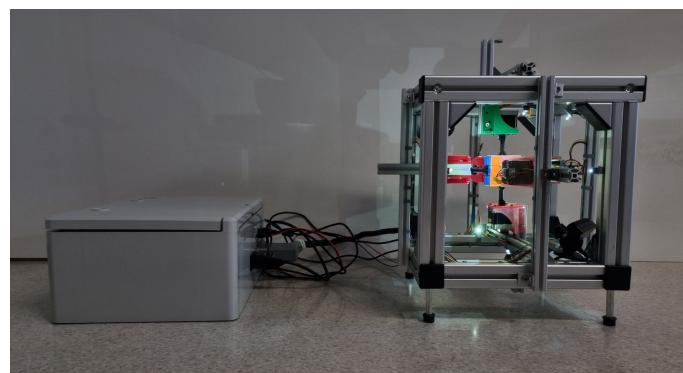


Kim Henrik Otte  
Kim Westphal  
Marvin Paetow  
Christoph Rathjen  
Sophie Kirchhoff

# Rubik's Cube Solving Machine

## Bachelorprojekt



The authors have made effort to ensure the provided information is correct. However, the information is provided without any warranty. Neither the authors, nor any other person or organization will be held liable for any damages caused or alleged to have been caused directly or indirectly by this document.

Rubik's Cube Solving Machine  
Original report: 04.07.2022  
Document version: 02.08.2023

© 2022 – 2023  
Authors: Otte, Westphal, Paetow, Rathjen, Kirchhoff  
Supervisor: Prof. Dr. Marc Hensel  
<http://www.haw-hamburg.de/marc-hensel>

Published under Creative Commons license CC BY-NC-ND 3.0  
Attribution, non-commercial, no derivatives  
<https://creativecommons.org/licenses/by-nc-nd/3.0/deed.en>

# Inhaltsverzeichnis

---

<b>Inhaltsverzeichnis</b>	<b>3</b>
<b>Abbildungsverzeichnis</b>	<b>4</b>
<b>Tabellenverzeichnis</b>	<b>5</b>
<b>1 Einleitung</b>	<b>7</b>
<b>2 Hardware-Aufbau</b>	<b>8</b>
2.1 Würfelgestell . . . . .	8
2.2 Schaltschrank . . . . .	9
2.3 Materialliste . . . . .	11
2.4 Schaltplan . . . . .	12
<b>3 Software</b>	<b>13</b>
3.1 Arduino TB6600 (Main) . . . . .	13
3.1.1 Befehlsformat Move-Befehle . . . . .	14
3.1.2 TB6600 (Main Datei) . . . . .	14
3.1.3 StepperStepControl . . . . .	15
3.1.4 clsTB6600 . . . . .	15
3.1.5 TB6600_Globals . . . . .	15
3.2 Visualisierung . . . . .	17
3.2.1 Hauptansicht . . . . .	17
3.2.2 Arduino . . . . .	18
3.2.3 Kamera . . . . .	20
3.2.4 Detections . . . . .	21
3.2.5 Manual Move . . . . .	21
<b>4 Bilderkennung</b>	<b>25</b>
4.1 Farbvereinfachung . . . . .	26
4.2 Farbzuordnung . . . . .	26
4.2.1 Ähnlichkeit zweier Farben . . . . .	26
4.2.2 Gruppierung . . . . .	26
4.3 Implementierung . . . . .	28
<b>5 Solver</b>	<b>30</b>
<b>6 Kameratreiber</b>	<b>31</b>
<b>7 Troubleshooting</b>	<b>32</b>

## Abbildungsverzeichnis

---

2.1	Gesamtaufbau . . . . .	8
2.2	Aufbau Würfelgestell . . . . .	8
2.3	Stecker Schaltschrank . . . . .	9
2.4	Aufbau Schaltschrank . . . . .	10
3.1	Arduino Main UML-State-Machine . . . . .	13
3.2	Klasse <i>TB6600</i> . . . . .	15
3.3	Klasse <i>StepperStepControl</i> . . . . .	16
3.4	Klasse <i>clsTB6600</i> . . . . .	16
3.5	Visualisierung mit spezifischem (1) und allgemeinem Kontrollfeld (2) . . . . .	17
3.6	Hauptansicht - Generate Algorithm . . . . .	18
3.7	Arduino . . . . .	19
3.8	Arduino (Anpassung Parameter) . . . . .	19
3.9	Arduino (Anpassung Schritte) . . . . .	20
3.10	Kamera . . . . .	20
3.11	Kamera (Scan Cube) . . . . .	21
3.12	Kamera (Start & End State) . . . . .	22
3.13	Kamera (Fehler korrigieren) . . . . .	22
3.14	Kamera (No Signal) . . . . .	23
3.15	Detections . . . . .	23
3.16	Manual Move . . . . .	24
4.1	Aufbau Rubik's Cube . . . . .	25
4.2	Gruppierung von Farben in sechs 4-er Gruppen . . . . .	27
4.3	Ungültige Flächenzuordnung . . . . .	28

## Tabellenverzeichnis

---

2.1	Materialliste . . . . .	11
3.1	Move-Befehlskodierung . . . . .	14
3.2	Move Befehle für die Motoren . . . . .	14



---

## **Kapitel 1**

### **Einleitung**

Die „Rubik’s Cube Solving Machine“ soll einen Rubik’s Cube, oder umgangssprachlich auch „Zauberwürfel“ genannt, zufällig „scramble“ (mischen), seinen Zustand mittels Bilderkennung erkennen und anschließend in kürzester Zeit ohne menschliche Einwirkung vollautomatisch lösen.

---

## Kapitel 2

### Hardware-Aufbau

Die „Rubik’s Cube Solving Machine“ besteht aus zwei Hauptkomponenten, dem Schaltschrank und dem Würfelgestell (Abb. 2.1). Verbunden werden diese über Stecker für die Motoren, die Beleuchtung und die beiden Kameras.

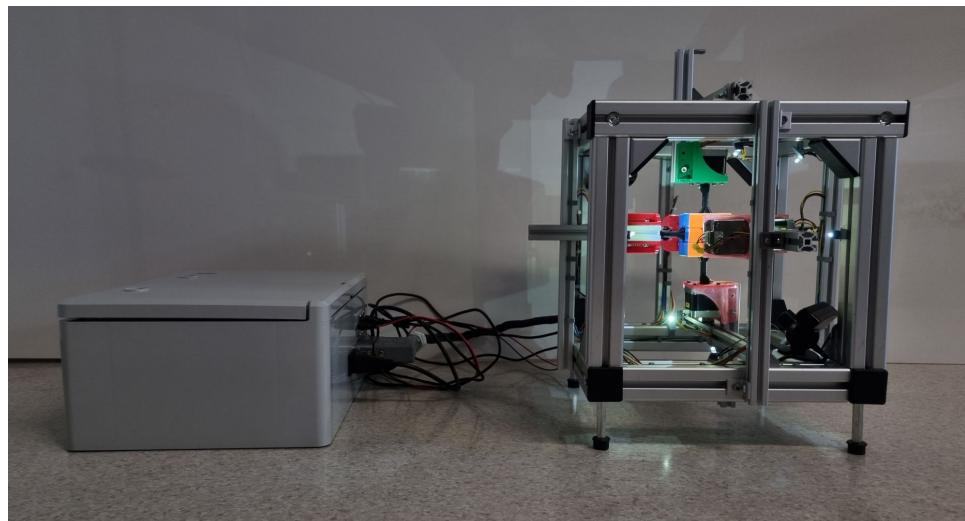


Abbildung 2.1: Gesamtaufbau

#### 2.1 Würfelgestell

Der Rubik’s Cube sitzt mittig, innerhalb eines aus 40x40 mm Item-Profilen bestehenden Würfelgestells (Abb. 2.2). Die Außenmaße des Würfelgestells betragen 33,5 cm. Befestigt wird der Rubik’s Cube an seinen sechs Mittelsteinen, die mittels 3D-gedruckten Motoraufnahmen mit den

