

POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRONIKI I TECHNIK INFORMACYJNYCH
INSTYTUT AUTOMATYKI I INFORMATYKI STOSOWANEJ



PRACA DYPLOMOWA MAGISTERSKA

inż. Przemysław Krajewski

**System robotyczny na bazie ROS/OROCOS
układający kostkę Rubika**

Opiekun pracy:
dr inż. Tomasz Winiarski

Ocena pracy:

.....
Data i podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Streszczenie

Celem pracy było opracowanie systemu robotycznego potrafiącego ułożyć kostkę Rubika. Zadanie zostało podzielone na cztery części: pokazanie poszczególnych ścianek kostki kamerze, lokalizacja jej w obrazie, wyznaczenie układu kostki Rubika na podstawie kolorów pól oraz jej ułożenie. System składa się z dwóch modułów.

Jeden z nich jest odpowiedzialny za sterowanie manipulatorami IRp-6 w celu manipulacji kostką Rubika. Sterowanie odbywa się z wykorzystaniem systemu ROS oraz planera trajektorii ruchu OpenRAVE umożliwiający uniknięcie kolizji.

Drugim modułem jest system wizyjny, którego zadaniem jest lokalizacja kostki w obrazie i pobranie kolorów jej pól. Do przetwarzania obrazu wykorzystano programową strukturę ramową DisCODE oraz bibliotekę OpenCV.

Słowa kluczowe: manipulator IRp-6, kostka Rubika, DisCODE, OpenCV, OpenRAVE, OMPL

Abstract

Title: *Robotic system based on ROS/OROCOS solving Rubik's cube*

The aim of this thesis was to develop a robotic system able to solve Rubik's cube. This task has been divided into four parts: presenting each cube's face to the camera, locating cube in the image, determining Rubik's cube permutation using colors of tiles, solving cube. System consists of two parts.

One of them is responsible for controlling of IRp-6 manipulators in order to manipulate Rubik's Cube. Control is performed using ROS system and OpenRAVE motion planner enabling to avoid collisions.

The second part is a vision system which task is locating the cube in the image and extracting values of its colors. The DisCODE framework and OpenCV library are used for image processing.

Keywords: IRp-6 manipulator, Rubik's cube, DisCODE, OpenCV, OpenRAVE, OMPL

Spis treści

1 Wstęp	7
1.1 Wprowadzenie	7
1.2 Motywacja	7
1.3 Cel pracy	8
1.4 Układ pracy	8
2 Wykorzystane technologie	9
2.1 Wykorzystany sprzęt	9
2.1.1 Manipulatory Irp6	9
2.1.2 Kamera	9
2.1.3 Kostka Rubika	10
2.2 Wykorzystane oprogramowanie	11
2.2.1 ROS	11
2.2.2 IRPOS	12
2.2.3 DisCODE	12
2.2.4 OpenCV	15
2.2.5 OMPL	15
2.2.6 OpenRAVE	15
2.3 Wykorzystane algorytmy	15
2.3.1 Redukcja szumu w obrazie	15
2.3.2 Dylatacja i erozja	17
2.3.3 Wyrównywanie histogramu	17

2.3.4	Detekcja krawędzi	19
2.3.5	Detekcja konturów w obrazie	24
2.3.6	Algorytmy planowania trajektorii ruchu	25
2.3.7	Algorytmy układania kostki Rubika	25
3	Opis rozwiązania	27
3.1	Stanowisko badawcze	27
3.2	Struktura systemu	28
3.2.1	Moduł DisCODE	28
3.2.2	Moduł ROS	30
3.3	Lokalizacja kostki Rubika i wyznaczenie kolorów pól	31
3.3.1	Wstępna obróbka obrazu	31
3.3.2	Lokalizacja kostki Rubika w obrazie	35
3.4	Sterowanie manipulatorami	42
3.4.1	Kinematyka manipulatora Irp-6	42
3.4.2	Planowanie trajektorii ruchu	47
3.4.3	Proces manipulacji kostką	51
3.5	Obliczanie układu pól kostki Rubika	57
3.6	Układanie kostki Rubika	61
4	Wyniki testów	63
4.1	Etapy układania kostki Rubika	63
4.2	Serie testowe	65
4.3	Obserwacje i wnioski	66
4.4	Podsumowanie	68

Rozdział 1

Wstęp

1.1 Wprowadzenie

Robotyka bardzo szybko znalazła zastosowanie w różnych dziedzinach ludzkiego życia, a w szczególności w przemyśle. Roboty dzięki swojej sile, precyzji czy wytrzymałości są w stanie wykonywać niektóre czynności efektywniej niż człowiek. Większość maszyn jednak opiera się na dużej powtarzalności swojej pracy i wcześniej ustalonego środowiska. Bardziej złożone zadania wymagają zdolności obserwacji świata czy podejmowania decyzji. Jednym z ich przykładów jest manipulacja obiektem. Przed manipulacją przedmiotu należy najpierw go poddać analizie, w której można wykorzystać przetwarzanie obrazu. Dziedzina ta obejmuje takie problemy jak lokalizacja przedmiotów w obrazie czy ich analiza. Istnieją jednak sytuacje utrudniające bądź wręcz uniemożliwiające rozpoznanie obrazu, gdy np. przedmiot jest przesłonięty lub niedoświetlony. W celu uniknięcia tego problemu można za pomocą manipulatorów odpowiednio wyeksponować przedmiot kamerze. Sterowanie manipulatorami stanowi wsparcie dla przetwarzania obrazu oraz jest niezbędnym elementem manipulacji przedmiotem. Operując manipulatorami należy odpowiednio dobrą dla nich trajektorię ruchu pozwalającą uniknięcie kolizji. System jest bardziej elastyczny, gdy jest w stanie automatycznie wyznaczyć trajektorię. Dzięki takiemu podejęciu roboty są zdolne poruszać się w nieznanym wcześniej otoczeniu.

1.2 Motywacja

Roboty coraz częściej zastępują ludzi w ich codziennych obowiązkach. W związku z tym wymaga się od nich między innymi umiejętności analizy obiektów i odpowiedniej nimi manipulacji. W niniejszej pracy obiektem manipulacji jest kostka Rubika ze względu na jej reprezentatywność w robotyce usługowej.

1.3 Cel pracy

Celem pracy było stworzenie systemu robotycznego, który jest w stanie układać kostkę Rubika zakładając, że jeden z manipulatorów trzyma kostkę. Jest ona tak ustawiana, aby kamera przymocowana do drugiego manipulatora była w stanie obejrzeć przedmiot z wszystkich stron. Następnie kostka jest przekładana w końcówce manipulatora w celu obejrzenia wcześniej zasłoniętych przez chwytak pól. Po rozpoznaniu wszystkich pól wyznaczany jest układ kostki. Na koniec jest ona układana. W niniejszej pracy zakłada się, że dwa chwyty są wystarczające do określenia pełnego stanu kostki. Przy manipulacji obiektem określone są tylko docelowe pozycje w jakich manipulatory mają wykonywać operacje, natomiast trajektorie ruchu są wyznaczane automatycznie. Podczas przetwarzania obrazu priorytet stanowi bezbłędność, choć proces analizy powinien wykonywać się w rozsądny czasie zakładając, że w tle obiektu mogą występować inne przedmioty, lecz nie mogą intencjonalnie zakłócać algorytmu. Do realizacji tego projektu wykorzystano dwa zmodyfikowane manipulatory IRp-6.

1.4 Układ pracy

Niniejsza praca składa się z pięciu rozdziałów. Rozdział 2 opisuje oprogramowanie, sprzęt oraz algorytmy wykorzystane w pracy. W rozdziale 3 została omówiona realizacja projektu. W rozdziale 4 przedstawiono wyniki pracy.

Rozdział 2

Wykorzystane technologie

W niniejszym rozdziale został przedstawiony wykorzystany sprzęt (sekcja 2.1), oprogramowanie (sekcja 2.2) oraz algorytmy (sekcja 2.3).

2.1 Wykorzystany sprzęt

Sekcja ta zawiera opis zmodyfikowanych manipulatorów IRP-6 (sekcja 2.1.1), przymocowanej do jednego z nich kamery (sekcja 2.1.2), oraz kostki Rubika (sekcja 2.1.3).

2.1.1 Manipulatory Irp6

Do manipulacji kostką Rubika wykorzystano dwa zmodyfikowane manipulatory IRP-6 o nazwach *Track* i *Postument* (fotografia 2.1). Manipulatorem nazywamy urządzenia za pomocą, których można oddziaływać na środowisko. Każdy manipulator charakteryzuje przestrzeń robocza określająca, w której robot może pracować.

Robot *Postument* został zainstalowany na stacjonarnej platformie i posiada sześć stopni swobody. Natomiast *Track* znajduje się na torze jezdny umożliwiający przesuwanie się manipulatora wzdłuż jednej osi. Z tego powodu *Track* w sumie ma siedem stopni swobody. Każdy z manipulatorów posiada chwytkę, którym może podnosić przedmioty. Przy każdym chwytku umieszczone zostały kamery wykorzystane w tej pracy do analizy kostki. Dodatkowo końcówki manipulatorów posiadają czujniki siły będące w stanie rejestrować wywierane na nie siły wzdłuż oraz wokół osi *X*, *Y* oraz *Z*.

2.1.2 Kamera

W celu analizy kostki Rubika wykorzystano przymocowaną do chwytyka kamerę Point Grey Blackfly [4] (fotografia 2.2). Urządzenie to rejestruje obraz o rozdziel-



(a) Fotografia manipulatora IRP-6 Postument.



(b) Fotografia manipulatora IRP-6 Track.

Rysunek 2.1: Fotografie manipulatorów IRP-6

czości 1286x1024.



Rysunek 2.2: Fotografia kamery Point Grey Blackfly

2.1.3 Kostka Rubika

Badanym obiektem przez moduł, który powstał w ramach niniejszej pracy, jest kostka Rubika. Została ona wynaleziona przez Ernő Rubika w 1974 roku. W 1976 roku taką samą kostkę skonstruował i opatentował w Japonii Terutoshi Ishige.

Kostka Rubika składa się z dwudziestu sześciu sześcianów połączone przegubem. Przegub ten umożliwia obracanie dowolnej warstwy wokół osi prostopadłej do danej warstwy oraz przechodzącej przez przegub. Zabawa kostką polega na obrocie poszczególnych warstw w taki sposób, aby wszystkie pola na poszczególnych ścianach kostki miały identyczny kolor.

Każda kostka posiada charakterystyczne dla niej parametry jak: kolory pól, potrzebna siła do obrotu warstw, metoda wykonania. Kostki wysokiej klasy wykorzystywane przez zawodowców do układania jej na czas niekoniecznie są optymalne

dla manipulatorów. Zabawka powinna mieć jasne kolory, aby mogła być łatwiejsza w analizie. Kamery czy algorytmy przetwarzające obraz nie są na razie tak doskonałe jak ludzkie oko. Warstwa, podczas poruszania, powinna stawać lekki opór, gdyż w przeciwnym razie może się samoistnie obrócić i podczas obracania innej warstwy manipulator może zniszczyć kostkę. Wybierając zabawkę należy także zwrócić uwagę na metodę jej wykonania. Najtańsze kostki posiadają przyklejane pola, które z czasem mogą odpadać. Te droższe posiadają dodatkową ochronną folię, która przy złym sterowaniu może zostać zniszczona lub całkiem zerwana. Istnieją także kostki, których pola wykonane są z jednolitego plastiku. Drobne obtarcia nie wpływają na jakość takiej zabawki, jednak silniejsze uderzenie i tak może ją uszkodzić. Stan pól może mieć znaczący wpływ na proces przetwarzania obrazu. Nieuyszkozone pola są łatwiejsze w analizie. Nie należy zapominać, że człowiek oglądający kostkę wykorzystuje inne elementy oprócz wzroku, np. dotyk. Podczas testowania oprogramowania niejednokrotnie dojdzie do uszkodzenia przedmiotu i trudno jest dobrać właściwą kostkę. W niniejszej pracy w początkowej fazie testowania wykorzystywano tańsze zabawki. Gdy manipulatory przestały niszczyć kostki zaczęto stosować te wytrzymalsze.

2.2 Wykorzystane oprogramowanie

Niniejsza sekcja zawiera opis platformy programistycznej ROS (sekcja 2.2.1), platformy programistycznej DisCODE (sekcja 2.2.3), biblioteki OpenCV (sekcja 2.2.4), biblioteki OMPL (sekcja 2.2.5) oraz symulator OpenRAVE (sekcja 2.2.6).

2.2.1 ROS

System odpowiedzialny za sterowanie manipulatorami funkcjonuje na bazie ROS [6] (Robotic Operating System). Jest to platforma programistyczna wspomagająca tworzenie oprogramowania dla robotów. Struktura aplikacji opiera się na grafie składającym się z węzłów, które mogą przesyłać między sobą dowolne dane. Taka budowa systemu ułatwia różnym zespołom na świecie dzielenie się swoimi komponentami. ROS został wydany na licencji BSD oraz wolnego oprogramowania.

Aplikacja ROS składa się z węzłów, które dla uproszczenia ich użycia znajdują się w pakietach. Każdy węzeł jest odpowiedzialny za realizację konkretnego zadania, np. komunikacji ze sterownikami czy wizualizacji wyników. Węzły mogą komunikować się ze sobą na dwa sposoby. Pierwszy z nich odbywa się za pomocą tematów (topic), czyli buforów, w których wszystkie węzły mogą umieszczać dane bądź je pobierać. Ten sposób komunikacji pozwala na wysyłanie informacji między wieloma węzłami. Drugi sposób opiera się na obsłudze (service). Polega na wysyłaniu między węzłami

żądań (request) i odpowiedzi (response). Ten sposób pozwala na komunikację między dwoma węzłami. Wykonując dowolny program można pojedyńczo uruchamiać każdy węzeł, ale także można stworzyć specjalny skrypt (launchfile), który pozwala na automatyczne stworzenie i konfigurację węzłów.

2.2.2 IRPOS

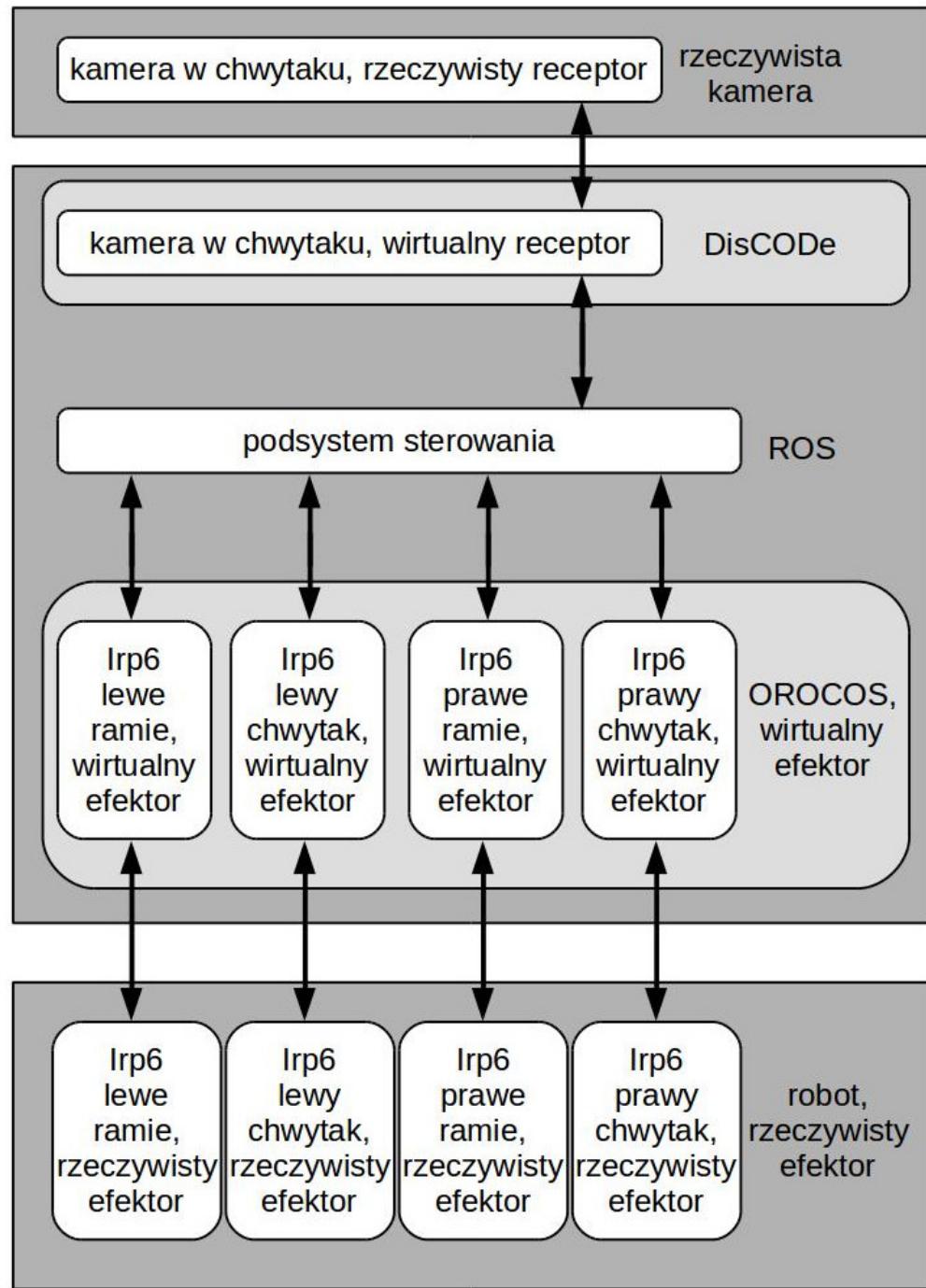
Sterowanie manipulatorami IRp-6 odbywa się z wykorzystaniem systemu IRPOS [7]. Składa się z podsystemów odpowiedzialnych za efektory i receptory rzeczywiste, jak i wirtualne oraz sterowanie. Rzeczywistymi receptorami są między innymi kamery przymocowane do chwytek manipulatorów, natomiast wirtualne receptory stanowią algorytmy przetwarzające dane (np. lokalizacja przedmiotów) realizowane przez DisCODE. Właściwe sterowanie rzeczywistymi efektorami odbywa się przez niskopoziomowe sterowniki pozycji wykorzystujące enkodery. Zadane pozycje są wyznaczane na podstawie wyjść wirtualnych efektorów wykorzystujących OROCOS. OROCOS dostaje polecenia od podsystemu sterowania. Strukturę systemu przedstawia ilustracja 2.3.

W celu wykonania dowolnego programu korzystającego z manipulatorów IRp-6 należy najpierw uruchomić komputer (serwer *gerwazy*) oraz podłączone do niego sprzętowe sterowniki robotów. Następnie po zalogowaniu się na serwer trzeba wykonać dwa skrypty (launchfile), które uruchamiają cały system. Na koniec należy uruchomić węzeł odpowiedzialny za realizację konkretnego zadania.

2.2.3 DisCODE

Podczas implementacji algorytmów lokalizujących kostkę Rubika wykorzystano DisCODE [5]. Jest to platforma programistyczna służąca do tworzenia oprogramowania przetwarzającego obraz. Pozwala w łatwy sposób zarządzać wątkami aplikacji, strumieniami danych, zdarzeniami oraz umożliwia podział programu na komponenty.

Aplikacja realizowana w DisCODE jest podzielona na komponenty, które są odpowiedzialne za pojedyncze operacje, np. desaturację obrazu czy jego rozmycie. Dla łatwiejszej nawigacji w projekcie umieszczono je bibliotekach zwanych DCL (DisCODE Component Library). Wszystkie komponenty komunikują się za pomocą strumieni danych. Za ich pomocą komponenty przesyłają między sobą wyniki kolejnych etapów przetwarzania danych. Komunikacja może także się odbywać za pomocą zdarzeń. Są one wykorzystywane do przesyłania między sobą informacji o określonych sytuacjach, lecz głównie wykorzystuje się strumienie danych. Istnieją biblioteki z gotowymi komponentami realizującymi typowe zadania. W niniejszej pracy wykorzystano DCL jak: *CvCoreTypes*, *CameraPGR*, *CvBasic* oraz *ROSProxy*. Oprócz

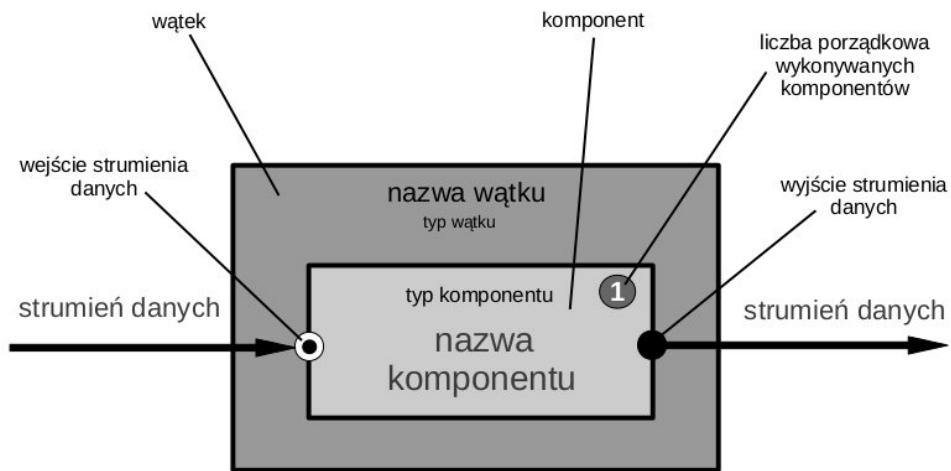


Rysunek 2.3: Struktura agenta IRPOS

definicji komponentów, DisCOPe potrzebuje pliku konfiguracyjnego, który określa z jakich komponentów składa się aplikacja i w jaki sposób są podłączone strumienie danych.

