Jędrzej Kuryło

Interaktywne programowanie robotów przy pomocy bezprzewodowego interfejsu sterującego

Praca magisterska pod kierunkiem
dra inż. Tomasza Winiarskiego

Instytut Automatyki i Informatyki Stosowanej
Politechniki Warszawskiej

Warszawa, Wrzesień 2010

Spis treści

Streszczenie 5

1 Wstęp 7

1.1 Cel pracy . . . 7

1.2 Struktura pracy . . . 7

2 Programowanie robotów 9

2.1 Programowanie off-line . . . 9

2.1.1 Języki programowania robotów . . . 10

2.1.2 Graficzne środowiska programowania robotów . . . 12

2.2 Programowanie on-line . . . 12

2.3 Zadawanie ruchu . . . 13

3 Narzędzia 19

3.1 Manipulator IRp-6 . . . 19

3.2 Struktura ramowa MRROC++ . . . 23

3.2.1 Programowanie robotów . . . 25

3.2.2 Zadania . . . 27

3.2.3 Czujniki wirtualne . . . 30

3.2.4 Generatory trajektorii . . . 32

3.2.5 Generator trajektorii smooth . . . 35

3.3 Zadajnik ruchu . . . 36

3.3.1 Wymagania . . . 36

3.3.2 Dostępne rozwiązania . . . 38

3.3.3 Nintendo Wiimote . . . 44

3.3.4 Wykorzystywane biblioteki programowe . . . 48

4 Wymagania funkcjonalne 51

5 Implementacja 55

5.1 Architektura . . . 55

5.2 Proces VSP . . . 56

5.2.1 Kalibracja zadajnika . . . 60

5.2.2 Wyznaczanie wartości przyspieszenia . . . 62

5.2.3 Wyznaczanie orientacji zadajnika . . . 63

5.3 Proces ECP . . . 64

5.4 Komunikacja . . . 70

5.5 Generatory trajektorii . . . 72
5.5.1 Generator bazowy . . . 74
5.5.2 Ruch w przestrzeni stawów . . . 78
5.5.3 Ruch w układzie o orientacji końcówki . . . 80
5.5.4 Ruch w układzie odniesienia układu bazowego . . . 80

6 Obsługa zadania **83**
6.1 Trajektoria . . . 83
6.2 Uruchomienie zadania . . . 83
6.3 Tryby pracy . . . 84
6.4 Tryb zarządzania trajektorią . . . 84
6.5 Tryb zadawania ruchu . . . 85
6.6 Konfiguracja zadania . . . 86

7 Podsumowanie **89**

Literatura **91**

Streszczenie

Celem pracy magisterskiej było zaproponowanie i opracowanie rozwiązania wspomagającego programowanie zadania dla pojedynczego manipulatora poprzez generację punktów interpolacji dla pozycyjnych generatorów trajektorii. Zakres pracy obejmował wybór zadajnika ruchu, wykorzystywanego do przemieszczania końcówki manipulatora, i włączenie go do ramowej struktury programowej MRROC++, pod której kontrolą działa manipulator, a następnie stworzenie oprogramowania umożliwiającego programowanie pojedynczego robota.

Od zadajnika ruchu wymagano, by umożliwiał bezprzewodowe i jednoręczne sterowanie manipulatorem. Do realizacji zadania wybrany został zadajnik Wiimote firmy Nintendo. W strukturze MRROC++ zaimplementowany został proces VSP działający pod kontrolą systemu operacyjnego Linux, komunikujący się z urządzeniem za pomocą protokołu Bluetooth.

Po stronie struktury MRROC++ zaprojektowane i zaimplementowane zostało zadanie uczenie robota trajektorii przy wykorzystaniu bezprzewodowego zadajnika. Powstały również trzy generatory trajektorii, wykorzystujące odczyty z zadajnika i umożliwiające ruch końcówką manipulatora w układzie współrzędnych zaczepionym w końcówce o orientacji układu bazowego lub końcówki oraz ruch w przestrzeni stawów.

Testy powstałego rozwiązania przeprowadzono przy użyciu manipulatora IRp-6 na torze jezdnym.

1 Wstęp

1.1 Cel pracy

Współczesne roboty wkraczają w coraz to nowe dziedziny życia, wyręczając człowieka w najbardziej żmudnych pracach. Pod względem wydajności przewyższają znacznie ludzkie ręce, nic więc dziwnego że w dobie ogromnej konkurencji na rynku i coraz większej presji ekonomicznej przedsiębiorstwa produkcyjne często sięgają po zautomatyzowane procesy wytwórcze, w tym roboty przemysłowe.

Wydajność systemu robotycznego w procesie wytwórczym w prosty sposób przekłada się na zysk w postaci czasu, który z kolei można przeliczyć na pieniądze. Zwiększanie więc wydajności systemów robotycznych pozwala na realne oszczędności. Systemy robotyczne mają jednak ograniczenia, często trzeba więc szukać oszczędności gdzie indziej.

Istotnym kosztem w życiu systemu robotycznego jest proces wytwarzania oprogramowania nim sterującego. Jest to złożony proces, w którym za pomocą różnych środków programista zapisuje wiedzę na temat oczekiwanego działania systemu robotycznego. Im wyższa jakość i wydajność dostępnych środków, tym krótsze, a więc tańsze, będzie przygotowanie robota do pracy.

Celem mojej pracy magisterskiej było opracowanie bezprzewodowego narzędzia, które wspomoże programistę w tworzeniu fragmentów aplikacji poprzez generowanie trajektorii bez sprzężenia zwrotnego, tak jak to miało miejsce w tradycyjnej robotyce przemysłowej. Implementacja rozwiązania miała być poprzedzona analizą dostępnych metod programowania robotów i sposobów zadawania ruchu.

1.2 Struktura pracy

Rozdział 2 ustanawia kontekst teoretyczny niniejszej pracy. Przedstawiam i porównuję w nim główne rodziny metod programowania robotów - programowanie on-line oraz programowanie off-line. Opisuję tutaj cechy charakterystyczne obu metod oraz przedstawiam ich wady i zalety. W rozdziale tym przedstawiam również sposoby zadawania ruchu robotowi.

W 3 rozdziale opisuję podstawowe narzędzia wykorzystywane w pracy. Przedstawiam programową strukturę ramową MRROC++ - środowisko, w oparciu o które realizowana jest niniejsza praca magisterska. Pokazuję, jak w oparciu o tę strukturę zaimplementować wirtualny czujnik, generator trajektorii oraz zadanie. Opisuję również generator smooth, zaimplementowany w ramach pracy inżynierskiej[1] przez Rafała Tulwina, a wykorzystywany przeze mnie w procesie uczenia robota. Nakreśliwszy w 2 rozdziale teorię programowania robotów, w tym rozdziale skupiam się na metodach programowania robotów dostęp-

nych obecnie w strukturze MRROC++. W drugiej części rozdziału opisuję zagadnienia związane z wyborem i obsługą zastosowanego w pracy zadajnika ruchu. Przedstawiam wymagania postawione zadajnikowi oraz przeprowadzam analizę porównawczą oferowanych na rynku gier zadajników pod kątem spełnienia tych wymagań. Następnie opisuję wybrany do realizacji pracy zadajnik Wiimote firmy Nintendo oraz biblioteki programowe wspomagające jego wykorzystanie.

W rozdziale 4 przedstawiłem wymagania funkcjonalne postawione przed stworzonym przeze mnie rozwiązaniem. Spełnienie tych wymagań powinno skutkować powstaniem rozwiązania użytecznego dla operatora systemu robotycznego, które ułatwi i przyspieszy zadanie uczenia robota trajektorii.

Rozdział 5 zawiera zagadnienia związane z implementacją zadania. Opisuję w nim architekturę mojego rozwiązania, a następnie poszczególne komponenty wchodzące w skład rozwiązania. Na początku przedstawiam proces VSP, czyli implementację wirtualnego czujnika struktury MRROC++, realizującego komunikację z zadajnikiem ruchu. Następnie przedstawiam proces ECP realizujący zadanie programowania robotów, wykorzystujące wspomniany czujnik wirtualny. Kolejnym krokiem jest opisanie sposobu komunikacji pomiędzy dwoma wspomnianymi procesami oraz przesyłanych między nimi struktur danych. Ostatnia część rozdziału 5 zawiera opis używanych przez zadanie generatorów ruchu, wykorzystujących odczyty z zadajnika do generowania trajektorii, które również powstały w ramach niniejszej pracy.

Rozdział 6 niniejszej pracy przybliża obsługę zadania uczenia robota i stanowić ma namiastkę podręcznika użytkowania rozwiązania. Opisuję w nim sposób sterowania wykonaniem zadania programowania robota za pomocą zadajnika ruchu oraz metody zadawania ruchu manipulatorowi.

Na końcu znajduje się podsumowanie wykonanej pracy oraz perspektywy rozwoju zaimplementowanego rozwiązania.

2 Programowanie robotów

Współczesne roboty wkraczają w coraz to nowe dziedziny życia, wyręczając człowieka w najbardziej żmudnych pracach. Pod względem wydajności przewyższają znacznie ludzkie ręce, nic więc dziwnego że w dobie ogromnej konkurencji na rynku i coraz większej presji ekonomicznej przedsiębiorstwa produkcyjne często sięgają po zautomatyzowane procesy wytwórcze, w tym roboty przemysłowe.

W literaturze brak jest jednoznacznej definicji robota przemysłowego. Norma ISO TR 8373 przyjmuje następującą definicję[2]:

robot przemysłowy to automatyczny, wielofunkcyjny, programowalny manipulator, mający kilka stopni swobody, wyposażony w serwonapędy, zdolny do przemieszczania materiałów, części narzędzi lub specjalizowanych urządzeń, wykonujący różne programowalne operacje w celu realizacji różnorodnych zadań

O ile sama definicja robota przemysłowego jest dosyć ogólna, to zwraca uwagę na istotną ich cechę - programowalność. Programowanie robota to formułowanie zadania dla robota, tj. zapisywanie wiedzy o oczekiwanym jego zachowaniu. I tak jak od systemu robotycznego oczekuje się wysokiej wydajności pracy, tak samo od programisty tego systemu oczekuje się możliwie wydajnego przygotowania robota do pracy. Dlatego też równolegle z rozwojem technologicznym samych robotów przemysłowych rozwijały się metody ich programowania.

Pierwsze roboty były urządzeniami typowo mechanicznymi i pozbawione były centralnego układu sterującego. Jedynym sposobem oddziaływania na nie przez programistę było bezpośrednie wpływanie na mechanizmy. Rozwój technologii i wzrost zaawansowania zarówno układów sterujących, jak i samych robotów, stymulowały z kolei rozwój metod programowania robotów. Współczesny programista systemów robotycznych ma do dyspozycji bardzo szerokie spektrum metod - od ręcznego programowania mechanicznych manipulatorów po programowanie off-line - oparte często na zaawansowanych rozwiązaniach komputerowych i nie angażujące rzeczywistych robotów. Niniejszy rozdział ma na celu przybliżenie dostępnych programiście środków programowania robotów.

Ogół metod programowania podzielić można na rodzinę metod off-line, która nie wymaga zaangażowania rzeczywistego robota w procesie definiowania zadania, oraz metod on-line, które go aktywnie wykorzystują.

2.1 Programowanie off-line

Czas, w którym robot jest zaangażowany w proces programowania, z ekonomicznego punktu widzenia jest czasem nieproduktywnym. Metody programowania off-line rozu-

miane są jako metody, które w proces definiowania zadania nie angażują rzeczywistego robota, dzięki czemu nieproduktywny czas robota ograniczają do minimum. Rozwój metod off-line umożliwił szybki rozwój technik komputerowych, dzięki czemu możliwe było stworzenie zaawansowanych środków programowania robotów.

Narzędzia metod off-line umożliwiają definiowanie zadania za pomocą języków wysokiego poziomu i wyrażanie ich w postaci tekstowej lub graficznej. Od wykorzystywanego narzędzia oczekuje się przede wszystkim tego, że bazując na modelu rzeczywistej sceny i rzeczywistego robota będzie w stanie zweryfikować poprawność stworzonego programu i zamodelować jego wykonanie. Spełnienie tego wymagania bowiem pozwoli uniknąć potrzeby angażowania rzeczywistego robota i rzeczywistej sceny w proces programowania.

Języki programowania wykorzystywane w metodach off-line charakteryzują się dużą liczbą dostępnych instrukcji, dzięki czemu umożliwiają oprogramowanie nawet bardzo złożonych zadań. Instrukcje warunkowe pozwalają na wykorzystanie danych sensorycznych i uzależnienie przebiegu wykonania zadania od danych z czujników. Tekstowa lub graficzna forma reprezentacji zadania pozwala na łatwą i tanią modyfikację zadania.

Podstawową trudnością metod off-line jest konieczność zamodelowania sceny i robota oraz opisania wzajemnych powiązań między wszystkimi elementami. Jest to proces czasochłonny, podczas którego łatwo o błędy. Samo zdefiniowanie zadania również nie musi być trywialne i w zależności od stopnia zaawansowania środowiska programistycznego wymagać może od operatora znacznych umiejętności programistycznych. Dlatego też metody off-line najlepiej sprawdzają się w sytuacjach, kiedy koszt czasu nieproduktywnego robota jest istotnie wysoki.

2.1.1 Języki programowania robotów

Systemy programowania robotów off-line pozwalają zapisać sekwencję instrukcji w postaci tekstowej lub graficznej. Zadanie definiuje się przy użyciu jednego z dostępnych języków wysokiego poziomu, w którym zapisuje się oczekiwaną logikę oraz zależności wykonania zadania od danych sensorycznych czy danych pobranych od operatora.

Języki programowania pozwalają na sterowanie robotem na różnych poziomach szczegółowości, a więc różnią się poziomem automatyzacji ruchu. G. Kost wprowadził następującą klasyfikację języków programowania[2].

1. *poziom ruchu* pozwala na programowanie ruchu manipulatora niezależnie na każdej z osi jego kinematyki. Zdefiniowanie zadania w takiej reprezentacji jest złożone i nieintuicyjne, a sama reprezentacja nie znalazła wielu zastosowań.

2. *poziom manipulacji* pozwala na programowanie pełnego ruchu manipulatora, tj. zadawania ruchu końcówki. Wymaga to zdefiniowania w systemie modelu kinematyki

robota.

3. *poziom obiektu* pozwala na zadawanie ruchu w kontekście obiektów znajdujących się w scenie robota, co wymaga stworzenia modelu sceny i wykorzystywania danych sensorycznych.

4. *poziom zadania* umożliwia opisywanie złożonych zadań poprzez odwoływanie się do instrukcji reprezentujących typowe zadania

Języki programowania robotów implementowane są jako języki specjalizowane oraz rozszerzenia istniejących języków ogólnego przeznaczenia[3].

Najpopularniejszymi językami programowania robotów są języki dedykowane konkretnym robotom i wspierane przez ich producentów. Firma KUKA dostarcza wraz ze swoimi manipulatorami środowisko KUKA.Sim[4], umożliwiające programowanie off-line ich robotów. Zadania implementuje się w stworzonym przez firmę KUKA języku programowania, udostępniającym proste instrukcje na poziomie manipulacji. Producent dostarcza dodatkowo również środowisko symulacyjne umożliwiające weryfikowanie poprawności programu.

Wadą języków dedykowanych jest przede wszystkim ich nieprzenośność. Zaprogramowanie systemu robotycznego, składającego się z robotów różnych producentów, wymaga znajomości kilku języków programowania, co znacznie podnosi koszt i czas przygotowania systemu do pracy.

Wady tej pozbawione są języki ogólnego przeznaczenia. Są one rozszerzane o biblioteki wspierające obsługę poszczególnych robotów przemysłowych. Oprócz standardowego zestawu operacji czy typów danych udostępniają one operacje związane ze sterowaniem wybranym robotem przemysłowym. Umożliwiają one definiowanie ruchu czy korzystanie z danych sensorycznych. Innym sposobem jest wprowadzanie dodatkowej warstwy abstrakcji nad klasami reprezentującymi poszczególne roboty. Przykładem takiego rozwiązania jest Motion Description Language[1][5], rozszerzenie dla języka Java, dedykowane programowaniu robotów mobilnych. Podobne rozszerzenia istnieją również dla języka C++[6] czy Python[7]. Do opisu zachowania robotów wykorzystywany jest również język XML[8], który przed wykonaniem zadania jest interpretowany w ciąg instrukcji dla sterownika robota.

W oparciu o języki ogólnego przeznaczenia tworzy się z kolei struktury programowe. Są to najczęściej zbiory bibliotek i modułów ułatwiających wytwarzanie sterowników dla systemów robotycznych. Struktury takie zawierają również narzędzia wspomagające programowanie robotów. Przykładami takiej struktury jest opisana w rozdziale 3.2 struktura MRROC++.

[1]Język opisu ruchu

2.1.2 Graficzne środowiska programowania robotów

Drugą grupą środowisk programowania robotów są języki graficzne. Oczekiwane zachowanie robota wyraża się w nich za pomocą ikon lub symboli. Języki te poświęcają część możliwości języków tekstowych, ograniczając użytkownika do używania zestawu predefiniowanych instrukcji i operacji. Są jednak przystępniejsze i nie wymagają z reguły zaawansowanej wiedzy programistycznej. Zachowanie robota wyraża się w tych językach za pomocą diagramów przepływu sterowania lub grafów. Przykładem takiego środowiska jest narzędzie *dd-designer* wykorzystujące grafy do opisu zachowania robota[9].

2.2 Programowanie on-line

Rodzina metod programowania robotów on-line charakteryzuje się tym, że rzeczywisty robot jest aktywnie wykorzystywany w procesie definiowania zadania. Metody te oparte są na uczeniu robota oczekiwanej trajektorii. Uczenie robota trajektorii polega na poprowadzenia ramienia manipulatora wzdłuż oczekiwanej ścieżki. Operator porusza ramieniem robota ręcznie lub używając napędów robota, a kolejne punkty trajektorii zostają zapamiętane. Wykonanie programu stworzonego w ten sposób polega na automatycznym odtworzeniu przez robota zaprogramowanej ścieżki.

Zapamiętywanie trajektorii realizowane jest na dwa sposoby, różniące się zakresem zapamiętywanych danych oraz metodą odtwarzania na ich podstawie trajektorii.

Point-to-point W tej metodzie operator robota przemieszcza ramie dowolną ścieżką pomiędzy kolejnymi punktami charakterystycznymi trajektorii, a następnie zapamiętuje te punkty. Stworzona w ten sposób trajektoria zawiera zatem wiedzę o tych punktach, nie opisuje zaś w żaden sposób trajektorii ruchu pomiędzy węzłami. Odtworzenie na tej podstawie trajektorii wymaga interpolacji ścieżki ruchu na podstawie zapamiętanych punktów charakterystycznych, a operator nie ma wpływu na ścieżkę ruchu pomiędzy węzłami interpolacji.

Continuous Path W tej metodzie operator robota przemieszcza ramie wzdłuż oczekiwanej trajektorii, a aktualne położenie ramienia jest zapamiętywane co stały kwant czasu. Zapisana w ten sposób zawiera o wiele większą ilość węzłów, dzięki czemu w przybliżeniu oddaje rzeczywistą ścieżkę przebytą przez ramię robota. Metoda ta pozwala również odwzorować prędkość ruchu, gdyż czas pomiędzy osiągnięciem przez ramię robota kolejnych zapamiętanych węzłów jest stały. Kolejne węzły znajdują się na tyle blisko, że zbędna staje się procedura interpolacji, dzięki czemu odtworzenie takiej trajektorii jest o wiele prostsze.

Podstawową zaletą metod on-line jest ich prostota. Nie wymagają one od operatora umiejętności programistycznych, a jedynie wiedzę o pożądanej trajektorii ruchu.

W podstawowej postaci metody on-line mają wiele wad. Kod programu to niewiele mówiąca sekwencja punktów, wymagana jest zatem dodatkowa forma dokumentacji. Prostota zadawania trajektorii uniemożliwia wykorzystanie danych sensorycznych czy warunkowe wykonanie części programu. Zrealizowanie tych mechanizmów jest możliwe, wymaga jednak o wiele bardziej złożononego środowiska programistycznego oraz bardziej złożonego zapisu programu niż sekwencja współrzędnych węzłów.

Największą jednak wadą metod on-line jest nieproduktywny czas robota poświęcony na jego programowanie.