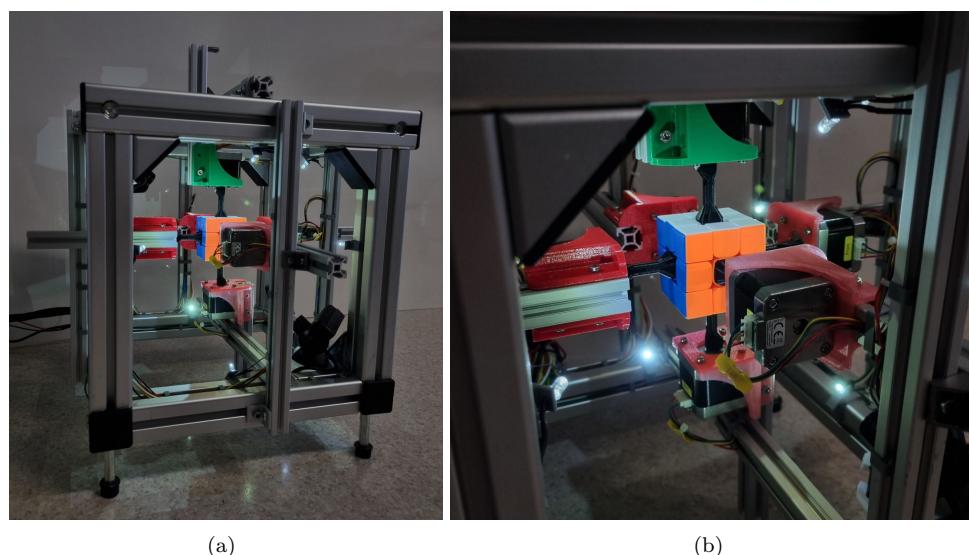


Abbildung 2.2: Aufbau Würfelgestell

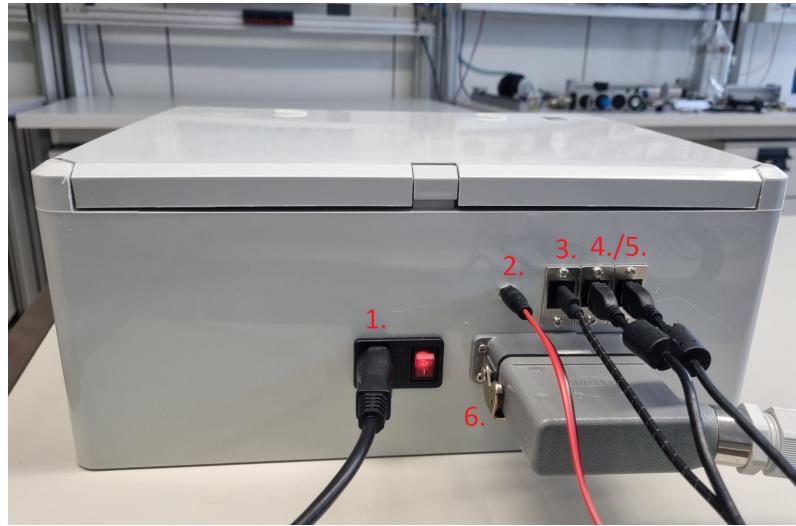


Abbildung 2.3: Stecker Schaltschrank

Achsen des Motors verbunden werden. Für die Drehung der unterschiedlichen Schichten werden Schrittmotoren mit entsprechenden Motortreibern verwendet. Die Schrittmotoren wurden mittels 3D-gedruckten Motorhalterungen an 20x20 mm Item-Profilen befestigt. Der Zustand des Würfels wird mit zwei PS3-Kameras ermittelt. Diese wurden mittels 3D-gedruckten Kamerahalterungen befestigt. Der ermittelte Zustand wird an den Berechnungsalgorithmus auf dem Computer übergeben. Der Algorithmus gibt seine Ergebnisse an einen Arduino weiter, der die Schrittmotoren ansteuert.

## 2.2 Schaltschrank

Zur Verwendung der „Rubik’s Cube Solving Machine“ müssen folgende Stecker eingesteckt werden (Abb. 2.3):

1. Netzstecker
2. Beleuchtung
3. USB-C Computer
4. und 5. 2x Kameras
6. Schrittmotoren

Im Schaltschrank sind folgende Komponenten vorhanden (Abb. 2.4):

1. Netzteil 1
2. Netzteil 2
3. 6x Motortreiber
4. USB-Hub
5. Klemme
6. Arduino



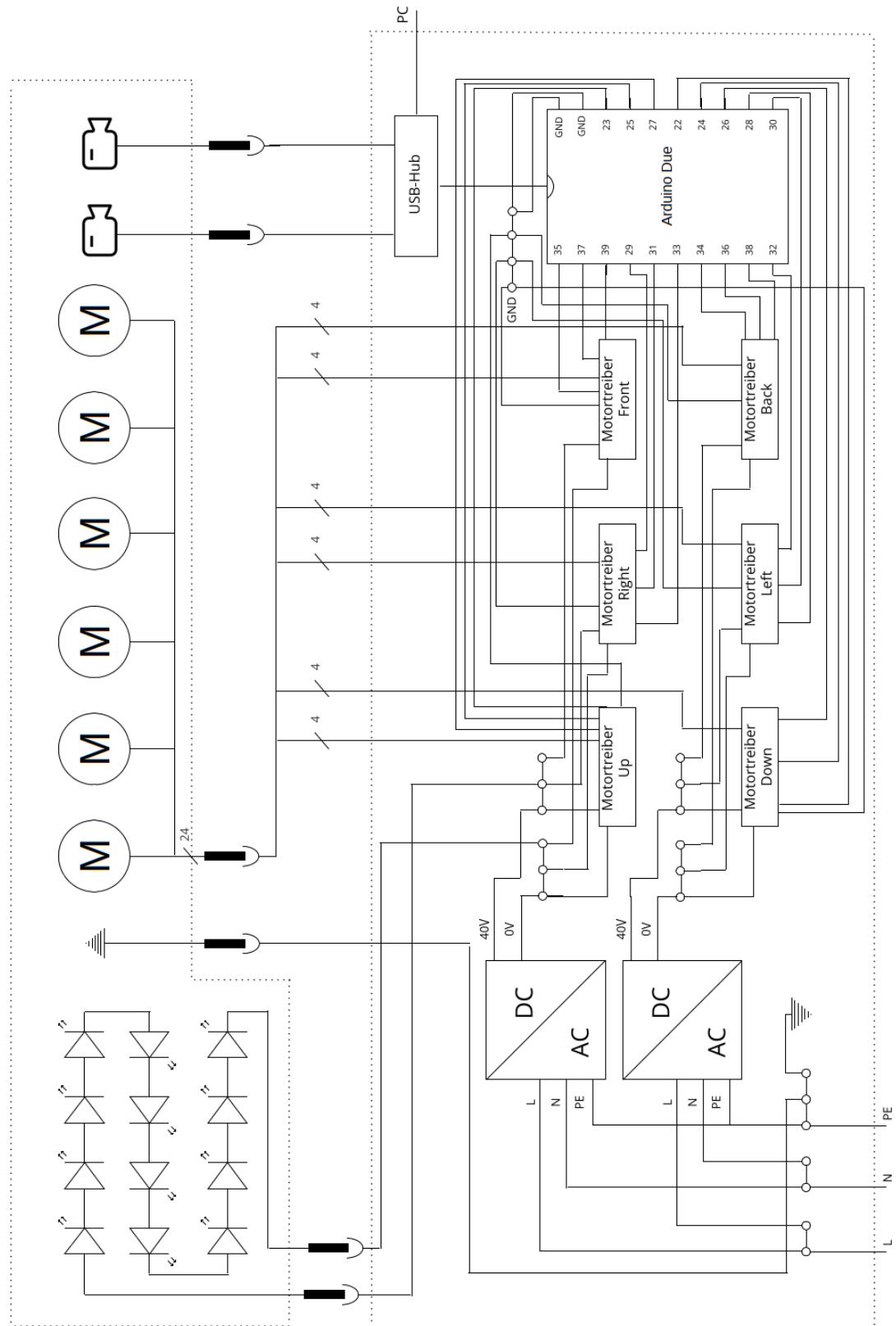
Abbildung 2.4: Aufbau Schaltschrank

## 2.3 Materialliste

Bezeichnung	Modell	Menge
Item Profil (1 m)	40 x 40 mm	3,4
Item Profil (1 m)	20 x 20 mm	3
Item Profilwinkel	40 x 40 mm	8
Item Profilwinkel	20 x 20 mm	26
Item Profil Nutenstein	40 x 40 mm	28
Item Profil Nutenstein	20 x 20 mm	58
Schrauben	unterschiedlich	118
Filament (500 g)	PLA	1
Leitungen (1 m)	0,38 mm <sup>2</sup>	48
Leitungen (1 m)	0,5 mm <sup>2</sup>	3
Stecker	25-polig	1
Schaltschrank	Boxexpert BXPPABSG300400170-F01 Schaltschrank 300 x 400 x 170	1
Netzteil	Schaltnetzteil, geschlossen, 150 W, 36 V, 4,3 A	2
Arduino	Due	1
USB-C Einbau-Buchse	Keystone USB3.1 Gen 2 Typ C	1
USB-A Einbau-Buchse/Stecker	Keystone USB3.0 A	2
Gehäusehalterung	Keystone Halterung	3
USB-Hub	USB 3.0 Typ-C, 4 Port Hub USB Typ-A	1
USB-A auf Mikro-USB-Kabel	USB 2.0 Kabel, A Stecker auf Micro B Stecker, 0,6 m	1
Kabelkanäle (1 m)	40 x 25 mm	0,5
Hutschiene (1 m)	35 x 7,5 mm	0,5

Tabelle 2.1: Materialliste

## 2.4 Schaltplan



### 3.1 Arduino TB6600 (Main)

Der Arduino dient zur Ansteuerung der Schrittmotoren mit den TB6600 Treibern (nach Implementierung einer entsprechenden Treiber-Klasse können auch andere Treiber verwendet werden). Er erhält von der Visu die entsprechenden Befehle. Zudem ist eine Konfiguration der Geschwindigkeiten über die Visu möglich. Dafür werden dem Arduino die Wartezeiten zwischen den einzelnen Schritten als float Arrays per UART geschickt.

Als Alternative ist ebenfalls eine feste Berechnung anhand der entsprechenden Parameter durch den Arduino möglich. Dies erfordert die Verwendung der *MoveSteps()*- und *setSteps()*-Funktionen der *StepperStepControl* oder eine Erweiterung um einen UART-Befehl in der UART Config, der die Funktion *calculateStepTimes()* aufruft. Diese Funktion wird ebenfalls im Setup dafür verwendet, um die Default Werte für die Turn\_times Arrays zu berechnen.