Każdy komponent posiada fragment kodu, który jest wywoływany przez egzekutor. Egzekutorem nazywamy wątek, który jest odpowiedzialny za realizację przypisanych mu komponentów, informowanie o zdarzeniach oraz synchronizację z komponentami innych egzekutorów. Jeden komponent (główny komponent) jest bez-

pośrednio wywoływanego przez egzekutor. Pozostałe komponenty są wykonywane za pośrednictwem zdarzeń. Najczęściej głównym komponentem jest ten, który zbiera dane z urządzenia zewnętrznego (np. kamery). Następnie wykonywane są komponenty posiadające niepusty wejściowy strumień danych, bądź otrzymały sygnał zdarzenia. Egzekutor może pracować w trybie cyklicznym (realizuje się raz na określony czas) bądź ciągłym. Definicja przynależności komponentów do egzekutora znajduje się w pliku konfiguracyjnym.



Rysunek 2.4: Schemat diagramu komponentów

Struktura komponentu została przedstawiona na schemacie 2.4. Ze względu na funkcje jakie mogą pełnić komponenty możemy je podzielić na:

- Źródła (sources) - pobierają informacje zewnętrznych urządzeń (np. kamera czy klawiatura) lub plików. Tego typu komponent zawsze zawiera przynajmniej jeden strumień wyjściowy.
- Przetwórcy danych (data processors) - przetwarzają dane jakie otrzymują na strumieniu wejściowym (od innych komponentów). Przetworzone dane kierowane są na strumień wyjściowy.
- Ujścia (sinks) - wysyłają przetworzone dane do urządzeń zewnętrznych przeznaczonych do prezentacji wyniku (np. ekran, bądź plik). Tego typu komponent zawsze zawiera przynajmniej jeden strumień wejściowy.
- Pośrednicy (proxies) - odpowiadają za komunikację między innymi modułami.

2.2.4 OpenCV

Do przetwarzania, obrazu pobieranego przez kamerę, wykorzystano bibliotekę OpenCV [1] (Open Source Computer Vision). Służy ona do tworzenia algorytmów przetwarzających obraz w czasie rzeczywistym. Wspiera takie języki programowania jak: C, C++, Java (Android), Python. Działa pod systemami operacyjnymi jak: Windows, Linux, Android, iOS, Mac OS. Biblioteka zawiera ponad dwa tysiące pięćset zoptymalizowanych algorytmów odpowiedzialnych między innymi za: zmianę formatu obrazu, detekcję krawędzi, szukanie prymitywów w obrazie, rysowanie figur.

2.2.5 OMPL

OMPL [2] (Open Motion Planning Library) jest biblioteką stanowiącą zbiór algorytmów planowania trajektorii ruchu manipulatorów. Biblioteka ta nie zawiera sama w sobie funkcji związanych z detekcją kolizji czy wizualizacją. Taka decyzja pozwala na łatwą integrację z innymi systemami np. OpenRAVE. OMPL został napisany w C++, ale także posiada Python'ową oprawę (binding).

2.2.6 OpenRAVE

Do wyznaczania trajektorii ruchu manipulatorów wykorzystano OpenRAVE [3] (Open Robotics Automation Virtual Environment). Jest to symulator służący do tworzenia, testowania czy wykorzystywania planera trajektorii ruchu. Zawiera funkcje wyznaczające kinematykę manipulatorów oraz ich kolizje. Do symulatora można dodatkowo zainstalować zewnętrzny silnik fizyczny (np. ODE czy Bullet), ale nie jest on niezbędny. OpenRAVE umożliwia wstawianie własnych obiektów czy robotów opisanych w formacie COLLADA (COLLAborative Design Activity). Istnieje też możliwość tworzenia własnych wtyczek. Symulator może działać w środowisku ROS.

2.3 Wykorzystane algorytmy

W niniejszej sekcji zostały przedstawione wykorzystane algorytmy związane z: redukcją szumu w obrazie (sekcja 2.3.1), dylatacją i korozją (sekcja 2.3.2), wyrównywaniem histogramu (sekcja 2.3.3), funkcją Canny (sekcja 2.3.4), detekcją konturów w obrazie (sekcja 2.3.5), planowaniem trajektorii ruchu (sekcja 2.3.6).

2.3.1 Redukcja szumu w obrazie

Obraz rejestrowany przez kamerę obarczony jest szumem, który objawia się losowym umiejscowieniem zmian jasności i koloru pikseli. Może być powodowany wadami

matrycy cyfrowej, zbyt wysoką czułością matrycy bądź wyraźną zmianą temperatury matrycy. Istnieje możliwość zredukowania w pewnym stopniu szumu np. za pomocą filtra uśredniającego bądź filtra Gaussa.

Filtr uśredniający

Jedną z najprostszych metod redukcji szumu jest filtr uśredniający [8]. Polega na wyznaczeniu dla każdego piksela średniej wartości z jego sąsiedztwa. Sąsiedztwem określamy macierz pikseli o wymiarach $n \times n$ (gdzie $n = 2m + 3, m \in N$). Jedną z przykładowych masek wykorzystywanych do realizacji filtra uśredniającego przedstawia ilustracja 2.5.

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

Rysunek 2.5: Maska 3×3 filtra uśredniającego

Filtr Gaussa

Sposób redukcji szumu przez filtr Gaussa [8] jest podobny do filtra uśredniającego. Różnica polega na innym doborze wag dla każdego piksela. Wagi te są dobierane na podstawie dwuwymiarowej funkcji Gaussa postaci 2.1 tak, aby maska filtra 2.6 przypominała rozkład Gaussa.

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) \quad (2.1)$$

(x, y) - współrzędne punktu, dla którego obliczana jest waga przy wyznaczaniu wartości punktu $(0,0)$,

$G(x, y)$ - wartość wagi w punkcie (x, y) ,

σ^2 - wariancja.

$\frac{1}{273} *$	1	4	7	4	1
	4	16	26	16	4
	7	26	41	26	7
	4	16	26	16	4
	1	4	7	4	1

Rysunek 2.6: Implementacja filtra Gaussa w postaci maski 5×5

Filtr medianowy

Filtr medianowy [8] jest metodą redukcji szumu, której zasada działania różni się od filtru rozmywającego. Polega na wyznaczeniu mediany z sąsiedztwa pikseli, czyli uszeregowania ich rosnąco i wyznaczenia środkowej wartości.

2.3.2 Dylatacja i erozja

Innym sposobem redukcji szumu jest dylatacja i erozja [8]. Algorytm ten jest głównie przeznaczony do monochromatycznych obrazów, czyli posiadających tylko dwie barwy. Ich działanie polega na sprawdzeniu dla każdego piksela sąsiedztwa i odpowiednie ustawienie punktu. Sąsiedztwem nazywana jest macierz pikseli o rozmiarze $n \times n$ (gdzie $n = 2m + 3, m \in N$).

Erozja polega na przeszukaniu lokalnego sąsiedztwa każdego piksela i nadaniu punktowi najwyższej, znalezionej wartości. Dla obrazów monochromatycznych piksel jest ustawiany na „zero”, gdy chociaż jeden piksel z sąsiedztwa jest „zgaszony”. W wyniku takiego działania znikają drobne zakłócenia, lecz kosztem pikseli należących do badanego obiektu.

Dylatacja ma odwrotny efekt niż erozja, choć zasada działania jest podobna. Polega na znalezieniu w sąsiedztwie najwyższej wartości. W obrazie monochromatycznym ustawiane są te piksele, które mają w swoim otoczeniu chociaż jednego „niezerowego” sąsiada. Na rysunku 2.7 został przedstawiony wynik zastosowania dylatacji i erozji.

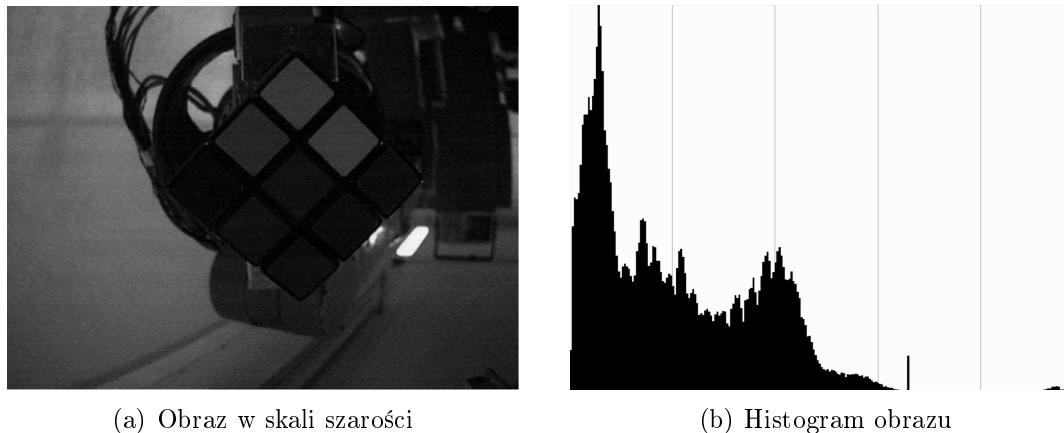


Rysunek 2.7: Efekt zastosowania dylatacji i erozji.

2.3.3 Wyrównywanie histogramu

Kamera rejestrująca obraz potrzebuje odpowiedniego naświetlenia matrycy, aby właściwie odwzorować rozkład oświetlenia w otoczeniu. Niejednokrotnie dochodzi do niedoświetlenia bądź prześwietlenia komórki światłoczułej, co prowadzi do powstania kontrastu, którego sensor nie jest w stanie obsłużyć. Sytuacja taka objawia

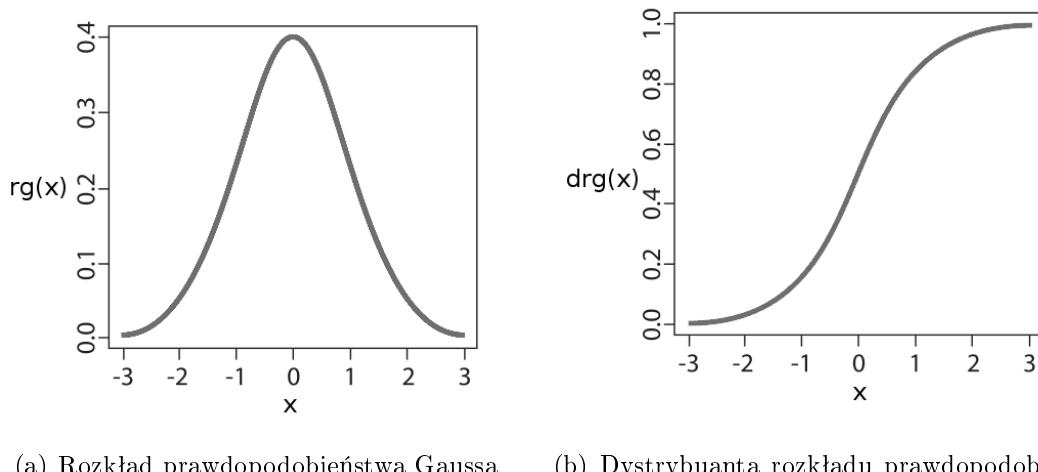
się zaciemnionym obrazem bądź jego przepaleniem, czyli pojawiają się odpowiednio ciemne lub jasne obszary.



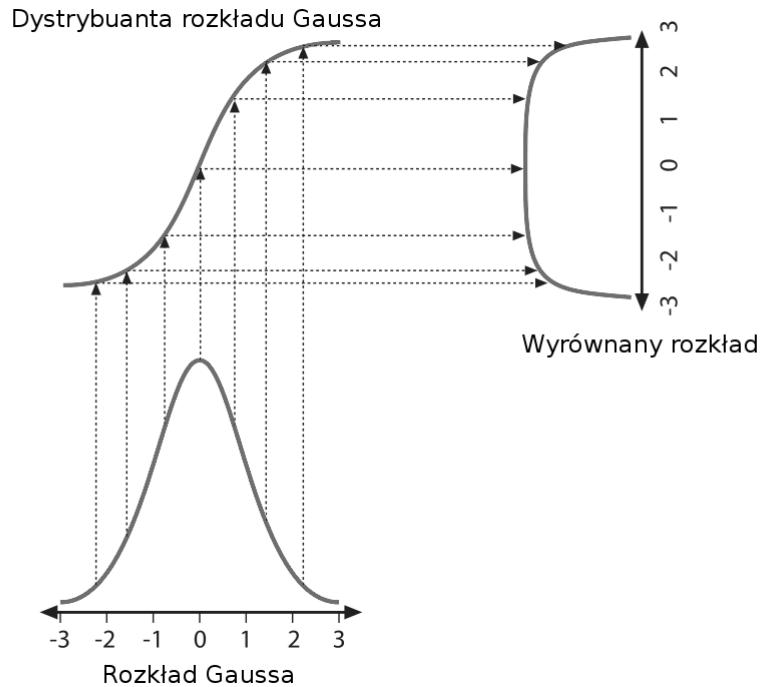
Rysunek 2.8: Zestawienie obrazu w skali szarości z jego histogramem

Można zredukować ten efekt wyrównując histogram [8]. Z ilustracji 2.8 wynika, że obraz nie wykorzystuje pełnego zakresu jasności, co wpływa na jego czytelność.

Wyrównywanie histogramu polega na przekształceniu pikseli tak, aby histogram wykorzystywał pełen zakres wartości. Operacja ta sprowadza się do mapowania jednego rozkładu pikseli w inny wykorzystując odpowiednią funkcję mapującą, czyli zależność określającą jaką wartość przypisuje się pikselom o danej, starej wartości. W tym algorytmie funkcją mapującą jest dystrybuanta histogramu przetwarzanego obrazu. Na ilustracji 2.9 zestawiony został rozkład prawdopodobieństwa Gaussa z jego dystrybuantą. Stosując dystrybuantę możliwe jest mapowanie histogramu w histogram, który posiada równomierny rozkład, przydzielając każdączęstość występowania piksela do odpowiedniej jasności zgodnie z ilustracją 2.10.

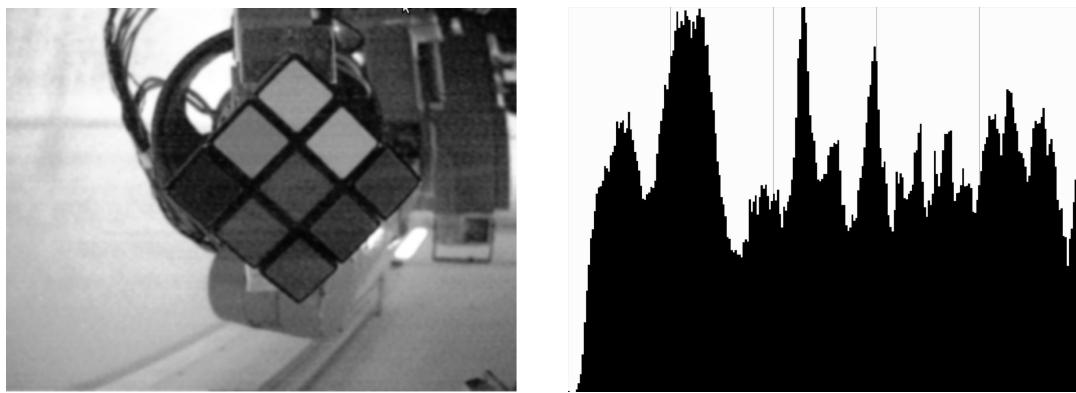


Rysunek 2.9: Zestawienie rozkładu Gaussa z jego dystrybuantą



Rysunek 2.10: Przykładowy sposób wyrównywania rozkładu Gaussa wykorzystując jego dystrybuantę

Posiadając ciągłą dystrybuantę histogramu można wyznaczyć nowy rozkład pikseli, lecz dla dyskretnej można uzyskać jedynie jej aproksymację. Wyrównywanie histogramu powoduje przekształcenie obrazu z ilustracji 2.8 w obraz na rysunku 2.11.



(a) Obraz w skali szarości po wyrównaniu jego histogramu

(b) Histogram obrazu

Rysunek 2.11: Zestawienie obrazu w skali szarości po wyrównaniu jego rozkładu pikseli z jego histogramem

2.3.4 Detekcja krawędzi

Jedną z użytecznych informacji jakie zawiera obraz jest rozkład krawędzi, czyli raptownych zmian funkcji obrazu. Algorytmy wykrywające krawędzie mogą polegać

na lokalizacji maksimów bądź minimów lokalnych pierwszej pochodnej funkcji obrazu czy zerowych wartości drugiej pochodnej funkcji obrazu [8].

$$\nabla f(x, y) = \begin{pmatrix} f_x(x, y) \\ f_y(x, y) \end{pmatrix} = \begin{pmatrix} \frac{\partial f(x, y)}{\partial x} \\ \frac{\partial f(x, y)}{\partial y} \end{pmatrix} \quad (2.2)$$

gdzie:

∇f - pochodna z dwuwymiarowej funkcji f ,

f_x - pochodna z funkcji f po x ,

f_y - pochodna z funkcji f po y .

Aproksymacja gradientu dyskretnymi różnicami dla funkcji dyskretnej $\hat{f}(x, y)$ będącej aproksymacją funkcji ciągłej $f(x, y)$ jest postaci:

$$\hat{f}_x(i, j) = \hat{f}(i + 1, j) - \hat{f}(i, j) \quad (2.3)$$

$$\hat{f}_y(i, j) = \hat{f}(i, j + 1) - \hat{f}(i, j) \quad (2.4)$$

$$\hat{f}_x(x, y) = \lim_{h \rightarrow 0} \frac{\hat{f}(x + h, y) - \hat{f}(x, y)}{h} \quad (2.5)$$

$$\hat{f}_y(x, y) = \lim_{h \rightarrow 0} \frac{\hat{f}(x, y + h) - \hat{f}(x, y)}{h} \quad (2.6)$$

gdzie:

\hat{f} - dyskretna funkcja będąca aproksymacją funkcji ciągłej f ,

\hat{f}_x - dyskretna pochodna z funkcji \hat{f} po x ,

\hat{f}_y - dyskretna pochodna z funkcji \hat{f} po y ,

W wyniku tej operacji powstaje amplituda krawędzi (s lub s') i kierunek krawędzi (r)

$$s = \sqrt{f_x^2 + f_y^2} \quad \text{lub} \quad s' = |f_x| + |f_y|, \quad r = \arctg\left(\frac{f_y}{f_x}\right) \quad (2.7)$$

gdzie:

s - rozkład amplitudy („stromości”) krawędzi w funkcji obrazu wyznaczany jako odległość euklidesowa,

s' - rozkład amplitudy („stromości”) krawędzi w funkcji obrazu wyznaczany jako odległość Manhattan,

r - rozkład kierunku krawędzi w funkcji obrazu,

f_x - pochodna funkcji f po x ,

f_y - pochodna funkcji f po y .

Kierunek r odpowiada wektorowi normalnemu w każdym punkcie linii konturu w obrazie. Omówione w niniejszej pracy algorytmy wyznaczania krawędzi zwracają obraz, w którym piskle o większych wartościach reprezentują większy „spadek” w funkcji obrazu.

Krzyż Robertsa

Jedną z najwcześniejszych i najprostszych metod wykrywania krawędzi jest krzyż Robertsa [8]. Operator ten wykrywa krawędzie o amplitudzie z zakresu (0, 255) dla ośmiorozowowych obrazów, zatem sposób prezentacji obrazu wejściowego i wyjściowego może być tak sam. Krzyż Robertsa polega na wyznaczeniu dla każdego piksela dyskretnych różnic wzduż dwóch prostopadłych kierunków, zatem jego realizacja opiera się na dwóch równaniach:

$$k_x(i, j) = f(i, j) - f(i + 1, j + 1) \quad (2.8)$$

$$k_y(i, j) = f(i + 1, j) - f(i, j + 1) \quad (2.9)$$

gdzie:

f - funkcja obrazu dla której wyznaczane są dyskretne różnice,

$k_x(i, j)$ - dyskretna różnica funkcji obrazu wzduż x dla punktu (i, j) .

$k_y(i, j)$ - dyskretna różnica funkcji obrazu wzduż y dla punktu (i, j) .