2.3 Zadawanie ruchu

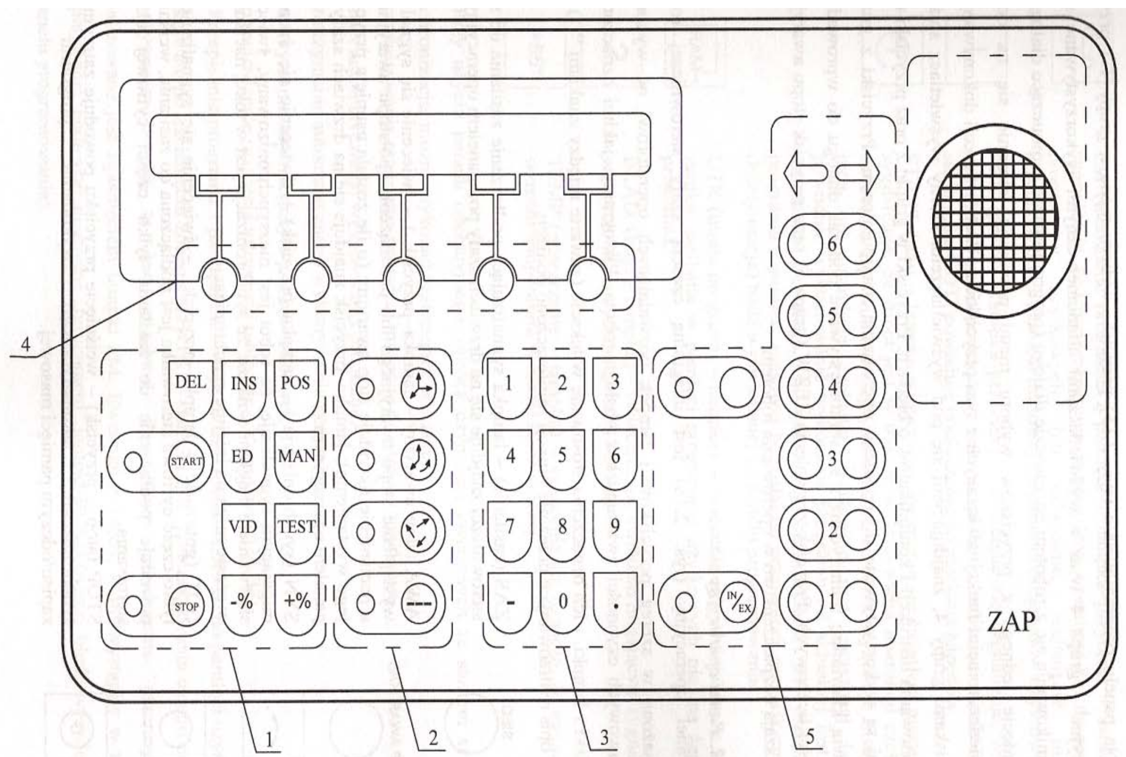
Operator może zadawać ruch robota ręcznie lub wykorzystać udostępniany przez system robotyczny panel sterowniczy. Samo wykonanie zadania uczenia robota, np. dodanie punktu do trajektorii czy jej odtworzenie, wymaga z kolei udostępnienia przez system robotyczny interfejsu sterowania zadaniem. Najczęściej panele sterownicze i i interfejs sterowania zadaniem są zintegrowane w jeden panel programowania.

Panel programowania robota (teach pendant, teach box) to urządzenie pozwalające sterować robotem spoza jego przestrzeni roboczej. Panele te najczęściej zawierają przycisk stopu awaryjnego, wyświetlacz, klawiaturę umożliwiającą wpisywanie instrukcji czy zestaw przycisków z przypisanymi funkcjami. Panel taki, oprócz poruszania ramieniem robota pozwala również sterować przebiegiem wykonania zadania uczenia robota. Dodatkowo panel udostępnia operatorowi informację zwrotną z systemu robotycznego.

Poszczególne panele różnią się sposobem sterowania przebiegiem wykonania zadania oraz zakresem informacji zwrotnej udostępnianej operatorowi. Ze względu na tematykę niniejszej pracy na szczególną uwagę zasługują oferowane przez nie metody zadawania ruchu. Ich mnogość oraz podejmowane próby opracowania nowych rozwiązań pokazują, że zagadnienie zadawania ruchu jest zagadnieniem nietrywialnym.

Panel programowania robota Irp-6 Dużą grupę paneli programowania stanowią panele oferowane przez producentów robotów. Schemat panelu programowania robota Irp-6[14] firmy ABB przedstawia rysunek 1. Panel składa się z wyświetlacza alfanumerycznego rozmiaru 2x40 znaków, klawiatury z predefiniowanymi funkcjami przycisków oraz przycisku stopu awaryjnego. Sam panel jest przenośny i połączony jest z szafą sterowniczą za pomocą przewodu.

Na panelu można wyróżnić pięć grup przycisków:



Rysunek 1: Panel programowania robota Irp-6[14]

1. przyciski menu - umożliwiają przeglądanie listy dostępnych programów oraz ich uruchamianie.
2. przyciski trybu poruszania robotem - pozwalają na wybór układu współrzędnych, w których zadawany będzie ruch. Do wyboru jest układ kartezjański, cylindryczny, układ współrzędnych wewnętrznych robota oraz układ współrzędnych z redukcją prędkości ruchu robota.
3. klawiatura numeryczna, pozwalająca wprowadzać liczbowe parametry zadania
4. przyciski funkcyjne, których rola zależy od aktualnie wykonywanego zadania
5. przyciski ruchów ręcznych, umożliwiające ręczne poruszanie robotem poprzez zadawanie przyrostów na poszczególnych osiach robota

Panel programowania robota Irp-6 umożliwia definiowanie nawet złożonych działań dzięki bogatemu zbiorowi dostępnych instrukcji. Umożliwia także ruchy ręczne robotem poprzez zadawanie współrzędnych docelowych oraz ruch przyrostowy na poszczególnych osiach, dzięki czemu możliwe jest jego wykorzystanie do uczenia robota trajektorii. Zadawanie ruchu poprzez wprowadzanie współrzędnych czy zadawanie przyrostów jest czasochłonne i mało intuicyjne, jednak metoda ta jest najczęściej oferowana przez panele programowania robotów przemysłowych.

Konsola sterownicza struktury MRROC++ Graficzna konsola struktury MRROC++ (rysunek 8) stanowi programową realizację konsoli sterowniczej robota. Umożliwia ona ruchy ręczne obsługiwanymi przez strukturę robotami poprzez zadawanie współrzędnych docelowych oraz zadawanie przyrostów na poszczególnych osiach. Zadawanie ruchu możliwe jest w przestrzeni zmiennych wewnętrznych robota, tj. stawów i napędów, oraz w zewnętrznych układach współrzędnych oś-kąt i układzie kątów Eulera. Konsola umożliwia również uruchamianie zadań i sterowanie przebiegiem ich wykonania.

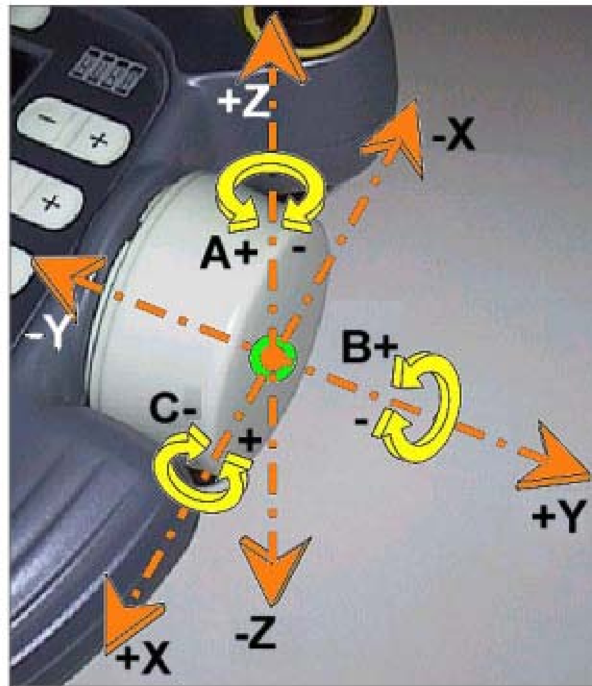
Struktura MRROC++ udostępnia również zadanie uczenia robota trajektorii pozycyjnej. Zadanie to pozwala na zapamiętywanie sekwencji położeń końcówki manipulatora oraz jej odtwarzanie. W połączeniu graficzną konsolą sterowniczą stanowi programową realizację panelu programowania robota.

Panel programowania robota KUKA Ciekawe rozwiązanie zadawania ruchu robotowi oferuje panel programowania[10] dla robotów KUKA (rysunek 2). Pracuje on pod kontrolą systemu operacyjnego Windows. Wyposażony jest w kolorowy wyświetlacz VGA, pełnowymiarową klawiaturę alfanumeryczną, panel umożliwiający uruchamianie i wyłączanie robota, a także przycisk stopu awaryjnego. Panel pozwala na programowanie robota przy użyciu języka KRL, a także zadawanie ruchów ręcznych poprzez wprowadzanie współrzędnych docelowych lub zadawanie przyrostów na poszczególnych osiach manipulatora.

Rysunek 2: Panel programowania robotów KUKA

Najciekawszym elementem panelu jest umieszczona z prawej strony panelu mysz przestrzenna o sześciu stopniach swobody (rysunek 3). Pozwala ona zadawać nie tylko kieru-

nek przemieszczenia się robota czy obrót wokół osi wybranego układu współrzędnych, ale również zadawanie prędkości poprzez przykładanie do myszy odpowiedniej siły.



Rysunek 3: Mysz przestrzenna KUKA

Panel programowania robota MOTOMAN Firma MOTOMAN opracowała własne rozwiązanie MOTOMAN-ET[11], umożliwiające zadawanie ruchu, wykorzystujące czujnik siły i momentu siły. Zasada działania jest podobna do myszą przestrzennej firmy KUKA, w których czujnik siły umieszczony był w panelu programowania. W rozwiązaniu firmy MOTOMAN czujnik umieszczony jest w końcówce manipulatora, a operator za pomocą uchwytu połączonego z czujnikiem może zadawać ruch. Dodatkowe oprogramowanie na podstawie odczytów z czujnika porusza robotem zgodnie z wolą operatora. Zaletą takiej formy zadawania ruchu jest jej intuicyjność. Oczywistą wadą jest jednak fakt obecności operatora w przestrzeni roboczej manipulatora, co jest niewskazane ze względów bezpieczeństwa.

Metody eksperymentalne W poszukiwaniu lepszych sposobów zadawania ruchu inżynierowie coraz częściej sięgają po niestandardowe metody sterowania robotem.

Yong Xu, Matthieu Guillemot i Toyoaki Nishida do sterowania robotem mobilnym postanowili wykorzystać dłonie operatora[12]. Danymi sterującymi był jednak nie obraz z kamery, lecz odczyty z akcelerometru przymocowanego do dłoni operatora. Z tego też powodu nieistotne było ułożenie dłoni, a jedynie rozkład sił grawitacji na poszczególnych

osiach akcelerometru. Do różnych konfiguracji rozkładu przypisany został niewielki zbiór instrukcji dla robota. Okazał się on być wystarczający do wykonania prostego zadania, polegającego przemieszczaniu robota mobilnego na płaszczyźnie.

Pedro Neto i J. Norberto Pires także podjęli próbę[13] wykorzystania akcelerometrów do sterowania robotem. Na lewej i prawej ręce operatora zamontowali oni trójosiowy akcelerometr. Odczyty z prawej ręki były wykorzystane do rozpoznawania gestów przy użyciu sztucznej sieci neuronowej. Odczyty z lewej ręki były wykorzystywane do sterowania systemem, a w szczególności do inicjalizowania rozpoznawania gestu oraz do przerywania ruchu robota. Rozpoznawane gesty ograniczone zostały do przemieszczeń akcelerometru wzdłuż jego osi oraz obrotów wokół nich. Na tej podstawie inicjalizowany był ruch robotem na pojedynczej współrzędnej ze stałą prędkością, trwający do momentu przerwania ruchu przez operatora gestem lewej ręki. Opisane rozwiązanie umożliwiło przemieszczanie końcówki robota przemysłowego w przestrzeni.

3 Narzędzia

Rozwiązanie umożliwiające interaktywne programowanie robotów zostało stworzone i sprawdzone przy wykorzystaniu robota przemysłowego IRp-6. Środowiskiem programowym była struktura programowa MRROC++, służąca do tworzenia sterowników dla systemów wielorobotowych. Interfejs człowiek-maszyna stanowił zadajnik Wiimote firmy Nintendo. Wspomniane elementy zostaną dokładnie opisane w niniejszym rozdziale.

3.1 Manipulator IRp-6

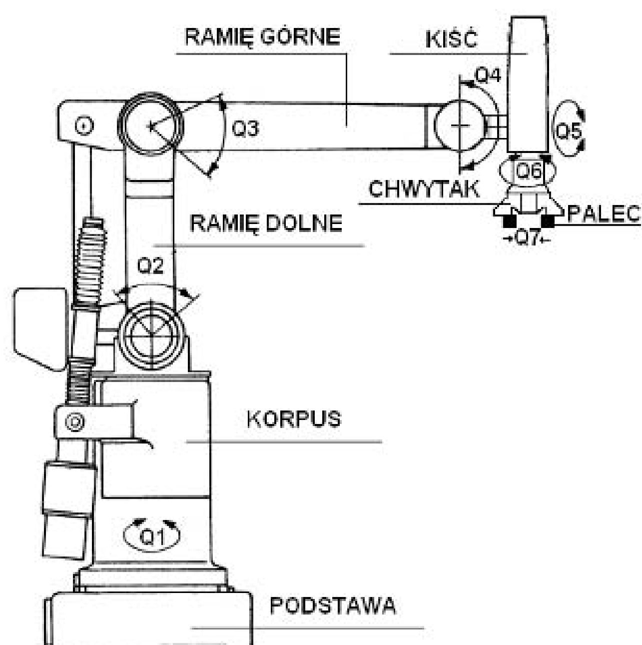
Pierwowzór manipulatora IRp-6 został opracowany w latach siedemdziesiątych XX wieku przez szwedzką firmę ASEA, istniejącą na rynku elektroenergetycznym i transportowym od 1883 roku, a od 1988 roku wchodzącą w skład grupy ABB. W 1977 roku Przemysłowy Instytut Automatyki i Pomiarów (PIAP) zakupił licencję na produkcję tego typu robotów i na jej podstawie od końca lat siedemdziesiątych produkował roboty serii IRp i IRb. Stanowisko laboratoryjne robota IRp-6 składa się z dwóch części oddzielonych konstrukcyjnie - manipulacyjnej i sterowniczej.

Rysunek 4: Manipulator IRp-6

Część manipulacyjna robota IRp-6 Część manipulacyjną stanowiska laboratoryjnego stanowi manipulator IRp-6. Schemat łańcucha kinematycznego robota IRp-6 przed-

stawiłem na rysunku 5. Oryginalnie manipulator składa się z:

- podstawy
- korpusu obrotowego
- ramienia dolnego
- ramienia górnego
- członu roboczego
- kiści
- chwytaka



Rysunek 5: Łańcuch kinematyczny manipulatora IRp-6

Manipulator znajdujący się w Laboratorium Robotyki został dodatkowo umiejscowiony na torze jezdnym(0), zrealizowanym poprzez pozycjoner liniowy PL-02c/1000. Umożliwia on przemieszczanie liniowe części manipulacyjnej, zwiększając ilość stopni swobody o jeden. Łańcuch kinematyczny został również rozszerzony o obrotową kiść, w której umieszczony został chwytak. Ograniczenia ruchu manipulatora znajdują się w tabeli 1.

Robot Irp-6 na stanowisku laboratoryjnym posiada zatem 6 osi manipulacji oraz jedną oś związaną z torowiskiem pozycjonera. Można zatem w zastosowanym zmodyfikowanym manipulatorze wyróżnić następujące pary kinematyczne:

Q0 przesuwna para kinematyczna piątej klasy, tj. posiadające 5 ograniczeń ruchu, a więc jeden stopień swobody

Q1 Q2 Q3 Q4 Q5 Q6 pary kinematyczne obrotowe piątej klasy

Zmodyfikowany manipulator IRp-6 jest zatem robotem o siedmiu stopniach swobody i jednoznacznie określonym ruchu wszystkich członów. Liczbę stopni swobody można wyznaczyć z zależności:

$$W = 6 * n - 7 * p5 = 6 * 7 - 7 * 5 = 42 - 35 = 7$$

gdzie:

n liczba członów ruchomych (człony 1, 2, 3, 4, 5, 6, 7)

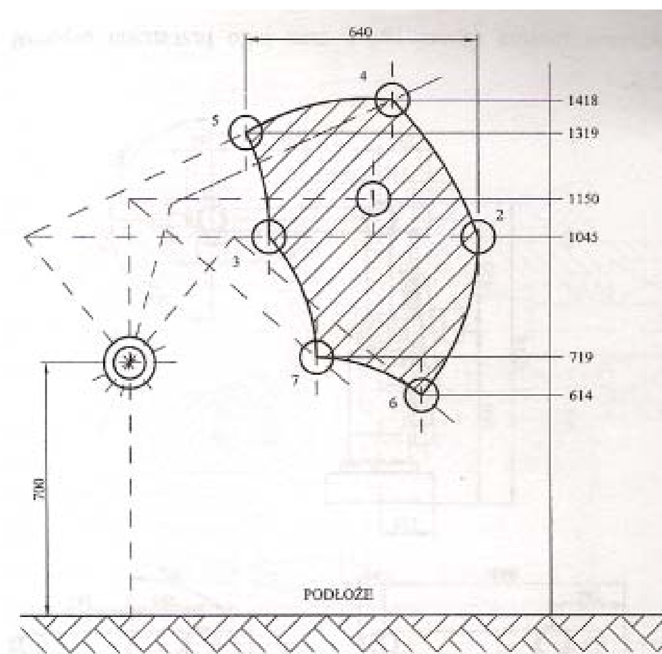
p5 liczba par kinematycznych piątej klasy (na stanowisku laboratoryjnym - wszystkie pary kinematyczne)

Para kinematyczna	**Ruch**	**Zakres ruchu**	**Ograniczenia prędkości**
Q0	przesunięcie podstawy na torze jezdnym	$1m$	*b.d.*
Q1	obrót korpusu względem podstawy	$340\,°$	$95\,°/s$
Q2	obrót ramienia dolnego	$\pm 40\,°$	
Q3	obrót ramienia górnego	$-25\,°$ - $40\,°$	
Q2 Q3	poziomy ruch ramienia	*n.d.*	$0.75m/s$
Q2 Q3	pionowy ruch ramienia	*n.d.*	$1.1m/s$
Q4	pochylenie kiści	$90\,°$	$115\,°/s$
Q5	obrót kiści	$\pm 180\,°$	$195\,°/s$
Q6	obrót chwytaka	*b.d.*	*b.d.*

Tablica 1: Ograniczenia ruchu manipulatora IRp-6[14]

Przestrzeń robocza manipulatora przedstawiona została na rysunku 6.

IRp-6 jest robotem elektrycznym. Ruch części manipulacyjnej realizowany jest przez zespoły silników prądu stałego, odpowiednich przekładni, resolvera lub enkodera do pomiaru przemieszczenia kątowego i prądnicy tachometrycznej do pomiaru rzeczywistej prędkości obrotowej.



Rysunek 6: Przestrzeń robocza manipulatora IRp-6[14]

Pozycjoner liniowy przemieszcza część manipulacyjną robota IRp-6. Składa się z podstawy, stanowiącej konstrukcję nośną zespołu jezdnego. W podstawie znajduje się zespół prowadnic i śruba kulowa. Po podstawie porusza się wózek, składający się z płyty nośnej, korpusu i wspornika łożyska. Ruch wózka po platformie jest realizowany przez zespół śruby kulowej zamontowanej w podstawie oraz przekładni pasowej zębatej, umieszczonej w wózku. Pozycjoner zapewnia zakres przesuwu długości 1m oraz dokładność pozycjonowania 0.1mm. Masa własna pozycjonera to 200kg, a nośność 150kg.

Obrót wokół podstawy umożliwia umieszczony w górnej części podstawy zespół silnika z przekładnią falową.

Ruch ramienia dolnego i górnego realizowany jest przez dwa silniki umieszczone w korpusie i przekładnie śrubowe toczne. Dolne ramie jest ułożyskowane obrotowo w korpusie, a ruch przekładni przekazywany jest do niego za pomocą dźwigni. W górnej części dolnego ramienia ułożyskowane zostało ramię górne, do którego ruch obrotowy silnika jest przekazywany za pomocą przekładni na dwa pręty przymocowane do zakrętki przekładni, które tworzą z dolnym i górnym ramieniem równoległobok. Manipulator jest wyposażony w układ korekty ruchu obrotowego ramion, umożliwiający poruszanie końcówki manipulatora ruchem prostoliniowym wzdłuż osi poziomej i osi pionowej.

Ruch przegubem roboczym wywoływany jest przez przekładnie falowe i zespół cięgien. Dodatkowo wewnątrz ramienia górnego robota poprowadzone są przewody elektryczne zasilające chwytak lub dowolne inne narzędzie wykonawcze umieszczone w kiści.

Ramiona robota zostały wykonane jako odlewy aluminiowe. Zapewnia to podzespołom robota dobrą ochronę przed warunkami zewnętrznymi i uszkodzeniami fizycznymi. Masa części manipulacyjnej robota wynosi 125kg, a dopuszczalne obciążenie łącznie z ciężarem

chwytaka to 6kg.