Die grundlegende Softwarestruktur des Arduinos ist eine State-Machine mit den folgenden Zuständen:

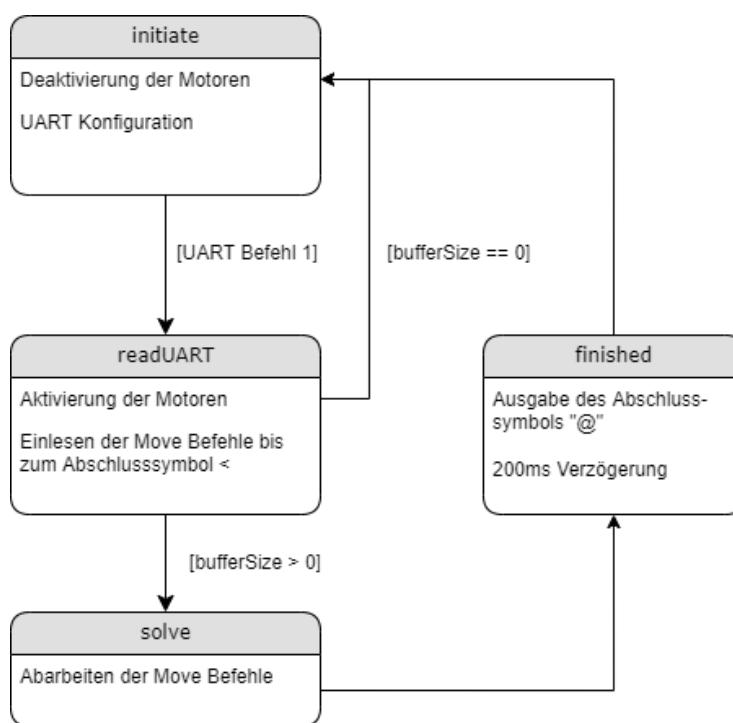


Abbildung 3.1: Arduino Main UML-State-Machine

Im Initialisierungszustand werden die Motortreiber deaktiviert, um freies Bewegen der Motoren zu ermöglichen. Mit der Visu können über UART Arduino Parameter konfiguriert werden. Mit dem Befehl 1 (als Byte) wird in den *readUART* Zustand gewechselt. In diesem werden die Befehle für die Bewegungen als einzelne Bytes über UART gelesen bis das Abschlussymbol "<" gelesen wird.

Nachdem die Befehle im *solve* Zustand bearbeitet wurden, gibt der Arduino im *finished* Zustand "@" aus, um der Visu mitzuteilen, dass alle Züge beendet wurden.

### 3.1.1 Befehlsformat Move-Befehle

Die Befehle der einzelnen Züge werden dem Arduino über UART von der Visu mitgeteilt. Dabei repräsentiert ein Byte jeweils einen Befehl für eine Achse ⇒ Up/Down, Left/Right oder Front/Back. Der Aufbau des Befehls ist in der folgenden Tabelle dargestellt.

Bit Nummer	Bedeutung
0,1	Achsen Auswahl: Up/Down, Right/Left, Front/Back
2,3,4	Bewegung erster Motor der Achse ⇒ Up/Right/Front
5,6,7	Bewegung zweiter Motor der Achse ⇒ Down/Left/Back

Tabelle 3.1: Move-Befehlskodierung

Die Befehle für die einzelnen Motoren sind folgendermaßen zu verstehen: cw = Clockwise, ccw = Counterclockwise.

Bit Werte	Bewegung
000	Keine Bewegung
001	90° cw
010	90° ccw
011	180° cw
100	180° ccw
Rest	Nicht definiert

Tabelle 3.2: Move Befehle für die Motoren

In den folgenden Abschnitten folgt die Darstellung einzelner Klassen/Elemente der Arduino Software.

### 3.1.2 TB6600 (Main Datei)

Abbildung 3.2 zeigt das Klassendiagramm. Einige Funktionen und Variablen sind nicht (mehr) im aktuellen Stand der Software integriert. Die Funktion *testmove1()* und die variable dd dienten zum Test der Motortreiber und werden nicht mehr regulär verwendet. Zudem ist keine Option zum Corner Cutting (Der nächste Zug beginnt bevor der vorherige abgeschlossen ist.) implementiert. Dies wäre entweder durch ein Deaktivieren des vorherigen Motors → Weniger Steps und der Würfel wird durch die nächste sich bewegende Seite ausgerichtet oder durch das parallele betreiben zweier Achsen (potentiell höhere Gefahr den Würfel zu verkanten) möglich.

- ▶ *void calculateStepTimes()*  
Berechnet die Turn\_times... Arrays mit den Verzögerungen zwischen den Schritten mit den aktuellen Arduino Parametern neu.
- ▶ *int com2Steps(char com)*  
Wandelt die Bit-Gruppen für die Motoren in die entsprechende Anzahl an Schritten um. Dafür werden die Schritt Konstanten aus der TB6600\_Globals Datei (siehe 3.1.5) verwendet.
- ▶ *void Move(char command)*  
Führt einen Move-Befehl aus. Der command ist dabei ein von der Visu erhaltenes Byte im in 3.1.1 beschriebenen Format.
- ▶ *bool uartSendParam()*  
Nicht mehr verwendet. Kann zum Senden der Arduino Parameter an die Visu genutzt werden.
- ▶ *void uartConfig()*  
Liest die von der Visu gesendeten Befehle über UART und führt die entsprechenden Anweisungen aus.

TB6600 (Main Datei)
<pre>+ enum class state initiate = 0, readUART = 1, solve = 2, finished = 3 + state main_state = state :: initiate + byte readBuffer[100] + byte inByte + int bufferSize + bool DEBUG = true + float minStepspsStart = 1000 + float minStepspsEnd = 800 + float MaxAccTime = 30 + float MaxBreakTime = 30 + float maxStepsps = 1900 + float* Turn_times90deg + float* Turn_times180deg + int d = 10 + int dd = 50 + int cc = 0 + bool disableBetweenMoves = false + TB6600 Driver1, Driver2, Driver3, Driver4, Driver5, Driver6 + StepperStepControl&lt;TB6600&gt; M1, M2, M3, M4, M5, M6</pre>
<pre>+ void testmove1() + void calculateStepTimes() + int com2steps(char com) + void Move(char command) + bool uartSendParam() + bool uartConfig() + void setup() + void loop()</pre>

Abbildung 3.2: Klasse *TB6600*

### 3.1.3 StepperStepControl

Generische Klasse gemäß Abbildung 3.3 zum Ansteuern eines beliebigen Treibers mit eigener Treiber Klasse. Diese muss die entsprechenden Funktionen implementieren (siehe *clsTB6600*). Je nach Aufruf kann die Klasse die vorberechneten Turn\_times Arrays für den Abstand zwischen den Schritten verwenden (*MoveSteps2()* und *setSteps2()*) oder die Zeiten während des Ablaufs selbst anhand der gegebenen Eigenschaften (Referenzen zu den Parametern in der Main TB6600 Datei) berechnen (*MoveSteps()* und *setSteps()*).

Zum Ausführen des Bewegungsbefehls muss zunächst die verwendete *setSteps()*-Funktion aufgerufen werden. Danach erfolgt ein zyklischer Aufruf von *MoveSteps()*, bis der Rückgabewert "false" ist → keine Schritte mehr zu machen.

### 3.1.4 clsTB6600

Klasse gemäß Abbildung 3.4 zur Ansteuerung der TB6600 Motortreiber. Es werden drei GPIO Pins zur Ansteuerung benötigt. Die Anschlussart ist High-Active. Die Instanziierung der Treiber erfolgt im TB6600 Hauptprogramm. Dabei müssen die Nummern der verwendeten GPIO Pins gesetzt werden.

### 3.1.5 TB6600\_Globals

Hier wird das Enum für die Drehrichtung → CW/CCW festgelegt. Zudem die Anzahl der benötigten Schritte für eine 90°- und 180°- Drehung. Bei den hier verwendeten Einstellungen der

StepperStepControl
<pre>+ float&amp; minStepspsStart + float&amp; minStepspsEnd + float&amp; MaxAccTime + float&amp; MaxBreakTime + float&amp; maxStepsps + int acAcTime = 0 + unsigned long acDelay = 0 + unsigned long waitStart + unsigned long waitTarget + bool lastStepdone + float* waitingTimes + float*&amp; Turn_times90deg + float*&amp; Turn_times180deg + int steps + int stepsstepped + float acF</pre>
<pre>+ void setSteps(int numSteps) + bool MoveSteps() + void setSteps2(int numSteps) + bool MoveSteps2() + void disableDriver() + void enableDriver() + StepperStepControl(const T stepper, float&amp; MinStepspsStart,                      float&amp; MinStepspsEnd, float&amp; maxAccTime,                      float&amp; maxBreakTime, float&amp; MaxStepsps,                      float*&amp; turn_times90deg, float*&amp; turn_times180deg)</pre>

Abbildung 3.3: Klasse *StepperStepControl*

clsTB6600
<pre>+ int ENABLE + int DIRECTION + int STEP</pre>
<pre>+ TB6600(int enable, int dir, int step1) + void step1() + void setDir(dir_type dir1) + void enableDriver() + void disableDriver()</pre>

Abbildung 3.4: Klasse *clsTB6600*

Schrittmotortreiber werden 200 Schritte für eine volle Umdrehung benötigt. Also 1.8° pro Schritt.

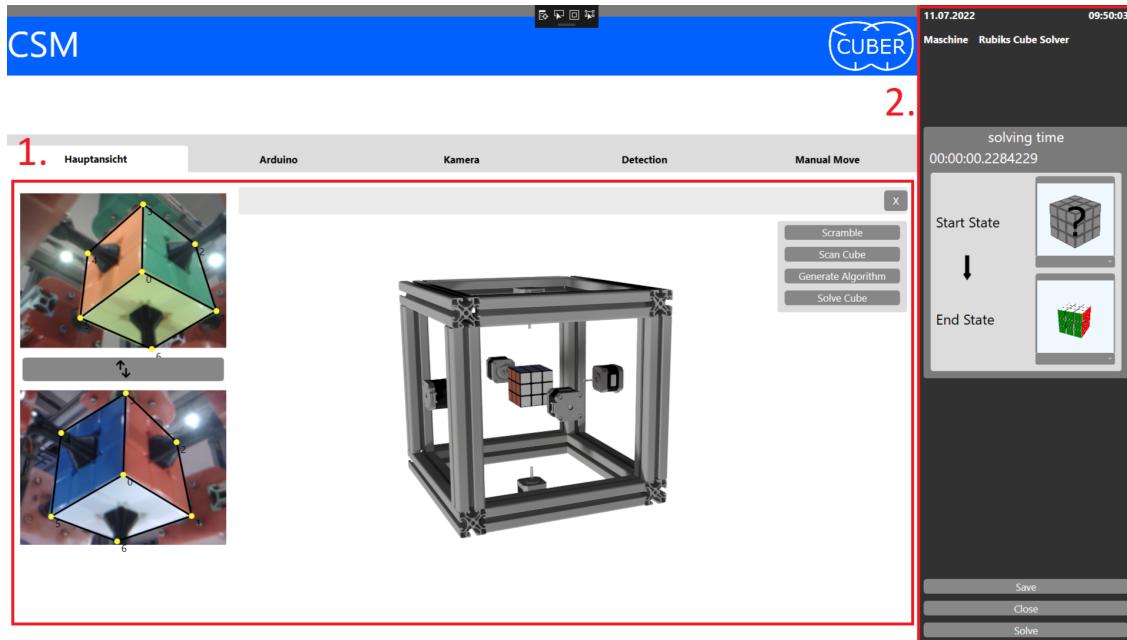


Abbildung 3.5: Visualisierung mit spezifischem (1) und allgemeinem Kontrollfeld (2)

## 3.2 Visualisierung

Mit der in Abbildung 3.5 dargestellten Visualisierung lässt sich die „Rubik’s Cube Solving Machine“ bedienen. Dabei verändert sich das rechte allgemeine Kontrollfeld nicht. Dieses kann zu jedem Zeitpunkt bedient werden. Dabei kann der Start- und Endzustand des Würfels angegeben werden.