Operację tą można zrealizować za pomocą dyskretnego splotu dwóch masek:

<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>+1</td><td>0</td></tr> <tr><td>0</td><td>-1</td></tr> </table> (a) k_x	+1	0	0	-1	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>+1</td></tr> <tr><td>-1</td><td>0</td></tr> </table> (b) k_y	0	+1	-1	0
+1	0								
0	-1								
0	+1								
-1	0								

Rysunek 2.12: Maski krzyża Robertsa - obliczenia dla lewego, górnego piksela

Amplituda (stromość) krawędzi może być wyznaczona w różnych przestrzeniach metrycznych:

$$s = \sqrt{k_x^2 + k_y^2}, \quad s' = |k_x| + |k_y| \quad (2.10)$$

gdzie:

s - rozkład amplitudy („stromości”) krawędzi w funkcji obrazu wyznaczany w metryce euklidesowej,

s' - rozkład amplitudy („stromości”) krawędzi w funkcji obrazu wyznaczany w metryce Manhattan,

k_y - dyskretna różnica funkcji obrazu wzdłuż y ,

k_x - dyskretna różnica funkcji obrazu wzdłuż x .

Natomiast kierunek krawędzi jest postaci:

$$r_{Roberts} = \operatorname{arctg} \left(\frac{k_y}{k_x} \right) - \frac{3}{4}\pi \quad (2.11)$$

gdzie:

s - rozkład kierunku krawędzi w funkcji obrazu,

k_y - dyskretna różnica funkcji obrazu wzdłuż y ,

k_x - dyskretna różnica funkcji obrazu wzdłuż x .

Zastosowane maski (rysunki 2.12(a), 2.12(b)) są obrócone względem osi współrzędnych obrazu o kąt $\frac{1}{4}\pi$ i z tego powodu należy dokonać korekty kierunku krawędzi o kąt $-\frac{3}{4}\pi$. Ze względu na mały badany obszar algorytm ten jest wrażliwy na szum.

Operacja splotu

Operacja splotu [8] z funkcją maski ($g = f * h$) w dyskretnej postaci wynosi:

$$g_{i,j} = \sum_{m=-M}^{M} \sum_{n=-N}^{N} h[m, n]f[i+m, j+n] \quad (2.12)$$

gdzie:

$g_{i,j}$ - wynik operacji splotu dla punktu (i, j) ,

h - maska z którą wyznaczany jest splot,

f - funkcja dla której wyznaczany jest splot.

Wyznaczanie krawędzi może się odbywać z wykorzystaniem dyskretnego splotu,

który jest realizowany przez odpowiednie maski ($h[m, n]$). Najbardziej znane maski zostały przedstawione na ilustracji 2.13. Aproksymują one pierwszą pochodną funkcji obrazu. Typowymi operatorami są:

- **operator różnic elementów centralnych** - prosty w implementacji, wrażliwy na szum, maksymalna amplituda wykrytych krawędzi wynosi 255.
- **operator Sobela** - łatwa implementacja, osiąga dobre rezultaty, maksymalna amplituda wykrytych krawędzi wynosi 2040.
- **operator Prewitta** - nietrudna realizacja, właściwości prawie tak dobre jak dla operatora Sobela, maksymalna amplituda wykrytych krawędzi wynosi 1020.

Kierunek krawędzi został zkwantyzowany do ośmiu kierunków. Ich podział został przedstawiony na ilustracji 2.14. Wyznaczanie dyskretnego kierunku sprowadza się do porównania składowych gradientu dla każdego piksela obrazu.

-1	0	1
0		
1		

(a) operator różnic centralnych

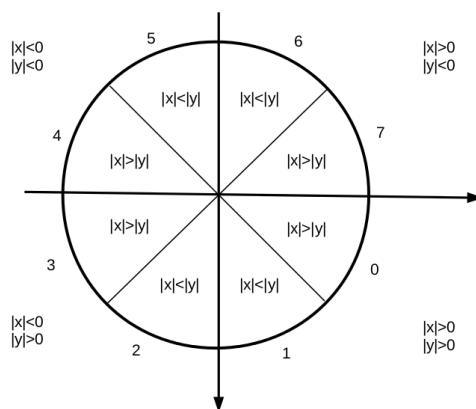
-1	0	1
-2	0	2
-1	0	1

(b) operator Sobela

-1	0	1
-1	0	2
-1	0	1

(c) operator Prewitta

Rysunek 2.13: Maski operatorów krawędziowych



Rysunek 2.14: Osiem dyskretnych kierunków przestrzeni obrazu. Dyskretyzację kierunku uzyskuje się porównując składowe x i y wektora gradientu

Funkcja Canny

Funkcja Canny została opracowana przez Johna F. Canny w 1986 roku. Jest wielostopniowym algorytmem wykrywania raptownych zmian funkcji obrazu wykorzystujący istniejące operatory. Głównym celem tej funkcji jest spełnienie trzech kryteriów:

- niski współczynnik błędu - oznaczanie wyłącznie istniejących krawędzi,
- dobre umiejscowienie - odległość między wykrytą a rzeczywistą krawędzią powinna być jak najmniejsza,
- minimalna odpowiedź - prawdziwa krawędź powinna być oznaczona tylko raz oraz zakłócenia (między innymi szum) nie powinny być wykrywane jako krawędzie.

Funkcja Canny składa się z czterech etapów. Pierwszym z nich jest redukcja szumu wykorzystująca filtr Gaussa. Drugim etapem jest wyznaczenie rozkładu natężeń gradientu za pomocą operatora Sobela. Kolejnym krokiem jest usunięcie z tego rozkładu niemaksymalnych wartości. W wyniku tej operacji powstają linie reprezentujące krawędzie o grubości jednego piksela. Ostatnim etapem jest eliminacja tych krawędzi, które nie przekraczają określonego progu. W tym algorytmie zastosowano dwa progi. Krawędź jest zachowywana, gdy jest powyżej górnej granicy i jest prowadzona do momentu, gdy jej stromość nie spadnie poniżej dolnego progu. Takie rozwiązanie zapobiega dzieleniu krawędzi w miejscach słabszego konturu.

2.3.5 Detekcja konturów w obrazie

Z pomocą algorytmów opisanych w sekcji 2.3.4 można uzyskać rozkład krawędzi. Jednak krawędzie w formie pikseli są trudne w analizie. Z tego powodu w niniejszej pracy wykorzystano algorytm Suzuki85 [1] zaimplementowany w ramach biblioteki OpenCV. Funkcja jako wynik może zwracać kontury w różnej postaci, np. listę pikseli należących do konturu.

Każdemu konturowi funkcja ta przypisuje liczbę porządkową. Na początku algorytm znajduje pierwszy piksel, który należy do krawędzi oraz z lewej i prawej strony otaczają go piksele tła. Następnie śledzi brzeg, czyli przestrzeń między piksemlem będącym krawędzią a piksemlem należącym do tła. Podczas śledzenia brzegu nadaje pikselom krawędzi odpowiedni numer:

- Jeśli po lewej stronie brzegu znajduje się piksel należący do krawędzi a po prawej piksel należący do tła - przypisuje pikselowi ujemną wartość numeru aktualnie wyznaczanego konturu,

- w przeciwnym wypadku - jeśli krawędź była nieodwiedzana przypisuje pikselowi dodatnią wartość numeru aktualnie wyznaczanego konturu.

Śledzenie brzegu trwa do momentu dojścia do punktu startu. Następnie algorytm poszukuje kolejny piksel spełniający warunek punktu startowego. Nowe kontury mogą zaczynać się od odwiedzanych pikseli pod warunkiem, że zostały im przypisane dodatnie wartości. Procedura trwa do momentu przeskanowania całego obrazu. Cała ta procedura została dokładniej wyjaśniona w artykule Satoshi Suzuki [11].

2.3.6 Algorytmy planowania trajektorii ruchu

Manipulatory do poruszenia się potrzebują trajektorii składającej się z dwóch punktów: punkt startowy i punkt końcowy. Istnieją jednak takie sytuacje, gdy pojawiają się przeszkody i należy wzbogacić trajektorię o dodatkowe punkty, aby uniknąć kolizji. W tym celu powstały algorytmy planowania trajektorii ruchu [2]. Większość z nich polega na budowie drzewiastego grafu, w którym każdy węzeł reprezentuje pozycję niekolidującego manipulatora. Przykładowymi algorytmami planowania są:

- RRT(Rapidly-exploring Random Tree)- opiera się na grafie o drzewiastej strukturze, gdzie korzeniem jest punkt startowy. Algorytm polega na tworzeniu losowych pozycji i łączeniu ich z wcześniej dodanymi do momentu, gdy w drzewie pojawi się punkt docelowy,
- RRTConnect – zasada działania przypomina RRT. Różnica polega na budowaniu dwóch drzew: od pozycji startowej oraz od pozycji końcowej. Algorytm dąży do połączenia tych drzew,
- RRT* - opiera się na algorytmie RTT z tą różnicą, że drzewo jest budowane do momentu, gdy znajdzie się dostatecznie krótką ścieżkę bądź skończy się czas,
- EST(Expansive Space Trees) – algorytm polega na budowaniu drzewa podobnie jak w RRT z tą różnicą, że drzewo jest budowane w najmniej odwiedzanych obszarach.

2.3.7 Algorytmy układania kostki Rubika

Celem układania kostki Rubika jest przekręcanie takich ścianek, aby pola na poszczególnych ściankach miały taki sam kolor. Możliwe jest ułożenie jej z dowolnego stanu w maksymalnie dwudziestu ruchach. Istnieją algorytmy, które ułatwiają to zadanie [10]. Metody układania kostki wykorzystywane w robotyce różnią się od tych przeznaczonych dla człowieka. Przykładowymi algorytmami są:

- Algorytm Boga - algorytm wykorzystujący bazę danych posiadającą informacje o optymalnym ruchu dla każdego stanu. Biorąc pod uwagę ilość możliwych ułożen kostki rozwiązywanie to posiada duże wymagania pamięciowe,
- Ręczne rozwiązywanie - polega na podzieleniu procesu układania kostki na kilka etapów (kolejne warstwy). Wymaga intuicji i jest wykorzystywana głównie przez człowieka,
- Algorytm Thistlethwaite'a - algorytm opiera się na podziale układania kostki na etapy wyznaczone na matematycznych podstawach,
- Przeszukiwanie drzewa – jego działanie polega na zbudowaniu drzewa, gdzie podstawą jest stan początkowy kostki, odgałęzieniami drzewa są stany po wykonaniu konkretnego ruchu. Algorytm polega na odnalezieniu stanu, w którym kostka jest ułożona.

Rozdział 3

Opis rozwiązania

Niniejszy rozdział przedstawia sposób realizacji projektu. Opisane zostało zbudowane stanowisko eksperymentalne (sekcja 3.1), struktura systemu (sekcja 3.2), metoda lokalizacji kostki Rubika w obrazie (sekcja 3.3), algorytm sterowania manipulatorami (sekcja 3.4), proces wyznaczania układu pól (sekcja 3.5) oraz algorytm układania kostki Rubika (sekcja 3.6).

3.1 Stanowisko badawcze

Stanowisko badawcze zbudowane jest z dwóch komputerów. Pierwszy z nich jest bezpośrednio połączony do kamery oraz do sprzętowego interfejsu sterowników manipulatorów IRP-6. Komputer ten pełni rolę serwera o nazwie *gerwazy* i za jego pośrednictwem odbywa się sterowanie manipulatorami oraz praca z kamerą. W wyjątkowych sytuacjach istnieje możliwość sterowania robotami bezpośrednio za pomocą interfejsu sprzętowych sterowników. Dodatkowo istnieją awaryjne przyciski wyłączające roboty na wypadek niebezpiecznych sytuacji. Drugi komputer przetwarza obraz, realizuje algorytm układania kostki Rubika oraz wysyła zadania związane z robotami. Łączy się z pierwszym komputerem za pośrednictwem sieci lokalnej i jako klient przesyła polecenia dla manipulatorów.

Testowanie nowych algorytmów może wiązać się z ryzykiem, np. kolizji robotów. Z tego powodu istnieje możliwość uruchomienia systemu w trybie symulacji, która nie steruje prawdziwymi manipulatorami lecz ich wirtualnymi odpowiednikami przedstawionymi za pomocą grafiki trójwymiarowej. Wizualizacja odbywa się za pomocą pakietu *RVIZ*.

Uruchamianie polega na wykonaniu odpowiedniego skryptu włączającego cały system bez sterowania prawdziwymi manipulatorami. Następnie należy uruchomić węzeł Rviz, aby mieć wgląd na symulację. Na koniec trzeba włączyć węzeł odpowiedzialny za realizację konkretnego zadania.

W niniejszej pracy sterowanie manipulatorami odbywa się za pośrednictwem simuladora OpenRAVE. Z tego powodu istnieje możliwość wglądu na trajektorię ruchu tuż przed jego wykonaniem, co umożliwia interwencję przed powstaniem niebezpiecznej sytuacji. W tym celu należy dla funkcji inicjującej ustawić odpowiedni parametr.

3.2 Struktura systemu

Program składa się z dwóch modułów, które komunikują się zgodnie ze schematem 3.1. Pierwszy moduł został zrealizowany z wykorzystaniem platformy programistycznej DisCODE, natomiast drugi używając systemu ROS. Komunikacja między modułami jest jednokierunkowa. Odbywa się za pomocą mechanizmu tematów. Moduł DisCODE posiada komponent (*CubeProxy*) publikujący wynik analizy ścianki kostki Rubika wykorzystując funkcję dostarczoną przez system ROS:

```
1 | rubik_cube::Cube_face_color msg;
2 | pub.publish(msg);
```

msg - obiekt przechowujący informacje, które mają być upublikowane

pub - zainicjowany publikator (publisher).

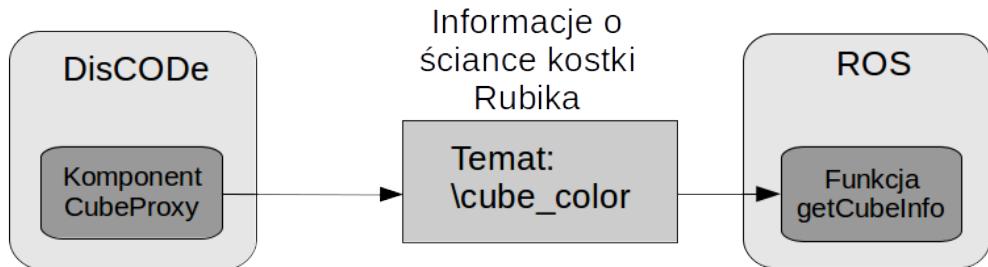
Publikowane informacje odczytuje moduł ROS w standardowy sposób. Wiadomość (message) zawierająca dane ścianki kostki Rubika jest postaci:

```
1 | //Tile_color
2 | uint8 red
3 | uint8 green
4 | uint8 blue
5 |
6 | //Cube_color
7 | rubik_cube/Tile_color tile1
8 | rubik_cube/Tile_color tile2
9 | rubik_cube/Tile_color tile3
10 | rubik_cube/Tile_color tile4
11 | rubik_cube/Tile_color tile5
12 | rubik_cube/Tile_color tile6
13 | rubik_cube/Tile_color tile7
14 | rubik_cube/Tile_color tile8
15 | rubik_cube/Tile_color tile9
16 | float64 x
17 | float64 y
18 | float64 z_rot
```

Program działa pod systemem operacyjnym Linux.

3.2.1 Moduł DisCODE

Moduł realizowany przez DisCODE jest odpowiedzialny za wyznaczenie pozycji oraz koloru pól kostki Rubika na podstawie obrazu. Jego działanie jest podzielone



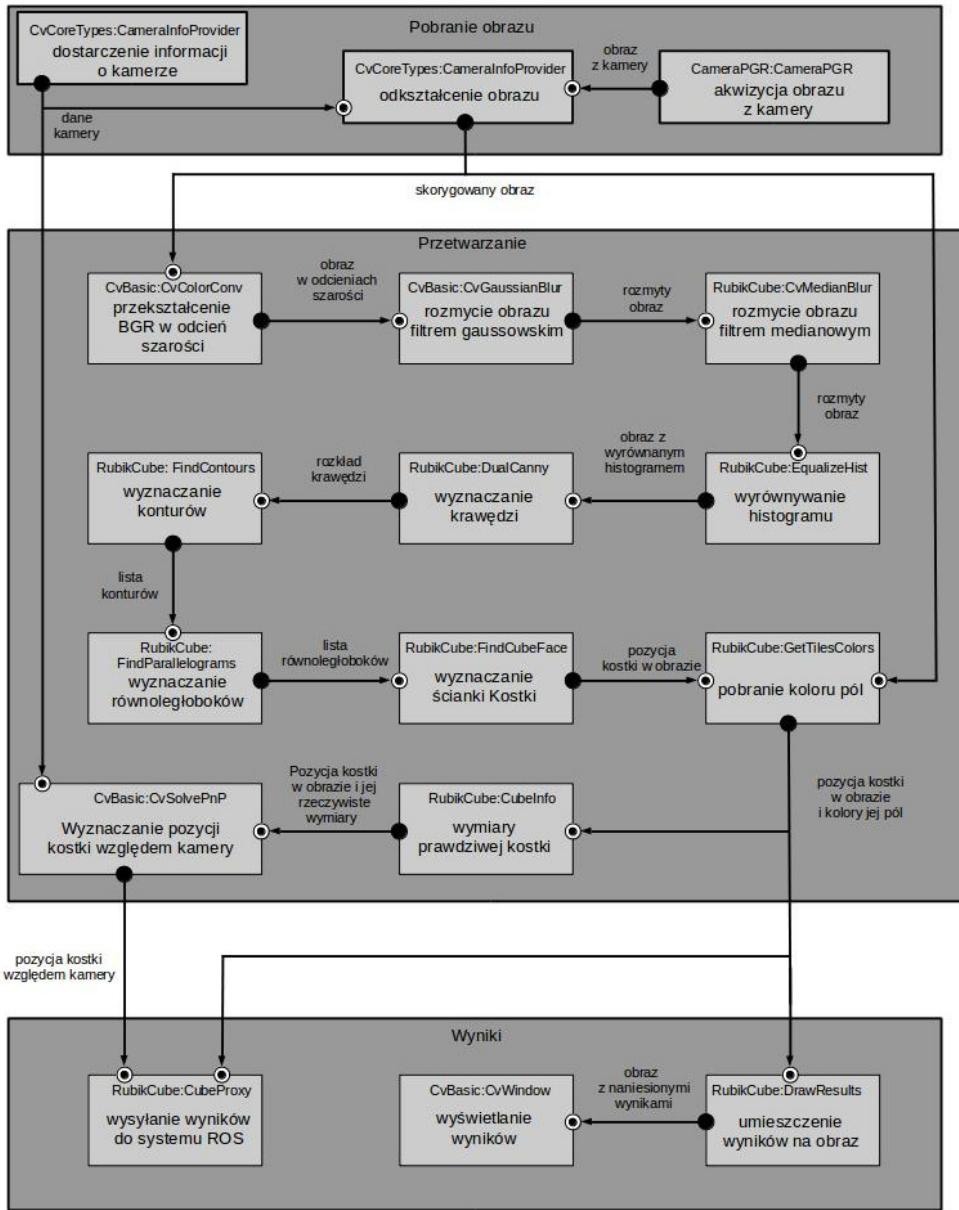
Rysunek 3.1: Schemat komunikacji dwóch modułów

na komponenty tworzące jeden potok. Do przetwarzania obrazu wykorzystano bibliotekę OpenCV. Moduł jest realizowany przez trzy wątki. Większość obliczeń jest wykonywanych przez drugi wątek. Podział procesu na wątki pozwoli wykorzystać kilka rdzeni procesora. Ogólną strukturę modułu przedstawia schemat 3.2

Pierwszy wątek składa się z trzech komponentów i jest odpowiedzialny za dostarczenie obrazu z kamery. Jeden z nich dostarcza informacje o kamerze. Posiada w sobie na stałe wpisane parametry, które przekazuje dalej. Drugi jest odpowiedzialny za pobranie obrazu z kamery. Trzeci posiadając obraz z kamery i jej dane dokonuje korekty, czyli usuwa zniekształcenia z obrazu jak np. efekt „rybiego oka”.

Drugi wątek realizuje algorytm przetwarzania obrazu i składa się z jedenastu komponentów. Pierwszy komponent po otrzymaniu obrazu dokonuje jego desaturacji. Dwa kolejne komponenty redukują szum w obrazie (sekcja 2.3.1) wykorzystując filtr gaussowski i filtr medianowy zaimplementowane w ramach biblioteki OpenCV. W celu poprawy jasności obrazu czwarty komponent wyrównuje jego histogram (sekcja 2.3.3) używając gotową funkcję biblioteki OpenCV. Następnie zostają wyznaczone krawędzie w sposób opisany w sekcji 2.3.4. Szósty komponent wyznacza na podstawie krawędzi kontury wykorzystując algorytm Suzuki85 (sekcja 2.3.5) zwierający listę pikseli tworzących kontur. Siódmy komponent określa równoległy boki znajdujące się w obrazie (sekcja 3.3.2) reprezentowane przez cztery punkty. Posiadając równoległy boki ósmy komponent wyznacza położenie kostki Rubika w obrazie (sekcja 3.3.2). Algorytm znając lokalizację kostki pobiera próbki kolorów. Dziesiąty komponent wzbogaca lokalizację kostki o informacje o jej rzeczywistych wymiarach. Ostatni komponent na podstawie położenia obiektu na obrazie oraz jego prawdziwych rozmiarów wyznacza lokalizację kostki względem kamery wykorzystując funkcję OpenCV rozwiązującą problem Perspective-n-Point.

