammelt und angewendet werden.

Das Projekt kann als erfolgreich bewertet werden. Für eine zweite Entwicklungsphase des Solvers sind jedoch folgende Verbesserungsvorschläge vorgesehen:

- **Optimierung der Farbkalibrierung der Webcams:** Durch den Einsatz definierter Farbflächen im Erfassungsbereich der Kameras kann die Genauigkeit der Zustandserfassung des Würfels erhöht werden.
- **Erweiterung der mechanischen Komponenten:** Verbesserungen an den mechanischen Komponenten sollen das Einsetzen des Würfels in den Solver erleichtern und die Handhabung insgesamt benutzerfreundlicher gestalten.
- **Entwicklung und Implementierung eines automatischen Mechanismus:** Ein automatischer Mechanismus für das Öffnen und Schließen des Solvers soll die Benutzerfreundlichkeit weiter steigern und den Betrieb des Roboters effizienter gestalten.

Literatur

[1] Rachel Swatman. *Fastest robot to solve a Rubik's cube record falls again as German engineer takes title*. 2016. URL: https://www.guinnessworldrecords.com/news/2016/2/new-footage-proves-fastest-robot-to-solve-a-rubiks-cube-record-has-been-broken-y-418225 (besucht am 19. 05. 2024).

[2] MIT. *Solving a Rubik's Cube in record time: A robot developed by MIT students Ben Katz and Jared Di Carlo can solve a Rubik's Cube in a record-breaking 0.38 seconds.* 2018. URL: https://news.mit.edu/2018/featured-video-solving-rubiks-cube-record-time-0316 (besucht am 19. 05. 2024).

[3] Undark Hope Reese. *A Brief History of the Rubik's Cube: Nearly half a century after its humble invention, the cube continues to be a global sensation. What's the secret?* 2020. URL: https://www.smithsonianmag.com/innovation/brief-history-rubiks-cube-180975911/ (besucht am 20. 05. 2024).

[4] Denes Ferenc. *Rubik's Cube solution with advanced Fridrich (CFOP) method*. 2024. URL: https://ruwix.com/the-rubiks-cube/advanced-cfop-fridrich/ (besucht am 08. 03. 2024).

[5] Richard E. Korf. *Finding Optimal Solutions to Rubik's Cube Using Pattern Databases*. 1997. URL: https://www.cs.princeton.edu/courses/archive/fall06/cos402/papers/korfrubik.pdf (besucht am 08. 03. 2024).

[6] Tomas Rokicki. *God's Number is 20: Every position of Rubik's Cube™ can be solved in twenty moves or less.* URL: https://www.cube20.org/ (besucht am 20. 05. 2024).

[7] Jaap Scherphuis. *Thistlethwaite's 52-move algorithm*. 2024. URL: https://www.jaapsch.net/puzzles/thistle.htm (besucht am 08. 03. 2024).

[8] polylog. *The hidden beauty of the A* algorithm*. 2023. URL: https://www.youtube.com/watch?v=A60q6dcoCjw&ab_channel=polylog (besucht am 01. 06. 2024).

[9] Miguell Malacad. *Comparing Dijkstra's and A* Search Algorithm*. 2022. URL: https://medium.com/@miguell.m/dijkstras-and-a-search-algorithm-2e67029d7749 (besucht am 01. 06. 2024).

[10] PlatformIO. *What is PlatformIO?* URL: https://docs.platformio.org/en/latest/what-is-platformio.html (besucht am 05. 06. 2024).

[11] Mike McCauley. *AccelStepper library for Arduino*. 2022. URL: https://www.airspayce.com/mikem/arduino/AccelStepper/ (besucht am 05. 06. 2024).

[12] Adafruit. *Adafruit NeoPixel Library: Arduino library for controlling single-wire LED pixels (NeoPixel, WS2812, etc.)* URL: https://github.com/adafruit/Adafruit_NeoPixel (besucht am 05. 06. 2024).

[13] digint. *TinyFSM: A simple C++ finite state machine library*. 2022. URL: https://github.com/digint/tinyfsm (besucht am 05. 06. 2024).

[14] Ben Botto. *Implementing an Optimal Rubik's Cube Solver using Korf's Algorithm: And a Quick Solver Using Thistlethwaite's Algorithm*. 2020. URL: https://medium.com/@benjamin.botto/implementing-an-optimal-rubiks-cube-solver-using-korf-s-algorithm-bf750b332cf9 (besucht am 07. 06. 2024).

[15] Ben Botto. *Sequentially Indexing Permutations: A Linear Algorithm for Computing Lexicographic Rank*. 2019. URL: https://medium.com/@benjamin.botto/sequentially-indexing-permutations-a-linear-algorithm-for-computing-lexicographic-rank-a22220ffd6e3 (besucht am 07. 06. 2024).

[16] benbotto. *generic.cpp: Sequentially Indexing Permutations O(n) (Generic)*. 2019. URL: https://gist.github.com/benbotto/021b3c78d6a0b5a35479e97fd301617d#file-generic-cpp (besucht am 07. 06. 2024).

[17] Richard E. Korf. *Time complexity of iterative-deepening-A∗*. 2000. URL: https://www.sciencedirect.com/science/article/pii/S0004370201000947 (besucht am 08. 06. 2024).

[18] Cubically. *Thistlethwaite: A better cube solver.* URL: https://github.com/Cubically/thistlethwaite/tree/master (besucht am 07. 06. 2024).

[19] jaapsch. *Thistlethwaite's 52-move algorithm*. URL: https://www.jaapsch.net/puzzles/thistle.htm (besucht am 08. 06. 2024).