- ▶ „Save“ speichert allgemeine Einstellungen der Visualisierung.
- ▶ „Close“ beendet die Visualisierung.
- ▶ „Solve“ löst den Würfel zum ausgewählten Endzustand.

Je nach Auswahl des Reiters verändert sich die Oberfläche des spezifischen Kontrollfelds. Man kann unter folgenden Reitern wählen:

- ▶ Hauptansicht
- ▶ Arduino
- ▶ Kamera
- ▶ Detection
- ▶ Manual Move

### 3.2.1 Hauptansicht

Im Kontrollfeld „Hauptansicht“ ist mittig das Gestell, inklusive Motoren und Würfel dargestellt. Auf der linken Seite sind die Ansichten der beiden Kameras zu sehen. Über dem Gestell befindet sich eine „Move-Zeile“ in welcher die Züge angezeigt und bearbeitet werden können. Dabei beschreiben die einzelnen Buchstaben-Zahlen-Kombination, bzw. diese in Klammern, einen Zug. In Abbildung 3.6 werden somit drei Züge angezeigt. Rechts kann unter verschiedenen Optionen gewählt werden.

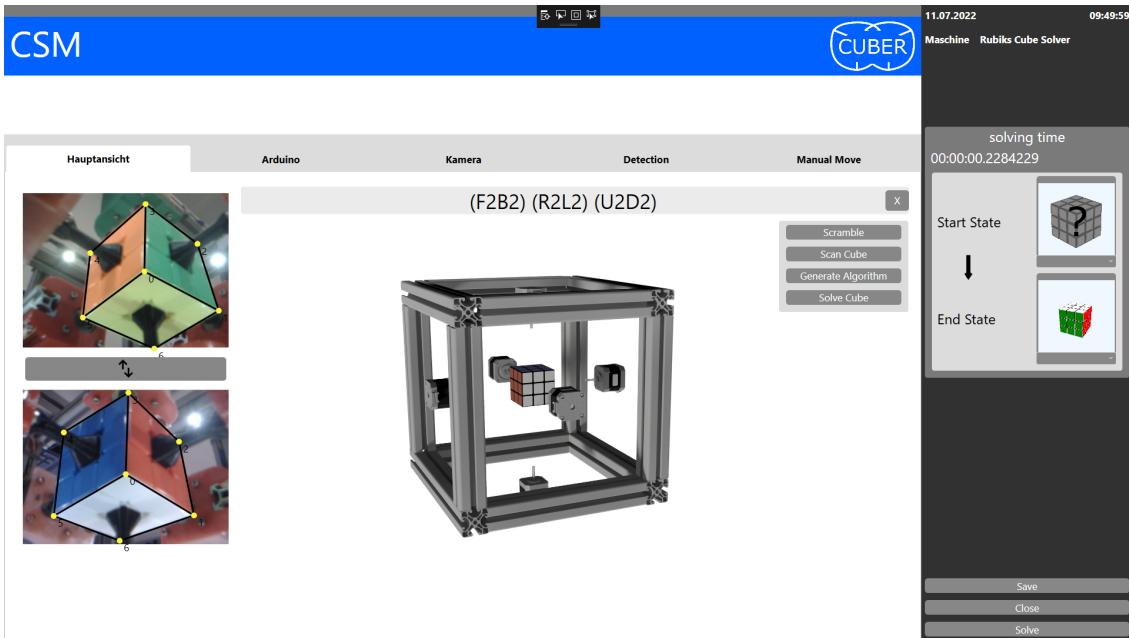


Abbildung 3.6: Hauptansicht - Generate Algorithm

- „Scramble“ verdreht den Würfel.
- „Scan Cube“ scannt alle Würfelseiten mittels zwei Kameras.
- „Generate Algorithm“ berechnet den Algorithmus bzw. die Züge zum Lösen des Würfels vom „Start State“ zum „End State“. Sollte der „Start State“ undefiniert (?) sein wird das Scannen des Würfels inkludiert.
- „Solve Cube“ sendet die Befehle aus der Move-Zeile über dem Gestell an den Arduino. Sollte die Move-Zeile leer sein, wird „Generate Algorithm“ aufgerufen. Es werden die Schrittmotoren angesteuert.

### 3.2.2 Arduino

Im Kontrollfeld „Arduino“ (Abb. 3.7) werden die Abstände zwischen den Schritten der Schrittmotoren angezeigt. Somit können diese einzeln angepasst werden. Der obere Graph ist den 90°-Drehungen und der untere den 180°-Drehungen zuzuordnen. Links kann unter verschiedenen Optionen gewählt werden:

- „Update“ sendet die Move-Zeile an den Arduino.
- „SerialPort“ wählt den Serialport zum Arduino aus.
- „Delay“ wählt das Delay zwischen den Zügen aus.
- „DelayScramble“ wählt das Delay zwischen den Zügen beim Scramben aus.
- „Debug“ setzt den Arduino in den Debug Modus.
- „SPSMax“ setzt maximale steps per second.
- „SPSStart“ setzt steps per second am Anfang des Move.
- „SPSEnd“ setzt steps per second am Ende des Move.

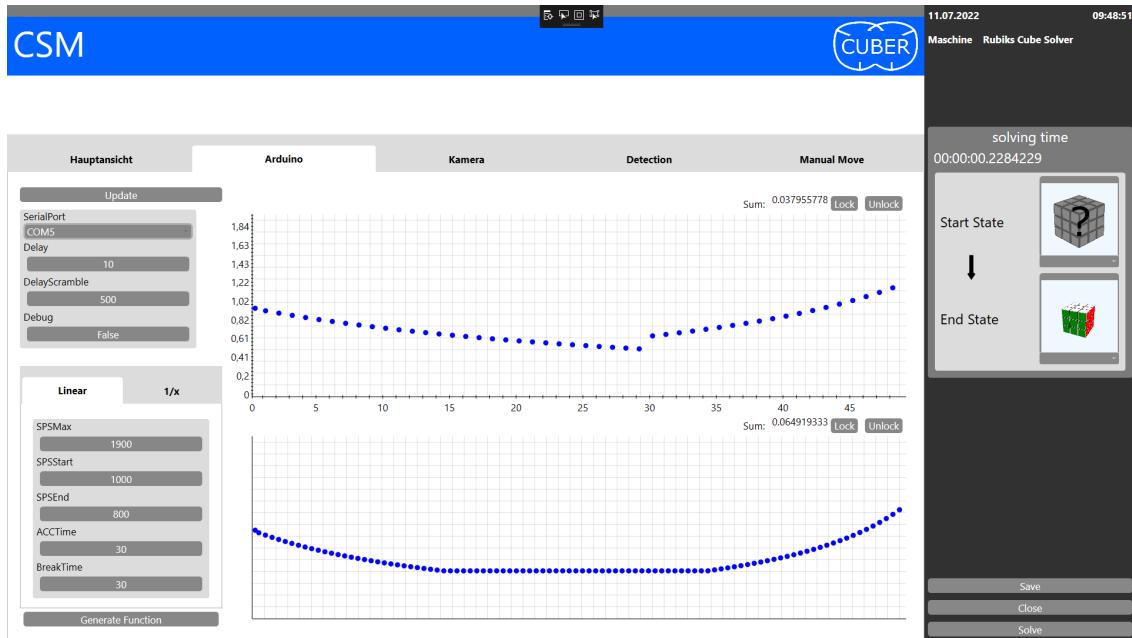


Abbildung 3.7: Arduino

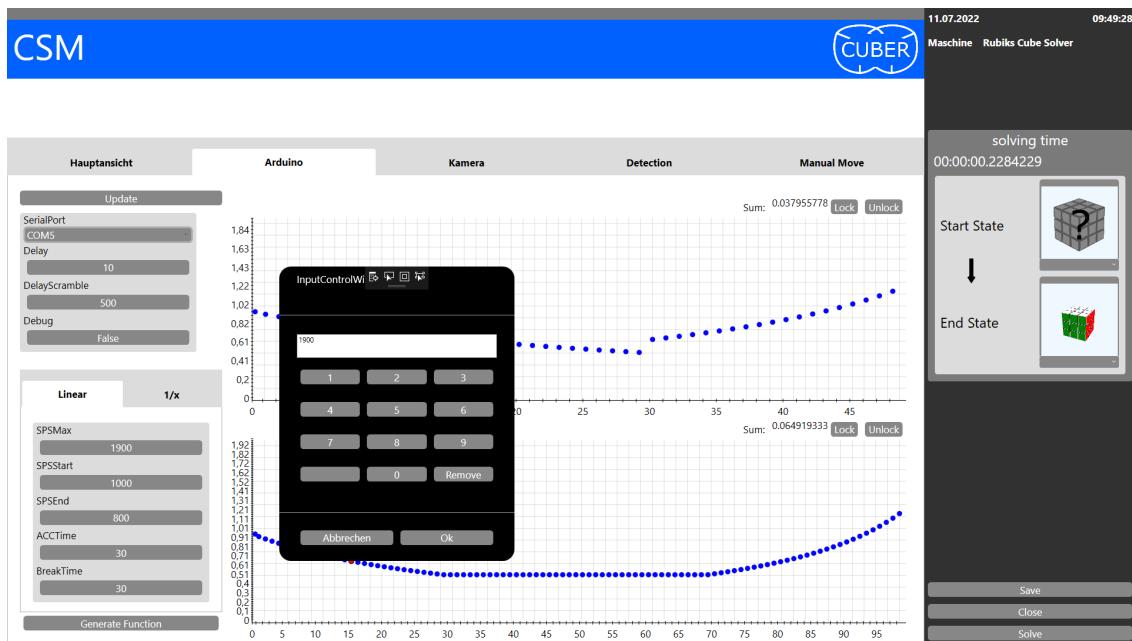


Abbildung 3.8: Arduino (Anpassung Parameter)

- ▶ „ACCTime“ setzt die Anzahl der Steps, die beschleunigt werden.
- ▶ „BreakTime“ setzt die Anzahl der Steps, die gebremst werden.
- ▶ „Generate Function“ generiert eine Funktion basierend auf den angegebenen Parametern und plottet diese.