Część sterownicza robota IRp-6 Część sterowniczą robota stanowi szafa sterownicza. Jest to konstrukcja z blachy stalowej i zawiera cztery zespoły funkcjonalne:

- zespół zasilania układów elektrycznych i elektronicznych, składający się z zestawu zasilaczy sieciowych i układów przekaźnikowo-stycznikowych
- zespół zasilania silników prądu stałego, składający się z regulatorów prędkości obrotowej silników i stanowiący część wykonawczą sterowania silnikami części manipulacyjnej, sterowany z zespołu sterującego
- zespół sterujący
- obwód stopu awaryjnego, służący do odłączenia zasilania silników w przypadku wciśnięcia przycisku stopu awaryjnego lub zadziałania jednego z zabezpieczeń nadmiarowo-prądowych

3.2 Struktura ramowa MRROC++

MRROC++[15] to programowa struktura ramowa wspomagająca wytwarzanie sterowników dla systemów wielorobotowych. Struktura ta powstała i jest rozwijana w Instytucie Automatyki i Informatyki Stosowanej Politechniki Warszawskiej. Historia struktury sięga początku lat dziewięćdziesiątych - wtedy to powstał jej pierwowzór RORC[2][16], wspomagający tworzenie sterowników dla pojedynczych robotów. W 1995 powstała struktura MRROC[3][16], wspierająca systemy wielorobotowe. MRROC stworzony został z wykorzystaniem najpopularniejszego w tamtych czasach proceduralnego paradygmatu programowania, w przeciwieństwie do powstałej kilka lat później obiektowej struktury MRROC++ stworzonej w języku C++. Struktura ta rozwijana jest po dziś dzień - w dużej mierze jest to zasługa faktu, że stanowi ona podstawowe narzędzie wykorzystywane w pracach dyplomowych realizowanych w Laboratorium Robotyki.

Struktura MRROC++ od chwili powstania została wykorzystana do sterowania wykonaniem różnorodnych zadań, m.in:

- polerowania i frezowania[17] przez robota RNT
- układania kostki Rubika[18] przez dwa zmodyfikowane manipulatory IRP-6 (rys. 7)
- gry w warcaby[17, 19]

[2]Research-Oriented Robot Controller
[3]Multi-Robot Research-Oriented Controller

- podążanie za konturem[17]
- rysowanie[17]

Rysunek 7: Dwa roboty IRp-6 układające kostkę rubika

Struktura MRROC++ zawiera bibliotekę modułów programowych oraz wzorców, na których bazie można zbudować sterownik dedykowany systemowi wielorobotowymi. Ogromną zaletą MRROC++ jest jej otwarta struktura, rozszerzalna o nowe moduły zaimplementowane w języku C++ w całości, lub powstałe na bazie istniejących modułów. Umożliwia to łatwe rozszerzanie struktury o sterowniki nowych czujników czy manipulatorów. Poniższy opis struktur MRROC++ zaczerpnięty został z [20].

Sterownik, powstały w oparciu o strukturę MRROC++ działa pod kontrolą systemu operacyjnego czasu rzeczywistego QNX 6.5.0. Jest to w rzeczywistości zbiór procesów, działających w węzłach sieci lokalnej, z których każdy ma ściśle określone zadanie.

Najniższą warstwę systemu stanowi warstwa sprzętowa. W tej warstwie działają procesy EDP[4], mające bezpośredni dostęp do sprzętu i sterujące ruchem pojedynczego efektora. Ich podstawowym zadaniem jest rozwiązywanie prostego i odwrotnego zadania kinematyki oraz realizacja funkcji charakterystycznych dla danego typu robota. Od strony programistycznej EDP stanowi wirtualny efektor, wykonujący zadane ruchy.

Obok procesów EDP działają również procesy VSP[5] reprezentujące wirtualny sensor agregujący dane otrzymane z rzeczywistych czujników w wirtualny odczyt, stanowiący już informację użyteczną dla systemu. Odczyty czujników rzeczywistych wykonywane są co określony interwał czasu na żądanie procesu ECP lub MP.

[4]Effector Driver Process
[5]Virtual Sensor Process

Warstwa druga odpowiedzialna jest za wykonanie zadania na pojedynczym efektorze. W jej skład wchodzą procesy ECP[6] . Ich ilość, podobnie jak procesów EDP, jest równa liczbie efektorów w systemie. Procesy te realizują algorytm sterowania pojedynczego manipulatora. W celu realizacji programu użytkowego komunikują się z procesem EDP sterującym odpowiadającym im robotom oraz procesami MP, VSP czy UI. Jednym z zadań pojedynczego procesu ECP może być generacja trajektorii dla robota pod jego kontrolą w przypadku niezależnej lub luźnej współpracy.

Najwyższą warstwę stanowi pojedynczy proces MP[7], koordynujący współpracę robotów i sterujący wykonaniem zadania. Synchronizuje w czasie działanie efektorów, a w przypadku ich ścisłej współpracy generuje trajektorie. Dodatkowo proces ten realizuje również polecenia użytkownika otrzymane z procesu UI[8].

Oprócz wyżej wymienionych w systemie istnieje również warstwa niezależna od wykonywanego zadania, odpowiedzialna za komunikację z użytkownikiem. W warstwie tej pracuje proces UI. Proces UI realizuje graficzny interfejs użytkownika, oparty na środowisku graficznym Photon MicroGUI, wchodzącym w skład systemu operacyjnego QNX. Proces ten odpowiedzialny jest za wyświetlanie stanu systemu sterującego.

Procesy ECP i MP są zależne od wykonywanego zadania, podczas gdy procesy EDP są ściśle związane ze sprzętem. Oznacza to, iż modyfikacja zadania sprowadza się do modyfikacji kodu procesów MP i ECP. Zmiany w sprzęcie z kolei pociągają za sobą konieczność modyfikacji kodu procesów EDP w przypadku efektora i VSP w przypadku czujnika. Taki rozdział warstwy zadania od warstwy sprzętowej ułatwia tworzenie sterowników, gdyż umożliwia oprogramowanie zadania bez konieczności ingerencji w kod sterujący efektorów i czujników, a kod sterujący zadaniem uniezależnia od użytego sprzętu.

3.2.1 Programowanie robotów

Struktura MRROC++ udostępnia obecnie trzy metody programowania trajektorii robota:

- pliki trajektorii
- ruchy ręczne za pomocą konsoli sterowniczej
- ruchy ręczne podatnym robotem

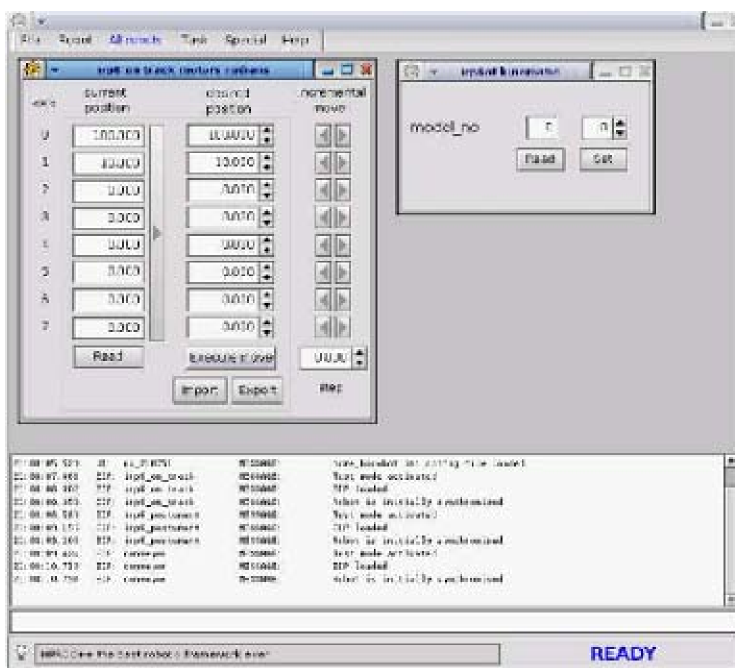
Pliki trajektorii Operator systemu robotycznego może opisać oczekiwane zachowanie robota poprzez definicję kolejnych węzłów interpolacji w plikach trajektorii. Pliki takie

[6]Effector Control Process
[7]Master Process
[8]User Interface

zawierają w kolejnych liniach współrzędne punktów w wybranym układzie współrzędnych. Odpowiednie zadanie systemu MRROC++ wczytuje pliki trajektorii i wykonuje trajektorię przy użyciu wybranego generatora ruchu.

Metoda ta pozwala na bardzo dokładne zadawanie położenia węzłów interpolacji przez wprowadzanie współrzędnych z wybraną dokładnością. Nie wymaga również zaangażowania robota podczas programowania. Nie jest jednak zbyt intuicyjna, nie pozwala również na weryfikację poprawności zapisanej trajektorii na bieżąco.

Ruchy ręczne za pomocą konsoli sterowniczej Struktura MRROC++ udostępnia graficzną konsolę sterowniczej robota Irp-6. Pierwotna wersja oparta była o środowisko graficzne Photon (rysunek 8), dostępna jest jednak wersja przenośna oparta o język Java[20].



Rysunek 8: Graficzna konsola sterownicza systemu MRROC++

Konsola sterownicza umożliwia ruchy ręczne robotem w wybranym układzie współrzędnych. Operator ma możliwość wprowadzania bezwzględnych wartości współrzędnych lub poruszania robotem w trybie przyrostowym. W strukturze zaimplementowane zostało również zadanie pozwalające operatorowi zapamiętywanie wybranych węzłów, a następnie odtwarzanie zapamiętanej trajektorii.

Jest to metoda z rodziny metod on-line, angażujących rzeczywistego robota. Struktura MRROC++ umożliwia jednak pracę w trybie testowym, w którym obliczenia wykonywane są na modelu kinematyki robota, a sam robot nie jest używany. W ramach pracy

inżynierskiej[21] M. Żbikowski zaimplementował środowisko symulacyjne robota Irp-6. Połączenie tych dwóch rozwiązań pozwala na wygodne programowanie off-line oraz weryfikację poprawności programu.

Ruchy ręczne podatnym robotem W strukturze MRROC++ zaimplementowane zostało zadanie umożliwiające poruszanie podatnym robotem przez operatora. Dużą wadą takiej metody zadawania pozycji jest obecność operatora w przestrzeni roboczej manipulatora, co jest niewskazane ze względów bezpieczeństwa. Dodatkowo do obsługi takiego zadania niezbędna jest dodatkowa osoba przy konsoli sterowniczej, sterująca przebiegiem wykonania zadania uczenia robota. Metoda ta, chociaż bardzo intuicyjna, nie pozwala na dokładne zadanie położenia końcówki manipulatora, gdyż jest oparta na wzrokowej analizie sceny przez operatora.

3.2.2 Zadania

Obiekty zadań realizują wybrany algorytm pracy systemu robotycznego, tj. sekwencję operacji mającą doprowadzić do osiągnięcia przez niego zamierzonego przez operatora celu. Sterują one przebiegiem wykonania zadania, komunikując się zarówno z systemem robotycznym, jak i z operatorem. Implementacja zadania w oparciu o strukturę MRROC++ jest ułatwiona dzięki dostarczeniu w ramach struktury narzędzi wspomagających tworzenie sterownika systemu robotycznego. Szerszy opis prezentowanych wanych tutaj zagadnień można znaleźć w [15] - w dalszej części rozdziału przybliżone zostaną aspekty istotne z punktu widzenia realizacji niniejszej pracy. Opis implementacji zadania na potrzeby niniejszej pracy znajduje się w rozdziale 5.3.

Klasa *ecp_mp::task::task* Klasą bazową wszystkich globalnych procesów ECP i MP jest klasa *ecp_mp::task::task*. Inicjalizuje ona komunikację z procesem UI, udostępnia też szereg metod umożliwiających komunikację z operatorem systemu i czujnikami wirtualnymi zdefiniowanymi na potrzeby zadania (tablica 2). Zapewnia ona również obiektom klas po niej dziedziczących dostęp do innych obiektów istniejących w systemie (tablica 3).

Metoda	**Opis**
bool operator_reaction(const char* question)	Przesyła pytanie do procesu UI i zwraca odpowiedź operatora - odpowiedź na pytanie może być typu *bool*, tj. tylko *prawda/fałsz.*

uint8_t choose_option(const char* question, uint8_t nr_of_options_input)	Przesyła do procesu UI pytanie i zwraca odpowiedź operatora - odpowiedź na pytanie to liczba z zakresu *1-nr_of_options_input*. Nie są do procesu UI przesyłane dostępne do wyboru opcje, dlatego też treść pytania powinna zawierać przyporządkowanie liczb do opcji.
int input_integer (const char* question) double input_double (const char* question)	Przesyła do procesu UI pytanie i zwraca liczbę całkowitą wprowadzoną przez operatora. Przesyła do procesu UI pytanie i zwraca liczbę zmiennoprzecinkową wprowadzoną przez operatora.
bool show_message(const char* message)	Przesyła do procesu UI komunikat, wymagając przy tym potwierdzenia jego odbioru przez operatora.
void all_sensors_initiate_reading (sensors_t &_sensor_m)	Przesyła polecenie *INITIATE_READING* do wszystkich czujników znajdujących się w tablicy *_sensor_m*.
void all_sensors_get_reading (sensors_t &_sensor_m)	Przesyła polecenie *GET_READING* do wszystkich czujników, znajdujących się w tablicy *_sensor_m*.

Tablica 2: Interfejs klasy ecp_mp::task::task

Obiekt	**Opis**
sensors_t sensor_m	Mapa wszystkich czujników zdefiniowanych dla zadania
transmitters_t transmitter_m	Mapa wszystkich transmiterów zdefiniowanych dla zadania
lib::sr_ecp* sr_ecp_msg	Obiekt służący do komunikacji z procesem SR - umożliwia m.in. przesyłanie komunikatów do operatora
lib::configurator config	Obiekt udostępniający konfigurację zadania i poszczególnych obiektów je realizujących

Tablica 3: Obiekty udostępniane klasom dziedziczącym z klasy ecp_mp::task::task

Klasa *ecp::common::task::task* Kolejna w hierarchii klas zadań struktury MRROC++ jest klasa *ecp::common::task::task* rozszerzająca klasę *ecp_mp::task::task*. Stanowi ona klasę bazową całego procesu MP i implementuje metody służące do komunikacji z nim (tablica 4). Udostępnia również klasom dziedziczącym obiekty wymienione w tablicy 5.

Metoda	**Opis**
void initialize_communication()	Inicjalizuje komunikację z procesami ECP, MP i SR
void send_pulse_to_mp()	Wysyła puls do procesu MP
int receive_mp_message()	Pobiera komunikat z procesu MP
lib::MP_COMMAND mp_command_type()	Zwraca typ polecenia z procesu MP
void main_task_algorithm(void)	Metoda, która powinna zostać nadpisana w klasach dziedziczących, tak by implementowała algorytm wykonania zadania
bool ecp_wait_for_start()	Oczekuje na polecenie START z procesu MP
bool ecp_wait_for_stop()	Oczekuje na polecenie STOP z procesu MP

Tablica 4: Interfejs klasy ecp::common::task::task

Obiekt	**Opis**
lib::MP_COMMAND_PACKAGE mp_command	Polecenie z procesu MP
ecp_robot* ecp_m_robot	Obiekt reprezentujący robota, umożliwiający komunikację z nim i udostępniający aktualny stan robota

Tablica 5: Obiekty udostępniane klasom dziedziczącym z klasy ecp::common::task::task

Implementacja zadania w oparciu o strukture MRROC++ Stworzenie nowego zadania obejmuje implementację metody *main_task_algorithm()*. Powinna ona, przy wy-

korzystaniu innych obiektów zadania, tj. czujników, generatorów czy obiektu robota, oraz metod służących do komunikacji z innymi procesami, zrealizować algorytm przewidziany dla zadania. Przykład implementacji zadania został opisany w rozdziale 5.3, gdzie przedstawiłem implementację zadania uczenia robota trajektorii.

3.2.3 Czujniki wirtualne

Czujniki wykorzystywane przez sterowniki systemów robotowych to fizyczne urządzenia udostępniające dowolnego rodzaju odczyty. W strukturze MRROC++ ich obsługa jest implementowana w klasach czujników wirtualnych, ukrywających szczegóły komunikacji z fizycznym czujnikiem oraz agregujących otrzymane z niego dane i udostępniających zagregowaną użyteczną informację obiektom generatorów trajektorii, zadań lub robotów. Szerszy opis opisywanych tutaj zagadnień można znaleźć w [15] - w dalszej części rozdziału przybliżone aspekty istotne z punktu widzenia realizacji niniejszej pracy. Opis implementacji czujnika wirtualnego na potrzeby niniejszej pracy znajduje się w rozdziale 5.2.

Klasa lib::sensor_interface Klasą bazową dla wszystkich klas czujników wirtualnych jest abstrakcyjna klasa *lib::sensor_interface*, zdefiniowana w pliku *lib/sensor.h*. Klasy od niej pochodne, reprezentują w procesie ECP konkretne czujniki, muszą implementować deklarowane przez nią metody (tablica 6).

Metoda	**Opis**
void configure_sensor()	Konfiguruje czujnik, tj. wykonuje wszystkie czynności niezbędne do rozpoczęcia pobierania odczytów z czujnika wirtualnego.
void initiate_reading()	Inicjalizuje pracę czujnika w przypadku, gdy wymagane są dodatkowe działania przed pobraniem odczytu
void get_reading()	Pobiera odczyt z czujnika, jeżeli jest dostępny

Tablica 6: Interfejs klasy lib::sensor_interface

Plik nagłówkowy *lib/sensor.h* definiuje również klasę *sensor_error*, reprezentującą błędy czujnika. Dodatkowo zdefiniowane są tutaj typy wyliczeniowe VSP_COMMAND_T i VSP_REPORT_T, wykorzystywane przy komunikacji z czujnikiem i określające odpowiednio polecenie dla procesu VSP oraz status odpowiedzi z procesu czujnika (listing 1).

```
typedef enum _VSP_COMMAND
{
```

```
  VSP_CONFIGURE_SENSOR,  //konfiguracja czujnika
  VSP_INITIATE_READING,  //zlecenie odczytu
  VSP_GET_READING,       //pobranie odczytu
  VSP_TERMINATE          //zakonczenie pracy czujnika
} VSP_COMMAND_t;

typedef enum _VSP_REPORT
{
  VSP_REPLY_OK,                 //OK
  VSP_SENSOR_NOT_CONFIGURED,    //czujnik nie zostal skonfigurowany
  VSP_READING_NOT_READY,        //odczyt nie jest gotowy
  INVALID_VSP_COMMAND           //nieprawidlowe polecenie
} VSP_REPORT_t;
```

Listing 1: Stale uzywane do komunikacji z procesem VSP

Klasa ecp_mp::sensor Klasą bazową dla wszystkich czujników po stronie procesu ECP i MP jest klasa (a raczej szablon klas) *ecp_mp::sensor*, zdefiniowana w pliku *ecp_mp/ecp_mp_sensor.h*. Utworzenie obiektu tej klasy w procesie ECP powoduje uruchomienie procesu VSP na zdefiniowanym w pliku konfiguracyjnym węźle sieci. Czujniki tej klasy posiadają nazwę (*VSP_NAME*), identyfikator procesu VSP (*pid*) i tzw. obraz czujnika (*image*), czyli aktualny odczyt. Implementują również domyślną logikę metod deklarowanych przez klasę *lib::sensor*.

Dodatkowo klasa *ecp_mp::sensor* definiuje struktury wykorzystywane w komunikacji z czujnikiem (listing 2) - ECP_VSP_MSG, służącą do przesyłania poleceń do procesu VSP, oraz VSP_ECP_MSG, służącą do przesyłania odpowiedzi z czujnika. Struktura ECP_VSP_MSG przechowuje polecenie dla procesu VSP oraz dane typu CONFIGURE_DATA. Struktura VSP_ECP_MSG przechowuje status odpowiedzi VSP oraz tzw. obraz czujnika typu SENSOR_IMAGE. Typy CONFIGURE_DATA oraz SENSOR_IMAGE są przekazywane jako parametry szablonu klasy.

```
struct ECP_VSP_MSG
{
  lib::VSP_COMMAND_t i_code;   //polecenie
  CONFIGURE_DATA command;      //struktura sterujaca
} to_vsp;

struct VSP_ECP_MSG
{
```

```
    lib::VSP_REPORT_t vsp_report;     //status odpowiedzi
    SENSOR_IMAGE comm_image;          //obraz czujnika
} from_vsp;
```

Listing 2: Struktury uzywane do komunikacji z procesem VSP

Komunikacja z czujnikiem wirtualnym Do komunikacji z czujnikiem wirtualnym służą metody zadeklarowane w klasie *lib::sensor*. Każda z nich wysyła do czujnika strukturę ECP_VSP_MSG, zawierającą odpowiednią komendę (*i_code*) i ewentualnie dodatkowe parametry (*command*), zwłaszcza w przypadku polecenia konfiguracji czujnika. Po wysłaniu komendy sterującej pobierana jest struktura VSP_ECP_MSG, zawierająca kod odpowiedzi (*vsp_report*), a w przypadku pobierania odczytu z czujnika, również wypełnioną strukturę obrazu czujnika (*comm_image*), która jest kopiowana do zmiennej *image*.