Trzeci wątek wyświetla i publikuje wynik programu. Pierwszy komponent umieszcza na obrazie z kamery rezultat algorytmu, a następny komponent wyświetla go na ekranie. Trzeci komponent wysyła wynik do systemu ROS.



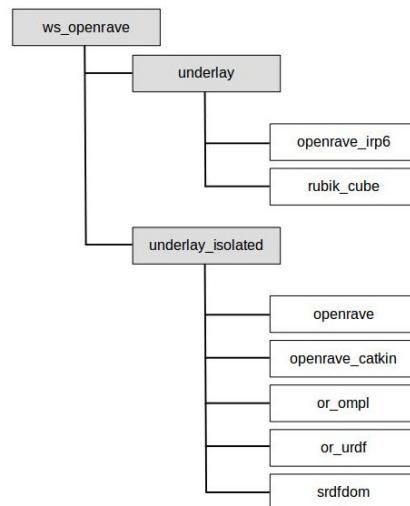
Rysunek 3.2: Schemat modułu DisCODe

3.2.2 Moduł ROS

Moduł realizowany przez ROS jest odpowiedzialny za sterowanie manipulatorami, wyznaczenie układu pól kostki na podstawie kolorów oraz realizację algorytmu układania jej. W celu sterowania manipulatorami łączy się jako klient z serwerem *gerwazy*, który jest bezpośrednio połączony z sprzętowym interfejsem sterowników manipulatorów i wysyła odpowiednie polecenia. Trajektorie ruchu manipulatorów są tworzone z wykorzystaniem planera zbudowanego z symulatora OpenRAVE oraz biblioteki OMPL.

Środowisko planera znajduje się w środowisku ROS przedstawionym na schemacie 3.3. Zostało ono podzielone na dwa podkatalogi, w której jeden

(*underlay*) zawiera pakiety stworzone na potrzeby niniejszej pracy, natomiast drugi (*underlay_isolated*) posiada pakiety już istniejące. Pakiet *openrave* zawiera skonfigurowany symulator OpenRAVE i dzięki pakietowi *openrave_catkin* może funkcjonować w systemie ROS. Pakiet *or_ompl* stanowi skonfigurowaną bibliotekę OMPL. OpenRAVE domyślnie obsługuje modele robotów w formacie COLLADA. Wykorzystano pakiet *or_urdf* w celu umożliwienia planerowi obsługę modeli robotów, także w formacie URDF. Pakiet *srdfdom* umożliwia czytanie danych semantycznych robotów (pliki SRDF) i jest niezbędny do działania pakietu *or_urdf*. Istniejące pakiety zostały stworzone przez laboratorium Personal Robotics mieszczące się na uniwersytecie badawczym w Pittsburghu. Pakiet *openrave_irp6* stanowi interfejs do symulatora OpenRAVE, który automatycznie wczytuje modele robotów i posiada funkcje sterowania manipulatorami. Pakiet *rubik_cube* zawiera funkcje służące do oglądania i przekładania kostki Rubika.



Rysunek 3.3: Drzewo katalogów planera (szare katalogi, białe pakiety)

3.3 Lokalizacja kostki Rubika i wyznaczenie kolorów pól

Niniejsza sekcja opisuje sposób przetwarzania obrazu. Algorytm na początku dokonuje wcześniejszej obróbki obrazu w celu wyodrębnienia potrzebnych informacji (sekcja 3.3.1), następnie dokonuje właściwą analizę obrazu (sekcja 3.3.2).

3.3.1 Wstępna obróbka obrazu

Obrazem nazywamy dwuwymiarową macierz wartości reprezentujących jasność i barwę pikseli. W takiej macierzy oprócz badanego obiektu są też nieużywane dane

należące miedzy innymi do otoczenia, które należy odrzucić. Ponadto informacja o obiekcie jest zakłócona i modyfikowana takimi czynnikami jak: słabe oświetlenie, odbicia światła, szum obrazu. Z tego powodu warto jest poddać obraz wstępnej obróbce przed jej właściwą analizą. Przetwarzanie obrazu składa się z operacji jak: wstępna filtracja, konturowanie, wyodrębnianie obiektów.

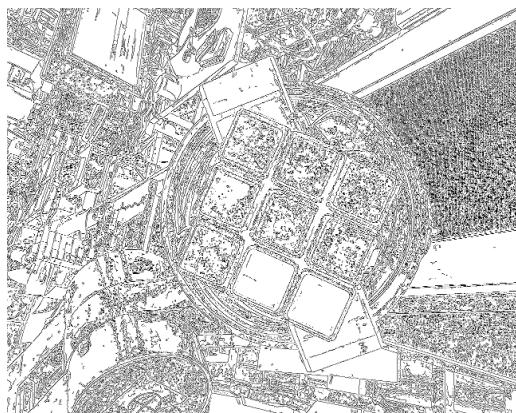
Nieodłącznym elementem obrazu jest szum objawiający się losowo umiejscowionymi zmianami pikseli, który skutecznie obniża jakość wykrywanych krawędzi. W wyniku jego obecności pojawiają się dodatkowe, losowo umieszczone krawędzie. Zastosowano filtr gaussowski oraz filtr medianowy, aby zredukować ten efekt. Ilustracja 3.4 przedstawia wpływ efektu rozmycia na wykrywane krawędzie. Zastosowano gotową funkcję OpenCV:

```

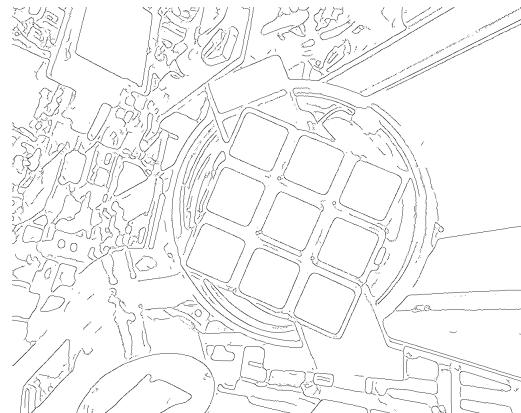
1 | cv::GaussianBlur(img_in, img_out, cv::Size(kernel_width,
|   kernel_height));
2 | cv::medianBlur(img_in, img_out, kernelSize);

```

- 1 - rozmycie gaussowskie, gdzie dwa pierwsze parametry to obraz wejściowy i wyjściowy a trzeci to szerokość i wysokość maski,
- 2 - rozmycie medianowe, gdzie dwa pierwsze parametry to obraz wejściowy i wyjściowy a trzeci to rozmiar maski.



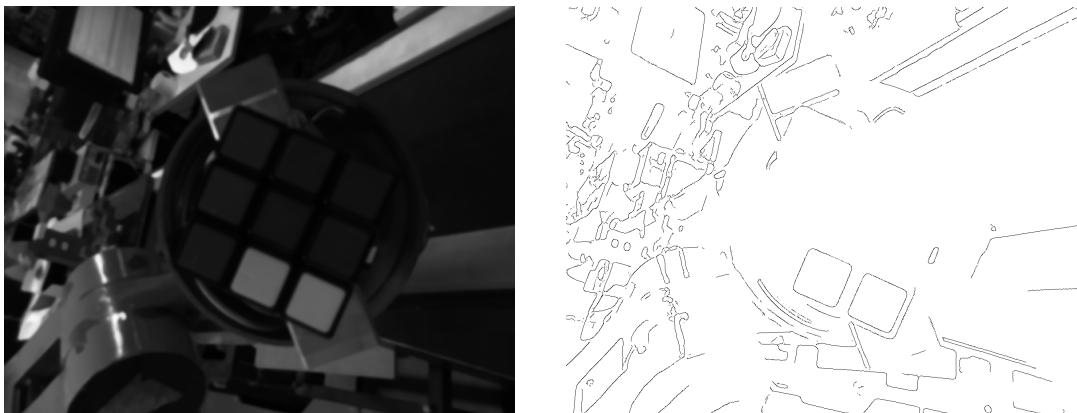
(a) Bez rozmywania obrazu



(b) Z rozmywaniem obrazu

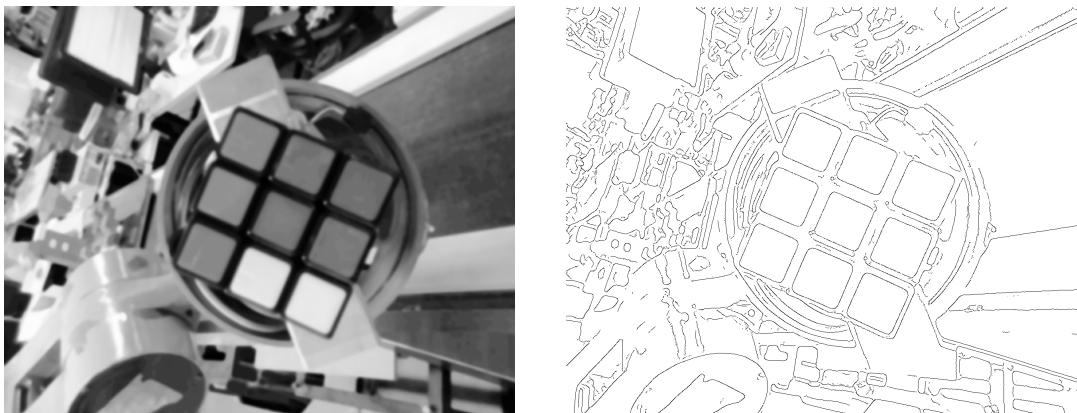
Rysunek 3.4: Wykryte krawędzie kostki Rubika.

Jednym z problemów jakie można napotkać podczas przetwarzania obrazu jest źle dobrany czas naświetlania matrycy. Objawia się on zaciemnionym obrazem niewykorzystującym pełnego zakresu wartości pikseli, co drastycznie wpływa na jakość wykrywania krawędzi. Na ilustracji 3.5 przedstawiono zaciemniony obraz wraz z jego wykrytymi krawędziami.



Rysunek 3.5: Zaciemniony obraz kostki Rubika i jego wykryte krawędzie.

W celu zniwelowania tego efektu wykorzystano algorytm wyrównywania histogramu opisany w sekcji 2.3.3, który w tym przypadku rozjaśnia obraz. Takie podejście pozwala na zwiększenie ilości wykrytych krawędzi, co można zauważyć na ilustracji 3.6.



Rysunek 3.6: Obraz kostki Rubika z wyrównanym histogramem i jego wykryte krawędzie.

Jedną z przydatnych informacji o obrazie jest rozkład występujących krawędzi. Do ich wyznaczenia użyto funkcji Canny (sekcja 2.3.4) zaimplementowanej w ramach biblioteki OpenCV:

```
1 | cv::Canny( img_in, img_out, lowerThreshold,higherThreshold);
```

Efekt działania funkcji przedstawia ilustracja 3.8(a). Istnieją sytuacje, w których krawędzie pól są czasem scalane z krawędziami kostki lub są przerywane (ilustracja 3.7). W celu poprawy rezultatu zastosowano inne podejście. Na początek wyznaczane są krawędzie funkcją Canny. Następnie są pogrubiane stostując dylatację (sekcja 2.3.2), tak aby krawędzie pól scalili się z sąsiednimi polami lub kostką. Później krawędzie są poddawane erozji, tak aby były cieńsze. Na koniec ponownie na całym obrazie stosowana jest funkcja Canny. Dzięki takiej operacji, więcej pól jest

reprezentowanych jako ciągła linia tworząca równoległybok. Poszczególne etapy tego podejścia przedstawia ilustracja 3.8. Kod programu jest następujący:

```

1 | Canny( img, img, lowerThreshold,higherThreshold);
2 |
3 | Mat dilationElement = getStructuringElement(
|     MORPH_RECT,Size(dilationSize, dilationSize),Point(
|         (int)(dilationSize/2), (int)(dilationSize/2) ) );
4 | dilate( img, img, dilationElement );
5 |
6 | Mat erosionElement = getStructuringElement(
|     MORPH_RECT,Size(erosionSize, erosionSize),Point(
|         (int)(erosionSize/2), (int)(erosionSize/2) ) );
7 | erode( img, img, erosionElement );
8 |
9 | Canny( img, img, lowerThreshold,higherThreshold);

```

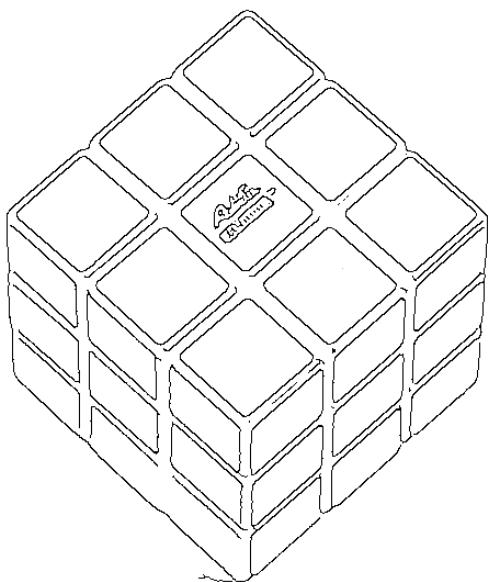
1 - pierwsza funkcja Canny,

3,6 - tworzenie struktury określającej wielkość maski,

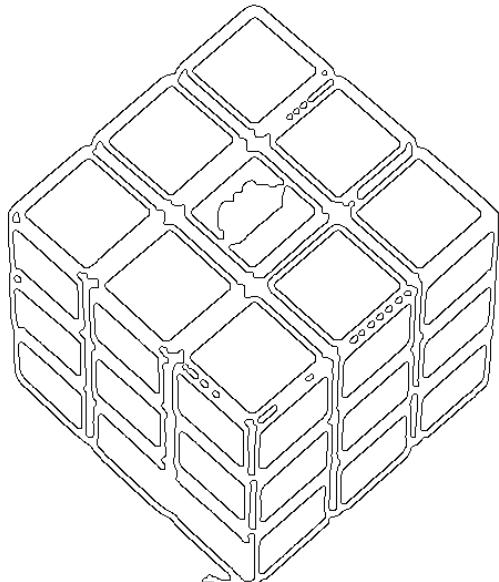
4 - dylatacja obrazu,

7 - erozja obrazu,

9 - druga funkcja Canny.



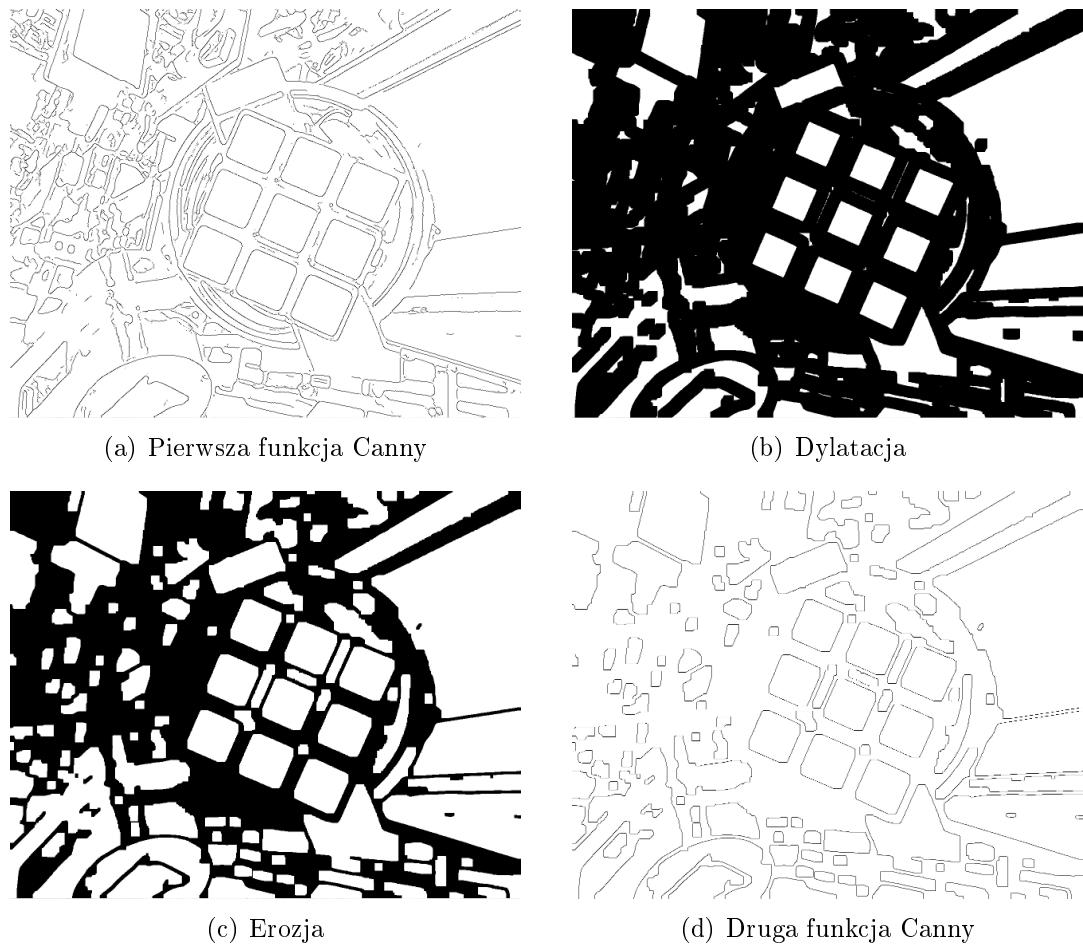
(a) Standardowa funkcja Canny



(b) Podwojna funkcja Canny

Rysunek 3.7: Wykryte krawędzie.

Posiadając obraz w takiej postaci algorytm przystępuje do właściwej analizy kostki Rubika.



Rysunek 3.8: Poszczególne etapy drugiego podejścia wykrywania krawędzi.

3.3.2 Lokalizacja kostki Rubika w obrazie

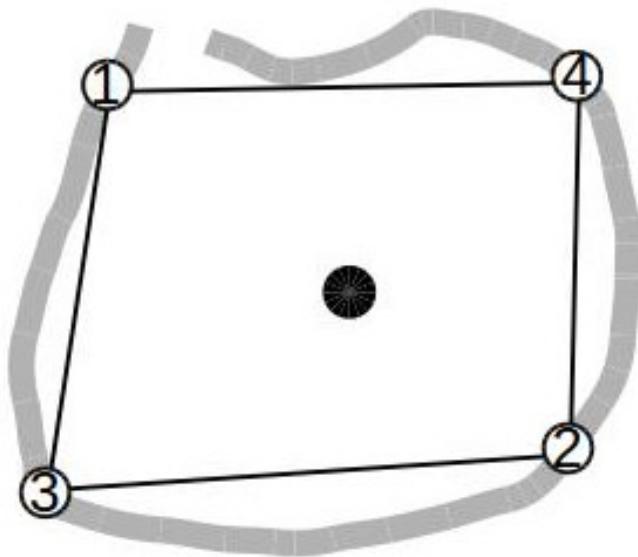
Rozkład krawędzi w obrazie niesie ze sobą dużo informacji lecz są trudne w analizie. Należy przekształcić zbiór pikseli w matematyczną postać, łatwiejszą w przetwarzaniu. W tym celu wykorzystano algorytm Suzuki85 (sekcja 2.3.5). Jego implementacja znajduje się w bibliotece OpenCV i zwraca listę konturów, czyli zbiory punktów tworzących figurę. Sposób wywołania funkcji jest postaci:

```
1 | findContours(img, contours, CV_RETR_LIST, CV_CHAIN_APPROX_NONE);
```

1 - funkcja znajdowania konturów, gdzie pierwszym argumentem jest obraz wejściowy, drugim struktura do której zostanie zapisany wynik, trzeci argument określa hierarchię konturów (w tym przypadku jest to lista), czwarty określa sposób organizacji punktów (w tym wywołaniu każdemu pikselowi przypisany jest jeden punkt).

Następnym krokiem wyznaczania lokalizacji kostki jest określenie, które kontury mogą być równoległobokami. Proces ten składa się z czterech etapów. Pierwszym

z nich jest odnalezienie punktu należącego do konturu, który znajduje się najdalej od jego środka. Punkt ten stanowi pierwszy kąt równoległoboku. Następnym etapem jest wyznaczenie drugiego punktu leżącego po przeciwniej stronie konturu. Trzecim etapem jest wyznaczenie dwóch kolejnych przeciwnie skierowanych punktów, tak aby powstała figura o jak największym polu. Na koniec sprawdzany jest warunek, czy przeciwwiegle boki i kąty równoległoboku są podobne. W przeciwnym wypadku figura jest odrzucana. Proces został przedstawiony na ilustracji 3.9.