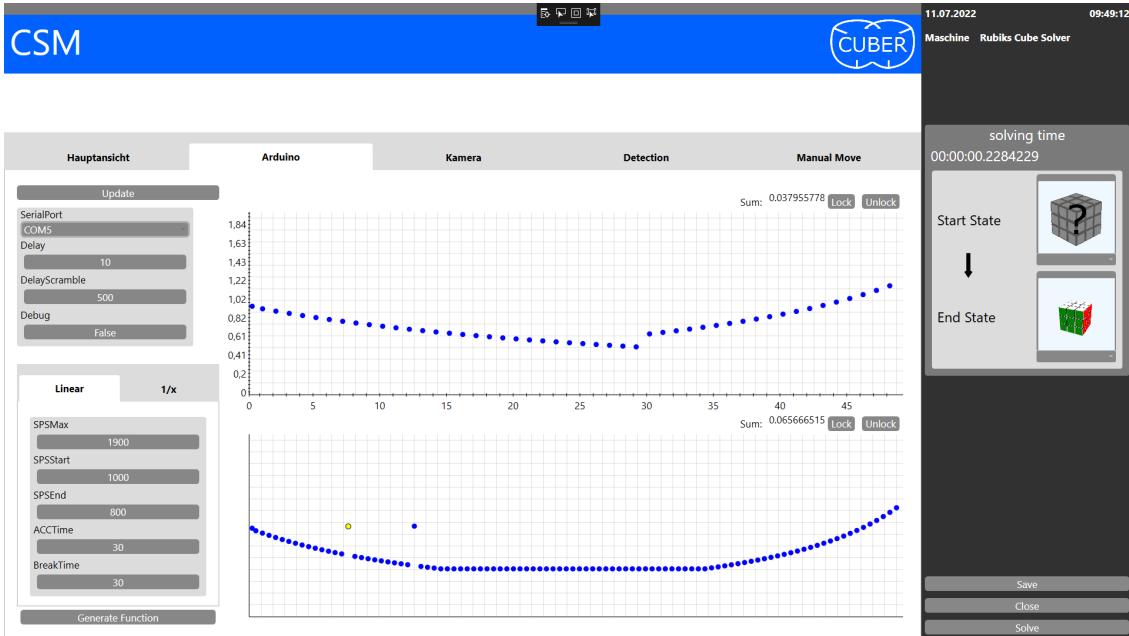


Abbildung 3.9: Arduino (Anpassung Schritte)

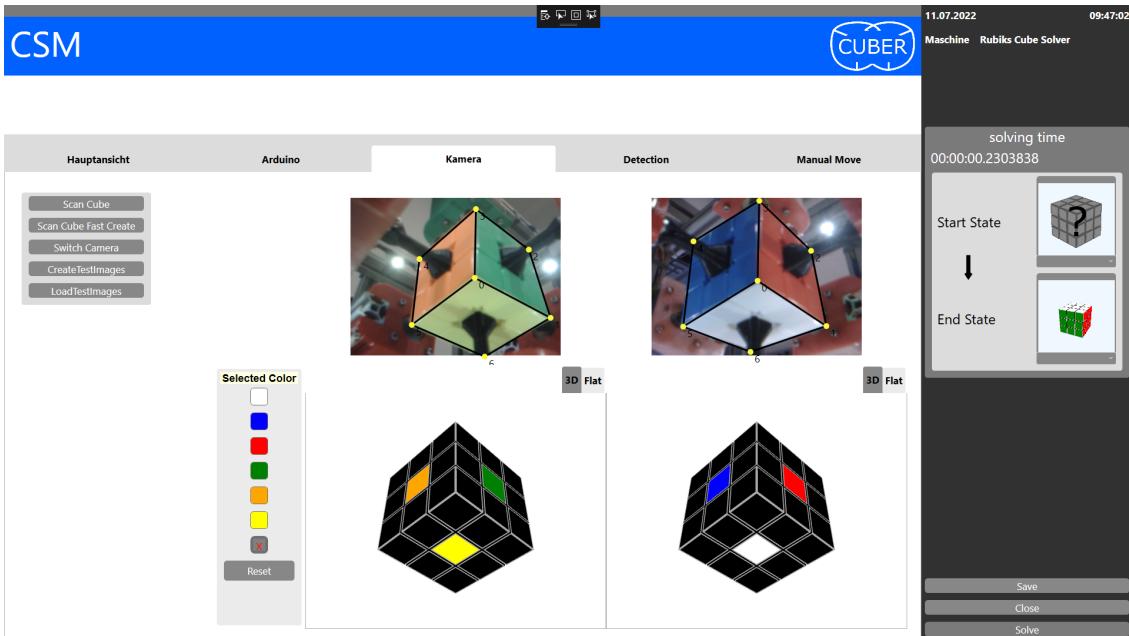


Abbildung 3.10: Kamera

### 3.2.3 Kamera

Im Kontrollfeld „Kamera“ sind mittig die Kameraansichten zu erkennen. Die gelben Punkte müssen in richtiger Reihenfolge (siehe Abb. 3.10) auf die Ecken des Würfels positioniert werden. Links kann unter verschiedenen Optionen gewählt werden:

- „Scan Cube“ scannt alle Würfelseiten mittels zwei Kameras (Abb. 3.11).
- „Scan Cube Fast Create“ scannt den Würfel mit einem anderen Algorithmus (nur zum Te-

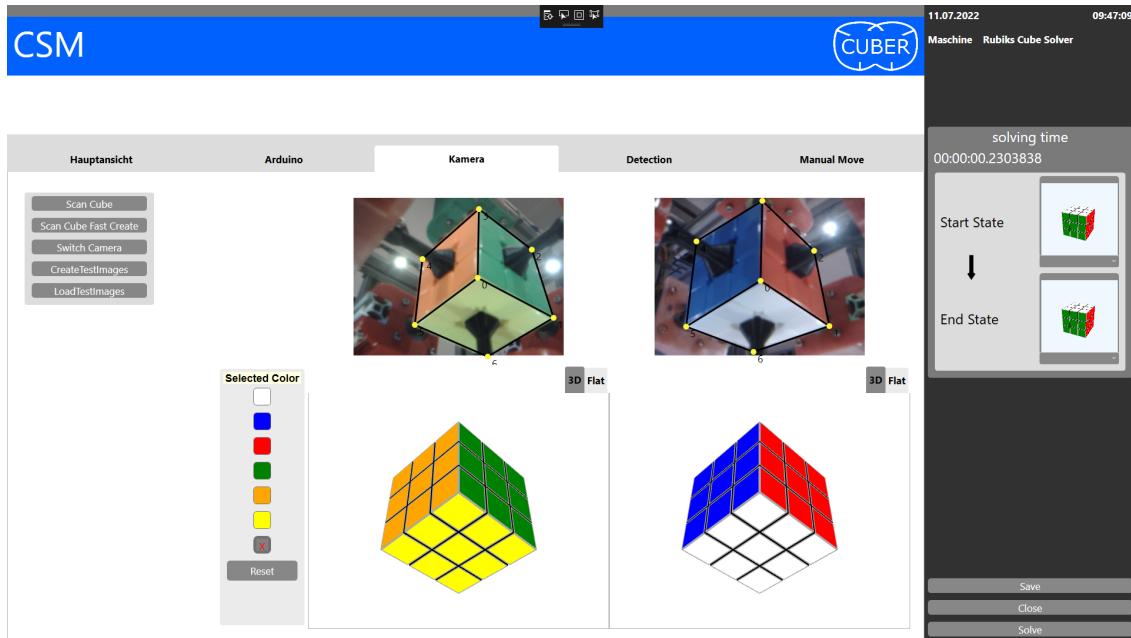


Abbildung 3.11: Kamera (Scan Cube)

sten).

- ▶ „Switch Camera“ tauscht die Fenster der Kameraansicht.
- ▶ „CreateTestImage“ erstellt Testbilder (macht zufällige Moves und speichert dann die Bilder der PS3-Kameras).
- ▶ „LoadTestImages“ lädt ein Bild simulativ (als würden die PS3-Kameras dieses gerade anzeigen).

Zum Einstellen des Würfels ist der „Start State“ als undefined auszuwählen. Der „End State“ kann je nach Anforderung gewählt werden (Abb. 3.12). Ist es beim Scannen zu einem Fehler gekommen, kann dieser manuell behoben werden (Abb. 3.13). Dazu ist unter „Selected Color“ die gewünschte Farbe und dann eine beliebige Fläche auszuwählen.

Ist die Verbindung zwischen Schaltschrank und Kameras nicht gegeben, erscheint die Fehlermeldung „No Signal“ (Abb. 3.14).