Implementacja czujnika wirtualnego w oparciu o strukturę MRROC++ Stworzenie nowej klasy, implementującej obsługę czujnika wirtualnego w oparciu o strukturę MRROC++ obejmuje:

- zdefiniowanie struktury sterującej (*CONFIGURE_DATA*) przesyłanej do czujnika w strukturze ECP_VSP_MSG
- zdefiniowanie struktury opisującej odczyt z czujnika (*SENSOR_IMAGE*) przesyłanej w odpowiedzi z czujnika w strukturze ECP_VSP_MSG
- implementację metod zadeklarowanych w klasie *lib::sensor*, jeżeli ich domyślna implementacja jest niewystarczająca
- implementację metod udostępniających zagregowane dane z odczytu, jeżeli surowe dane w zmiennej *image* są niewystarczające

3.2.4 Generatory trajektorii

Za generowanie trajektorii w strukturze MRROC++ odpowiedzialne są obiekty generatorów. Do ich zadań należy zainicjalizowanie ruchu, wyznaczanie współrzędnych kolejnych węzłów trajektorii oraz sprawdzanie spełnienia warunków stopu. Struktura MRROC++ udostępnia narzędzia przyspieszające implementację generatorów, zawiera również szereg generatorów, na których można się wzorować lub które można rozszerzać. Szerszy opis opisywanych tutaj zagadnień można znaleźć w [15] - w dalszej części rozdziału przybliżone aspekty istotne z punktu widzenia realizacji niniejszej pracy. Opis implementacji trzech generatorów na potrzeby niniejszej pracy znajduje się w rozdziale 5.5.

Klasa ecp_mp::generator::generator Klasą bazową dla wszystkich generatorów trajektorii w strukturze MRROC++ jest abstrakcyjna klasa *ecp_mp::generator::generator*. Zadania korzystają z konkretnych klas generatorów dziedziczących po tej klasie. Klasa ta umożliwia dostęp do obiektów wspólnych dla wszystkich generatorów, implementuje wspólne dla nich funkcje i deklaruje szereg metod, które klasy ją rozszerzające powinny implementować.

Obiekty dostępne dla wszystkich generatorów, dziedziczących z klasy *ecp_mp::generator::generator* zebrane są w tabeli 7.

Obiekt	**Opis**
lib::sr_ecp sr_ecp_msg	obiekt służący do komunikacji z procesem SR, umożliwiający przekazywanie komunikatów do użytkownika
sensors_t sensor_m	mapa wszystkich zainicjalizowanych dla danego zadania czujników wirtualnych
transmitters_t transmitter_m	mapa wszystkich zainicjalizowanych dla danego zadania transmiterów

Tablica 7: Obiekty udostępniane klasom dziedziczącym z klasy ecp_mp::generator::generator

Dodatkowo abstrakcyjna klasa *ecp_mp::generator::generator* deklaruje następujące metody:

Metoda	**Opis**
bool check_and_null_trigger()	zwraca informację o tym, czy otrzymano impuls *trigger* i czyści tę flagę
abstract bool first_step()	abstrakcyjna metoda, której zadaniem jest generowanie pierwszego kroku ruchu, który najczęściej różni się znacznie od kolejnych ruchów np. nie wykorzystuje czujników, inicjalizuje i konfiguruje robota lub rusza go do pozycji startowej, stąd potrzeba wydzielenia odrębnej metody

abstract bool next_step()	abstrakcyjna metoda, której konkretna wersja powinna generować kolejne kroki manipulatora. Rolą tej metody jest wygenerowanie na podstawie aktualnego stanu robota i docelowego działania manipulatora nowych współrzędnych w wybranym układzie współrzędnych i wypełnienie odpowiednich struktur sterujących manipulatorem. Dodatkowo metoda ta powinna sprawdzać warunek stopu i w przypadku jego spełnienia zwrócić wartość 0 lub wartość niezerową w przeciwnym wypadku.

Tablica 8: Interfejs klasy ecp_mp::generator::generator

Klasa ecp::common::generator::generator Klasa bazowa *ecp_mp::generator::generator* rozszerzana jest przez kolejną klasę abstrakcyjną *ecp::common::generator::generator*. Udostępnia ona również klasom potomnym generatorów szereg obiektów, które zebrałem w tabeli 9.

Obiekt	**Opis**
ecp::common::task::task ecp_t	obiekt zadania wykonywanego w procesie ECP
ecp_robot the_robot	obiekt reprezentujący manipulator, dla którego generowana jest trajektoria, umożliwiający komunikację z robotem, a w szczególności wypełnianie struktur sterujących
lib::trajectory_description td	zawiera opis aktualnej trajektorii manipulatora

Tablica 9: Obiekty udostępniane klasom dziedziczącym z klasy ecp::common::generator::generator

Klasa *ecp_mp::generator::generator* implementuje również następujące metody:

Metoda	**Opis**
void move_init()	uruchamia komunikację z manipulatorem i czyści odpowiednie struktury komunikacyjne

void Move()	inicjalizuje ruch manipulatora, a następnie nim porusza. Ruch rozpoczyna wykonanie metody *move_init()* w celu inicjalizacji komunikacji. Następnie wykonywana jest metoda *first_step()* w celu konfiguracji manipulatora i inicjalizacji ruchu. W kolejnym kroku wołana jest w pętli metoda *next_step()*, która powinna wyznaczyć kolejny krok ruchu oraz sprawdzić warunek stopu. Metoda Move() wykonuje się do momentu spełnienia przez robota warunku stopu.
bool is_EDP_error()	sprawdza, czy nie wystąpił błąd w procesie EDP
void execute_motion()	każe robotowi wykonać ruch na podstawie współrzędnych w strukturach sterujących ustawionych przez generator trajektorii

Tablica 10: Interfejs klasy ecp::common::generator::generator

Implementacja generatora trajektorii w oparciu o strukturę MRROC++ Stworzenie nowej klasy generatora trajektorii w oparciu o strukturę MRROC++ obejmuje implementację metod *first_step()* i *next_step()*, które powinny realizować specyficzny dla danego generatora algorytm generowania współrzędnych kolejnych węzłów trajektorii.

3.2.5 Generator trajektorii smooth

Generator trajektorii smooth stworzony został w ramach pracy inżynierskiej[1] Rafała Tulwina. Jego implementacja znajduje się w klasie ecp::common::generator::smooth.

Generator ten porusza końcówką manipulatora po trajektorii zdefiniowanej poprzez skończony zbiór punktów w przestrzeni roboczej. Umożliwia zdefiniowanie ograniczeń na przyspieszenie i prędkość ruchu niezależnie dla każdej osi. Prędkość początkowa i końcowa ruchu wynosi 0, jednak przejście przez poszczególne węzły trajektorii odbywa się bez zatrzymywania ruchu, dzięki czemu można osiągnąć ruch płynny i możliwie szybki przy zadanych ograniczeniach.

Generator smooth umożliwia generowanie trajektorii dla różnych reprezentacji:

- współrzędne wewnętrzne
  - przestrzeń napędów
  - przestrzeń stawów

- zewnętrzne układy współrzędnych
  - układ współrzędnych oś-kąt
  - układ współrzędnych Eulera

3.3 Zadajnik ruchu

Interaktywne programowanie robotów wymagało zastosowania urządzenia umożliwiającego zadawanie przemieszczeń manipulatora. Z założeń pracy wyniknął szereg wymagań, które zadajnik ruchu powinien spełnić. Na rynku dostępne jest wiele tego typu urządzeń, jednak tylko kilka spełnia postawione przed nimi wymagania. W niniejszym rozdziale przedstawię stawiane zadajnikowi wymagania, przeprowadzę analizę dostępnych na rynku rozwiązań, a następnie przedstawię urządzenie wybrane do realizacji niniejszej pracy.

3.3.1 Wymagania

Założenia pracy i oczekiwania wobec projektowanego rozwiązania pozwoliły zdefiniować następujące wymagania, które powinien spełniać zadajnik ruchu:

- bezprzewodowa komunikacja
- jednoręczna obsługa zadajnika
- duża ilość wejść
- płynne zadawanie ruchu
- możliwość sprzężenia zwrotnego
- dostępność handlowa
- łatwa obsługa programowa

W dalszej części rozdziału przedstawię, jak rozumiem powyższe wymagania, powody postawienia przed zadajnikiem właśnie takich wymagań, oraz oczekiwane zyski z zadajnika je spełniającego.

Komunikacja bezprzewodowa Jest to podstawowe wymaganie stawiane zadajnikowi. W tradycyjnych realizacjach sterowania systemem robotycznym operator używa panelu sterującego zrealizowanego jako aplikacja działająca na wybranym komputerze lub fizycznego panelu sterującego połączonego z systemem robotycznym. Przestrzeń robocza manipulatora jest duża, a operator systemu robotycznego powinien mieć możliwość przemieszczania się wokół niej np. w celu obserwacji jego pracy czy umieszczenia w tej przestrzeni

obiektów niezbędnych do wykonania zadnia. Kable ograniczają swobodę ruchu, uniemożliwiając operatorowi swobodny dostęp do przestrzeni roboczej manipulatora. Bezprzewodowe urządzenie pozwoliłoby pracować z systemem robotycznym z dowolnego miejsca, w którym zadajnik jest w stanie skomunikować się z systemem.

Jednoręczna obsługa To wymaganie podyktowane zostało względami bezpieczeństwa. Operator poruszający się w przestrzeni roboczej manipulatora musi mieć możliwość szybkiego odłączenia zasilania robota. W Laboratorium Robotyki udostępnione są w tym celu przyciski stopu awaryjnego, które operator dla własnego bezpieczeństwa powinien mieć zawsze przy sobie, by możliwie szybko reagować na niepożądane zachowanie robota. Dwuręczne urządzenie znacznie utrudniłoby pracę operatora, zmuszając go do rezygnacji z łatwego dostępu do stopu awaryjnego lub do obsługi dwuręcznego urządzenia jedną ręką. Pierwsze rozwiązanie nie jest zalecane ze względów bezpieczeństwa, drugie nie byłoby wygodne i nie pozwoliłoby na pełne wykorzystanie potencjału zadajnika ruchu.

Duża ilość wejść Urządzenie powinno być wyposażone we względnie dużą ilość wejść ze względu na mnogość sygnałów przesyłanych od operatora do systemu robotycznego. Operator musi mieć możliwość łatwego wyboru trybu czy kierunku ruchu, zadania prędkości lub przyspieszenia ruchu oraz przesyłania sygnałów sterujących do zadania wykonywanego przez system robotyczny. Dodatkowo fizyczne wejścia zadajnika (np. przyciski czy joystick) powinny znajdować się w miejscu, w którym dostęp do nich będzie nieutrudniony i możliwie wygodny, zwłaszcza biorąc pod uwagę wymaganie jednoręcznej obsługi zadajnika ruchu.

Płynne zadawanie ruchu Zadajnik musi umożliwiać zadanie wybranej zmiennej ruchu w pełnym zakresie od 0 do pewnej wartości maksymalnej. Niedopuszczalne jest zadawanie ruchu w sposób dwustanowy, tj. ruch z ustaloną prędkością lub jego brak, niewskazana jest również konieczność wyboru prędkości ruchu spośród kilku predefiniowanych wartości.

Sprzężenie zwrotne Wielką zaletą bezprzewodowego zadajnika ruchu jest fakt, iż operator systemu robotycznego nie musi przebywać cały czas przy konsoli sterowniczej. Wadą takiego rozwiązania jest to, że operator nie widząc informacji przesyłanych przez system robotyczny do konsoli sterowniczej nie jest informowany o sytuacjach wyjątkowych czy niepożądanych. Zadajnik ruchu musi zatem posiadać środki, pozwalające zrealizować sprzężenie zwrotne z systemem robotycznym i sygnalizację w sposób zauważalny dla operatora sytuacji niepożądanych, które mogą mieć miejsce w systemie robotycznym.

Dostępność handlowa Zadajnik ruchu musi być łatwy do nabycia i niedrogi. Dzięki temu, gdy zajdzie taka potrzeba, ewentualne rozszerzanie zestawu sterowniczego systemu robotycznego o kolejne zadajniki odbędzie się szybko i niskim kosztem. Nieskomplikowana będzie również wymiana zadajnika w przypadku jego uszkodzenia.

Możliwie łatwa obsługa programowa Podczas realizacji zadania konieczna będzie implementacja obsługi zadajnika jako czujnika wirtualnego systemu MRROC++. Aby było to możliwe, urządzenie powinno wykorzystywać do komunikacji popularny i dobrze określony protokół (np. Bluetooth) oraz powinno mieć dobrze udokumentowany interfejs. W idealnym przypadku powinna istnieć biblioteka programowa do obsługi urządzenia wspierana przez jego producenta.

Jestem przekonany, iż analiza dostępnych na rynku rozwiązań pod kątem spełnienia zdefiniowanych wyżej wymagań pozwoli ograniczyć spektrum możliwych rozwiązań do maksymalnie kilku urządzeń, a ich spełnienie przez zadajnik umożliwi efektywną i wygodną pracę z systemem robotycznym.

3.3.2 Dostępne rozwiązania

Poszukując zadajnika ruchu dla systemu robotycznego skupiłem się na rynku konsoli gier wideo. Różnorodność zadań stawianych w grach przed graczami, ogromna rzesza użytkowników o wysokich wymaganiach i duża konkurencja na rynku sprawiły, że rynek ten obecnie oferuje bardzo szeroki wachlarz zadajników ruchu sprawdzonych w wielu różnych zastosowaniach. W niniejszym rozdziale przedstawię najbardziej obiecujące rozwiązania oferowane przez rynek konsoli gier wideo w momencie pisania tej pracy. Wszystkie zostały przeanalizowane pod kątem spełnienia zdefiniowanych wcześniej wymagań.

Sony DualShock 3 Gdy firma Sony wypuściła na rynek swoją najnowszą konsolę Playstation 3 w listopadzie 2006 roku, dołączony był do niej bezprzewodowy kontroler Sony Sixaxis. Rok później zastąpiony został zadajnikiem DualShock 3, który oferował porównywalne możliwości, został jednak wzbogacony o mechanizm sprzężenia haptycznego. W niezmienionej postaci zadajnik oferowany do dziś jest jako podstawowy zadajnik konsoli Playstation 3. Cena zadajnika to obecnie około 170 zł.

DualShock 3 jest zadajnikiem bezprzewodowym, komunikującym się z konsolą za pomocą protokołu Bluetooth lub kabla USB. Twórcy zadajnika nie przewidzieli możliwości zastosowania zadajnika z urządzeniami innymi niż konsola Playstation 3, dlatego też zadajnik mimo wspierania standardu Bluetooth nie oferuje tzw. discovery mode, tj. trybu, w którym rozgłasza swój adres, a inne urządzenia mogą się z nim skomunikować. Wskutek

Rysunek 9: Sony Dualshock 3

tego przed skomunikowaniem zadajnika z konsolą poprzez interfejs bezprzewodowy wymagane jest wcześniejsze podłączenie zadajnika do konsoli lub komputera PC za pomocą kabla USB w celu synchronizacji urządzeń, które w najprostszym przypadku sprowadza się do odczytania przez urządzenie komunikujące się z zadajnikiem jego adresu fizycznego. Konieczność wstępnej synchronizacji za pomocą kabla USB komplikuje zastosowanie zadajnika w innych zastosowaniach niż współpraca z konsolą Playstation 3.

Urządzenie pracuje zgodnie z interfejsem Bluetooth HID, bezprzewodowym rozszerzeniem interfejsu HID[9]. Dzięki temu urządzenie można wykorzystać jako urządzenie wskazujące w środowisku graficznym systemów operacyjnych obsługujących ten interfejs, a w szczególności systemów Linux i Windows. Takie zastosowanie nie jest wystarczające na potrzeby niniejszej pracy, gdyż urządzenie widziane jest jako generyczny model zadajnika, nie są więc obsługiwane funkcje specyficzne danemu urządzeniu. Biblioteki służące do komunikacji z takimi urządzeniami umożliwiają jednak odczytywanie surowych danych otrzymanych z urządzenia, możliwe jest więc doimplementowanie obsługi brakujących funkcji zadajnika.

Podstawowymi komponentami zadajnika DualShock 3 są:

- trójosiowy czujnik przyspieszenia liniowego
- dwie analogowe gałki, które można wychylić w dowolnym kierunku na płaszczyźnie, o rozdzielczości 10 bitów, które można również używać jako przyciski (L3 i R3)
- dwa analogowe przyciski (L2 i R2)

[9]Human Interface Device

- 8 przycisków monostabilnych (tr, kol, iks, kwa, L1, R1, Start, Select)
- panel kierunkowy (tzw. krzyżak), tj. cztery przyciski-strzałki
- 4 diody LED, niewidoczne jednak dla operatora w czasie normalnego użytkowania zadajnika
- dwa silniki wibracyjne

Jak widać zadajnik oferuje bardzo dużą ilość wejść, jednak ceną za to jest konieczność dwuręcznej obsługi zadajnika. Można próbować obsługiwać go jedną ręką, traci się jednak wtedy dostęp do połowy wejść zadajnika, a samo trzymanie zadajnika również nie jest wygodne.

Zadajnik posiada cztery wejścia analogowe - dwie gałki i dwa przyciski. Umożliwiają one zadawanie analogowych sygnałów sterujących, jednak maksymalny kąt wychylenia gałki lub wciśnięcia przycisku jest na tyle mały, że trudno o precyzyjne zadawanie sygnału sterującego. Na szczęście Dualshock 3 posiada wbudowany akcelerometr, co umożliwia wysyłanie analogowych sygnałów sterujących za pomocą wychylania zadajnika, a maksymalny kąt wychylenia jest o wiele większy. Możliwie jest bowiem obracanie zadajnika o dowolny kąt, co w praktyce oznacza maksymalne możliwe wychylenie o 180 stopni.

Dodatkowo zadajnik umożliwia dwie formy komunikacji systemu robotycznego z operatorem - identyfikację stanu urządzenia za pomocą diod LED oraz sygnalizację sytuacji wyjątkowych za pomocą silników wibracyjnych.

Zadajnik Sony Dualshock 3 spełnia większość wymagań stawianych przed zadajnikiem ruchu dla systemu robotycznego, jedyną zaś jego wadą jest konieczność dwuręcznej jego obsługi.

Microsoft Xbox 360 Controller Urządzenie to jest podstawowym zadajnikiem konsoli Xbox 360 firmy Microsoft. Na rynku konsola z zadajnikiem pojawiły się w 2005 roku. Zadajnik dostępny jest zarówno w wersji przewodowej jak i bezprzewodowej, opartej na standardzie Bluetooth. Cena zadajnika w wersji bezprzewodowej to obecnie około 130 zł.

Obie wersje zadajnika można łatwo skomunikować z aplikacjami działającymi pod kontrolą systemu operacyjnego Windows lub Linux. W Linuksie udostępnia to biblioteka Userspace Xbox/Xbox360 USB Gamepad Driver for Linux[10], wspierająca ten oraz starsze zadajniki firmy Microsoft. Dla systemu Windows istnieją z kolei udostępnione przez firmę Microsoft sterowniki. Wspomniane biblioteki implementują obsługę wszystkich funkcji zadajnika. Dodatkowo zadajnik jest zgodny ze standardem Bluetooth HID,

[10] http://pingus.seul.org/ grumbel/xboxdrv/

Rysunek 10: Microsoft Xbox 360 Controller

tak więc podobnie jak w przypadku Dualshock 3 możliwy jest dostęp do surowych danych przesyłanych z zadajnika z poziomu bibliotek obsługujących urządzenia HID.

Podstawowymi komponentami zadajnika Xbox 360 są:

- dwie gałki analogowe, wychylające się w dowolnym kierunku na płaszczyźnie
- dwa analogowe przyciski
- 11 monostabilnych dwustanowych przycisków
- panel kierunkowy (tzw. krzyżak), tj. cztery przyciski-strzałki

Zadajnik ruchu firmy Microsoft ma podobną konstrukcję jak opisywany wcześniej zadajnik DualShock 3 firmy Sony, podobną liczbę przycisków, a co za tym idzie również wymaga obsługi dwuręcznej. Ma dodatkowo zdecydowanie mniejsze możliwości, nie posiada bowiem żadnej formy sygnalizacji stanu systemu robotycznego. Nie posiada wbudowanych akcelerometrów, których nie są w stanie zastąpić dwie analogowe gałki z powodu zbyt małej precyzji zadawania sygnałów.

Logitech Cordless Rumblepad 2 Na początku 2009 roku firma Logitech wypuściła na rynek bezprzewodowy zadajnik Cordless Rumblepad 2, dedykowany komputerom PC i Mac. Jest on obecnie dostępny w cenie ok. 140 zł. Jest urządzeniem pracującym zgodnie ze standardem HID, można więc używać tego zadajnika jako urządzenia wskazującego w większości systemów operacyjnych, a dodatkowo komunikować się z urządzeniem za pomocą przesyłania surowych danych.