Rysunek 3.9: Proces wyznaczania równoległoboku, gdzie numery punktów oznaczają kolejność ich wyznaczania)

Kod wyznaczania pierwszego i drugiego punktu jest postaci:

```

1 | double maximalDistance = 0;
2 | int numberOffurthestPoint = -1;
3 | for (int i = 0; i < distanceLayout.size(); i++)
4 |
5 |     if (distanceLayout[i] > maximalDistance or
6 |         numberOffurthestPoint == -1)
7 |     {
8 |         maximalDistance = distanceLayout[i];
9 |         numberOffurthestPoint = i;
10 |
11 |
12 |     parallelogram.setCorner( (*c)[(numberOffurthestPoint) %
13 |                               c->size()], 0);
14 |     parallelogram.setCorner( (*c)[(numberOffurthestPoint + 2 *
15 |                               c->size() / 4) % c->size()], 2);

```

3-10 - poszukiwania najdalszego punktu konturu, gdzie `distanceLayout` jest listą odległości poszczególnych punktów od środka konturu

12,13 - ustawienie odpowiedniego kąta równoległoboku. Pierwszym argumentem jest punkt. Drugi argument stanowi numer ustawianego kąta. (*c) jest aktualnie przetwarzanym konturem.

Implementacja wyznaczania trzeciego i czwartego punktu jest następująca:

```

1 | double biggestArea = 0;
2 | int numberOffurthestPoint2 = -1;
3 | for (int j = 0; j < c->size(); j++) {
4 |     Point p1 = (*c)[j];
5 |     Point p2 = (*c)[(j + c->size() / 2) % c->size()];
6 |     Point p0 = (*c)[numberOffurthestPoint];
7 |
8 |     double a01 = atan2(p0.y - p1.y, p0.x - p1.x);
9 |     double a02 = atan2(p0.y - p2.y, p0.x - p2.x);
10 |    double r01 = sqrt((p0.x - p1.x) * (p0.x - p1.x) + (p0.y -
11 |        p1.y) * (p0.y - p1.y));
12 |    double r02 = sqrt((p0.x - p2.x) * (p0.x - p2.x) + (p0.y -
13 |        p2.y) * (p0.y - p2.y));
14 |
15 |    if (sin(a01 - a02) * r01 * r02 > biggestArea) {
16 |        biggestArea = sin(a01 - a02) * r01 * r02;
17 |        numberOffurthestPoint2 = j;
18 |    }
19 |}
20 | parallelogram.setCorner((*c)[(numberOffurthestPoint2) %
21 |     c->size()], 1);
22 | parallelogram.setCorner((*c)[(numberOffurthestPoint2 + 2 *
23 |     c->size() / 4) % c->size()], 3);

```

3-17 - poszukiwania takich dwóch punktów, aby pole powierzchni figury było jak największe,

4,5 - pobranie kolejnej pary przeciwnieległych punktów,

6 - pobranie wcześniej wyznaczonego pierwszego kąta,

8,9 - wyznaczenie kąta nachylenia odcinków między pierwszym i trzecim oraz pierwszym i czwartym punktem. Różnicą tych dwóch wartości jest kąt między dwoma odcinkami,

10,11 - wyznaczenie długości dwóch przyległych boków równoległoboku,

13-15 - wyznaczenie połowy pola figury i porównanie z najwyższą dotychczas znalezioną wartością.

Weryfikacja czy figura jest równoległobokiem jest postaci:

```

1 | Point p0 = parallelogram.getCorner(0);
2 | Point p1 = parallelogram.getCorner(1);
3 | Point p2 = parallelogram.getCorner(2);
4 | Point p3 = parallelogram.getCorner(3);
5 | double a01 = atan2(p0.y - p1.y, p0.x - p1.x);
6 | double a03 = atan2(p0.y - p3.y, p0.x - p3.x);

```

```

7 |double a12 = atan2(p1.y - p2.y, p1.x - p2.x);
8 |double a32 = atan2(p3.y - p2.y, p3.x - p2.x);
9 |double r01 = sqrt((p0.x - p1.x) * (p0.x - p1.x) + (p0.y - p1.y) *
   (p0.y - p1.y));
10 |double r12 = sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) *
    (p1.y - p2.y));
11 |double r23 = sqrt((p2.x - p3.x) * (p2.x - p3.x) + (p2.y - p3.y) *
    (p2.y - p3.y));
12 |double r30 = sqrt((p3.x - p0.x) * (p3.x - p0.x) + (p3.y - p0.y) *
    (p3.y - p0.y));
13 |
14 |double distanceThreshold;
15 |double minLength = 40;
16 |double similarAngleThreshold = 10 * 3.14 / 180;
17 |if (r01 < r30) distanceThreshold = r01 / 5;
18 |else distanceThreshold = r30 / 5;
19 |
20 |
21 |if (abs(a01 - a32) < similarAngleThreshold and abs(a03 - a12) <
   similarAngleThreshold and
22 |      abs(r01 - r23) < distanceThreshold and abs(r30 - r12) <
      distanceThreshold and
23 |      r01 > minLength and r12 > minLength
24 |)
25 |
26 |{ parallelograms.push_back(parallelogram);
27 |}

```

1-4 - pobranie kolejnych punktów równoległoboku,

5-8 - wyznaczenie kątów nachylenia kolejnych odcinków równoległoboku,

9-12 - wyznaczenie długości kolejnych odcinków równoległoboku,

15 - minimalna dopuszczalna długość odcinka,

16 - maksymalna dopuszczalna różnica między przeciwnymi kątami równoległoboku,

17-18 - maksymalna dopuszczalna różnica długości między przeciwnymi bokami równoległoboku,

21-23 - warunek czy figura jest równoległobokiem,

26 - dodanie wyznaczonej figury do listy znalezionych równoległoboków.

Moduł posiadając listę równoległoboków przystępuje do wyznaczenia ścianki kostki Rubika. Pierwszym krokiem jest pogrupowanie równoległoboków podobnych do siebie sprowadzające się do porównywania kątów nachylenia i długości boków figur. Implementacja jest postaci:

```

1 |while(parallelograms.size() !=0)
2 |{
3 |    vector<Parallelogram> groupOfParallelograms;

```

```

4 |     double a01 = measureAngle(parallelograms[0].getCorner(0),
5 |                               parallelograms[0].getCorner(1));
6 |     double a03 = measureAngle(parallelograms[0].getCorner(0),
7 |                               parallelograms[0].getCorner(3));
8 |     double r01 = measureDistance(parallelograms[0].getCorner(0),
9 |                                   parallelograms[0].getCorner(1));
10 |    double r03 = measureDistance(parallelograms[0].getCorner(0),
11 |                                   parallelograms[0].getCorner(3));
12 |
13 |    if(a01<0) a01+=CV_PI;
14 |    if(a03<0) a03+=CV_PI;
15 |
16 |    if(a01>CV_PI) a01-=CV_PI;
17 |    if(a03>CV_PI) a03-=CV_PI;
18 |
19 |    double distanceThreshold;
20 |    if (r01 < r03) distanceThreshold = r01 / 3;
21 |    else distanceThreshold = r03 / 3;
22 |
23 |    groupOfParallelograms.push_back(parallelograms[0]);
24 |    parallelograms.erase(parallelograms.begin());
25 |
26 |    for(vector<Parallelogram>::iterator
27 |         j=parallelograms.begin();j!=parallelograms.end();)
28 |    {
29 |        double aj1=measureAngle(j->getCorner(0),j->getCorner(1));
30 |        double aj3= measureAngle(j->getCorner(0),j->getCorner(3));
31 |        double
32 |            rj1=measureDistance(j->getCorner(0),j->getCorner(1));
33 |        double
34 |            rj3=measureDistance(j->getCorner(0),j->getCorner(3));
35 |        if(aj1<0) aj1+=CV_PI;
36 |        if(aj3<0) aj3+=CV_PI;
37 |
38 |        if(aj1>CV_PI) aj1-=CV_PI;
39 |        if(aj3>CV_PI) aj3-=CV_PI;
40 |
41 |        if( ((abs(a01 - aj1)<angleThresholdGroupSquares &&
42 |              abs(a03 - aj3)<angleThresholdGroupSquares) ||
43 |              abs(a03 - aj1)<angleThresholdGroupSquares &&
44 |              abs(a01 - aj3)<angleThresholdGroupSquares) &&
45 |                  ((abs(r01 - rj1)<distanceThreshold &&
46 |                      abs(r03 - rj3)<distanceThreshold) || (abs(r03 -
47 |                          rj1)<distanceThreshold && abs(r01 -
48 |                          rj3)<distanceThreshold)) )
49 |        {
50 |            groupOfParallelograms.push_back(*j);
51 |            j=parallelograms.erase(j);
52 |        }
53 |        else j++;
54 |    }
55 |
56 |    if(groupOfParallelograms.size()>4)
57 |        {groupsOfParallelograms.push_back(groupOfParallelograms);}
58 |
59 |
60 |
61 |
62 |
63 |
64 |
65 |
66 |
67 |
68 |
69 |
70 |
71 |
72 |
73 |
74 |
75 |
76 |
77 |
78 |
79 |
80 |
81 |
82 |
83 |
84 |
85 |
86 |
87 |
88 |
89 |
90 |
91 |
92 |
93 |
94 |

```

parallelograms - równoległoboki jeszcze nie przydzielone do żadnej grupy,

3 - grupa równoległoboków posiadająca podobne figury,

4-17 - wyznaczanie parametrów i dołączenie do grupy kolejnego, nieprzydzielonego jeszcze równoległoboku,

4-7 - wyznaczenie kątów nachylenia oraz długości boków figury,

15-17 - określenie maksymalnej, dopuszczalnej różnicy długości i kąta,

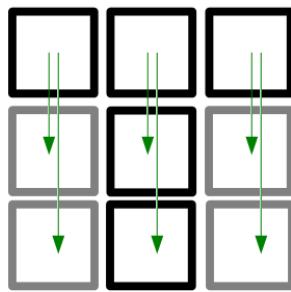
22-41 - przeszukiwanie wśród pozostałych figur podobnych równoległoboków,

24-27 - wyznaczenie kątów nachylenia oraz długości boków figury,

34-36 - sprawdzenie warunku podobieństwa równoległoboków,

43 - jeśli grupa jest wystarczająco liczna, jest zapamiętywana

Następnie algorytm określa ułożenie równoległoboków względem siebie i weryfikuje, które pole w ściance kostki reprezentuje dany równoległobok. Operacja polega na znalezieniu takiego równoległoboku względem, którego znajdują się dwie figury wzduż jednej krawędzi oraz dwa inne równoległoboki wzduż drugiej krawędzi. Rezultatem są dwie trójkie równoległoboków leżących w linii, gdzie jedna z figur należy jednocześnie do dwóch trójek, dając w sumie pięć równoległoboków reprezentujących pola kostki. Nie wszystkie pola zostały jeszcze wyznaczone. Są one obliczane na podstawie znajomości rozmiarów i odległości względem siebie znanych pól. Procedura ta została przedstawiona na ilustracji 3.10.



Rysunek 3.10: Uzupełnianie brakujących pól (czarne - znane pola, szare - uzupełniane pola)

Posiadając lokalizację kostki Rubika moduł pobiera próbki kolorów pól (ilustracja 3.11). Realizacja tego etapu przedstawia kod:

```

1 |     if(cubeFace.isFound() && !img.empty())
2 |     for (int i=0;i<3;i++)
3 |         for(int j=0;j<3;j++)
4 |     {
5 |         int x = cubeFace.getTile(i,j).getMiddle().x;
6 |         int y = cubeFace.getTile(i,j).getMiddle().y;
7 |         if(x >10 && x <1286 && y>10 && y < 1022)
8 |     {

```

```

9 |         int b =
10|         img.ptr<uchar>(y)[numberOfChannels*x+0] +
11|             img.ptr<uchar>(y+4)[numberOfChannels
12|                 * (x+4)+0]+
13|                 img.ptr<uchar>(y+4)[numberOfChannels
14|                     * (x-4)+0]+
15|                     img.ptr<uchar>(y-4)[numberOfChannels
16|                         * (x+4)+0]+
17|                         img.ptr<uchar>(y-4)[numberOfChannels
18|                             * (x-4)+0];
19|         int g =
20|         img.ptr<uchar>(y)[numberOfChannels*x+1] +
21|             img.ptr<uchar>(y+4)[numberOfChannels
22|                 * (x+4)+1]+
23|                 img.ptr<uchar>(y+4)[numberOfChannels
24|                     * (x-4)+1]+
25|                     img.ptr<uchar>(y-4)[numberOfChannels
26|                         * (x+4)+1]+
27|                         img.ptr<uchar>(y-4)[numberOfChannels
28|                             * (x-4)+1];
29|         int r =
30|         img.ptr<uchar>(y)[numberOfChannels*x+2] +
31|             img.ptr<uchar>(y+4)[numberOfChannels
32|                 * (x+4)+2]+
33|                 img.ptr<uchar>(y+4)[numberOfChannels
34|                     * (x-4)+2]+
35|                     img.ptr<uchar>(y-4)[numberOfChannels
36|                         * (x+4)+2]+
37|                         img.ptr<uchar>(y-4)[numberOfChannels
38|                             * (x-4)+2];
39|         b=b/5;g=g/5;r=r/5;
40|         cubeFace.setTileColor(i,j,cv::Scalar(b,g,r));
41|     }
42| }

```

2,3 - operacja jest wykonywana dla każdego pola kostki,

5,6 - wyznaczenie środka pola,

7 - warunek czy punkt mieści się w obrazie,

9-24 - pobranie koloru z 5 punktów będących blisko środka pola i wyciągnięcie z nich średniej,

25 - ustawienie wyznaczonego koloru.

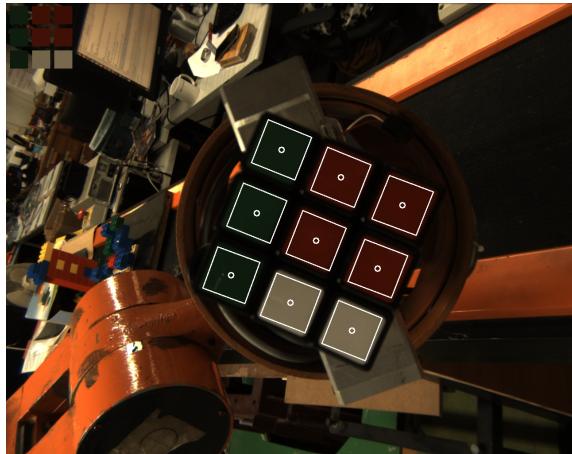
Dodatkowo w celu wyznaczenia pozycji kostki względem kamery wykorzystywana jest funkcja biblioteki OpenCV rozwiązująca problem PnP:

```

1 | solvePnP(modelPoints, imagePoints, camera_info.cameraMatrix(),
2 |           camera_info.distCoeffs(), rvec, tvec, false);

```

1-funkcja rozwiązująca problem pnp,gdzie pierwszym argumentem są wymiary rzeczywistej kostki, drugim wymiary znalezionej kostki, trzecim i czwartym dane kamery, piątym wynikowa macierz rotacji obiektu, szóstym wynikowa macierz translacji obiektu.



Rysunek 3.11: Proces pobierania koloru pól kostki

3.4 Sterowanie manipulatorami

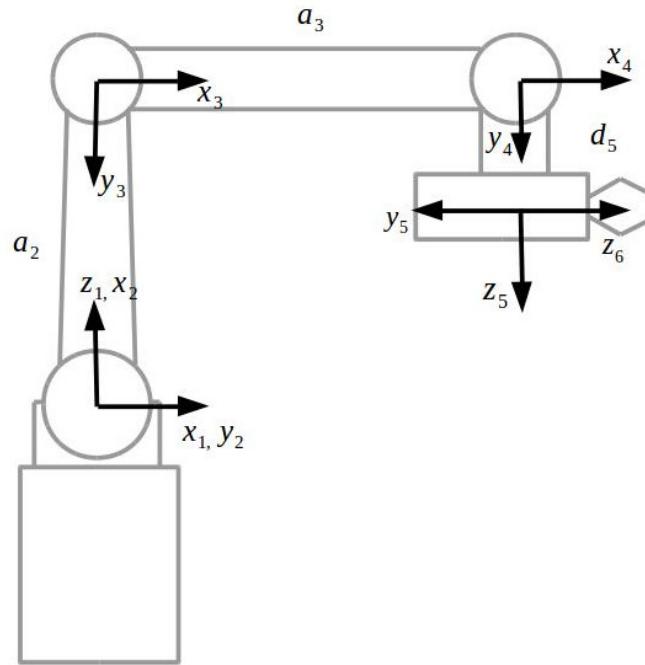
Sterowanie manipulatorami odbywa się za pośrednictwem interfejsu, który daje możliwość poruszania się po trajektorii określonej w pozycji silników, pozycji stawów bądź układzie kartezjańskim. Gdy manipulator ma za zadanie osiągnąć pozycję docelową, powinien wyznaczyć trajektorię ruchu pozwalającą uniknąć kolizji, przekroczenia limitów pozycji bądź prędkości w stawach. W tym celu wykorzystano OpenRAVE, który automatycznie wyznacza trajektorie ruchu (sekcja 3.4.2). Planer ten operuje wyłącznie na pozycjach w stawach. Trajektoria ruchu określana jest przez listę punktów, przez które manipulator powinien przejść. Każdy punkt determinuje pozycję robota, prędkość oraz czas osiągnięcia tej pozycji. Należy wyznaczyć odwrotną kinematykę manipulatora (sekcja 3.4.1), aby możliwe było zadanie docelowej pozycji w układzie kartezjańskim. Automatyczne wyznaczanie trajektorii pozwala na opracowanie sekwencji ruchów manipulatorów mających na celu ułożenie kostki Rubika (sekcja 3.4.3).

3.4.1 Kinematyka manipulatora Irp-6

Każdy manipulator charakteryzuje się kinematyką, która określa w jakiej przestrzeni manipulator może pracować. Istnieją takie sytuacje, w których wygodniej jest operować w układzie kartezjańskim niż w pozycji w stawach. W celu swobodnego przekształcania z jednego układu w drugi dla każdego manipulatora wyznacza się kinematykę prostą oraz kinematykę odwrotną, która została wyznaczona z wykorzystaniem wykładu [12].

Kinematyka prosta

Schemat manipulatora przedstawia ilustracja 3.12. Natomiast parametry w notacji Denavita-Hartenberga zostały umieszczone w tabeli 3.13.