### 3.2.4 Detections

Im Kontrollfeld „Detections“ (Abb. 3.15) sind die Farben der Würfelflächen den Motoren zuzuordnen. Dabei ist zu beachten, dass nur lösbarer Möglichkeiten eingegeben werden dürfen.

Unter „Detection Method“ können „Own“ und „Rescan“ ausgewählt werden. „Own“ nutzt eine eigene Bilderkennung, die in Abschnitt 4 weiter ausgeführt wird. „Rescan“ nutzt die Bilderkennung von *efrantar/rcscan*<sup>1</sup>.

### 3.2.5 Manual Move

Im Kontrollfeld „Manual Move“ können die Seiten des Würfels manuell gedreht werden (Abb. 3.16). Links kann unter verschiedenen Optionen gewählt werden:

- ▶ Mit „Axis“ lassen sich die Achsen UP/DOWN, RIGHT/LEFT, FRONT/BACK auswählen.

<sup>1</sup><https://github.com/efrantar/rcscan> (Besucht: 02.08.2023)

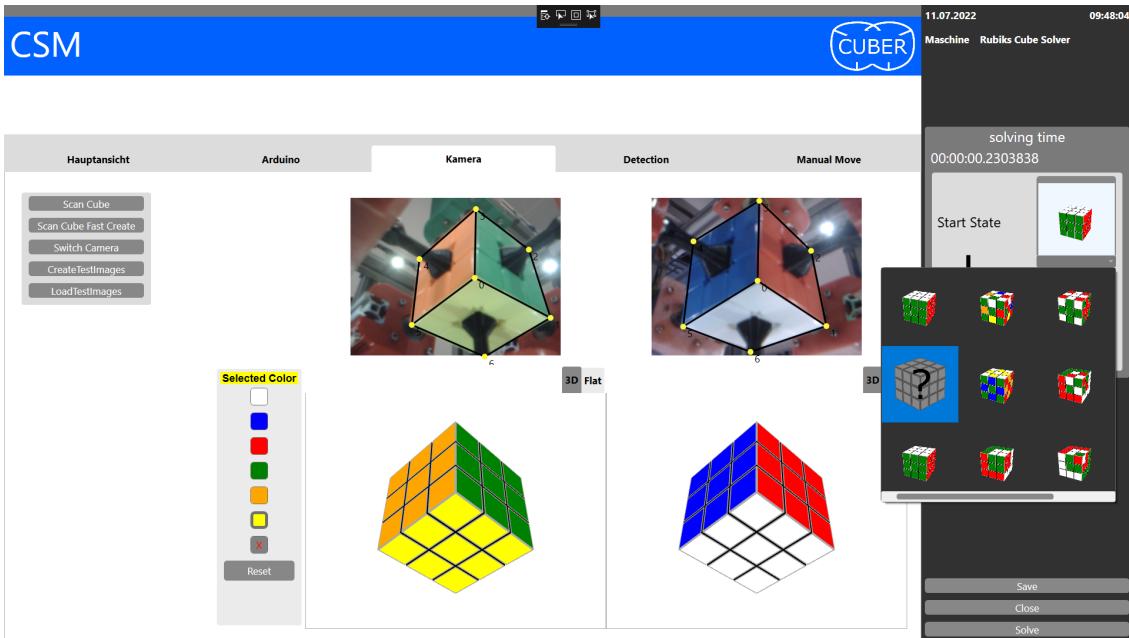


Abbildung 3.12: Kamera (Start & End State)

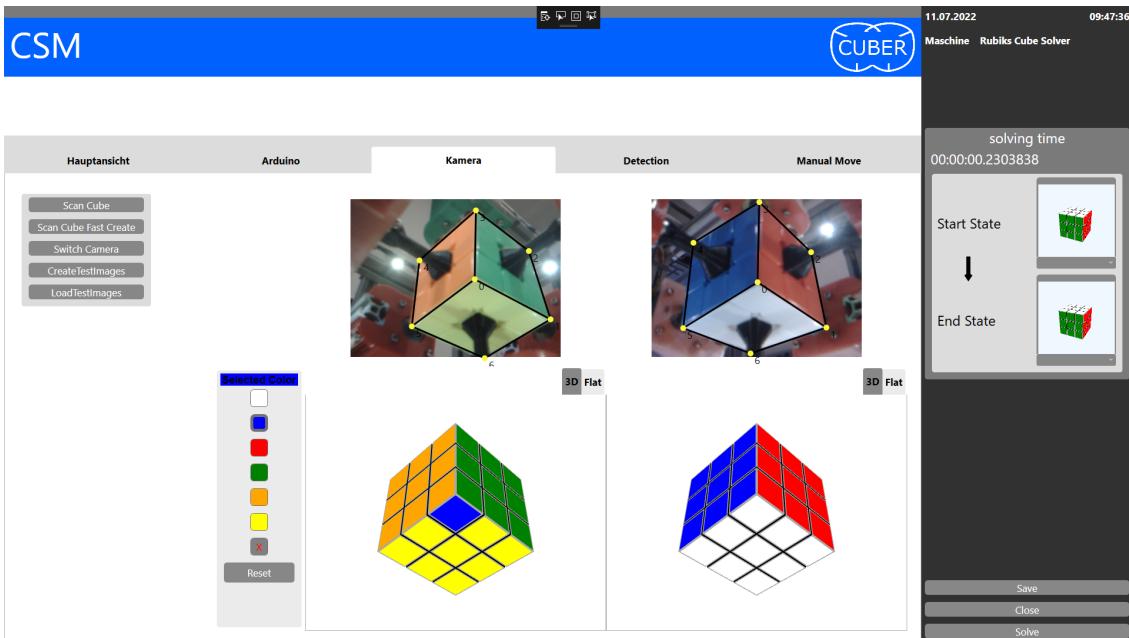


Abbildung 3.13: Kamera (Fehler korrigieren)

- Über „Side 1 Up/Right/Front“ kann für die jeweilige Seite eine 90°- bzw. 180°-Drehungen clockwise (cw) bzw. counter clockwise (ccw) ausgewählt werden.
- Über „Side 1 Down/Left/Back“ kann für die jeweilige Seite eine 90°- bzw. 180°-Drehungen clockwise (cw) bzw. counter clockwise (ccw) ausgewählt werden.
- „Manual Move“ startet die manuelle Drehung mit vorherigen Einstellungen.
- „Test Move“ wurde intern benutzt, um Züge, wie „U2R2U2R2“ zu testen. Das Feature wurde

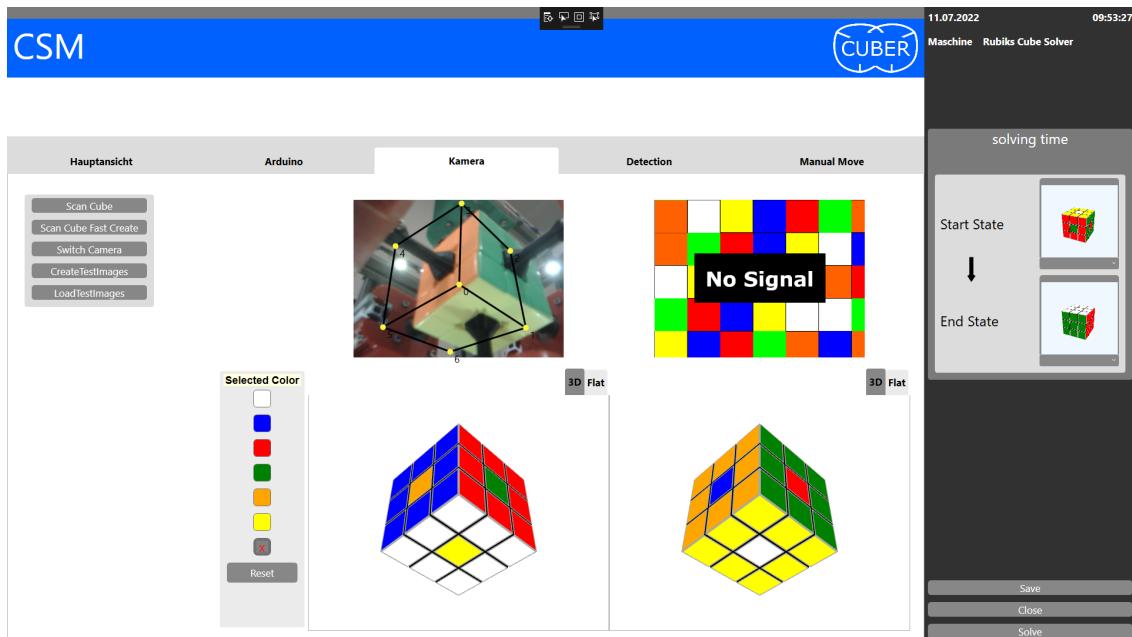


Abbildung 3.14: Kamera (No Signal)

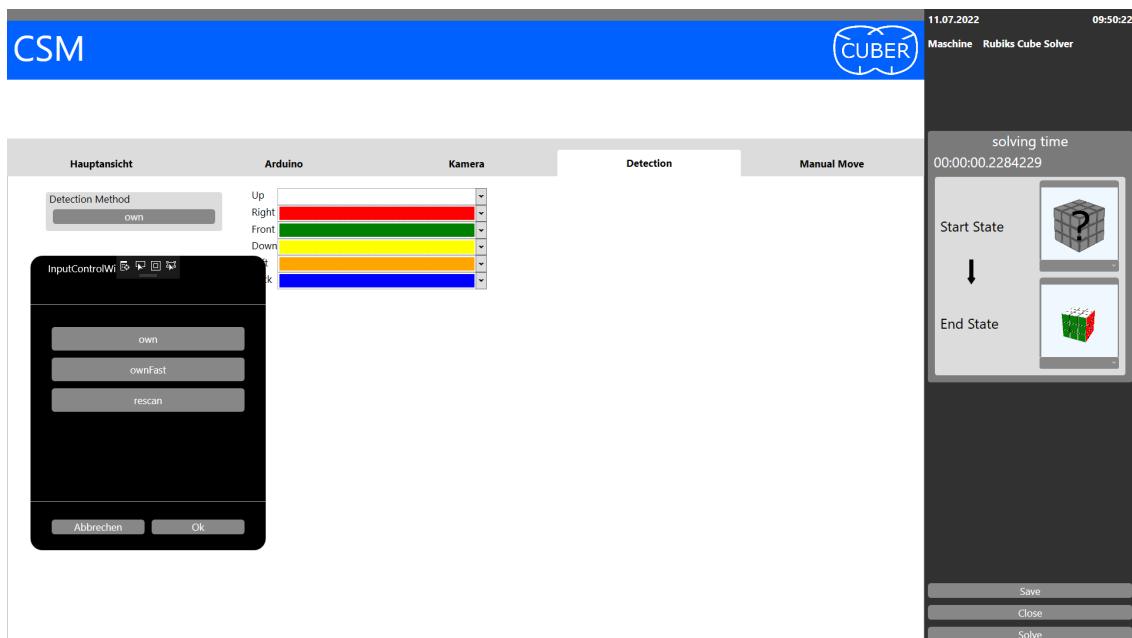


Abbildung 3.15: Detections

durch „Move-Zeile“ ersetzt.