Rysunek 11: Logitech Cordless Rumblepad 2

Nie da się nie zauważyć podobieństwa jego konstrukcji do opisywanego wcześniej zadajnika firmy Sony. Pod względem funkcjonalnym również nie różni się wiele od zadajnika Dualshock 3. Podstawowymi jego komponentami są:

- 10 programowalnych przycisków,
- dwie gałki analogowe
- dwa silniki wibracyjne

Od razu nasuwa się myśl, że zadajnik firmy Logitech stanowi uboższą wersję zadajnika firmy Sony, gdyż mimo ograniczonej funkcjonalności nie oferuje nic, czego nie oferowałby zadajnik Dualshock 3. Duża liczba przycisków wymusza dwuręczną obsługę urządzenia. Brak akcelerometru uniemożliwia precyzyjne zadawanie analogowych sygnałów sterujących, gdyż analogowe gałki z powodu niewystarczającej precyzji nie sprawdzają się w tej roli. Silniki wibracyjne wprawiają urządzenie w intensywniejsze wstrząsy niż silniki zadajnika Dualshock 3, jednak większa intensywność wstrząsów nie jest warta poświęcenia pozostałych funkcji.

Novint Falcon Game Controller Zadajnik Novint Falcon jest zupełnie innym rodzajem zadajnika niż opisywane wcześniej urządzenia. Wyprodukowany został przez firmę Novint Technologies, specjalizującą się w urządzeniach sterujących wykorzystujących sprzężenie haptyczne. Z tego też powodu oferuje możliwości niespotykane dotychczas w innych zadajnikach.

Novint Falcon składa się z części bazowej, podłączanej do komputera za pomocą kabla USB oraz części ruchomej. Nie jest więc urządzeniem bezprzewodowym, a bezprzewodo-

Rysunek 12: Novint Falcon Game Controller

wość jest podstawowym wymaganiem stawianym zadajnikowi ruchu systemu robotycznego.

Falcon jest tak naprawdę robotem o trzech ramionach trzymających jedną końcówkę, którą z kolei trzyma i przemieszcza użytkownik. Przestrzeń robocza zadajnika to sześcian o boku długości 4 cali (ok. 10 cm). Każde z ramion robota posiada dwa stawy i jest podatne, a opory stawiane przez ramiona robota są pomijalne. Użytkownik przemieszczając końcówkę robota porusza jego ramionami. Urządzenie na podstawie współrzędnych w poszczególnych stawach wszystkich ramion określa położenie końcówki, a rozdzielczość wskazań położenia wynosi 400 dpi. Dzięki temu możliwe jest zadawanie przemieszczenia w przestrzeni trójwymiarowej. Operacja odczytu położenia końcówki odbywa się z częstotliwością 1 kHz, co w praktyce umożliwia sterowanie systemem robotycznym w czasie rzeczywistym.

Standardowa końcówka robota nie posiada przycisków, możliwe jest jednak rozszerzenie jej o dodatkowe nakładki. Dostępna jest m.in. nakładka imitująca pistolet, wyposażona w spust i trzy przyciski dodatkowe. Tak wyposażony robot umożliwia zadawanie ruchu jedną ręką. Ilość przycisków jest bardzo mała, jednak możliwość zadawania ruchu poprzez poruszanie końcówką robota powoduje, że duża ilość przycisków przestaje być potrzebna, gdyż użyte one byłyby jedynie do sterowania przebiegiem wykonania zadania uczenia robota trajektorii, nie zaś do samego procesu zadawania przemieszczenia. Mimo to liczba przycisków wydaje się niewystarczająca.

Każde z ramion robota wyposażone jest w silnik, co daje ogromne możliwości wykorzystania sprzężenia haptycznego przy sterowaniu robotem. Silniki umożliwiają bowiem symulację ciężaru trzymanego obiektu czy pędu końcówki rzeczywistego robota, dzięki

czemu operator systemu otrzymuje bardzo istotne informacje na temat przebiegu wykonania zadania realizowanego przez manipulator. Możliwa jest też sygnalizacja sytuacji wyjątkowych poprzez chociażby wprowadzenie końcówki w wibracje, czy nawet sterowanie systemem robotycznym za pomocą gestów wykonywanych końcówką zadajnika Novint Falcon.

Firma Novint udostępnia za darmo oprogramowanie Novint Falcon SDK[11], umożliwiające wytwarzanie aplikacji wykorzystujących zadajnik Novint Falcon. Niestety, dostępne jest ono tylko dla systemów operacyjnych z rodziny Windows, do obsługi zadajnika szybko powstała więc alternatywna biblioteka o otwartym kodzie źródłowym.

libnifalcon umożliwia komunikację z urządzeniem w aplikacjach działających pod kontrolą systemu operacyjnego Windows, Linux lub OS X. Wykorzystuje biblioteki ftd2xx (Windows) lub libusb-1.0 (Linux, OS X) do obsługi połączenia poprzez port USB i udostępnia na dzień dzisiejszy surowe odczyty z urządzenia. Biblioteka jest jednak cały czas rozwijana i powoli uzupełniana o funkcje wyższego poziomu, agregujące odczyty i udostępniające bardziej użyteczne informacje, np. położenie końcówki robota.

Jako że zadajnik Novint Falcon stanowi obecnie nowinkę techniczną na rynku urządzeń sterujących, jego cena jest sporo wyższa niż pozostałych zadajników. Koszt urządzenia to około 200 dolarów. Urządzenie jest obecnie niedostępne w Polsce, więc do jego ceny należy doliczyć jeszcze koszt sprowadzenia urządzenia z zagranicy.

Nintedo Wiimote Wiimote to podstawowy zadajnik konsoli Wii firmy Nintendo. W porównaniu z pozostałymi dostępnymi na rynku urządzeniami najlepiej spełniał wymagania stawiane przed zadajnikiem ruchu systemu robotycznego, dlatego też został przeze mnie wybrany do realizacji zadania. Szczegółowy opis urządzenia i jego analiza pod kątem spełnienia wymagań znajduje się w rozdziale 3.3.3.

3.3.3 Nintendo Wiimote

Wii to konsola gier wideo opracowana przez japońską firmę Nintendo. Zaliczana jest do konsol siódmej generacji - konsol gier, które pojawiły się na rynku w latach 2005-2006. Jest następczynią popularnej w latach 2001-2003 konsoli Nintendo GameCube. Oprócz Wii do siódmej generacji zaliczana jest konsola Xbox 360 firmy Microsoft oraz konsola PlayStation 3 firmy Sony, jednak pod względem sprzedaży konsola Wii wiedzie obecnie prym na rynku konsol gier i do końca 2009 roku sprzedano 67mln egzemplarzy Wii.

Kolejne generacje konsol gier wprowadzały na rynek przełomowe nowinki techniczne. Konsole siódmej generacji podbiły rynek przede wszystkim obrazem wideo w wysokiej rozdzielczości. Wii oferuje obraz w jakości DVD Video (480p), a Xbox360 i PlayStation

[11] Software Development Kit

3 obraz w jakości High Definition (1080p). Wszystkie wymienione konsole umożliwiają również podłączenie bezprzewodowego zadajnika ruchu.

Konsola Wii oparta jest na mikroprocesorze Broadway zbudowanym w architekturze PowerPC o szybkości zegara 729Mhz[22]. Konsola wyposażona jest w 88MB pamięci operacyjnej oraz 512MB pamięci Flash. Karta graficzna urządzenia zbudowana jest na bazie procesora Hollywood firmy ATI taktowanego z prędkością 243Mhz. Konsola umożliwia dołączenie 4 zadajników Nintendo GameCube, oprócz tego jest w stanie obsłużyć do 16 dodatkowych bezprzewodowych urządzeń dzięki wbudowanemu modułowi Bluetooth. Konsola posiada również 2 porty USB umożliwiające podłączenie klawiatury lub adaptera sieci LAN, jak również moduł sieci bezprzewodowej WiFi.

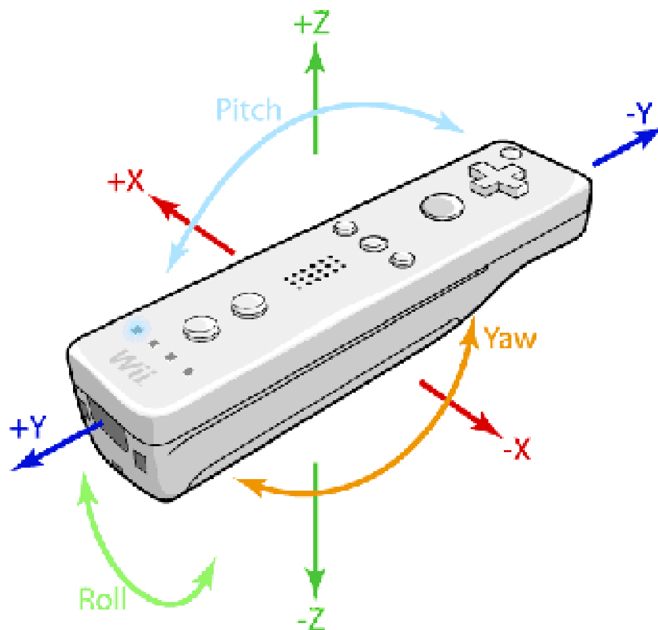
Nintendo *W*iimote (rys. 13), zwany także *Wii Remote* czy *Wiilot*, jest podstawowym zadajnikiem konsoli Nintendo Wii. Zadajniki ruchu innych konsol wykonane są jako tzw. *pady*, czyli urządzenia trzymane oburącz, do których obsługi używa się przede wszystkim kciuków obu dłoni. Wiimote różni się od nich znacząco, gdyż jest urządzeniem jednoręcznym.

Rysunek 13: Zadajnik Nintendo Wiimote

Wiimote wyposażony jest w 12 przycisków monostabilnych, obsługiwanych za pomocą kciuka i palca wskazującego. Zadajnik posiada wbudowany czujnik podczerwieni oraz trójosiowy akcelerometr liniowy. Oprócz tego posiada wbudowany głośnik, silnik wibracyjny oraz 4 diody. Dodatkowo posiada port rozszerzeń pozwalający podłączyć dodatkowe urządzenia.

Jako trójosiowy akcelerometr liniowy zastosowany został układ ADXL330[23] firmy

Analog Devices. Zakres odczytu urządzenia obejmuje zakres $[-3g; +3g]$ z dokładnością 10%. Układ mierzy w trzech osiach (rys. 14) wartości składowych sumy sił działających na niewielki obiekt umieszczony wewnątrz urządzenia i nie kompensuje w żaden sposób siły oddziaływania pola grawitacyjnego. Dlatego też czujnik urządzenia nieruchomego lub poruszającego się ruchem jednostajnym wskaże wartość wypadkowej sił działających na urządzenie wartości około 1g, a czujnik urządzenia spadającego swobodnie wskaże wartość sił równą 0. Między innymi z tego powodu nie można wyznaczyć bezwzględnych wartości przyspieszenia poruszającego się urządzenia. Jednak gdy urządzenie pozostaje w spoczynku lub gdy przyspieszenia są pomijalnie małe, a więc w sytuacji, gdy jedyną siłą działającą na urządzenie jest siła pola grawitacyjnego, ze stosunku składowych siły grawitacji na poszczególnych osiach akcelerometru można wyznaczyć kąty obrotu akcelerometru wokół osi X i Y. W dalszym ciągu niewiadomą pozostanie wartość obrotu wokół osi siły grawitacji, obrót ten bowiem nie powoduje zmiany wartości odczytywanych na poszczególnych osiach akcelerometru.



Rysunek 14: Osie akcelerometru wbudowanego w zadajni Wiimote

Czujnik podczerwieni stanowi monochromatyczna kamera o rozdzielczości 128x96 punktów interpolowana do rozdzielczości 1024x768 punktów. Oprogramowanie czujnika zwraca informację o maksymalnie czterech najjaśniejszych punktach wychwyconych przez czujnik. Czujnik wykorzystywany jest w parze z urządzeniem o nazwie Sensor Bar (rys. 15). Wbrew temu, co sugeruje nazwa, Sensor Bar nie zawiera czujników, a 2 zbiory diod

podczerwonych oddalonych od siebie około 20cm. Odległość na siatce pozycji tych dwóch źródeł wskazywanych przez odczyty czujnika podczerwieni pozwala w przybliżony sposób ustalić odległość zadajnika od Sensor Baru, a ich wzajemne umiejscowienie oraz kąt między prostą przechodzącą przez 2 punkty-źródła a krawędziami siatki, przy jednoczesnym wykorzystaniu odczytów z akcelerometrów, pozwala zgrubnie oszacować orientację zadajnika w przestrzeni. Głównym ograniczeniem jednak wykorzystania czujnika podczerwieni do celu ustalenia położenia i orientacji zadajnika jest stosunkowo niewielki kąt widzenia czujnika.

Rysunek 15: Sensor Bar

Wiimote posiada wbudowany głośnik piezoelektryczny o średnicy 21mm. Służy on do odtwarzania dźwięku transmitowanego strumieniowo do urządzenia. Oprogramowanie zadajnika obsługuje dźwięk w standardzie ADPCM próbkowany z rozdzielczością 4 bitów oraz dźwięk w standardzie PCM próbkowany z rozdzielczością 8 bitów. Domyślnym ustawieniem jest dźwięk ADPCM 4-bit z częstotliwością 3000Hz.

Wbudowany w urządzenie silnik obraca niewielką masę wokół osi nieprzechodzącej przez środek ciężkości tej masy, wprawiając na życzenie użytkownika urządzenie w wibracje. Urządzenie nie posiada możliwości regulacji siły wibracji. Silnik zasilany jest stałym napięciem 3.3V.

Urządzenie posiada również wbudowane 4 niebieskie diody LED zasilane stałym napięciem 2.66V. W standardowym zastosowaniu służą do sygnalizowania poziomu naładowania baterii oraz gotowości do nawiązania połączenia. Każda z nich może być zapalana niezależnie, przez co znajdują również inne zastosowania w poszczególnych aplikacjach.

Zadajnik Wiimote komunikuje się bezprzewodowo z konsolą za pomocą wbudowanego modułu BCM2042 firmy Broadcom, kompatybilnego ze standardem Bluetooth 2.0. Po nawiązaniu połączenia można przełączyć zadajnik w jeden z wielu trybów raportowania. Urządzenie przesyła raporty o aktualnych odczytach oraz swoim stanie z częstotliwością

do 50Hz. Raporty zawierają dane o wciśniętych przyciskach oraz dowolnie wybrane parametry spośród odczytu czujnika podczerwieni, odczytu akcelerometrów oraz odczytu z urządzenia podłączonego do portu rozszerzeń. Ze względu na ograniczony rozmiar przesyłanych raportów wybór większej liczby parametrów skutkuje zmniejszeniem rozdzielczości przesyłanych odczytów. Komunikacja z urządzeniem jest dwukierunkowa. Komunikaty przesyłane do urządzenia mogą zawierać sygnały sterujące dla diod i silnika wibracyjnego, kolejne fragmenty strumienia audio lub polecenia zmiany trybu raportowania.

Wiimote zasilany jest dwiema bateriami AA, pozwalającymi na ciągłe działanie do 60 godzin, gdy wykorzystywany jest tylko akcelerometr, oraz do 25 godzin, gdy włączony również jest czujnik podczerwieni. Urządzenie posiada dodatkowy układ składający się z kondensatora o pojemności 3300 mikrofaradów zabezpieczający przed sytuacjami krótkotrwałego braku zasilania wynikającego z drgań baterii związanych z gwałtownym ruchem zadajnikiem.

Zadajnik Wiimote dostępny jest na rynku za ok. 170 zł.

3.3.4 Wykorzystywane biblioteki programowe

Największą przeszkodą na drodze do wykorzystania Wiimote w aplikacji jest polityka firmy Nintendo. Producent konsoli nie udostępnia żadnych narzędzi służących do interakcji z urządzeniem, jak również jakiejkolwiek dokumentacji dotyczącej komunikacji z urządzeniem czy sterowników dla poszczególnych systemów operacyjnych. Możliwość programowej interakcji z urządzeniem zawdzięczać należy społeczności skupionej wokół idei "Wii homebrew", tj. wykorzystania konsoli Wii, jak również dedykowanych jej akcesoriów w sposób nieprzewidziany przez producenta. W ramach społeczności powstał między innymi projekt WiiLi[12], którego celem jest uruchomienie systemu operacyjnego Linux na konsoli. Specyfikacja komunikacji z urządzeniem została opracowana w ramach projektu WiiBrew[13] metodami inżynierii wstecznej. Na chwilę obecną udało się uzyskać niemal całkowitą wiedzę o interfejsie komunikacyjnym urządzenia. Bezproblemowo działa nawiązywanie połączenia Bluetooth. Struktura komunikatów sterujących głośnikiem, silnikiem wibracyjnym i diodami została całkowicie poznana, jak również struktura komunikatów zawierających odczyty czujnika podczerwieni, akcelerometru, stan przycisków i stan urządzenia. Nie są obsługiwane natomiast w pełni wszystkie rozszerzenia.

Zadajnik Nintendo Wiimote do komunikacji wykorzystuje protokół Bluetooth. Samodzielne tworzenie komunikatów i przesyłanie ich za pomocą natywnych funkcji protokołu Bluetooth jest jednak skomplikowane i czasochłonne. Od czasu wypuszczenia na rynek konsoli Wii powstało więc wiele bibliotek programowych wspomagających obsługę Wii-

[12]http://wiili.org
[13]http://wiibrew.org

mote. Dla systemu operacyjnego Windows dostępne są m.in. biblioteki GlovePIE[14], WiiYourself![15] oraz wiiuse[16]. Biblioteka wiiuse dostępna jest również dla systemu operacyjnego Linux. Inną popularną biblioteką dla Linuksa jest CWiid[17]. W aplikacji powstałej w ramach niniejszej pracy korzystam właśnie z biblioteki CWiid. Do komunikacji z urządzeniem wykorzystuje ona bibliotekę BlueZ[18], implementującą obsługę protokołu Bluetooth. Obie wspomniane biblioteki postaram się przybliżyć w niniejszym rozdziale.

Biblioteka BlueZ implementuje większość warstw i protokołów zawartych w standardzie Bluetooth 2.0. Biblioteka stanowi warstwę abstrakcji nad rzeczywistym sprzętem i umożliwia komunikację z urządzeniami przy pomocy gniazd. Umożliwia jednoczesną, wielowątkową obsługę wielu urządzeń zgodnych ze standardem. Zawiera sterowniki umożliwiające korzystanie z adapterów Bluetooth podłączonych za pomocą portów USB, UART lub PCIMCIA. Oprócz właściwiej biblioteki w skład pakietu BlueZ wchodzi szereg narzędzi służących konfiguracji i diagnostyce połączenia. Biblioteka działa na wielu architekturach sprzętowych - w szczególności Intel i AMD x86, AMD64, Sun SPARC, PowerPC i wielu innych. Jest również kompatybilna z najważniejszymi dystrybucjami systemu operacyjnego Linux - Debian GNU/Linux, Ubuntu, Fedora Core / Red Hat, OpenSuSE / SuSE, Mandrake.

Pakiet CWiid to zestaw napisanych w C narzędzi dla systemu operacyjnego Linux wspomagających obsługę zadajnika Wiimote lub wykorzystujących tenże zadajnik. W skład pakietu CWiid wchodzą następujące narzędzia:

- libcwiid - biblioteka programowa do obsługi urządzenia
- wminput - sterownik umożliwiający sterowanie kursorem myszy za pomocą zadajnika oraz wykorzystywanie zadajnika zamiast joysticka
- wmgui - graficzna konsola umożliwiająca wysyłanie komunikatów sterujących do zadajnika oraz wyświetlająca aktualne odczyty z czujników urządzenia
- wmdemo - aplikacja demonstrująca możliwości biblioteki i urządzenia

Biblioteka libcwiid wspomaga tworzenie aplikacji wykorzystujących zadajnik Wiimote. Udostępnia ona wysokopoziomowe funkcje sterujące urządzeniem czy pobierające z niego odczyty. Definiuje również szereg struktur reprezentujących komunikaty przesyłane do i z urządzenia. Struktury te wypełniane są przy pomocy zdefiniowanych przez bibliotekę stałych, dzięki czemu w sposób intuicyjny uzyskuje się czytelny i samodokumentujący

[14]http://sites.google.com/site/carlkenner/glovepie
[15]http://wiiyourself.gl.tter.org/
[16]http://wiiuse.net/
[17]http://abstrakraft.org/cwiid/
[18]http://www.bluez.org

się kod. Biblioteka obsługuje nie tylko sam zadajnik Wiimote, ale również szereg dodatkowych urządzeń podłączanych do portu rozszerzeń (m.in. Nintendo Nunchuck - prosty zadajnik wyposażony w joystick i 2 przyciski, czy klasyczny zadajnik (tzw. pad) konsoli GameCube).