Rysunek 3.12: Schemat manipulatora IRp-6 z osiami obrotu.

i	a_{i-1}	α_{i-1}	d_i	θ_i
1	0	0	0	θ_1
2	0	$-\frac{\pi}{2}$	0	θ_2
3	a_2	0	0	$\theta'_3 = \theta_3 - \theta_2$
4	a_3	0	0	$\theta'_4 = \theta_4 - \theta_3 - \frac{\pi}{2}$
5	0	$-\frac{\pi}{2}$	d_5	θ_5
6	0	$\frac{\pi}{2}$	0	θ_6

Rysunek 3.13: Parametry manipulatora IRp-6 w notacji Denavita-Hartenberga

Macierze transformacji między poszczególnymi punktami odniesienia przedstawiają wzory 3.1, 3.2, 3.3, 3.4, 3.5, 3.6.

$${}_1^0T = \begin{bmatrix} c\theta_1 & -s\theta_1 & 0 & 0 \\ s\theta_1 & c\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.1)$$

$${}_2^1T = \begin{bmatrix} c\theta_2 & -s\theta_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -s\theta_2 & -c\theta_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.2)$$

$${}_3^2T = \begin{bmatrix} c(\theta_3 - \theta_2) & -s(\theta_3 - \theta_2) & 0 & a_2 \\ s(\theta_3 - \theta_2) & c(\theta_3 - \theta_2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.3)$$

$${}_4^3T = \begin{bmatrix} -s(\theta_3 - \theta_4) & c(\theta_3 - \theta_4) & 0 & a_3 \\ -c(\theta_3 - \theta_4) & -s(\theta_3 - \theta_4) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.4)$$

$${}_5^4T = \begin{bmatrix} c\theta_5 & -s\theta_5 & 0 & 0 \\ 0 & 0 & 1 & d_5 \\ -s\theta_5 & -c\theta_5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.5)$$

$${}_6^5T = \begin{bmatrix} c\theta_6 & -s\theta_6 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ s\theta_6 & c\theta_6 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.6)$$

Pozycję końcówki względem początku układu przedstawia macierz transformacji 3.7, gdzie $c_i = c\theta_i, s_i = s\theta_i, c_{32} = c(\theta_3 - \theta_2), s_{32} = s(\theta_3 - \theta_2)$. Równanie to stanowi rozwiązanie kinematyki prostej, czyli pozwala wyznaczyć pozycję końcówki manipulatora na podstawie jego pozycji stawów.

$${}^0_6T = \begin{bmatrix} (c_1s_4c_5 + s_1s_5)c_6 + c_1c_4s_6 & -(c_1s_4c_5 + s_1s_5)s_6 + c_1c_4c_6 \\ (s_1s_4c_5 - c_1s_5)c_6 + s_1c_4s_6 & -(s_1s_4c_5 - c_1s_5)s_6 + s_1c_4c_6 \\ c_4c_5c_6 - s_4s_6 & -c_4c_5s_6 - s_4c_6 \\ 0 & 0 \\ c_1s_4s_5 - s_1c_5 & c_1(c_4d_5 + a_3c_3 + a_2c_2) \\ s_1s_4s_5 + c_1c_5 & s_1(c_4d_5 + a_3c_3 + a_2c_2) \\ c_4s_5 & -s_4d_5 - a_3s_3 - a_2s_2 \\ 0 & 1 \end{bmatrix} \quad (3.7)$$

Kinematyka odwrotna

Kinematyka odwrotna pozwala wyznaczyć pozycję stawów manipulatora tak, aby jego chwytek uzyskał zadaną pozycję w układzie kartezjańskim. Niech zadana pozycja w układzie kartezjańskim będzie określona przez macierz transformacji 3.8. Rozwiążanie kinematyki odwrotnej polega na wyznaczeniu pozycji stawów dla których spełnione jest równanie 3.9. Równanie to można obustronnie przemnożyć przez odwrotność transformacji ${}_1^0T$ zgodnie z wzorem 3.10, który po wstawieniu macierzy równanie ma postać 3.11.

$${}^0_6T_d = \begin{bmatrix} N_x & O_x & A_x & P_x \\ N_y & O_y & A_y & P_y \\ N_z & O_z & A_z & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.8)$$

$${}^0_6T = {}^0_6T_d \quad (3.9)$$

$${}^0_1T^{-1} {}^0_6T = {}^0_1T^{-1} {}^0_6T_d \quad (3.10)$$

$$\begin{bmatrix} s_4c_5c_6 + c_4s_6 & -s_4c_5s_6 + c_4c_6 & s_4s_5 & c_4d_5 + a_3c_3 + a_2c_2 \\ -s_5c_6 & s_5s_6 & c_5 & 0 \\ c_4c_5c_6 - s_4s_6 & -c_4c_5s_6 - s_4c_6 & c_4s_5 & -s_4d_5 - a_3s_3 - a_2s_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \quad (3.11)$$

$$\begin{bmatrix} c_1N_x + s_1N_y & c_1O_x + s_1O_y & c_1A_x + s_1A_y & c_1P_x + s_1P_y \\ c_1N_y - s_1N_x & c_1O_y - s_1O_x & c_1A_y - s_1A_x & c_1P_y - s_1P_x \\ N_z & O_z & A_z & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rozwiązywanie równania 3.11 przedstawia kod:

```

1 | Pz -= z_offset_const;
2 | Pz-= d6*Az;
3 | Py-= d6*Ay;
4 | Px-= d6*Ax;
5 |
6 | (dstJoints)[0] = (atan2(Py, Px));
7 | s0 = sin((double) (dstJoints)[0]);
8 | c0 = cos((double) (dstJoints)[0]);
9 |
10 | c4 = Ay * c0 - Ax * s0;
11 | if (fabs(c4 * c4 - 1) > EPS) s4 = sqrt(1 - c4 * c4);
12 | else s4 = 0;
13 |
14 | if (fabs(s4) < EPS)
15 | {
16 |     osobliwosc = true;
17 |     (dstJoints)[3] = 0.4;
18 |     t5 = atan2(c0 * Nx + s0 * Ny, c0 * 0x + s0 * 0y);
19 |     (dstJoints)[5] = t5-(dstJoints)[3];
20 | }
21 | else
22 | {
23 |     (dstJoints)[5] = t5 = atan2(-s0 * 0x + c0 * 0y, s0 * Nx - c0 *
24 |         Ny);
25 |
26 |     t_ok = atan2(c0 * Ax + s0 * Ay, Az);
27 |     if (fabs((double)(t_ok )) > fabs((double)(t_ok - M_PI ))) t_ok
28 |         = t_ok - M_PI;
29 |     if (fabs((double)(t_ok )) > fabs((double)(t_ok + M_PI ))) t_ok
30 |         = t_ok + M_PI;
31 |     (dstJoints)[3] = t_ok;
32 |
33 |     c3 = cos((double)(dstJoints)[3]);
34 |     s3 = sin((double)(dstJoints)[3]);
35 |
36 |     E = c0 * Px + s0 * Py - c3 * d5;
37 |     F = -Pz - s3 * d5;
38 |     G = 2 * E * a2;
39 |     H = 2 * F * a2;
40 |     K = E * E + F * F + a2 * a2 - a3 * a3;
41 |     ro = sqrt(G * G + H * H);
42 |
43 |     (dstJoints)[1] = atan2(K / ro, sqrt(1 - ((K * K) / (ro * ro)))) -
44 |         atan2(G, H);
45 |
46 |     s1 = sin((double)(dstJoints)[1]);
47 |     c1 = cos((double)(dstJoints)[1]);
48 |     (dstJoints)[2] = atan2(F - a2 * s1, E - a2 * c1);
49 |
50 |     (dstJoints)[4] = atan2(s4, c4);
51 |
52 |     (dstJoints)[2] -= (dstJoints)[1] + M_PI_2;
53 |     (dstJoints)[3] -= (dstJoints)[2] + (dstJoints)[1] + M_PI_2;
54 |
55 |     double c5 = cos(dstJoints[5]),s5 = sin(dstJoints[5]);
56 |     if(fabs(c3*c4*c5-s3*s5-Nz) > EPS && !osobliwosc)
57 |         dstJoints[5]-=M_PI;

```

1-4 - cofnięcie docelowego punktu o długość chwytaka, gdyż transformacja 0T jej nie uwzględnia,

6-8 - wyznaczenie kąta θ_1 z zależności ${}^1T_{24} = {}^1_6T_{d24}$,

10-12 - wyznaczenie $\cos\theta_5$ z zależności ${}^1T_{23} = {}^1_6T_{d23}$,

15-20 - przypadek, gdy $\sin\theta_5 = 0$ czyli jest osobliwość,

17-19 - wyznaczenie kąta $\theta_4 + \theta_6$ z zależności $\frac{{}^1T_{11}}{{}^1T_{12}} = \frac{{}^1T_{d11}}{{}^1T_{d12}}$. Wyznaczana jest suma kątów więc kąt θ_4 jest dowolny,

23-28 - przypadek, gdy $\sin\theta_5 \neq 0$,

23 - wyznaczenie kąta θ_6 z zależności $\frac{{}^1T_{22}}{{}^1T_{21}} = \frac{{}^1T_{d22}}{{}^1T_{d21}}$,

25-28 - wyznaczenie kąta θ_4 z zależności $\frac{{}^1T_{13}}{{}^1T_{33}} = \frac{{}^1T_{d13}}{{}^1T_{d33}}$,

34-41 - wyznaczenie kąta θ_2 z zależności ${}^1T_{14} = {}^1_6T_{d14}$ oraz ${}^1T_{34} = {}^1_6T_{d34}$,

45 - wyznaczenie kąta θ_3

47-50 - wyznaczenie kąta θ_5 z zależności ${}^1T_{13} = {}^1_6T_{d13}$ bądź ${}^1T_{33} = {}^1_6T_{d33}$. Wcześniej z jedynki trygonometrycznej została wyznaczona jedynie wartość bezwzględna z $\sin\theta_5$,

52-53 - poprawka w celu dostosowania do konwencji Denavita-Hartenberga

56 - poprawka kąta θ_6 o kąt $\frac{\pi}{2}$, gdy niespełniona jest zależność ${}^1T_{31} = {}^1_6T_{d31}$. W sytuacji, gdy $N_x = 0$ oraz $O_x = 0$ kąt θ_6 „traci” znak.

3.4.2 Planowanie trajektorii ruchu

Planowanie trajektorii ruchu odbywa się z wykorzystaniem symulatora OpenRAVE, który został skonfigurowany pod system ROS. W celu wykorzystywania planera, stworzono interfejs umożliwiający wydawanie poleceń (diagram klas 3.14).

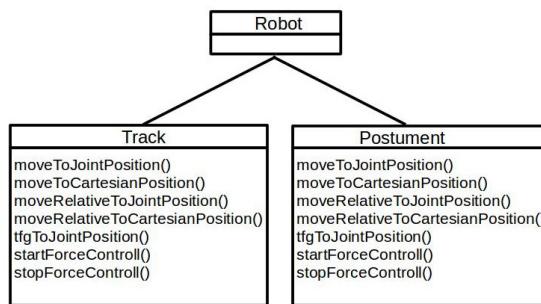
Inicjacja planera odbywa się za pomocą funkcji:

```
1 | initialize(mode='urdf', viewerEnabled=True, manageIrpov=True,
|     planner=None, simplifier='OMPL_Simplifier')
```

1 - funkcja inicjująca planer.

mode - format wczytywanego modelu robota: urdf lub collada,

viewerEnabled - zmienna określająca czy należy uruchomić wizualizację,



Rysunek 3.14: Diagram klas interface'u planera.

manageIrpos - czy planer będzie wysyłać wyznaczone trajektorie ruchu do prawdziwych manipulatorów

planer - konkretny algorytm planowania ruchu. W przypadku *None* jest używany domyślny.

simplifier - algorytm wygładzający trajektorie ruchu

Funkcja inicjuje symulator OpenRAVE, wczytuje model robota. Jako wynik zwraca obiekt *Robot*, który posiada dwa atrybuty *postument* i *track*. Posiadają po sześć funkcji służących do sterowania odpowiednio manipulatorem *Postument* oraz manipulatorem *Track*. Jedną z najważniejszych funkcji jest przemieszczenie się ramienia manipulatora do zadanej pozycji w stawach *moveToJointPosition()*. Jej implementacja jest postaci:

```

1 | def moveToJointPosition(self,dstJoints,simulate=True):
2 |     robot = self.manipulator.GetRobot()
3 |     self.updateManipulatorPosition()
4 |
5 |     robot.SetActiveManipulator(self.manipulator.GetName());
6 |     robot.SetActiveDOFs(self.manipulator.GetArmIndices());
7 |
8 |     traj = None
9 |     conf = None
10 |    if self.planner!=None:
11 |        planner = RaveCreatePlanner(self.env, self.planner)
12 |        simplifier = RaveCreatePlanner(self.env,
13 |                                         self.simplifier)
14 |        simplifier.InitPlan(robot, Planner.PlannerParameters())
15 |
16 |        params = Planner.PlannerParameters()
17 |        params.SetRobotActiveJoints(robot)
18 |        params.SetGoalConfig(dstJoints)
19 |        planner.InitPlan(robot, params)
20 |
21 |        traj = RaveCreateTrajectory(self.env, '')
22 |        result = planner.PlanPath(traj)
23 |        assert result == PlannerStatus.HasSolution
24 |        result = simplifier.PlanPath(traj)
25 |        assert result == PlannerStatus.HasSolution
26 |        result = planningutils.RetimeTrajectory(traj)
27 |        assert result == PlannerStatus.HasSolution
  
```

```

27 |
28 |     if simulate:
29 |         robot.GetController().SetPath(traj)
30 |         self.waitForRobot()
31 |         robot.SetDOFValues(dstJoints,
32 |             self.manipulator.GetArmIndices())
33 |         conf = traj.GetConfigurationSpecification();
34 |     else:
35 |         with self.env:
36 |             traj=self.baseManipulation.MoveActiveJoints(
37 |                 dstJoints, outputtrajobj=True, execute=simulate)
38 |         self.waitForRobot()
39 |         self.updateManipulatorPosition()
40 |         conf = traj.GetConfigurationSpecification();

```

1 - funkcja poruszająca manipulator do zadanej pozycji w stawach,

5-6 - aktywowanie w symulatorze odpowiedni manipulator,

10-32 - planowanie trajektorii ruchu,

11-13 - tworzenie instancji planera,

15-18 - konfiguracja planera,

20-22 - planowanie trajektorii ruchu,

23-24 - uproszczenie trajektorii,

25-26 - przypisanie punktom trajektorii czas, w jakim mają być osiągnięte,

28-30 - symulacja ruchu manipulatora,

31 - nadanie manipulatorowi w symulatorze pozycji docelowej,

34-38 - planowanie trajektorii domyślnym planerem OpenRAVE,

35 - planowanie trajektorii,

37 - nadanie manipulatorowi w symulatorze pozycji docelowej,

```

1 |     if self.irpos!=None and conf!=None:
2 |         try:
3 |             jointGroup = conf.GetGroupFromName("joint_values")
4 |         except openrave_exception:
5 |             jointGroup = None;
6 |         try:
7 |             velGroup =
8 |                 conf.GetGroupFromName("joint_velocities")
9 |         except openrave_exception:
10 |             velGroup = None;
11 |         try:
12 |             gr_tim = conf.GetGroupFromName("deltatime")
13 |         except openrave_exception:
14 |             gr_tim = None

```

```

14 |         times = None
15 |         joints = [];vels = [];accs = [];times = []
16 |         for i in range (0,traj.GetNumWaypoints()):
17 |             w=traj.GetWaypoint(i);
18 |             if self.manipulator.GetName()=='postument':
19 |                 joints.append([w[jointGroup.offset],
20 |                               w[jointGroup.offset+1],
21 |                               w[jointGroup.offset+2],
22 |                               w[jointGroup.offset+3],
23 |                               w[jointGroup.offset+4],
24 |                               w[jointGroup.offset+5]]);
25 |                 vels.append([w[velGroup.offset],
26 |                               w[velGroup.offset+1], w[velGroup.offset+2],
27 |                               w[velGroup.offset+3], w[velGroup.offset+4],
28 |                               w[velGroup.offset+5],
29 |                               w[velGroup.offset+6]]);
30 |                 times.append(w[gr_tim.offset])
31 |
32 |             points = []
33 |             delay=0;
34 |             for i in range(1,len(joints)):
35 |                 delay = delay+times[i]
36 |                 points.append( JointTrajectoryPoint(joints[i],
37 |                                               vels[i], [], [], rospy.Duration(delay)) )
38 |             if len(times)!=0:
39 |                 self.irpos.tfg_to_joint_position(position,time)

```

1-24 - uzyskanie informacji trajektorii ruchu,

25-29 - budowanie trajektorii ruchu,

30-31 - zadanie trajektorii ruchu rzeczywistemu manipulatorowi.

Funkcja odpowiedzialna za poruszanie chwytkiem *tfgToJointPosition()* posiada podobną implementację do funkcji poruszającej ramieniem. Sterowanie manipulatorem w układzie kartezjańskim (*moveToCartesianPosition()*) polega na przekształceniu docelowego punktu do pozycji w stawach (sekcja 3.4.1) i wykorzystaniu funkcji *moveToJointPosition()*. Funkcje *moveRelativeToJointPosition()* oraz *moveRelativeToCartesianPosition()* służą do względnego poruszania się odpowiednio w stawach bądź układzie kartezjańskim. Realizacja sprowadza się do pobrania pozycji manipulatora, przesunięcia pozycji o zadaną wartość i realizacji ruchu wykorzystując funkcję *moveToJointPosition()* bądź *moveToCartesianPosition()*.

Funkcje *startForceControll()* oraz *stopForceControll()* służą do siłowego sterowania manipulatorem. W realizacji tych dwóch funkcji nie wykorzystano planera, więc ograniczają się do wywołania odpowiedniej funkcji *IRPOS*:

```

1 | irpos.start_force_controller(Inertia(Vector3,Vector3),
|     ReciprocalDamping(Vector3,Vector3), Wrench(Vector3,Vector3),
|     Twist(Vector3,Vector3))
2 | irpos.stop_force_controller()

```

1 - funkcja zaczynająca sterowanie siłowe,

Inertia - bezwładność manipulatora,

ReciprocalDamping - współczynnik odwrotności tłumienia,

Wrench - zadana siła wywierana przez chwytkę,

Twist - zadana prędkość chwytaka,

2 - funkcja zatrzymująca sterowanie siłowe.

3.4.3 Proces manipulacji kostką

Posiadając podstawowe funkcje sterowania manipulatorami możliwe jest opracowanie sekwencji ruchów związanych z układaniem kostki Rubika. W tym celu został stworzony zbiór funkcji, które realizują konkretne zadania:

- *faceTrack()* - ustawia chwytek *Postumenta* naprzeciwko chwytaka *Tracka*,
- *facePostument()* - ustawia chwytek *Tracka* naprzeciwko chwytaka *Postumenta*,
- *sidePostument()* - ustawia chwytek *Tracka* z boku chwytaka *Postumenta*,
- *getCubeFromTrackToPostument()* - *Track* przekazuje kostkę *Postumentowi* z przodu,
- *getCubeFromPostumentToTrack()* - *Postument* przekazuje kostkę *Trackowi* z przodu,
- *getCubeFromPostumentToTrackSide()* - *Track* chwyta kostkę od *Postumenta* z boku,
- *setCorrectionFace()* - ustala poprawkę przy odbieraniu kostki z przodu,
- *setCorrectionSide()* - ustala poprawkę przy odbieraniu kostki z boku,
- *showCubeFaceToTrack()* - *Postument* pokazuje trzymaną kostkę *Trackowi*,

- *watchCubeTrack()* - szereg ruchów mających na celu rozpoznanie układu pól kostki Rubika, gdzie *Track* ogląda kostkę,
- *rotateCube()* - *Track* przekręca ściankę kostki trzymaną przez *Postument*,
- *solveRubikCube()* - *Postument* i *Track* układają kostkę Rubika.

Funkcja *faceTrack()* polega na pobraniu pozycji *Tracka* w układzie kartezjańskim, odpowiednim przekształceniu i zadaniu go *Postumentowi*. Istnieją sytuacje, gdy kinematyka manipulatora nie pozwala na to i należy zmienić pozycję *Tracka*.

Implementacja tej funkcji jest następująca:

```

1 | def faceTrack(self, z_dist=0.2, x_dist=0.0, y_dist=0.0,
2 |     showAngle=pi/2):
3 |     trackTrans = self.robot.track.getCartesianPosition()
4 |
5 |     trackRot = PyKDL.Rotation(trackTrans[0][0],
6 |                                 trackTrans[0][1], trackTrans[0][2], trackTrans[1][0],
7 |                                 trackTrans[1][1], trackTrans[1][2], trackTrans[2][0],
8 |                                 trackTrans[2][1], trackTrans[2][2])
9 |
10 |    postRot = trackRot
11 |    postRot.DoRotY(self.pi)
12 |    trackRot.DoRotZ(showAngle)
13 |
14 |    postQuaternion = Quaternion(postRot.GetQuaternion()[0],
15 |                                postRot.GetQuaternion()[1], postRot.GetQuaternion()[2],
16 |                                postRot.GetQuaternion()[3])
17 |
18 |    postPoint = Point (trackTrans[0][2]*z_dist +
19 |                        trackTrans[0][1]*y_dist + trackTrans[0][0]*x_dist +
20 |                        trackTrans[0][3],
21 |                        trackTrans[1][2]*z_dist +
22 |                        trackTrans[1][1]*y_dist +
23 |                        trackTrans[1][0]*x_dist +
24 |                        trackTrans[1][3] +
25 |                        self.robotsDistance,
26 |                        trackTrans[2][2]*z_dist +
27 |                        trackTrans[2][1]*y_dist +
28 |                        trackTrans[2][0]*x_dist +
29 |                        trackTrans[2][3])
30 |
31 |    self.robot.postument.moveToCartesianPosition(
32 |        Pose(postPoint, postQuaternion), self.simulate)
```

1 - funkcja ustawiająca *Postument* naprzeciwko *Tracka*,

z_dist, *x_dist*, *y_dist* - przesunięcie względem drugiego manipulatora,

showAngle - rotacja chwytaka,

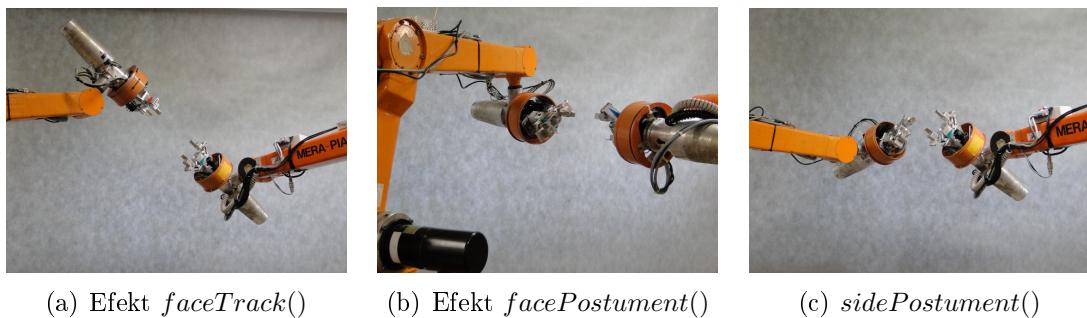
2 - pobranie pozycji *Tracka*,

6-8 - rotacja chwytaka tak, aby był naprzeciwko drugiego robota,

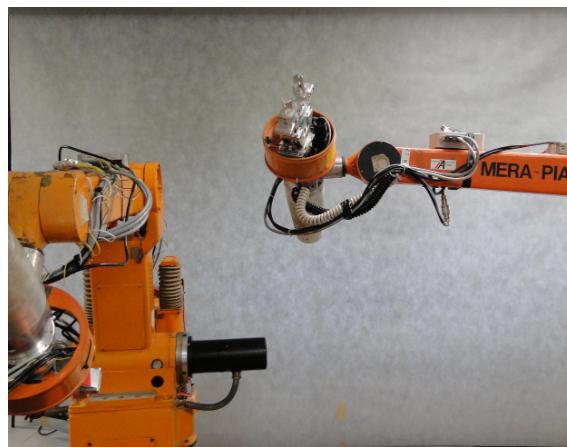
12 - translacja pozycji,

16 - realizacja ruchu.