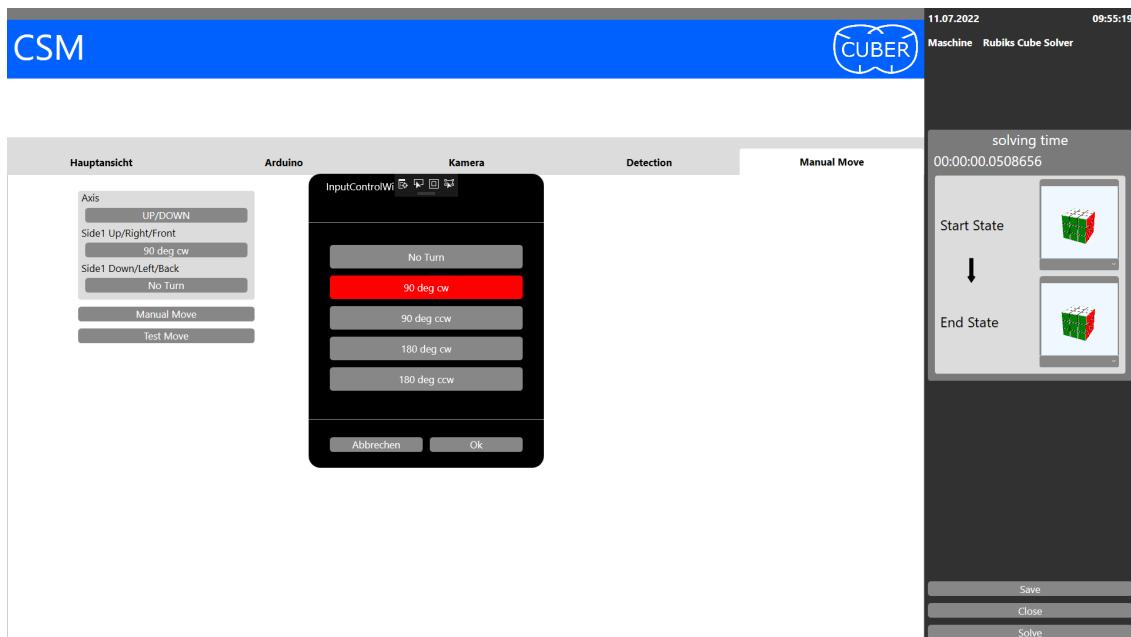


Abbildung 3.16: Manual Move

---

## Kapitel 4

### Bilderkennung

Im Folgenden werden folgende Würfelnamen definiert:

1. Mittelfläche
2. Kantenfläche
3. Eckfläche
4. Würfelseite

Eine Würfelseite besteht aus einer Mittelfläche, vier Kantenflächen und vier Eckflächen. Die Bilderkennung erzeugt aus zwei Bildern mit jeweils drei Würfelseiten (vgl. Abb. 4.1) ein 1x54 Array für die Zuordnung jeder Würfelfläche zu einer Farbe. Die Kodierung erfolgt nach dem gleichen Schema des Solvers von *efrantar/rob-twophase*<sup>1</sup>.

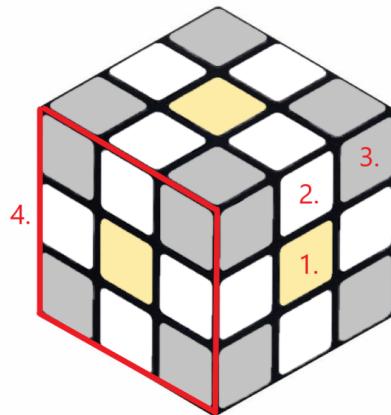


Abbildung 4.1: Aufbau Rubik's Cube

```
/*
 *      +---+
 *      | U1 | U2 | U3 |
 *      +---+---+---+
 *      | U4 | U5 | U6 |
 *      +---+---+---+
 *      | U7 | U8 | U9 |
 *      +---+---+---+
 *      | L1 | L2 | L3 | F1 | F2 | F3 | R1 | R2 | R3 | B1 | B2 | B3 |
 *      +---+---+---+---+---+---+---+---+---+---+---+---+
 *      | L4 | L5 | L6 | F4 | F5 | F6 | R4 | R5 | R6 | B4 | B5 | B6 |
 *      +---+---+---+---+---+---+---+---+---+---+---+---+
 *      | L7 | L8 | L9 | F7 | F8 | F9 | R7 | R8 | R9 | B7 | B8 | B9 |
 *      +---+---+---+---+---+---+---+---+---+---+---+---+
 *      | D1 | D2 | D3 |
 *      +---+---+---+
 *      | D4 | D5 | D6 |
 *      +---+---+---+
 *      | D7 | D8 | D9 |
 *      +---+---+---+
 */
```

Die Kodierung der Würfelfarben erfolgt in der Reihenfolge U, R, F, D, L, B, also ergibt sich insgesamt *U1U2U3U4U5U6U7U8U9R1R2...B8B9*. Als ASCII-String steht an jeder Stelle das Zeichen

<sup>1</sup><https://github.com/efrantar/rob-twophase> (Besucht: 02.08.2023)

”U”, ”R”, ”F”, ”D”, ”L” oder ”B” und beschreibt somit die Farbe welche Seite an dieser Stelle steht. Der gelöste Würfel hätte somit den nachfolgenden ASCII-String:

”UUUUUUUUURRRRRRRFFFFFFFFFFDDDDDDDDLLL LLLLBBBBBBBBB”.

## 4.1 Farbvereinfachung

Da der Würfel und die Kameras fest eingespannt sind, wird die Position/Lage des Würfels manuell festgelegt.

```
auto Dst1 = cv::Mat(IMAGE_SIZE, IMAGE_SIZE, CV_32FC3);

cv::Point2f source = {{p1x, p1y}, {p2x, p2y}, {p3x, p3y}, {p4x, p4y}};
cv::Point2f destination[] = {{0, IMAGE_SIZE}, {IMAGE_SIZE, IMAGE_SIZE}, {IMAGE_SIZE, 0}, {0, 0}};

const cv::Mat transform1 = getPerspectiveTransform(source, destination);
warpPerspective(img, Dst1, transform1, {IMAGE_SIZE, IMAGE_SIZE});
```

Mithilfe von *getPerspectiveTransform()* und *warpPerspective()* aus OpenCV erhält man aus den zwei schrägen Bildern sechs Bilder jeder Würfelseite. Aus den sechs Bildern werden dann in einem 3x3 Grid die Mittelwerte jeder Würfelfläche gebildet. So erhält man 54 RGB Farbwerte jeder kleinen Würfelfläche.

## 4.2 Farbzuzuordnung

Aus den 54 RGB Farbwerten muss nun die Zuordnung zu den sechs Seiten/Farben erfolgen. Anstatt alle Farben mit fixen Referenzfarben zu vergleichen (z. B. Rot = (255,0,0), alles was am nächsten an (255,0,0) ist wird Rot zugeordnet), wurde versucht in den 54 Farben direkt Farbgruppen zu suchen.

### 4.2.1 Ähnlichkeit zweier Farben

Man nehme zwei Farben im RGB Format  $c_1$  und  $c_2$ . Das Ziel ist es einen Wert zu erhalten, der beschreibt wie weit auseinander zwei Farben liegen. Die einfachste Methode ist der euklidischer Abstand

$$d(c_1, c_2) = \sqrt{(c_1^R - c_2^R)^2 + (c_1^G - c_2^G)^2 + (c_1^B - c_2^B)^2}.$$

Jedoch hat diese Methode im Tests zu nicht optimalen Ergebnisses geführt. Farben, die fürs menschliche Auge sehr nahe beieinander liegen, ergaben teilweise große Abstände. Farben, die fürs menschliche Auge unterschiedlich aussahen, ergaben im euklidischen Abstand teilweise niedrige Abstände.

Dieses Problem ist aufgrund der Wahrnehmung des Auges stark nicht-linear. Daher wird die Ähnlichkeit zweier Farben nach dem CIEDE2000 Verfahren durchgeführt. Diese basiert anstatt auf dem RGB-Farbraum, auf dem LAB-Farbraum und erzeugt plausiblere Ergebnisse.

### 4.2.2 Gruppierung

In dem RGB-Farbraum sollen sechs Gruppen von ähnlichen Farben gefunden werden. In Abbildung 4.2 ist die Gruppierung von sechs 4-er Gruppen dargestellt. Für den Menschen ist es recht einfach, die 4-er Gruppen zu bilden. In C++ muss allerdings über alle Farben iteriert, und diese einzelnen verglichen, werden. Zudem müssen Farben, die sehr dicht beieinander liegen, nicht zur gleichen Gruppe gehören.