Biblioteka umożliwia wysłanie do urządzenia czterech rodzajów poleceń:

- żądanie przesłania stanu urządzenia
- włączenie/wyłączenie wybranych diod
- włączenie/wyłączenie silnika wibracyjnego
- konfigurację sposobu raportowania

Wiimote umożliwia dwa tryby raportowania. Może udostępniać odczyty czy informacje o stanie urządzenia na żądanie. W zrealizowanej w ramach pracy aplikacji wykorzystuję jednak drugi tryb, w którym urządzenie w regularnych odstępach czasu przesyła raporty zawierające określony przy nawiązywaniu połączenia zbiór informacji. Do wyboru są następujące parametry:

- stan urządzenia (stan baterii, informacja o podłączonym rozszerzeniu)
- stan przycisków
- odczyty z akcelerometru
- odczyty czujnika podczerwieni
- raporty z dołączonego urządzenia - różne w zależności od rozszerzenia

Biblioteka libcwiid jest kompatybilna z systemami z rodziny Linux. Istnieją również prekompilowane pakiety dedykowane dla systemu ArchLinux, Debian i Gentoo.

4 Wymagania funkcjonalne

Celem mojej pracy było rozszerzenie struktury ramowej MRROC++ o narzędzie wspomagające programistę w tworzeniu fragmentów aplikacji poprzez generowanie trajektorii pozycyjnej. Wymagania stawiane zadajnikowi ruchu przedstawiłem w rozdziale 3.3. Analiza dostępnych na rynku rozwiązań pod kątem ich spełnienia miała ułatwić wybór optymalnego zadajnika na potrzeby niniejszej pracy. Wybór zadajnika i implementacja jego obsługi w strukturze stanowiły jednak tylko fragment pracy. Ważniejszym elementem była implementacja zadania uczenia robota trajektorii. Przed tym zadaniem postawiono również szereg wymagań funkcjonalnych, których spełnienie miało gwarantować powstanie produktu przede wszystkim przydatnego i łatwego w zastosowaniu.

Niniejszy rozdział przybliża następujące wymagania funkcjonalne stawiane zadaniu programowania robotów:

- możliwość zadawania przemieszczenia metodą przyrostową
- wybór układu współrzędnych, w którym zadawany będzie ruch
- możliwość dodawania/edycji/usuwania punktów do trajektorii
- wczytywanie i zapisywanie trajektorii z/do pliku
- możliwość odtwarzania trajektorii w dowolnym kierunku i o dowolną ilość węzłów
- sygnalizacja sytuacji wyjątkowych operatorowi

Zadawanie przemieszczenia metodą przyrostową Operator systemu robotycznego powinien mieć możliwość zadawania przemieszczeń końcówki manipulatora. Dostępne zadajniki (w tym wybrany do realizacji zadania zadajnik Wiimote) nie dostarczają wystarczających danych do tego, by móc zadawać za ich pomocą bezwzględną pozycję oczekiwaną końcówki manipulatora. Dlatego też zadanie uczenia robota powinno umożliwiać operatorowi zadawanie przemieszczeń względnych na wybranej osi w wybranym układzie współrzędnych. Możliwa powinna być obsługa ciągłego sygnału sterującego, tak by operator miał możliwość zadania dowolnej wartości prędkości czy przyspieszenia z dopuszczalnego przez mechaniczne ograniczenia zakresu. Umożliwić to powinno łatwe i możliwie precyzyjne osiągnięcie przez końcówkę manipulatora pożądanych przez operatora współrzędnych.

Wybór układu współrzędnych W zależności od aktualnej sytuacji w przestrzeni roboczej manipulatora oraz od ruchu manipulatorem, który operator systemu robotycznego zamierza wykonać, poszczególne układy współrzędnych są bardziej lub mniej do tego

celu odpowiednie. Ograniczenie ruchu do pojedynczej osi układu współrzędnych zmusza operatora do wcześniejszego planowania sekwencji przemieszczeń, która pozwoli osiągnąć zamierzony punkt. W takiej sytuacji dobór odpowiedniej reprezentacji ruchu i układ odniesienia może sprawić, że ruch manipulatorem będzie bardziej intuicyjny. Dodatkowo zmiana reprezentacji, w którym odbywa się ruch, pozwala zrealizować przejście przez punkty osobliwe. Zadanie programowania robotów powinno umożliwiać ruch w kilku układach współrzędnych oraz dać operatorowi systemu możliwość przełączania się pomiędzy nimi w trakcie realizacji zadania.

Dodawanie/edycja/usuwanie węzłów trajektorii Operator systemu robotycznego za pomocą zadajnika ruchu zadaje przemieszczenia końcówki manipulatora. Wskutek wykonania zadania programowania robota wygenerowana powinna zostać trajektoria, tj. sekwencja punktów w przestrzeni roboczej manipulatora. Operator powinien mieć możliwość tworzenia trajektorii, tj. dodawania do niej wybranych punktów, modyfikacji punktów już istniejących poprzez zastąpienie ich innymi punktami, a także usuwania zbędnych lub błędnych węzłów. Generacja trajektorii za pomocą zaimplementowanego zadania powinna składać się z przemieszczania końcówki manipulatora oraz wspomnianych wyżej operacji zarządzania trajektorią.

Wczytywanie i zapisywanie trajektorii z/do pliku Celem zadania programowania robotów nie jest samo stworzenie trajektorii i wykorzystanie jej w bieżącej instancji zadania, ale utrwalenie jej w celu późniejszego wykorzystania przez inne zadania. Operator systemu powinien mieć możliwość zapisu do pliku aktualnej trajektorii w dowolnym momencie wykonania zadania. Trajektoria powinna zostać zapisana w jasno zdefiniowanym formacie, zrozumiałym dla innych zadań działających w strukturze MRROC++. Co więcej, samo zadanie programowania robota powinno umożliwiać wczytanie trajektorii z pliku. Umożliwi to wprowadzanie poprawek w trajektoriach wygenerowanych podczas wcześniejszych przebiegów procesu uczenia, a również wykorzystanie dowolnej trajektorii jako trajektorii bazowej, którą operator systemu w dowolny sposób będzie modyfikował.

Odtwarzanie trajektorii Droga, którą pokona końcówka manipulatora podczas ruchu pomiędzy kolejnymi węzłami może być inna od tej, którą zadał operator systemu robotycznego, gdyż zadanie zapamiętuje tylko dodawane przez operatora węzły interpolacji trajektorii, a ruch pomiędzy nimi zależy od użytego do jej odtworzenia generatora. Ważne jest, by miał możliwość odtworzenia wygenerowanej dotychczas trajektorii w celu analizy ruchu, wprowadzenia poprawek lub dodania dodatkowych węzłów pośrednich w celu modyfikacji drogi pokonywanej przez końcówkę. Odtwarzanie trajektori powinno odbywać

się w dowolnym kierunku, tj. w rosnącej lub malejącej kolejności węzłów. Pozwoli to na przejście do dowolnego węzła trajektorii w celu jego modyfikacji.

Sygnalizacja sytuacji wyjątkowych Operator systemu robotycznego często przemieszcza się wokół przestrzeni roboczej manipulatora, chociażby w celu poprawy widoczności sceny robota. Nie zawsze znajduje się w pobliżu konsoli sterującej, potrzebna jest więc metoda sygnalizacji operatorowi sytuacji wyjątkowych w inny sposób. Od zadania programowania robota wymaga się zatem, by w jasny i zauważalny sposób dawało operatorowi znać, że system robotyczny osiągnął niepożądany stan. Sytuacje wyjątkowe nie powinny występować często, dlatego wystarczające będzie samo powiadomienie operatora o wystąpieniu takiej sytuacji, szczegółowy stan systemu dostępny jest bowiem dla niego z poziomu konsoli sterowniczej.

Jestem przekonany, że konieczność spełnienia powyższych wymagań przez zadanie programowania robotów spowoduje, że powstałe rozwiązanie będzie rozwiązaniem wartościowym z punktu widzenia operatora systemu robotycznego, tj. zmniejszy koszt programowania robota, rozumiany w szczególności jako czas niezbędny do wygenerowania oczekiwanej trajektorii, po której poruszać się powinna końcówka manipulatora.

5 Implementacja

5.1 Architektura

Powstałe w ramach pracy rozwiązanie dostarczyć miało narzędzi umożliwiających interaktywne programowanie manipulatora poprzez generowanie trajektorii dla generatorów pozycyjnych. Rozwiązanie miało stanowić element struktury ramowej MROCC++. Jednym z elementów rozwiązania miała być implementacja czujnika wirtualnego reprezentującego w systemie zadajnik Nintendo Wiimote.

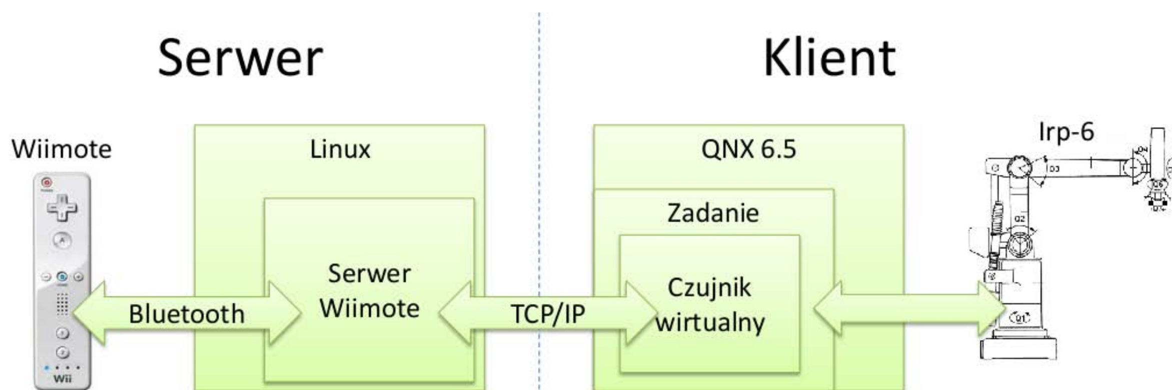
Struktura ramowa MRROC++ pracuje pod kontrolą systemu operacyjnego QNX 6.5.0. Zadajnik Wiimote komunikuje się za pośrednictwem protokołu Bluetooth, dla którego wsparcia ten system operacyjny nie zapewnia. Konieczne stało się więc wydzielenie obsługi zadajnika poza środowisko QNX. Do obsługi zadajnika wybrane zostało środowisko systemu operacyjnego Linux ze względu na dostępność w tym systemie gotowych bibliotek wspierających standard Bluetooth, jak również bibliotek programowych implementujących komunikację z zadajnikiem za pomocą wspomnianego protokołu, w tym użytej przeze mnie biblioteki libcwiid, opisanej w rozdziale 3.3.4.

Implementacja rozwiązania w oparciu o dwa heterogeniczne środowiska wymusiło realizację zadania z uwzględnieniem konieczności wymiany danych pomiędzy tymi środowiskami za pośrednictwem sieci. Rozwiązanie zrealizowane zostało w architekturze klient-serwer.

Architektura klient-serwer to jeden z modeli aplikacji rozproszonej, który zakłada rozdzielenie funkcji realizowanej przez aplikację pomiędzy serwer, który dostarcza usługę lub udostępnia pewien zasób, oraz dowolną ilość aplikacji klienckich, które z tych usług lub zasobów korzystają. Rozdzielenie aplikacji jest istotne tylko na poziomie funkcjonalnym, jednak bardzo często aplikacja zostaje rozdzielona fizycznie pomiędzy różne, często odległe maszyny, pomiędzy którymi komunikacja odbywa się za pomocą protokołów sieciowych. Istotną cechą tej architektury jest to, że stroną inicjalizującą wymianę danych jest aplikacja kliencka, która przesyła żądania do serwera. Serwer pozostaje bierny i nie komunikuje się z aplikacjami klienckimi za wyjątkiem sytuacji, w których na żądanie klienta przetwarza je i zwraca odpowiedź.

W opisanej architekturze zrealizowana została implementacja czujnika wirtualnego, opisana w rozdziale 5.2. Część serwerową stanowi proces działający pod kontrolą systemu operacyjnego Linux. Komunikuje się on z zadajnikiem Wiimote za pomocą protokołu Bluetooth i agreguje otrzymane od niego dane. Z definicji proces ten pełni jednocześnie funkcję usługową dla innych procesów, nasłuchując żądań, przetwarzając je i zwracając odpowiedź, np. ostatnio odczytany stan zadajnika. Procesy zadań działające w ramach struktury MRROC++ i wykorzystujące czujnik wirtualny Wiimote reprezentują w tej ar-

chitekturze aplikacje klienckie. Inicjalizują one połączenie z nasłuchującym na wybranym porcie serwerem i przesyłają polecenia konfiguracji zadajnika lub odczytu danych. Szerszy opis implementacji serwera i klienta znajduje się w rozdziale 5.2. Rysunek 16 przedstawia ogólną strukturę systemu.



Rysunek 16: Ogólna struktura systemu

5.2 Proces VSP

Zadajnik Nintendo Wiimote został zaimplementowany w systemie jako czujnik wirtualny. Ze względu na brak wsparcia dla obsługi standardu Bluetooth w systemie operacyjnym QNX, zadajnik obsługiwany jest przez osobny proces VSP[19] działający pod kontrolą systemu operacyjnego Linux. Proces ten działa w systemie jako serwer. Łączy się z zadajnikiem Wiimote i pobiera aktualne odczyty, a dodatkowo nasłuchuje na wybranym porcie na połączenia przychodzące ze strony systemu MRROC++. Po nawiązaniu połączenia serwer dodatkowo odbiera i wykonuje polecenia otrzymane z procesów MRROC++, tj. zmienia konfigurację zadajnika lub przesyła ostatnio pobrane odczyty.

Serwer Serwer reprezentowany jest przez obiekt klasy *wiimoteServer*, zdefiniowany w pliku nagłówkowym *WiimoteServer.h*. Jest to prosta implementacja jednowątkowego procesu serwera nasłuchującego na wybranym porcie TCP/IP. Posiada on wskaźnik na obiekt reprezentujący zadajnik Wiimote, którego ostatnio odczytany stan udostępnia. Pojedynczy obiekt jest w stanie utrzymywać jednocześnie tylko jedno połączenie.

Metoda	**Opis**
wiimoteServer()	inicjalizuje obiekt serwera
wiimoteServer(wiimote* wii)	inicjalizuje obiekt serwera i przypisuje mu wybraną instancję obiektu reprezentującego zadajnik Wiimote

[19]Virtual Sensor Process

void run(int port)	uruchamia serwer na zadanym porcie
void stop()	zatrzymuje serwer
bool getIsRunning()	zwraca true, gdy serwer jest uruchomiony
void setWiimote(wiimote* wii)	wiąże serwer z wybranym obiektem reprezentującym zadajnik

Tablica 11: Interfejs klasy wiimoteServer

Zadajnik W procesie serwera zadajnik reprezentowany jest przez obiekt klasy *wiimote* zdefiniowany w pliku nagłówkowym *wiimote.h*. Ukrywa on komunikację z zadajnikiem i na bieżąco udostępnia ostatnio pobrane z zadajnika odczyty. Udostępnia też szereg metod umożliwiających zmianę konfiguracji zadajnika.

Metoda	**Opis**
wiimote()	inicjalizuje obiekt
bool connect(char* address)	inicjalizuje połączenie z zadajnikiem reprezentowanym przez podany adres
void disconnect()	kończy połączenie z zadajnikiem
float** calibrate()	przeprowadza procedurę kalibracji urządzenia, zwraca tablicę współczynników kalibracji, które można zapisać i następnym razem pominąć etap kalibracji, korzystając z metody *precalibrate()*
void precalibrate(float calibration[3][4])	skalibrowanie urządzenia przy użyciu wyliczonych wcześniej współczynników
void setRumble(bool on)	włącza/wyłącza silnik wibracyjny
void setLeds(int leds)	ustawia stan diod LED
bool getIsConnected()	zwraca true, gdy obiekt jest połączony z zadajnikiem
bool getIsCalibrated()	zwraca true, gdy zadajnik jest skalibrowany
buttonStatus getButtons()	zwraca ostatnio pobrany stan przycisków
orientationStatus getOrientation()	zwraca ostatnio odczytaną orientację urządzenia, tj. kąty obrotu wokół osi zadajnika (rozdział 5.2.3)
accelerationStatus getAcceleration()	zwraca ostatnio odczytane przyspieszenia na poszczególnych osiach akcelerometru (rozdział 5.2.2)

Tablica 12: Interfejs klasy wiimote

Rejestr zadajników Rejestr zadajników to konstrukcja powstała ze względu na proceduralny charakter biblioteki libcwiid. Funkcje tej biblioteki w momencie otrzymania kolejnego odczytu z urządzenia uruchamiają zdefiniowaną przez użytkownika funkcję zwrotną, którą niestety nie może być metoda żadnego obiektu, w tym metoda klasy *wiimote*. Dlatego też dla każdej funkcji zwrotnej klasy *wiimote* powstała globalna funkcja (tzw. wrapper) wołana przez funkcjie biblioteki libcwiid, pobierająca na podstawie adresu urządzenia odpowiedni obiekt klasy *wiimote* z rejestru i uruchamiająca zdefiniowaną dla niego funkcję zwrotną. Klasa rejestru zadajników zdefiniowana jest w pliku nagłówkowym *WimoteRegistry.h*. Implementuje ona wzorzec projektowy Rejestru Obiektów.

Metoda	**Opis**
static void addWiimote(cwiid_wiimote_t* device,wiimote* wii)	dodaje do rejestr obiekt wiimote skojarzony z adresem obiektu biblioteki libcwiid
static wiimote* getWiimote(cwiid_wiimote_t* device)	zwraca obiekt wiimote skojarzony z adresem
static wiimote* removeWiimote(cwiid_wiimote_t* device)	usuwa wybrany obiekt z rejestru

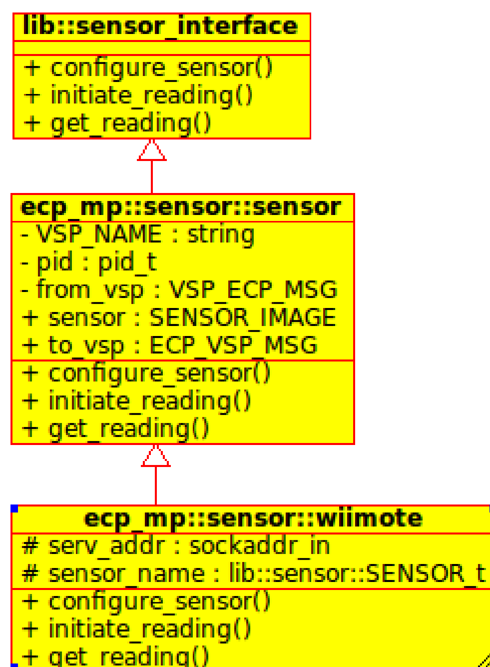
Tablica 13: Interfejs klasy wiimoteRegistry

Klient Aplikacją kliencką może być dowolne zadanie działające pod kontrolą struktury MRROC++. Dostęp do czujnika zapewnia obiekt klasy *wiimote*, reprezentujy czujnik wirtualny. Obiekt czujnika zapewnia dostęp do struktury *image*, zawierającej odczyt z czujnika.

Diagram klas czujnika znajduje się na rysunku 17.

Utworzenie procesu VSP Uruchomienie procesu VSP na dowolnej maszynie działającej pod kontrolą systemu operacyjnego Linux i wyposażonej w moduł Bluetooth jest nieskomplikowane, gdyż zdecydowaną wiekszość funkcji implementują przedstawione wcześniej klasy.

Do lokalnej komunikacji z zadajnikiem Wiimote wystarczy utworzyć obiekt klasy *wiimote*. Każde urządzenie pracujące w standardzie Bluetooth identyfikowane jest unikalnym 48-bitowym adresem, oznaczanym często symbolem BD_ADDR. Inicjalizacja połączenia z zadajnikiem następuje przez wywołanie metody *wiimote::connect()*, która jako jedyny argument przyjmuje adres zadajnika urządzenia zapisany w postaci



Rysunek 17: Diagram klas czujnika

XX:XX:XX:XX:XX:XX, gdzie X to cyfra w systemie szesnastkowym. Zadajnik Wiimote po pewnym czasie nieaktywności wyłącza się w celu oszczędzania baterii, dlatego też należy go aktywować wciskając jednocześnie przyciski 1 i 2. Zadajnik się włączy i zacznie nadawać swój sygnał, po czym nastąpi sparowanie zadajnika i obiektu klasy *wiimote*. Po nawiązaniu połączenia obiekt rejestruje się w rejestrze zadajników korzystając z metod statycznych klasy *wiimoteRegistry*.

Kolejnym krokiem po podłączeniu obiektu do zadajnika jest kalibracja urządzenia (opisana w rozdziale 5.2.1) poprzez wywołanie metody *wiimote::calibrate()*. Jeżeli zadajnik był już wcześniej kalibrowany, można zainicjalizować go wartościami uzyskanymi z poprzedniej kalibracji i pominąć ten krok, korzystając z metody *wiimote::precalibrate()*. Po poprawnym wykonaniu którejś z wspomnianych metod obiekt rejestruje przy pomocy metody biblioteki libcwiid funkcję zwrotną *global_msg_callback()*, która będzie wołana w momencie przesłania nowych odczytów z zadajnika. Funkcja ta za każdym wywołaniem będzie uruchamiała metodę *wiimote::msg_callback()*, która na podstawie odczytów z zadajnika aktualizuje stan obiektu go reprezentującego. Od tego momentu obiekt reprezentujący zadajnik nadaje się do użycia.