Funkja *facePostument()* działa w analogiczny sposób. Różnica pojawia się przy funkcji *sidePostument()*, gdzie manipulator jest ustawiany do boku chwytaka. Pozycja, w której jeden robot jest skierowany do boku drugiego nie jest jednoznaczna. Zbiór możliwych pozycji sprowadza się do okręgu, więc funkcja pozwala na określenie konkretnego punktu bądź algorytm samodzielnie znajduje dowolny, na który pozwala kinematyka robota. Wynik działania trzech funkcji przedstawia ilustracja 3.15. Należy zaznaczyć, że nie zawsze możliwe jest ustawienie robota z przodu bądź boku drugiego, ponieważ nie pozwala na to ich kinematyka (ilustracja 3.16).



Rysunek 3.15: Efekt funkcji ustawiających manipulatory naprzeciwko siebie.



Rysunek 3.16: Postument nie jest w stanie ustawić się przodem do *Tracka*.

Kolejne funkcje dotyczą przekładania kostki Rubika. Wynikiem *getCubeFromTrackToPostument()* jest przekazanie przez *Tracka* kostki *Postumentowi* z przodu. Realizacja tej funkcji jest postaci:

```

1 | def getCubeFromTrackToPostument(self,angle=0):
2 |     if angle==0:
3 |         angle=self.pi/2
4 |         fcx=self.fcx
5 |         fcy=self.fcy
6 |     else:
7 |         angle=-self.pi/2

```

```

8     fcx=self.fcx
9     fcy=self.fcy
10
11    self.robot.postument.moveToJointPosition( [-2.25, -1.44, 0.1,
12          0.10, 0.75, -1.91], self.simulate)
13
14    self.facePostument(z_dist=0.15, x_dist=fcx, y_dist=fcy,
15          showAngle=angle)
16
17    self.robot.track.releaseItem(self.cube)
18
19    self.robot.postument.tfgToJointPosition(position =
20          self.wideOpenJoint, simulate=self.simulate)
21
22    self.facePostument(z_dist=0.005, x_dist=fcx, y_dist=fcy,
23          showAngle=angle)
24
25    self.robot.track.startForceControl(tran_x=True, tran_y=True)
26    self.robot.postument.tfgToJointPosition(position = 0.068,
27          simulate=self.simulate)
28    self.robot.track.stopForceControl()
29
30    self.robot.track.startForceControl(tran_z=True, mov_z=0.013)
31    self.robot.postument.startForceControl(rot_x=True, rot_y=True)
32    time.sleep(5)
33    self.robot.postument.stopForceControl()
34    self.robot.track.stopForceControl()
35
36    self.robot.track.startForceControl(tran_x=True, tran_y=True)
37    self.robot.postument.tfgToJointPosition(position =
          self.gripJoint, simulate=self.simulate)
      self.robot.track.stopForceControl()
      self.robot.track.tfgToJointPosition(position =
          self.wideOpenJoint, simulate=self.simulate)
38
39    self.facePostument(z_dist=0.25, x_dist=fcx, y_dist=fcy,
40          showAngle=angle)
41    self.robot.postument.attachItem(self.cube)

```

1 - funkcja przekładająca kostkę z *Tracka* do *Postumenta*,

angle - czy chwytkazabierający kostkę powinien posiadać domyślną pozycję, czy przekreconą o 180°,

2-9 - ustawianie kąta przekręcenia chwytaka i parametrów korekcyjnych pozycji manipulatora,

11 - ustawienie *Postumenta* do pozycji odbierania przedmiotu,

13 - przygotowanie *Tracka* do przekazania obiektu,

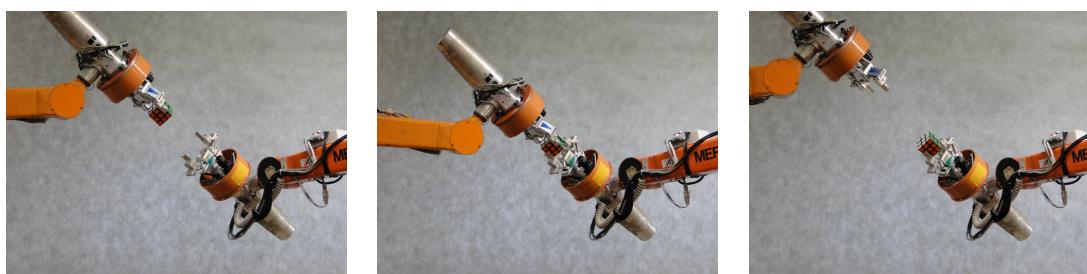
15 - usunięcie kostki z planera na czas jej manipulacji,

17-19 - przybliżenie *Tracka* do obiektu,

21-23 - lekkie zamknięcie chwytaka, żeby została mała przestrzeń między przedmiotem a manipulatorem,

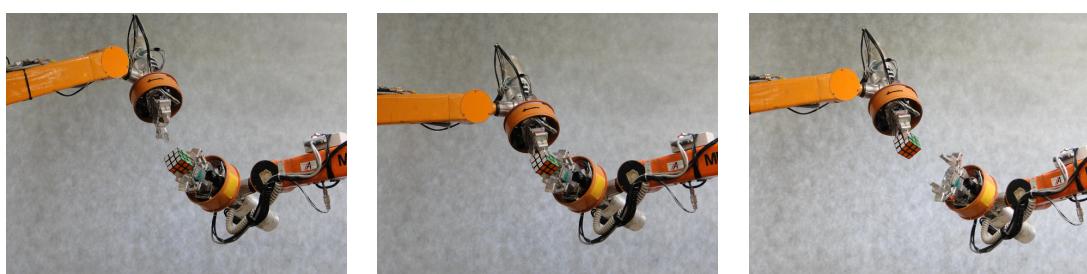
- 25-29 - dosunięcie *Tracka*, aby manipulator chwytający zablokował się na kostce,
 31-33 - *Postument* chwytą obiekt,
 34 - *Track* całkowicie puszcza kostkę,
 36 - manipulatory oddalają się od siebie,
 37 - przywrócenie kostki do planera.

Implementacja funkcji *getCubeFromPostumentToTrack()* jest analogiczna. Funkcja *getCubeFromPostumentToTrackSide()* też jest podobna, lecz kostka jest chwytna z boku drugiego manipulatora. Proces przekazywania kostki przedstawiają ilustracje 3.17 oraz 3.18.



(a) Manipulatory przygotowane do przekazania obiektu (b) Właściwy proces przekazywania kostki (c) Manipulatory oddalają się od siebie

Rysunek 3.17: Proces przekazywania kostku z przodu.



(a) Manipulatory przygotowane do przekazania obiektu (b) Właściwy proces przekazywania kostki (c) Manipulatory oddalają się od siebie

Rysunek 3.18: Proces przekazywania kostku z boku.

Podczas przekazywania kostki wskazana jest precyzja ruchu. Niestety odczyty z receptorów oraz pozycje manipulatorów nie są perfekcyjne. Z tego powodu w nieszej pracy, zastosowano korekcję pozycji z wykorzystaniem wizji. Proces polega na ustalenie manipulatorów w pozycji przekazywania kostki i dokonaniu korekty na podstawie lokalizacji kostki względem kamery (sekcja 3.3). Odpowiedzialne za to są funkcje *setCorrectionFace()* oraz *setCorrectionSide()*. Ich działanie jest podobne. Implementacja jednej z nich ma postać:

```

1 | def setCorrectionFace(self):
2 |     self.fcx=0;
3 |     self.fcy=0;
4 |     self.showCubeFaceToPostument(side=9)
5 |
6 |     time.sleep(3)
7 |     self(cubeFaceColors = None
8 |     while not hasattr(self(cubeFaceColors,'x'):
9 |         time.sleep(0.5)
10 |
11 |     self.fcy = (self(cubeFaceColors.y/1000)
12 |     self.fcx = (-self(cubeFaceColors.x/1000)

```

1 - funkcja wyznaczające parametry korekty przekładania,

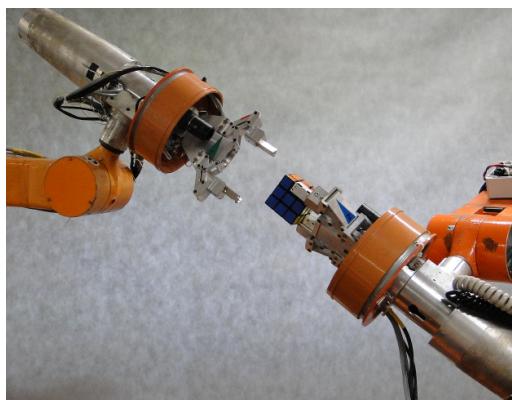
2-3 - zerowanie współczynników korekty,

4 - ustawianie manipulatorów, aby kamera widziała kostkę,

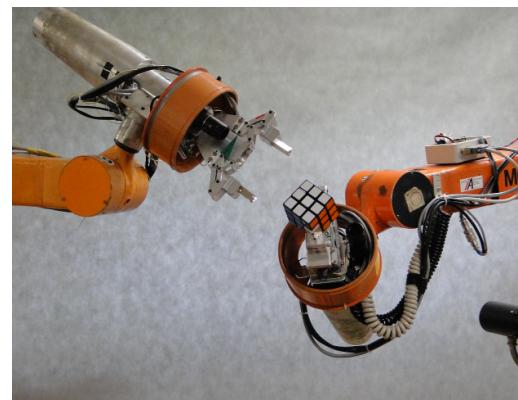
6-9 - oczekiwanie na wynik lokalizacji kostki,

11-12 - wyznaczenie nowych parametrów korekty, żeby były wyrażone w metrach.

Elementarną funkcją rozpoznawania układu pól jest *showCubeFaceToTrack()*, która umożliwia eksponację kostki kamerze. Realizacja sprowadza się do odpowiedniego ustawienia dwóch manipulatorów zgodnie z ilustracją 4.1.



(a) Kamera ogląda kostkę z przodu



(b) Kamera ogląda kostkę z boku

Rysunek 3.19: Proces oglądania kostki.

Funkcja *watchCubeTrack()* stanowi połączenie poprzednich funkcji mających na celu rozpoznanie układu pól kostki Rubika. Jej implementacja jest następująca:

```

1 | def watchCubeTrack(self):
2 |     gotCube = self.getCubeFromHumanToPostument()
3 |     if not gotCube:
4 |         print "Where is the cube?"
5 |         return
6 |
7 |     self.setCorrectionFace()
8 |

```

```

9 |     self.showCubeFaceToPostument(side=0)
10|     self.fillCubeColor(0)
11|
12|     self.setCorrectionSide()
13|
14|     self.showCubeFaceToPostument(side=1)
15|     self.fillCubeColor(1)
16|     self.showCubeFaceToPostument(side=2)
17|     self.fillCubeColor(2)
18|     self.showCubeFaceToPostument(side=3)
19|     self.fillCubeColor(3)
20|     self.showCubeFaceToPostument(side=4)
21|     self.fillCubeColor(4)
22|
23|     self.flipCubePostument()
24|
25|     self.showCubeFaceToPostument(side=0)
26|     self.fillCubeColor(5)
27|
28|     self.showCubeFaceToPostument(side=1)
29|     self.fillCubeColor(8)
30|     self.showCubeFaceToPostument(side=2)
31|     self.fillCubeColor(7)
32|     self.showCubeFaceToPostument(side=3)
33|     self.fillCubeColor(6)
34|     self.showCubeFaceToPostument(side=4)
35|     self.fillCubeColor(9)
36|
37|     self.giveBackCubePostument()
38|
39|     self.determineColors()
40|     self.moveToSynchroPosition()

```

1 - funkcja wyznaczająca układ pól kostki Rubika,

2 - *Postument* otrzymuje kostkę od człowieka,

3-5 - funkcja jest przerywana, gdy brak kostki,

7 - wyznaczenie korekcji dla przekazywania przedmiotu z przodu,

9-10 - oglądanie kostki z przodu,

12 - wyznaczenie korekcji dla przekazywania przedmiotu z boku,

14-21 - oglądanie kostki z przodu,

11-12 - wyznaczenie nowych parametrów korekty, żeby były wyrażone w metrach.

3.5 Obliczanie układu pól kostki Rubika

Posiadając próbki koloru wszystkich pól kostki Rubika możliwe jest wyznaczenie jej układu. Jednym z sposobów jest pogrupowanie kolorów według sztywnych granic definiujących barwę w przestrzeni RGB. Takie rozwiązanie jest wrażliwe na zmianę

kolorów kostki bądź oświetlenia. Innym podejściem do problemu jest podzielenie próbek na sześć równych grup, gdzie każda grupa reprezentuje poszczególną barwę kostki. W niniejszej pracy grupowanie polega na dobraniu dla kolejnych kolorów osiem innych, najbliższych, nieprzydzielonych barw. Algorytm trwa do momentu wyczerpania wszystkich próbek. Odległość między kolorami jest wyznaczana zgodnie z wzorem 3.12.

$$\sqrt[3]{(a_r - b_r)^2 + (a_g - b_g)^2 + (a_b - b_b)^2} = d \quad (3.12)$$

gdzie:

a_r, a_g, a_b - składowe RGB pierwszego koloru,

b_r, b_g, b_b - składowe RGB drugiego koloru,

d - odległość między dwiema barwami.

Z ilustracji 3.20 wynika, że dystans między próbками tej samej barwy są zdecydowanie mniejsze niż odległość próbek różnych barw. Wyjątek stanowi jedynie kolor czerwony i pomarańczowy. Istnieje między tymi dwoma zbiorami odległość, lecz jest porównywalna z dystansem między próbками tej samej barwy (ilustracja 3.21). Kolory będące blisko tej granicy mają w swoim najbliższym sąsiedztwie próbki należące do różnych barw, więc istnieje ryzyko błędного pogrupowania. Z tego powodu algorytm dobiera kolejne kolory pól do najjaśniejszej próbki w zbiorze. Taki warunek powoduje wybieranie punktów będących na brzegu zbioru wszystkich próbek. Takie rozwiązanie pozwala na odróżnianie podobnych barw pod warunkiem, że zbiory są rozłączne. Jako wynik algorytm zwraca sześć równych grup. Jest w stanie odróżniać poszczególne kolory, ale nie potrafi ich nazwać (stwierdzić czy dany kolor to niebieski czy żółty). Do układania kostki Rubika ważna jest informacja o podobieństwie poszczególnych kolorów, a ich nazwy są zbędne. Implementacja tego algorytmu ma postać:

```

1 | for l in range(0,6):
2 |     firstSampleIdx=0
3 |     maxDistance=-1
4 |     for i in range(0,54):
5 |         sampleColor = self.cubeColors.getColor(i);
6 |         distance = math.sqrt( (sampleColor.red)*(sampleColor.red)
7 |             + (sampleColor.green)*(sampleColor.green) +
8 |             (sampleColor.blue)*(sampleColor.blue) )
9 |         if maxDistance<distance and sampleColor.number== -1:
10 |             maxDistance=distance
11 |             firstSampleIdx=i
12 |
13 |
14 |             restColorDistance =
15 |                 [999999,999999,999999,999999,999999,999999,999999,999999]
16 |             restColorIdx = [-1, -1, -1, -1, -1, -1,-1,-1]
```

```

17 |     for i in range(0,54):
18 |         sampleColor = self.cubeColors.getColor(i)
19 |         if sampleColor.number!=-1:
20 |             continue
21 |
22 |         distance = self.cubeColors.calculateDistance(sampleColor,
23 |                                         samplefirstColor)
24 |
25 |         maxDistance=0
26 |         furthestIdx=0
27 |         for j in range(0,8):
28 |             if maxDistance < restColorDistance[j]:
29 |                 furthestIdx = j
30 |                 maxDistance = restColorDistance[j]
31 |
32 |             if restColorDistance[furthestIdx] > distance:
33 |                 restColorDistance[furthestIdx] = distance
34 |                 restColorIdx[furthestIdx] = i
35 |         for j in range(0,8):
36 |             self.cubeColors.setColor(restColorIdx[j],l1)

```

kolory są przechowywane w liście, a ich uzyskanie odbywa się za pomocą funkcji
`self.cubeColors.getColor(i)`

1 - pętla wykonuje się dla każdej z sześciu barw,

4-9 - wybierana jest najjaśniejsza, nieprzydzielona próbka koloru

11 - nadanie numeru koloru dla pierwszej próbki ,

12 - pobranie koloru pierwszej próbki,

14-15 - odległości dotychczas znalezionych ośmiu najbliższych próbek do pierwszej
 i ich indeksy w liście wszystkich próbek,

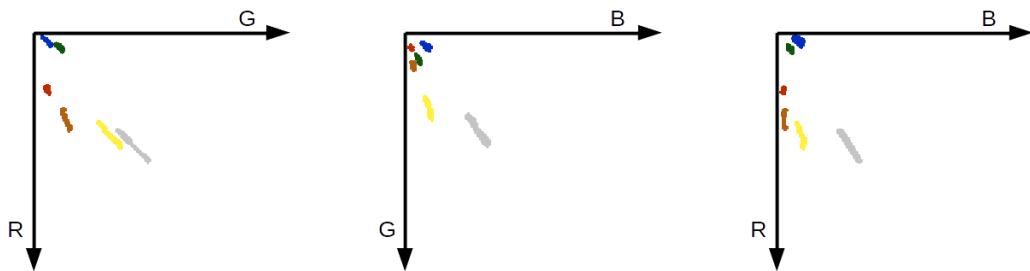
17-30 - wyznaczenie ośmiu innych najbliższych próbek,

22 - wyznaczenie odległości między przeglądana próbką a pierwszą

26-29 - poszukiwanie, która z ośmiu próbek jest najdalej

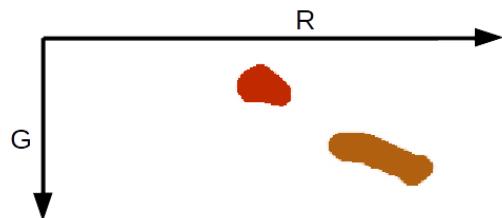
31-33 - jeśli aktualnie analizowana próbka jest bliżej niż jedna z ośmiu, to jest ona
 podmieniana

34-35 - przydzielanie pozostałym ośmiu próbkom numer koloru.

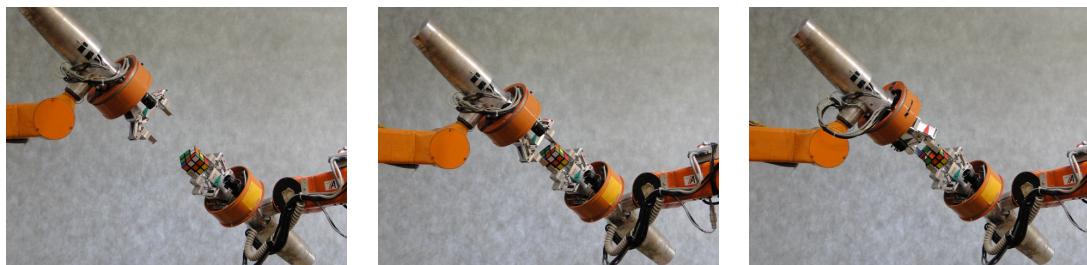


(a) Rozkład kolorów pól kostki Rubika w przestrzeni RG (b) Rozkład kolorów pól kostki Rubika w przestrzeni GB (c) Rozkład kolorów pól kostki Rubika w przestrzeni RB

Rysunek 3.20: Zarys rozkładu kolorów pól kostki Rubika w przestrzeni RGB.



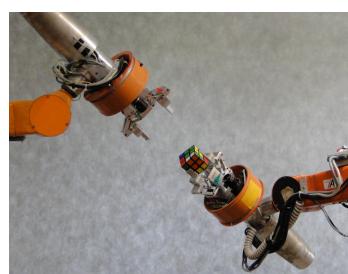
Rysunek 3.21: Przykładowy układ kolorów pól czerwonych i pomarańczowych w przestrzeni RG. Wartości B koloru czerwonego i pomarańczowego są podobne.