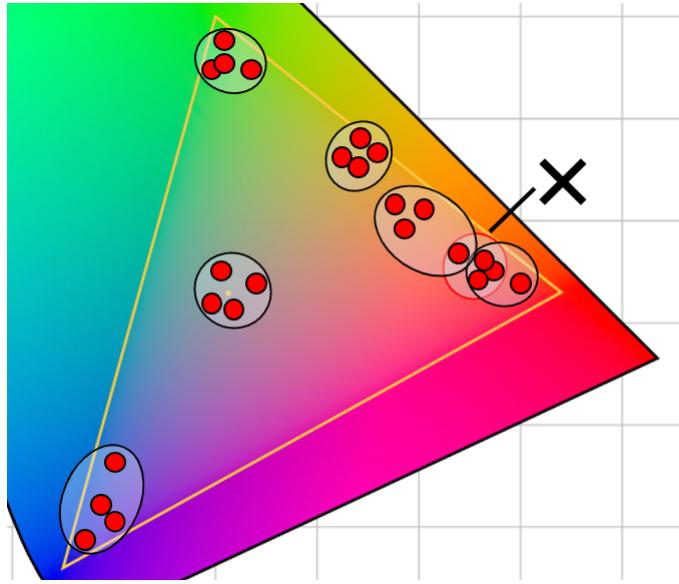


Abbildung 4.2: Gruppierung von Farben in sechs 4-er Gruppen

In Abbildung 4.2 liegen zwischen den Farben Orange und Rot vier Punkte, die sich im Vergleich zu allen anderen möglichen Gruppen am ähnlichsten sind. Allerdings ist diese Zuordnung falsch, da den anderen Gruppen dann Elemente fehlen. Um alle Elemente richtig zu gruppieren, wird somit nicht die beste einzelne Gruppe gesucht. Es muss die Summe der Unterschiedlichkeit aller sechs Gruppen minimiert werden.

Es sei  $g = g_1, g_2, g_3, \dots, g_m$  eine Gruppe von  $m$  Farben. Der Fehler  $err$  (d. h. wie unterschiedlich die Farben sind) der Gruppe wird dann folgendermaßen berechnet:

$$\begin{aligned}\mu &= \sum_{x=1}^m \frac{g[x]}{m} \\ err_1 &= \sum_{x=1}^m CIEDE2000(g[x], \mu)\end{aligned}$$

Der zu minimierende Fehler ist dann die Summe der Fehler jeder gebildeten Gruppe  $err_1$  bis  $err_6$ . Da man sechs Mal neun Farbwerte finden muss, wäre die erste Idee die besten sechs 9-er Gruppen aus 54 zu suchen. Hierbei ergeben sich jedoch viele nicht valide Werte. Zum Beispiel ist es nicht möglich, dass mehr als vier Kantenflächen einer Gruppe zugeordnet werden (es gibt genau vier Kantenflächen jeder Farbe).

Um das Problem zu vereinfachen, wird in der kleineren Kanten und Eckenmenge gesucht und die besten Ergebnisse dann wieder zusammengeführt. Anstatt sechs 9-er Gruppen werden somit zwei Mal sechs 4-er Gruppen gesucht (siehe Abbildung 4.2). Die Anzahl der Möglichkeiten, sechs Mal 4-er Gruppen auszuwählen, liegt bei  $\frac{24!}{6 \cdot 4!} = 4.3 \cdot 10^{21}$ . Diese Menge ist zu groß, um alle Möglichkeiten durchzugehen. Daher wird zunächst die Anzahl der Möglichkeiten, eine 4-er Gruppe auszuwählen, betrachtet. Die Anzahl der Möglichkeiten vier Kanten auszuwählen liegt bei  $\frac{24 \cdot 23 \cdot 22 \cdot 21}{4!} = 10626$ . Auch hierbei ergeben sich allerdings Fälle, die nicht valide sind. Beide Flächen einer Kante können z. B. nicht der gleichen Gruppe angehören (vgl. Abb. 4.3). Somit kann man die Anzahl der Möglichkeiten weiter reduzieren auf:  $\frac{24 \cdot 22 \cdot 20 \cdot 18}{4!} = 7920$  Möglichkeiten eine Gruppe auszuwählen.

Für alle diese Gruppenmöglichkeiten wird dann der  $err$  der Gruppe berechnet. Da es nicht möglich ist alle Möglichkeiten durchzugehen, wird angenommen, dass der beste globale Fehler ( $err_1 + err_2 + \dots + err_6$ ) überwiegend aus den besten Einzelfehlern ( $err$ ) besteht. Die 7920

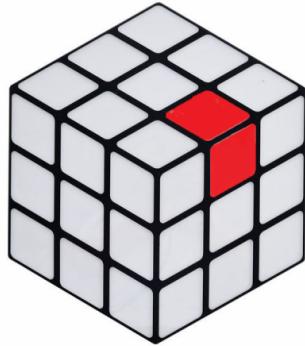


Abbildung 4.3: Ungültige Flächenzuordnung

Gruppenmöglichkeiten werden nach dem Fehler *err* sortiert und die besten  $n = 100$  Gruppen herausgesucht. Anschließend wird dann in den besten n 4-er Gruppen nach einer validen Lösungen gesucht, sodass sich keine Elemente der Gruppe überschneiden. (Angenommen Gruppe 1 wäre  $e_1 = [1, 3, 4, 22]$  und Gruppe 2  $e_2 = [4, 6, 7, 9]$ , dann wäre das keine mögliche Zusammenführung, da es eine Überschneidung der Fläche 4 gibt).

Nach dem gleichen Prinzip werden sechs 4-er Gruppen der Ecken gebildet. Diese werden anschließend mit den sechs Mitten zusammengeführt. Hierbei werden alle Möglichkeiten durchgegangen, sechs mit sechs Farben zuzuordnen ( $6! = 720$ ) und der beste Fehlerfall herausgesucht. Da die Mittelflächen von den Halterungen verdeckt sind, werden für die Mittelflächen fixe Referenzfarben angenommen. So kann definiert werden, wie der Würfel eingespannt ist. Bei den 4-er Gruppen wird der Durchschnitt der Farbe für die spätere Zuordnung verwendet.

### 4.3 Implementierung

Die Erstellung der Liste der besten 4-er Gruppen:

```
std::multimap<float, EdgeIndex> errors; //std::multimap is ordered by key!
for(unsigned char i1 = 0; i1 < 24; i1++){
    for(unsigned char i2 = i1 + 1; i2 < 24; i2++){
        for(unsigned char i3 = i2 + 1; i3 < 24; i3++){
            for(unsigned char i4 = i3 + 1; i4 < 24; i4++){
                EdgeIndex d = {i1, i2, i3, i4, ...};
                bool isValid = d.checkValid(isCorner); //e.g. check for same
                                                color on same Edge Faces
                if (isValid)
                {
                    auto dist = GetError(d, rgb); //SE like Error using
                                                ciede
                    errors.emplace(dist, d);
                }
            }
        }
    }
}
```

Die Suche der besten sechs 4-er Gruppen:

```

for (int a = 0; a < n; a++){
    float total1 = errorKey[a];
    if (total1 > bestError) continue;
    AllEdges isSetGlobalA;
    isSetGlobalA . SetEdgeIndex(errorValue[a]);
    for (int b = a + 1; b < n; b++){
        float total2 = total1 + errorKey[b];
        AllEdges isSetGlobalB = isSetGlobalA;
        if (!isSetGlobalB.IsValid(errorValue[b]) || total2 > bestError)
            continue;
        for (int c = b + 1; c < n; c++){
            ...
            for (int d = c + 1; d < n; d++){
                ...
                for (int e = d + 1; e < n; e++){
                    ...
                    //find last not set Indexes fIndexes
                    ...

                    const float totalError = total5 + GetError(fIndexes, rgb
                        );
                    if (totalError > bestError) continue;

                    EdgeIndex possibleState[6] = { errorValue[a], errorValue
                        [b] ,errorValue[c] ,errorValue[d] ,errorValue[e] ,
                        fIndexes };

                    if (checkGroupValid(...))
                    {
                        bestError = totalError;
                        result = possibleState;
                    }
                }
            }
        }
    }
}

```

## Kapitel 5

---

### Solver

Zum Erstellen des Algorithmus wird der Solver von *efrantar/rob-twophase*<sup>1</sup> verwendet. Dieser benötigt bei maximal 20 180°-Zügen weniger als 1 ms, um einem Lösungsalgorithmus zu finden. Um den Solver besser aus C# aufrufen zu können, wurde der Solver als \*.dll angepasst. Zudem wurde eine Funktion hinzugefügt, um zu einen bestimmten Rubik's Cube Zustand (z. B. ein Muster) zu lösen.

---

<sup>1</sup><https://github.com/efrantar/rob-twophase> (Besucht: 02.08.2023)

---

## Kapitel 6

### Kameratreiber

Die Kommunikation mit den PS3-Kameras erfolgt über eine als \*.dll angepasste Version von *inspirit/PS3EYEDriver*<sup>1</sup>. Dies ist ein C-Programm, welches über *Libusb* mit der Kamera kommuniziert. Damit ein Gerät über *Libusb* direkt angesprochen werden kann, muss für dieses ein Libusb-Treiber installiert werden. Dafür kann das Tool „zadig.exe“ verwendet werden.

---

<sup>1</sup><https://github.com/inspirit/PS3EYEDriver> (Besucht: 02.08.2023)

---

## **Kapitel 7**

### **Troubleshooting**

Nach einigen Tests wurden die Drehungen nicht mehr zu Ende ausgeführt. Es wurde festgestellt, dass sich die Schrauben der Motorhalterungen gelockert haben. Diese müssen regelmäßig nachgezogen oder ggf. eingeklebt werden. Bei der Verwendung der Visu ist die Skalierung auf 100% einzustellen, sonst kommt es in dieser zu Formatierungsfehlern.