Udostępnianie odczytów z zadajnika zdalnym procesom za pomocą gniazd TCP/IP wymaga użycia dodatkowo obiektu klasy *wiimoteServer*. Obiektowi tej klasy należy przypisać zadajnik korzystając z metody *wiimoteServer::setWiimote()*. Po przypisaniu zadajnika do serwera wystarczy uruchomić serwer na wybranym porcie za pomocą metody

wiimoteServer::run(), co spowoduje przejście serwera w tryb oczekiwania na połączenie. Po połączeniu serwer będzie czekał na polecenia ze zdalnego procesu i w zależności od niego konfigurował zadajnik lub przesyłał ostatnio otrzymane z zadajnika odczyty.

5.2.1 Kalibracja zadajnika

Zadajnik Wiimote przekazuje do procesu VSP informacje o swoim stanie, a w szczególności wartości odczytów poszczególnych osi akcelerometru. Nie są to jednak wartości zmierzonego przez niego przyspieszenia, a jedynie surowe odczyty czujnika, które proces VSP powinien dopiero zagregować, by uzyskać przyspieszenie na poszczególnych osiach.

Pryspieszenia mierzone na pojedynczej osi akcelerometru mogą być zarówno dodatnie, jak i ujemne. Wartości zwracane przez akcelerometr wbudowany w zadajnik Wiimote są zawsze dodatnie i nie jest określone, jaka wartość odczytu oznacza zerowe przyspieszenie na wybranej osi. Nie jest określona również zależność zwracanej przez akcelerometr wartości od rzeczywistego zmierzonego przyspieszenia. Z tych powodów konieczne jest przeprowadzenie procedury kalibracji urządzenia, a raczej oprogramowania, w celu określenia wartości wspomnianych niewiadomych.

Aby poprawnie określić zależność między przyspieszeniem mierzonym na poszczególnych osiach akcelerometru potrzebna jest wiedza o rzeczywistym przyspieszeniu i wartości zmierzonej. Poznanie rzeczywistego przyspieszenia w zakresie mierzonym przez akcelerometr wbudowany w Wiimote jest niemożliwe bez użycia innego, już skalibrowanego akcelerometru, konieczne więc było przyjęcie kilku założeń, co miało uprościć procedurę kalibracji.

Pierwszym założeniem jest to, iż zależność wartości odczytywanej przez akcelerometr od rzeczywistego przyspieszenia jest liniowa. Dzięki temu nie byłoby konieczne przeprowadzanie pomiarów wartości przyspieszenia w pełnym zakresie przyspieszenia mierzonego na pojedynczej osi. Wystarczyłby w takim wypadku pojedynczy pomiar, na którego podstawie, znając punkt zerowy skali, można wyznaczyć zależność wartości zmierzonego przyspieszenia od wartości zwracanej przez akcelerometr. Możnaby wtedy również wykorzystać przyspieszenie o znanej wartości, np. przyspieszenie ziemskie, dzięki czemu pomiar można wykonać bez użycia dodatkowego akcelerometru.

Drugim założeniem jest to, że układ akcelerometru jest umieszczony wewnątrz zadajnika w taki sposób, że poszczególne osie akcelerometru są prostopadłe do poszczególnych ścian zadajnika. W takiej sytuacji łatwe jest określenie kierunku osi akcelerometru na podstawie orientacji zadajnika. Dodatkowo możliwe jest łatwe wyizolowanie pojedynczej osi podczas pomiaru przyspieszenia ziemskiego poprzez położenie zadajnika wybranej ściance na wypoziomowanej powierzchni.

Powyższe założenia pozwoliły uprościć procedurę kalibracji do dwóch prostych kroków

niewymagających dodatkowych urządzeń.

Krok 1 - pomiar przyspieszenia ziemskiego na pojedynczej osi W pierwszym kroku należy dokonać pomiaru znanego wektora przyspieszenia równoległego do wybranej osi akcelerometru. Najłatwiej dostępnym znanym przyspieszeniem jest przyspieszenie ziemskie i ono też zostało użyte do kalibracji. Dzięki drugiemu ze wspomnianych założeń izolacji wybranej osi akcelerometru można dokonać poprzez położenie zadajnika na wybranych ściankach na wypoziomowanej powierzchni. W celu dokonania procedury kalibracji należy:

- położyć zadajnik tak, by przycisk A był na wierzchu, a następnie zapisać odczytane wartości x_1, y_1, z_1 - w tej orientacji oś X zadajnika pokrywa się z kierunkiem wektora siły grawitacji
- położyć zadajnik czujnikiem podczerwieni do dołu, a następnie zapisać odczytane wartości x_2, y_2, z_2 - w tej orientacji oś Y zadajnika pokrywa się z kierunkiem wektora siły grawitacji
- położyć zadajnik na boku, tak by jego lewa strona była na wierzchu, a następnie zapisać odczytane wartości x_3, y_3, z_3 - w tej orientacji oś Z zadajnika pokrywa się z kierunkiem wektora siły grawitacji

W każdej z opisanych orientacji zadajnika wartość zmierzona to wartość siły grawitacji na jednej z osi oraz wartość punktu zerowego na pozostałych osiach.

Krok 2 - wyznaczenie punktów zerowych Wartości punktów zerowych, tj. wartości odczytów akcelerometru oznaczających mierzone przyspieszenie równie sile grawitacji, można wyznaczyć z następujących zależności:

$$\begin{aligned} x_0 &= \frac{x_1 + x_2}{2} \\ y_0 &= \frac{y_1 + y_3}{2} \\ z_0 &= \frac{z_2 + z_3}{2} \end{aligned} \tag{1}$$

Wartości x_1 i x_2 to wartości zmierzone, gdy siła grawitacji działała wyłącznie na oś Y lub oś Z, zatem powinny być to wartości punktów zerowych dla osi X. Oba pomiary mogą się nieznacznie różnić, chociażby ze względu na ograniczenia konstrukcyjne zadajnika, wskutek których osie akcelerometru mogą nieco odbiegać od osi obudowy zadajnika. Dlatego też wartości zmierzonych punktów zerowych zostały uśrednione, i tą uśrednioną

wartość x_0 wykorzystuję przy wyznaczaniu wartości przyspieszenia działacego na zadajnik. Analogicznie należy interpretować zależności, z których uzyskałem wartości punktów zerowych y_0, y_0 i z_0.

5.2.2 Wyznaczanie wartości przyspieszenia

Wyznaczone w procedurze kalibracji wartości wykorzystywane są do określania przyspieszenia działającego na zadajnik. Współczynniki te są niezmienne dla danego egzemplarza zadajnika, dlatego też kalibrację wystarczy przeprowadzić raz.

Niech x_r, y_r i z_r oznaczają wartości przyspieszeń na poszczególnych osiach przesłane z zadajnika Wiimote. Wartość zmierzonego przyspieszenia (w "g") można wyznaczyć z następującej zależności:

$$
\begin{aligned}
x &= \frac{x_r - x_0}{x_3 - x_0} \\
y &= \frac{y_r - y_0}{y_2 - y_0} \\
z &= \frac{z_r - z_0}{z_1 - z_0}
\end{aligned}
\tag{2}
$$

Różnica między x_r i x_0 to wartość aktualnego odczytu względem określonego wcześniej punktu zerowego. Znak tej wartości oznacza kierunek działania siły. Wartość ta wyrażona jest w jednostkach wewnętrznych akcelerometru i sama nie stanowi wartości użytecznej, dlatego też należy na jej podstawie wyznaczyć wartość przyspieszenia w "g", tj. wielokrotności przyspieszenia ziemskiego. Zadanie to spełnia wartość w mianowniku, czyli różnica pomiędzy maksymalnym odczytem podczas kalibracji, tj. odczytem w chwili, gdy na akcelerometr działała tylko siła grawitacji, a wyznaczonym punktem zerowym. Wartość tej różnicy to wartość odczytu dla przyspieszenia równego 1g w jednostkach wewnętrznych akcelerometru. Uzyskana z powyższej zależności wartość x to wartość przyspieszenia odczytanego na osi X akcelerometru wyrażona w wielokrotności wartości przyspieszenia ziemskiego. Analogicznie interpretować należy zależności, na podstawie których wyznaczone zostały wartości Y i Z.

Odczyty z akcelerometru są niestety zaszumione. Odczyty z nieruchomo leżącego zadajnika w każdym odczycie mogą się różnić o 15%. Konieczne było zastosowanie metody filtracji szumów w celu uniknięcia dużych skoków wartośći przyspieszenia w kolejnych odczytach.

W implementacji procesu VSP powstałej na potrzeby niniejszej pracy zastosowałem metodę średniej kroczącej. Zastosowanie wartości średniej pozwoliło pozbyć się gwałtownych skoków odczytu wskutek dużego szumu sygnału. Zastosowanie średniej kroczącej

sprawiło, że wartość uzyskiwana dzięki prostej procedurze opisanej wcześniej była wystarczająca bliska rzeczywistej wartośći przyspieszenia. Wartość średnia dodatkowo wygładza krzywą przyspieszenia, dzięki czemu proces sterujący robotem otrzymuje informacje o przyspieszeniu pozbawioną nagłych i krótkotrwałych zmian, co też jest pożądane. Wadą tego rozwiązania jest wprowadzenie pewnej bezwładności odczytu, objawiającej się opóźnieniem między rzeczywistą zmianą wartości przyspieszenia a zmianą wartości zwracanej przez akcelerometr. Opóźnienie jest jednak na tyle małe, że niedogodności wynikające z zastosowania opisanej metody są pomijalne.

5.2.3 Wyznaczanie orientacji zadajnika

W rozdziale 5.2.2 opisuję wyznaczanie wartości przyspieszenia działającego na poszczególne osie akcelerometru. O wiele bardziej wartościową informacją byłaby orientacja zadajnika w przestrzeni, tj. kąt obrotu zadajnika wokół poszczególnych jego osi. Niemożliwe jest jednak wyznaczenie tych kątów na podstawie znanych wartości przyspieszenia bez przyjęcia dodatkowych założeń.

Wyznaczenie orientacji zadajnika umożliwia założenie, że na zadajnik działa tylko siła grawitacji. Oznaczałoby to, że wypadkowa sił działających na poszczególne osie akcelerometru stanowi wektor siły grawitacji. Jest to sytuacja zbliżona do rzeczywistości, gdyż przyspieszenia działające na akcelerometr wynikające z poruszania nim przez operatora są znacznie mniejsze niż wartość siły grawitacji, a sam czas ruchu jest również krótki w stosunku do czasu trwania zadania programowania robota.

Mimo przyjęcia powyższego założenia wciąż niemożliwe jest wyznaczenie orientacji zadajnika w przestrzeni tylko na podstawie odczytów akcelerometru. Wskazują one wartość siły grawitacji działającej na poszczególne osie zadajnika, obrót zadajnika zaś wokół wektora siły grawitacji, tj. w płaszczyźnie poziomej, nie powoduje zmiany wartości zmierzonego przyspieszenia na żadnej z osi akcelerometru. Dlatego też przy przyjęciu wspomnianego założenia możliwe jest wyznaczenie w danej chwili obrotu wokół dwóch z trzech osi akcelerometru.

Wyznaczenie orientacji odbywa się w dwóch krokach.

Krok 1 - normalizacja wartości przyspieszenia Pierwszy krok zapewnić ma realizację założenia wspomnianego wcześniej poprzez znormalizowanie wartości odczytanych na poszczególnych osiach akcelerometru tak, by wypadkowa sił była równa co do wartości wektorowi siły grawitacji. Dzięki takiemu przekształceniu dalsze obliczenia będą mogły być przeprowadzane tak, jakby założenie było spełnione, a wszelkie siły oddziałujące na zadajnik, inne niż siła grawitacji, będą wprowadzać do wyznaczonych kątów odchylenia od rzeczywistej orientacji kontrolera.

Normalizacja wartości odbywa się wg następujących wzorów:

$$\begin{aligned} norm &= \frac{1}{|xd| + |yd| + |zd|} \\ x' &= \frac{x}{norm} \\ y' &= \frac{y}{norm} \\ z' &= \frac{z}{norm} \end{aligned} \tag{3}$$

Krok 2 - wyznaczenie orientacji Kąty obrotu wokół poszczególnych osi akcelerometru wyznaczane są na podstawie wyznaczonych w poprzednim kroku znormalizowanych wartości przyspieszenia na poszczególnych osiach akcelerometru. Kąt wyznaczany jest na podstawie stosunku wartości sił działających na poszczególne pary osi akcelerometru. Kąt obrotu wokół osi X wyznaczany jest na podstawie stosunku wartości sił działających na osie Y i Z. Pomiędzy poszczególnymi osiami akcelerometru znajduje się kąt prosty, dlatego też do wyznaczenia obrotu można wykorzystać funkcję *arcus tangens*. Dziedziną funkcji jest zbiór liczb rzeczywistych - w tym przypadku stosunek wartości sił działających na pary osi akcelerometru. Wartością zwracaną jest liczba z zakresu $[-\pi, \pi]$, a więc poszukiwany przeze mnie kąt obrotu wokół osi.

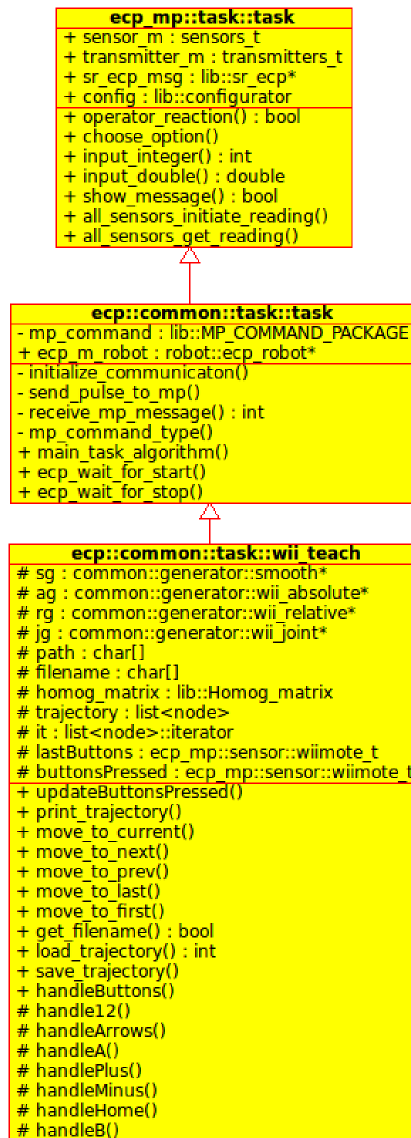
Kąt obrotu wokół osi akcelerometru można wyznaczyć zatem z zależności:

$$\begin{aligned} rotX &= atan(\frac{y'}{z'}) \\ rotY &= atan(\frac{x'}{z'}) \\ rotZ &= atan(\frac{x'}{y'}) \end{aligned} \tag{4}$$

5.3 Proces ECP

Proces ECP steruje przebiegiem wykonania zadania. Rolą procesu jest wykonanie przez pojedynczego robota algorytmu działania, które zapewni osiągnięcie pożądanego przez operatora systemu robotycznego celu. Cel ten jest osiągany poprzez wyznaczanie kolejnych ruchów robota na podstawie informacji o aktualnym położeniu i stanie robota oraz odczytów z czujników wirtualnych.

Zadanie programowania robotów przy pomocy zadajnika Wiimote zaimplementowane zostało w klasie *wii_teach*. Rozszerza ona opisaną w rozdziale 3.2.2 klasę *ecp::common::task::task*. Zadanie wii_teach umożliwia programowanie pojedynczego robota poprzez generowanie trajektorii pozycyjnej zgodnie z wymaganiami funkcjonalnymi określonymi w rozdziale 4. Diagram klas zadań znajduje się na rysunku 18.



Rysunek 18: Diagram klas zadania wii_teach

Klasa *wii_teach* dziedziczy z klasy nadrzędnej obiekty niezbędne każdemu zadaniu do poprawnego działania, dodatkowo definiuje szereg obiektów niezbędnych do wykonania algorytmu uczenia robota (tablica 14). Są to przede wszystkim generatory trajektorii, umożliwiające poruszanie robotem i odtwarzanie trajektorii oraz sama wygenerowana trajektoria.

Obiekt	**Opis**
common::generator::smooth* sg	Generator smooth2, autorstwa Rafała Tulwina, wykorzystywany do odtwarzania wygenerowanej trajektorii (rozdział 3.2.5)

common::generator::wii_absolute* ag	Ruch w układzie bazowym umieszczonym w końcówce manipulatora (rozdział 5.5.4)
common::generator::wii_relative* rg	Ruch w układzie końcówki manipulatora (rozdział 5.5.3)
common::generator::wii_joint* jg	Generator ruchu w przestrzeni stawów manipulatora (rozdział 5.5.2)
char path[80], char filename[20]	Zmienne przechowujące ścieżkę do pliku, z którego wczytana zostanie trajektoria i do którego wygenerowana trajektoria zostanie wpisana
lib::Homog_matrix homog_matrix	Pomocnicza macierz jednorodna
list<node> trajectory	Aktualna trajektoria robota - na początku zadania jest to trajektoria wczytana z pliku (jeżeli zawierał trajektorię) lub trajektoria pusta, w miarę przebiegu zadania uczenia zawiera aktualną wersję trajektorii, która zostanie zapisana do pliku
list<node>::iterator it	Iterator, wskazujący na aktualny węzeł trajektorii
ecp_mp::sensor::wiimote_t lastButtons	Ostatnio pobrany stan przycisków zadajnika
ecp_mp::sensor::wiimote_t buttonsPressed	Informacja o tym, które przyciski zadajnika zostały dotychczas wciśnięte. Procedury obsługujące wciśnięcie przycisku powinny czyścić flagę dla danego przycisku po obsłudze zdarzenia.

Tablica 14: Obiekty definiowane przez klasę wii_teach

Konstruktor obiektu zadania inicjalizuje środowisko, umożliwiając realizację algorytmu zadania. Tworzy on obiekt robota oraz obiekt czujnika wirtualnego *wii*, który udostępnia odczyty z zadajnika Wiimote. Inicjalizowane są w tym miejscu również struktury przechowujące informację o wygenerowanej trajektorii.

Funkcja *main_task_algorithm* implementuje algorytm uczenia robota. Komunikuje się ona z zadajnikiem Wiimote poprzez czujnik wirtualny i na podstawie poleceń operatora i zadawanych z zadajnika sygnałów sterujących podejmuje adekwatne działania. Funkcja ta zawiera podstawową logikę zadania - funkcje niższego poziomu, takie jak obsługa poszczególnych poleceń operatora, wydelegowane zostały do funkcji pomocniczych (tabela 15).

Metoda	**Opis**

void updateButtonsPressed()	Na podstawie informacji o ostatnio pobranym stanie przycisków aktualizuje strukturę buttonsPressed, ustawiając odpowiednie flagi, gdy poszczególne przyciski zostały wciśnięte. Ta informacja służy do wywołania odpowiednich metod obsługi zdarzeń. Informacja o wciśnięciu danego przycisku jest przesyłana z każdym raportem z zadajnika, więc pojedyncze wciśnięcie przycisku jest raportowane z każdym odczytem, aż do jego zwolnienia.
void print_trajectory(void)	Przesyła do procesu SR komunikaty zawierające współrzędne kolejnych węzłów wygenerowanej trajektorii. Proces SR powinien wypisać je na konsolę operatora.
void move_to_current(void), void move_to_next(void), void move_to_prev(void), void move_to_first(void), void move_to_last(void)	Przy użyciu generatora smooth2 porusza manipulatorem odpowiednio do ostatnio osiągniętej pozycji, pozycji następnej i poprzedniej lub bezpośrednio do pierwszego lub ostatniego węzła trajektorii.
bool get_filenames(void)	Pobiera od operatora ścieżkę pliku, pustego lub zawierającego trajektorię. Jeżeli plik zawiera trajektorię, zostanie ona wczytana i ustawiona jako aktualna trajektoria wygenerowana w zadaniu. Po zakończeniu zadania do podanego pliku zapisana zostanie wygenerowana trajektoria.
int load_trajectory(void)	Wczytuje trajektorię z pliku podanego przez operatora w metodzie *get_filenames()*.
void save_trajectory(void)	Zapisuje trajektorię do pliku podanego przez operatora w metodzie *get_filenames()*.
void handleButtons(void)	Obsługuje wciśnięcie przycisków, delegując obsługę poszczególnych zdarzeń do odpowiednich funkcji obsługi zdarzeń (tabela 16).