(a) Przygotowanie manipulatorów do obrócenia

(b) Chwycenie kostki

(c) Przekręcenie ścianki



(d) Odsunięcie manipulatora

Rysunek 3.22: Proces obracania ścianki kostki Rubika

3.6 Układanie kostki Rubika

Nieodłącznym elementem układania kostki Rubika jest obracanie jej ścianek. W tym celu została zaimplementowana funkcja *rotateCube()*, której działanie przedstawia ilustracja 3.22

Implementacja tej funkcji jest postaci:

```

1 def rotateCube(self,angle):
2     fcx=self.fcx
3     fcy=self.fcy
4
5     self.robot.postument.moveToJointPosition( [-2.25, -1.44,
6         0.1, 0.10, 0.75, -1.91], self.simulate)
7     self.facePostument(z_dist=0.15, x_dist=fcx, y_dist=fcy,
8         showAngle=0)
9
10    self.robot.postument.releaseItem(self.cube)
11
12    self.robot.track.tfgToJointPosition(
13        position=self.wideOpenJoint, simulate=self.simulate)
14
15    self.facePostument(z_dist=0.01, x_dist=fcx, y_dist=fcy,
16        showAngle=0)
17
18    self.robot.postument.startForceControl(tran_x=True,
19        tran_y=True)
20    self.robot.track.tfgToJointPosition( position=0.060,
21        simulate=self.simulate)
22    self.robot.postument.stopForceControl()
23
24    if angle==90:
25        position = (self.robot.track.getJointPosition())[6]
26        self.robot.track.startForceControl(rot_z=True,
27            force_z=0.2, value=0.2)
28        while math.fabs(position -
29            (self.robot.track.getJointPosition())[6]) < 1.57:
30            time.sleep(0.1)
31        self.robot.track.stopForceControl()
32    elif angle== -90:
33        position = (self.robot.track.getJointPosition())[6]
34        self.robot.track.startForceControl(rot_z=True,
35            force_z=-0.2, value=0.2)
36        while math.fabs(position -
37            (self.robot.track.getJointPosition())[6]) < 1.57:
38            time.sleep(0.1)
39        self.robot.track.stopForceControl()
40
41    elif angle==180:
42        position = (self.robot.track.getJointPosition())[6]
43        self.robot.track.startForceControl(rot_z=True,
44            force_z=0.2, value=0.2)
45        while math.fabs(position -
46            (self.robot.track.getJointPosition())[6]) < 3.14:
47            time.sleep(0.1)
48        self.robot.track.stopForceControl()
49
50    self.robot.track.tfgToJointPosition(position =
51        self.wideOpenJoint, simulate=self.simulate)
52
53    self.facePostument(z_dist=0.20, x_dist=fcx,y_dist=fcy,
54        showAngle=0)
55
```

```
41 |     self.robot.track.attachItem(self.cube)
```

1 - funkcja obracająca ściankę kostki Rubika, gdzie *angle* jest kątem o jaki przekręcana jest ściana,

2-3 - współczynniki korekcji chwytania kostki,

5-6 - manipulatory przygotowują się do obracania ścianki,

8 - usuwana kostka z modelu planera,

10 - *Track* otwiera chwytkę,

12 - *Track* zbliża się do kostki trzymanej przez *Postument*,

14-16 - *Track* zamyka chwytkę, lecz jeszcze jej nie chwyta,

18-35 - obracanie ścianki w zależności od zadanego kąta,

19, 25, 31 - zapamiętywanie pozycji chwytaka,

20-23, 26-29, 32-35 - właściwe przekręcanie ścianki wykorzystujące sterowanie siłowe,

21, 27, 33 - przekręcanie odbywa się do momenty, gdy różnica między aktualną pozycją a pozycją wcześniej zapamiętaną osiągnie zadaną wartość,

37 - *Track* puszczza kostkę,

39 - *Track* oddala się od *Postumenta*,

41 - kostka jest ponownie dodawana do modelu planera.

Sekwencja ruchów potrzebna do ułożenia kostki Rubika jest wyznaczana wykorzystując program zaimplementowany przez Andrew Brown'a, który bazował na artykułach Richard Korf'a [9]. Program ten wykorzystuje algorytm przeszukiwania drzewa. Jako wynik zwraca listę ścianek, które należy przekręcić. Układanie kostki przez manipulatory polega na wykorzystaniu funkcji *rotateCube()* oraz przekładaniu przedmiotu w zależności od kolejnych wyznaczonych ruchów. Układanie kostki Rubika realizuje funkcja *solveRubikCube()*.

Rozdział 4

Wyniki testów

W tym rozdziale zostały przedstawione etapy układania kostki Rubika (sekcja 4.1), zestawienie układu pól przed i po ułożeniu kostki (sekcja 4.2) oraz podsuumowanie (sekcja 4.4).

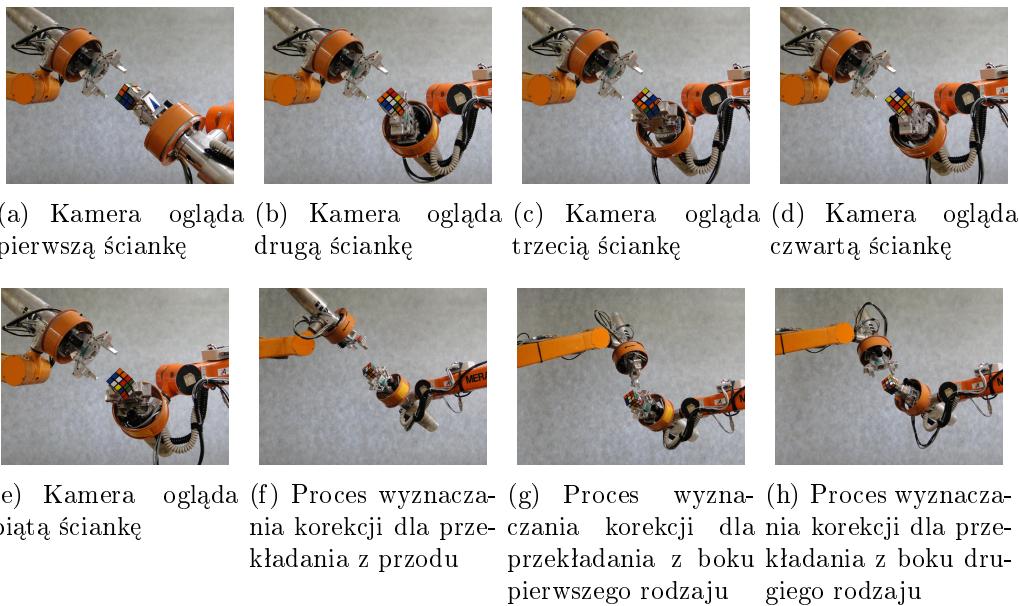
4.1 Etapy układania kostki Rubika

Pierwszą czynnością po podjęciu kostki Rubika z taśmociągu jest analiza wszystkich widocznych pól i wyznaczenie parametrów korekcji przekładania. Proces oglądania kostki przedstawia ilustracja 4.1.

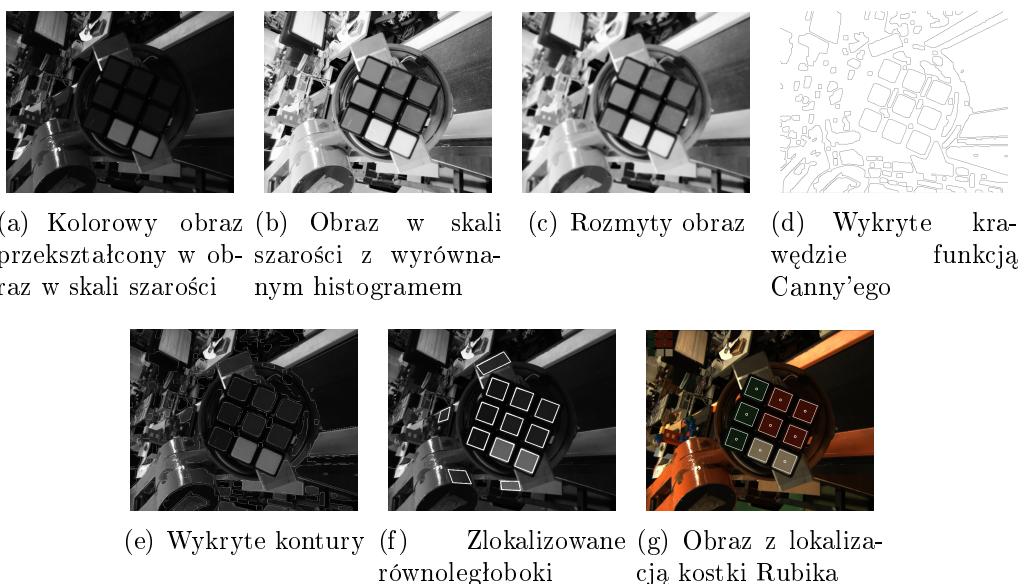
Podczas obserwacji kostki algorytm dokonuje analizy obrazu. Na początku dochodzi do desaturacji obrazu (ilustracja 4.2(a)). Następnym krokiem jest rozmycie obrazu filtrem gaussowskim oraz filtrem medianowym (rysunek 4.2(b)). Ilustracja 4.2(c) przedstawia wynik wyrównywania histogramu. Później wyznaczany jest rozkład krawędzi w obrazie (ilustracja 4.2(d)). Na rysunku 4.2(e) znajdują się wykryte kontury. Następnie algorytm oblicza równoległyboki (ilustracja 4.2(f)). Na koniec dochodzi do wyznaczenia pozycji kostki Rubika i pobrania koloru jej pól (ilustracja 4.2(g)).

Ilustracja 4.3 przedstawia sposób przekładania kostki w celu obejrzenia wcześniej zasłoniętych pól. Ponowna analiza odbywa się zgodnie z rysunkami 4.1 i 4.2. Wynik rozpoznania układu pól przedstawia rysunek 4.4.

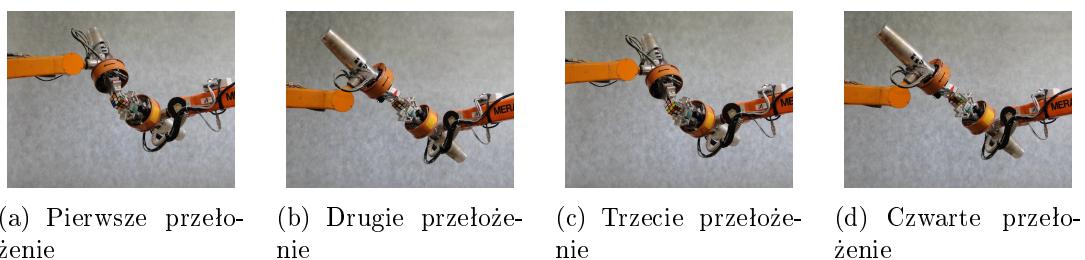
Po rozpoznaniu układu pól algorytm przystępuje do ułożenia kostki. Sposób obracania pojedyńczej ścianki przedstawia ilustracja 4.5. Wynik układania kostki Rubika przedstawia rysunek 4.6.



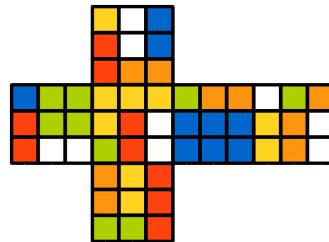
Rysunek 4.1: Siedem pozycji w jakich manipulatorzy eksponują ścianki kostki Rubika



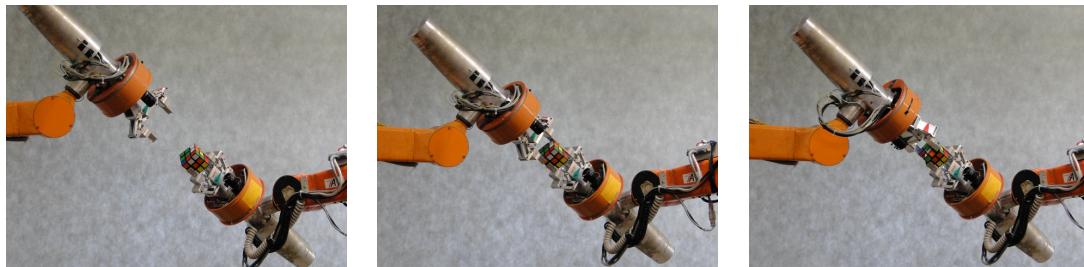
Rysunek 4.2: Etapy analizy obrazu na którym jest kostka Rubika



Rysunek 4.3: Cztery ruchy za pomocą których kostka Rubika jest obracana w chwytu o 180°



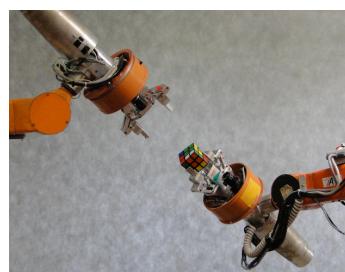
Rysunek 4.4: Obliczony układ pól kostki Rubika



(a) Przygotowanie manipulatorów do obrócenia

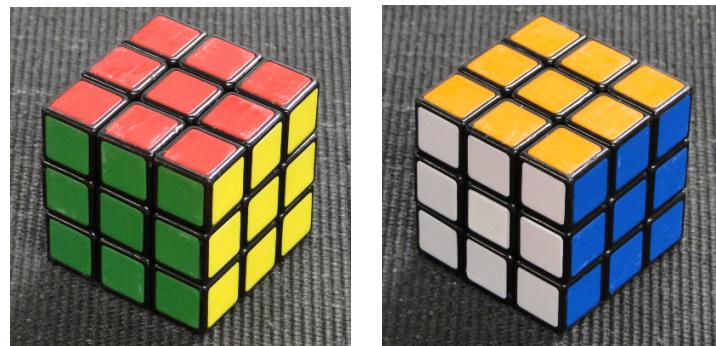
(b) Chwycenie kostki

(c) Przekręcenie ścianki



(d) Odsunięcie manipulatora

Rysunek 4.5: Proces obracania ścianki kostki Rubika



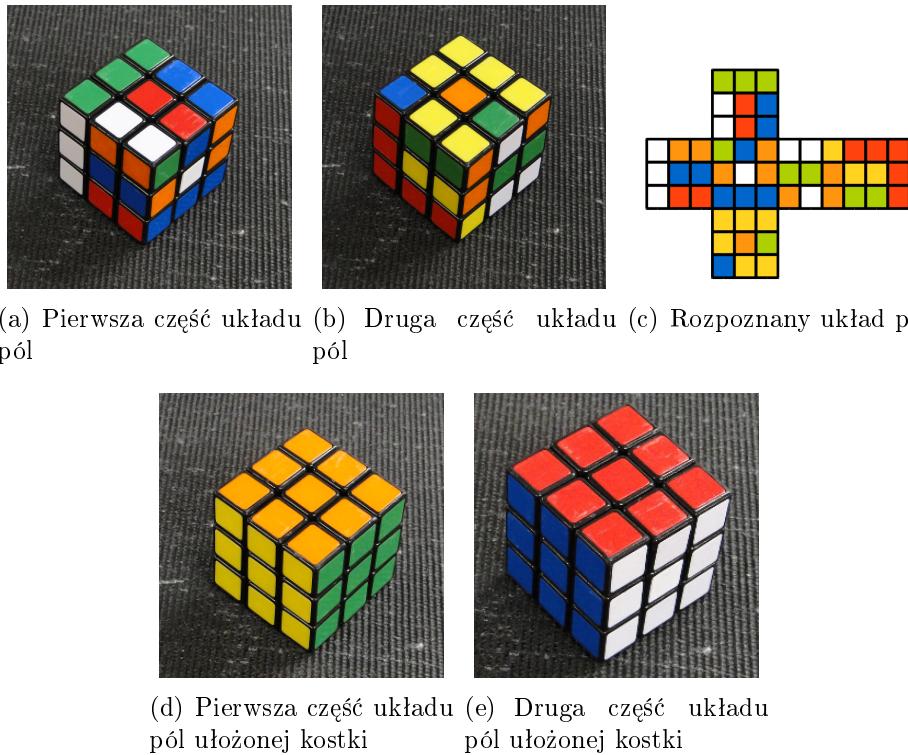
(a) Pierwsza część ułożonej kostki Rubika (b) Druga część ułożonej kostki Rubika

Rysunek 4.6: Wynik układania kostki Rubika

4.2 Serie testowe

Program został przetestowany dla dwudziestu kombinacji pól kostki Rubika w laboratoryjnych warunkach o stałym oświetleniu pomieszczenia. Należy zwrócić uwagę

na fakt, że pora dnia ma wpływ na oświetlenie stanowiska badawczego. Na ilustracjach 4.7, 4.8 i 4.9 przedstawione zostały wyniki rozpoznania układu pól kostki oraz jej ułożenia dla trzech przykładowych stanów kostki.



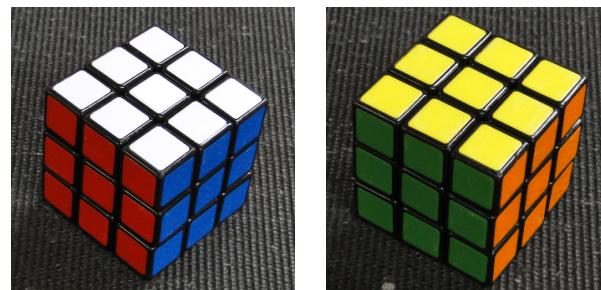
Rysunek 4.7: Wynik ułożenia kostki Rubika

4.3 Obserwacje i wnioski

Program przetestowano dla różnych kombinacji pól kostki Rubika w warunkach laboratoryjnych. Pomieszczenie posiadało stałe oświetlenie. Proces lokalizacji kostki w obrazie trwa poniżej 1 sekundy niezależnie, czy wszystkie pola są widoczne bądź część z nich jest przesłonięta. Przekładanie obiektu w manipulatorze o 90° zajmuje około 52 sekundy, natomiast przełożenie o 180° trwa około 197 sekund. Przekręcanie jednej ścianki trwa około 35 sekund. Należy zauważyć, że maksymalne prędkości robotów określone w modelu mają duży wpływ na czas manipulacji.



(a) Pierwsza część układu (b) Druga część układu (c) Rozpoznany układ pól
pół

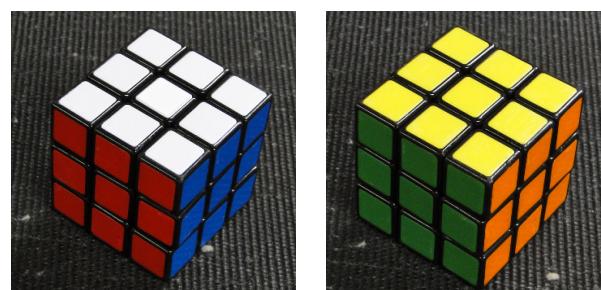


(d) Pierwsza część układu (e) Druga część układu
pół ułożonej kostki pół ułożonej kostki

Rysunek 4.8: Wynik ułożenia kostki Rubika



(a) Pierwsza część układu (b) Druga część układu (c) Rozpoznany układ pól
pół



(d) Pierwsza część układu (e) Druga część układu
pół ułożonej kostki pół ułożonej kostki

Rysunek 4.9: Wynik ułożenia kostki Rubika

4.4 Podsumowanie

Rozwój robotyki pozwala urządzeniom wykonywać coraz bardziej złożone zadania. Jednym z nich jest obserwacja i manipulacja przedmiotami.

Celem pracy było stworzenie programu, który wykorzystując dwa manipulatory ogląda kostkę Rubika, wyznacza jej układ pól i ją układa. W niniejszej pracy wykorzystano dwa manipulatory IRp-6. Projekt składa się z dwóch modułów. Jeden z nich jest odpowiedzialny za sterowanie manipulatorami, natomiast drugi za przetwarzanie obrazu z kamery. Dodatkowo w ramach niniejszej pracy stworzono środowisko umożliwiające planowanie trajektorii ruchu manipulatorów. Planer w znacznym stopniu usprawnia pracę likwidując potrzebę ręcznego projektowania trajektorii.

Priorytetem projektu była minimalizacja ryzyka błędu. Ważnym aspektem jest odpowiednia prezentacja przedmiotu kamerze. Źle dobrana pozycja może spowodować, że analiza obiektu będzie niemożliwa. Proces układania kostki musi się wykonywać w rozsądny czasie. Pojedyńcze przekręcenie ścianki kostki trwa około pół minuty, więc minimalizacja ilości ruchów jest kluczowa.

Bibliografia

- [1] Dokumentacja opencv. <http://docs.opencv.org>.
- [2] Oficjalna strona ompl. <http://ompl.kavrakilab.org/>.
- [3] Oficjalna strona openrave. <http://openrave.org/>.
- [4] Oficjalna strona point grey. <http://www.ptgrey.com>.
- [5] Oficjalna strona robot programming and pattern recognition group. <http://robotyka.ia.pw.edu.pl>.
- [6] Oficjalna strona ros. <http://www.ros.org/>.
- [7] Tomasz Winiarski Cezary Zieliński, Tomasz Kornuta. A systematic method of designing control systems for service and field robots.
- [8] Włodzimierz Kasprzak. *Rozpoznawanie obrazów i sygnałów mowy*. Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa, 2009.
- [9] Richard E. Korf. Finding optimal solutions to rubik's cube using pattern databases.
- [10] Jadwiga Sałacka. Szybkie algorytmy przeszukiwania drzew rozwiązań w celu znalezienia sekwencji ruchów ścianami kostki rubika. Praca magisterska, Politechnika Warszawska, Wydział Elektroniki i Technik Informacyjnych, 2007.
- [11] Satoshi Suzuki. Topological structural analysis of digitized binary images by border following.
- [12] Cezary Zieliński. Modeling and control of manipulators.