Tablica 15: Metody definiowane przez klasę wii_teach

Metoda	**Opis**

void handle12()	Obsługuje wciśnięcie przycisków 1 i 2. Przełączają one zadanie pomiędzy poszczególnymi generatorami.
void handleA()	Obsługuje wciśnięcie przycisku A. Wywołuje metodę Move() na aktualnie wybranym generatorze. Zadanie wraca z tej funkcji, gdy zostanie spełniony warunek stopu sprawdzany w generatorze, tj. zostanie zakończony zadany ruch i ponownie zostanie wciśnięty przycisk A.
void handleArrows()	Obsługuje wciśnięcia strzałek. W zależności od wciśniętego kierunku wywołuje jedną z metod move_to_next(void), move_to_prev(void), move_to_first(void) lub move_to_last(void) w celu zadania robotowi przemieszczenia do wybranego węzła trajektorii.
void handlePlus()	Obsługuje wciśnięcie przycisku +. Dodaje aktualnie osiągniętą pozycję manipulatora za aktualnym węzłem trajektorii.
void handleMinus()	Obsługuje wciśnięcie przycisku -. Usuwa aktualny węzeł trajektorii.
void handleHome()	Obsługuje wciśnięcie przycisku Home. Zamienia aktualny węzeł trajektorii na aktualną pozycję manipulatora.
void handleB()	Zapisuje trajektorię do pliku pobranego w metodzie get_filenames() przy pomocy metody save_trajectory().

Tablica 16: Metody obsługi zdarzeń, definiowane przez klasę wii_teach

Sama funkcja *main_task_algorithm* inicjalizuje tylko wspomniane wcześniej cztery generatory trajektorii, pobiera od operatora systemu nazwę pliku, w którym zapisana powinna zostać wygenerowana trajektoria, wczytuje trajektorię, jeżeli plik ją zawierał, a następnie w pętli pobiera polecenia od operatora i wywołuje odpowiednie funkcje obsługi. Logikę tej funkcji w pseudokodzie przedstawiłem na listingu 3.

```
void main_task_algorithm()
{
```

```
  generator_smooth          = utworz_generator_smooth();
  generator_wii_absolute = utworz_generator_wii_absolute();
  generator_wii_relative = utworz_generator_wii_relative();
  generator_wii_joint     = utworz_generator_wii_joint();

  aktualny_generator = wyznacz_generator();

  plik = pobierz_nazwe_pliku();
  wczytaj_trajektorie(plik);

  while(true)
  {
    odczyt = pobierz_odczyt();
    polecenie = wyznacz_polecenie(odczyt);

    if(czy_polecenie_ruchu(polecenie))
        aktualny_generator->Move();
    else if(czy_polecenie_zmiany_generatora(polecenie))
        aktualny_generator = wyznacz_generator(polecenie);
    else if(czy_polecenie_odtwarzania_trajektorii(polecenie))
        generator_smooth->Move();
    else if(czy_polecenie_zapisu())
        zapisz_trajektorie(plik);
    else obsluga(polecenie);
  }
}
```

Listing 3: Logika funkcji main_task_algorithm zadania wii_teach

Trajektoria Dotychczas wygenerowana trajektoria jest przechowywana w pamięci procesu. W aplikacjach struktury MRROC++ trajektoria taka najczęściej przechowywana jest w obiekcie generatora. W przypadku zaimplementowanego w ramach pracy zadania *wii_teach* jest to niemożliwe, gdyż korzysta ono aż z trzech generatorów. Każdy z nich wykorzystywany jest do przemieszczenia końcówki manipulatora, ale nie posiada informacji o pozycjach, które zostały dodane jako nowe węzły trajektorii. Informację taką posiada obiekt zadania i to tutaj przechowywana jest dotychczas wygenerowana trajektoria.

Trajektoria przechowywana jest jako obiekt klasy std::list<node> (listing 4). Dodatkowo obiekt zadania posiada iterator *it*, wskazujący na aktualny węzeł trajektorii, służący

do przechodzenia pomiędzy kolejnymi węzłami trajektorii w dowolnym kierunku. Dwukierunkowość była kluczowym wymaganiem stawianym strukturze danych przechowującej trajektorię, gdyż istotne było umożliwienie odtwarzania trajektorii w przód i w tył. Dodatkowo niezbędne są wskaźniki na początkowy i końcowy węzeł, by zrealizować przejście bezpośrednio do tych węzłów bez przechodzenia przez węzły pośrednie.

```
struct
{
  public:
    int id;
    double* position;
} node;

std::list<node> trajectory;
std::list<node>::iterator* it;
```

Listing 4: Struktury przechowujace trajektorie w zadaniu wii_teach

5.4 Komunikacja

Komunikacja pomiędzy procesem VSP a ECP odbywa się dwukierunkowo. Z procesu ECP do procesu VSP przesyłane są polecenia konfiguracyjne oraz żądania odczytu stanu zadajnika. Proces VSP zaś przesyła do procesu ECP aktualny stan zadajnika.

Struktury komunikacyjne Sposób komunikacji pomiędzy procesami ECP i VSP opisany został w rozdziale 3.2.3, poświęconym implementacji czujnika wirtualnego w strukturze MRROC++. W identyczny sposób komunikują się obiekty zaimplementowane na potrzeby realizowanego przeze mnie zadania.

Pomiędzy systemem MRROC++, a serwerem obsługującym zadajnik Nintendo Wiimote, przesyłane są dwa typy komunikatów - polecenia dla procesu VSP przesyłane przez procesy ECP, oraz komunikaty zwrotne zawierające dane z zadajnika. Reprezentowane są one przez struktury ECP_VSP_MSG i VSP_ECP_MSG. Po stronie klienta, tj. struktury MRROC++, zdefiniowane zostały one w klasie lib::sensor_interface. Po stronie serwera zaś, tj. dla procesu działającego pod kontrolą systemu operacyjnego Linux, struktury komunikacyjne zdefiniowane są w pliku nagłówkowym *server_structs.h*.

Wspomniane struktury zostały szerzej opisane w rozdziale 3.2.3. Zawierają one dane typu CONFIGURE_DATA oraz SENSOR_IMAGE, które każda implementacja czujnika powinna zdefiniować tak, by umożliwiały przesyłanie danych pomiędzy konkretnym czujnikiem, a procesem ECP.

Struktura ECP_VSP_MSG definiuje komunikaty przesyłane z procesów ECP systemu MRROC++ do serwera. Zawiera ona kod polecenia (i_code) oraz dodatkową strukturę CONFIGURE_DATA, określającą pożądany stan zadajnika, tj. stan diod LED oraz silnika wibracyjnego. Kod polecenia to wartość typu wyliczeniowego VSP_COMMAND. Struktura, zawierająca dane konfiguracyjne, przedstawiona została na listingu 5

```
typedef struct
{
  bool led_change;
  unsigned int led_status;
  bool rumble;
} CONFIGURE_DATA;
```

Listing 5: CONFIGURE_DATA

Zmienna led_status określa pożądany stan diod LED, zapisany na jej najmłodszych czterech bitach. Dowolna kombinacja najmłodszych czterech bitów tej zmiennej jest poprawną konfiguracją diod. Dlatego też do struktury została dodana flaga led_change. Określa ona, czy przesłane dane sterujące określają pożądany stan diod LED, tj. czy diody zadajnika powinny zostać przełączone w stan określony przez najmłodsze cztery bity. Zmienna rumble określa pożądany stan silnika wibracyjnego wbudowanego w zadajnik - ustawiona na *true* powoduje wibrację urządzenia.

Drugi typ komunikatu to informacja zwrota przesyłana z serwera zadajnika do procesu ECP. Zawiera ona status odpowiedzi, tj. stałą typu wyliczeniowego VSP_REPORT (rozdział 3.2.3) oraz stan przycisków i przyspieszenia zmierzone na trzech osiach akcelerometru. Strukturę SENSOR_IMAGE przedstawia listing 6.

```
typedef struct
{
  buttonStatus buttons;
  orientationStatus orientation;
} SENSOR_IMAGE;

struct buttonStatus
{
  //false - not pressed
  //true - pressed
  bool left;
  bool right;
  bool up;
```

```
  bool down;
  bool buttonA;
  bool buttonB;
  bool button1;
  bool button2;
  bool buttonPlus;
  bool buttonMinus;
  bool buttonHome;
};

struct acc
{
  float acc_x;
  float acc_y;
  float acc_z;
};
```

Listing 6: SENSOR_IMAGE

Struktura buttonStatus określa stan przycisków zadajnika. Wartość true oznacza, że przycisk podczas ostatniego odczytu był wciśnięty. Zmienne te ustawiane są przy pierwszym odczycie, w którym stan przycisku został zmieniony na wciśnięty, i pozostają w tym stanie do czasu zwolnienia przycisku. Dlatego też potrzebna jest dodatkowa implementacja mechanizmu, który pozwoli wykrywać sam moment wciśnięcia przycisku - w przeciwnym wypadku funkcje obsługi zdarzeń wciśnięcia przycisku byłyby wołane z każdym pobranym odczytem. Mechanizm ten zaimplementowany jest w metodzie updateButtonsPressed(), która na podstawie aktualnego i poprzedniego stanu przycisków aktualizuje strukturę buttonsPressed, ustawiając flagi oznaczające wciśnięcie poszczególnych przycisków.

Struktura acc przechowuje przyspieszenia odczytane na poszczególnych osiach akcelerometrów. Wyrażone są one w wielokrotności g, tj. przyspieszenia ziemskiego.

5.5 Generatory trajektorii

Proces ECP komunikuje się z procesem VSP i pobiera stan zadajnika Wiimote. Dane te przekazywane są do jednego z trzech generatorów ruchu, które na podstawie wychylenia zadajnika i wciśniętych przycisków poruszają manipulatorem. Na potrzeby zadania powstały trzy generatory ruchu dla systemu MRROC++. Taka mnogość opcji poruszania manipulatorem umożliwiła intuicyjne zadawanie przemieszczenia końcówki generatorem odpowiednim do sytuacji, co pozwoliło uniknąć słabości wybranych reprezentacji pozycji.

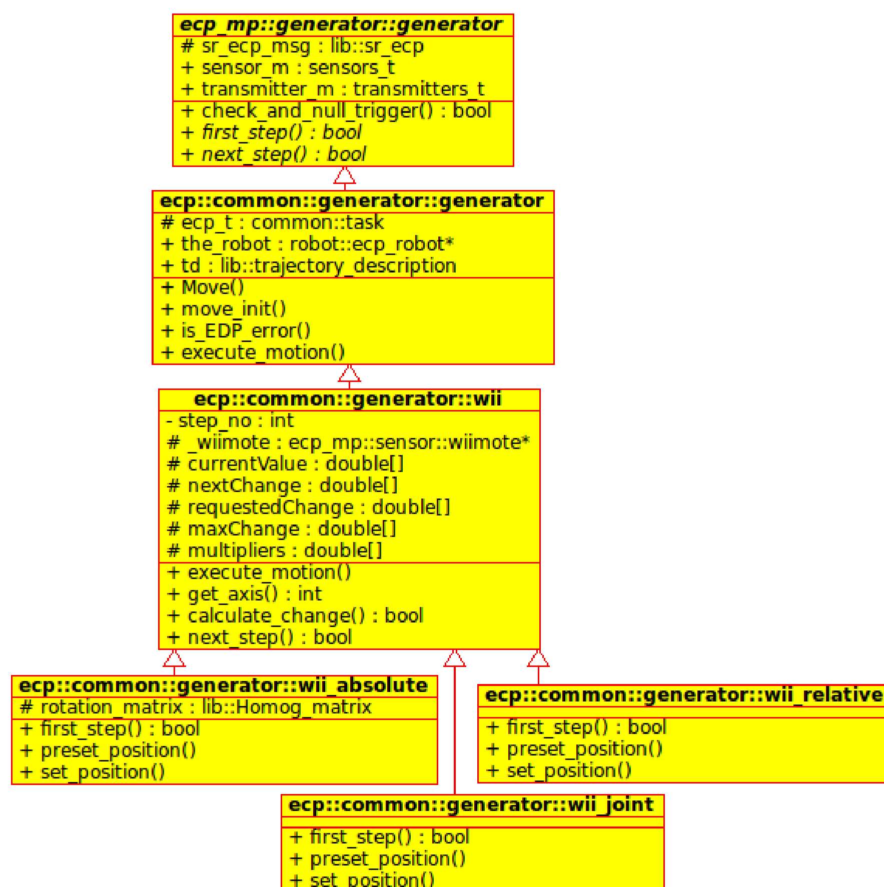
Powstałe w ramach realizacji pracy generatory współdzielą część kodu odpowiedzialną za komunikację z manipulatorem czy jego konfigurację, która została zamknięta w abstrakcyjnej klasie nadrzędnej *wii*. Klasa ta może zostać wykorzystana do zbudowania kolejnych generatorów ruchu opartych o odczyty z Wiimote.

Pierwszym z zaimplementowanych generatorów jest generator ruchu końcówką manipulatora względem układu związanego z końcówką. Cechą charakterystyczną tego modelu ruchu jest to, iż układ odniesienia może się obracać wskutek ruchu manipulatora.

Drugim z generatorów jest generator ruchu końcówką manipulatora względem układu umieszczonego w końcówce o orientacji układu bazowego. W przeciwieństwie do poprzedniego generatora, tutaj układ odniesienia nie obraca się wraz z obrotem końcówki manipulatora.

Kolejny generator to generator ruchu przyrostowego w przestrzeni stawów manipulatora. Generator ten transformuje wychylenie zadajnika na odpowiednią prędkość przyrostu w wybranym stawie robota.

Diagram klas generatorów znajduje się na rysunku 19.



Rysunek 19: Diagram klas generatorów

5.5.1 Generator bazowy

Generator wii Zaimplementowane w ramach pracy generatory trajektorii działające w oparciu o sygnały sterujące z zadajnika Nintendo Wii współdzielą znaczną część logiki. Różnica w ich działaniu sprowadza się do różnych algorytmów wyznaczania kolejnego kroku w trajektorii manipulatora. Wspólne funkcje generatorów zostały zagregowane w klasie *wii*, dzięki czemu udało się uniknąć powielania kodu, a tworzenie kolejnych generatorów pozycyjnych opartych o zadajnik Wiimote sprowadza się do rozszerzenia tej klasy i zaimplementowania jedynie tego, co wyróżnia poszczególne generatory, a więc samego algorytmu generowania trajektorii, tj. funkcji wyznaczającej na podstawie wartości sygnału sterującego i aktualnego położenia końcówki manipulatora następnej pozycji, która ma być osiągnięta w kolejnym makrokroku.

Obiekty klasy *wii*, jako pozbawione możliwości wyznaczania trajektorii, nie mają racji bytu, dlatego też klasa ta jest klasą abstrakcyjną. Zawiera jednak implementację generacji trajektorii w oparciu o sygnały z zadajnika Wiimote, umożliwia konfigurację zadajnika i zapewnia obsługę błędów aplikacji. Obiekty klas rozszerzających klasę *wii* mają dostęp do następujących zmiennych:

int **step_no** numer makrokroku, tj. ilość dotychczasowych wywołań metody *next_step*

ecp_mp::sensor::wiimote* **wiimote** obiekt reprezentujący zadajnik Wiimote widziany jako czujnik wirtualny, udostępniający odczyty z zadajnika oraz umożliwiający jego konfigurację

double* **multipliers** współczynniki w ilości równej ilości stopni swobody manipulatora pobierane z pliku konfiguracyjnego zadania - w zaimplementowanych w tej pracy generatorach używane do skalowania sygnałów sterujących dla poszczególnych osia

double* **maxChange** wartości ograniczeń w ilości równej ilości stopni swobody manipulatora pobierane z pliku konfiguracyjnego zadania - w zaimplementowanych przeze mnie generatorach używane do ograniczania przyrostów wartości prędkości ruchu w poszczególnych osiach manipulatora w pojedynczym makrokroku

double* **desiredChange** tablica aktualnych wartości sygnałów sterujących w ilości równej ilości stopni swobody manipulatora przemnożonych przez współczynniki *multipliers*

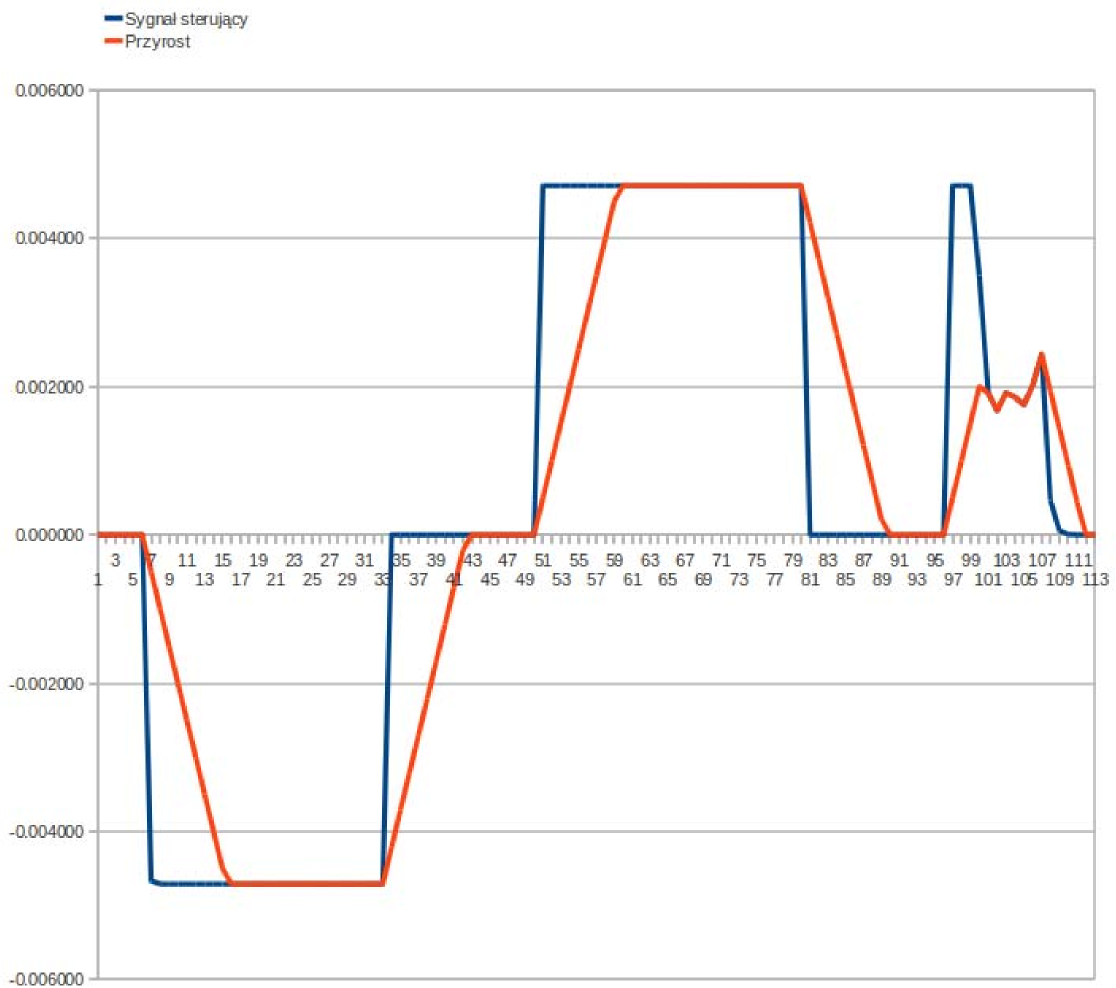
double* **nextChange** wartości sygnałów sterujących w ilości równej ilości stopni swobody manipulatora po uwzględnieniu ograniczeń i współczynników

Metoda	Opis
wii (ecp::common::task::task, ecp_mp::sensor::wiimote*)	konstruktor generatora, jako parametry przyjmujący obiekt zadania i obiekt reprezentujący zadajnik Wiimote jako czujnik wirtualny, inicjalizuje zmienne klasy, m.in. zeruje wymienione wcześniej tablice współczynników i współrzędnych
void execute_motion()	rozszerza odziedziczoną metodę o przechwytywanie błędów w procesie ECP - w momencie osiągnięcia przez manipulator ograniczenia na jednej z osi uruchamia silnik wibracyjny w zadajniku w celu zasygnalizowania operatorowi sytuacji wyjątkowej, a po wyjściu z ograniczenia wyłącza silnik wibracyjny
int get_axis()	na podstawie stanu zadajnika, a w szczególności stanu przycisków, określa i zwraca numer osi manipulatora, w której będzie wykonywany ruch
bool calculate_change(int axis, double value)	wyznacza pożądany przyrost prędkości ruchu w wybranej osi manipulatora przy uwzględnieniu ograniczeń na maksymalny przyrost prędkości w makrokroku
bool next_step()	wykorzystuje pozostałe metody klasy do wyznaczenia osi ruchu (*get_axis()*), wyznaczenia pożądanych przyrostów (*calculate_change()*) i wpisania ich w struktury sterujące manipulatorem (*set_position()*)

Tablica 17: Interfejs klasy wii

Ze względu na istnienie fizycznych ograniczeń przyspieszeń, które bezpiecznie może realizować manipulator, konieczne okazało się wprowadzenie możliwości definiowania limitów przyspieseń przez operatora. Ograniczenia na maksymalne przyrosty prędkości dla każdej z osi manipulatora w pojedynczym makrokroku zapisywane są w pliku konfiguracyjnym zadania. Metoda *calculate_change()* uwzględnia te ograniczenia przy wyznaczanio wartości prędkości dla danego makrokroku, co ilustruje rysunek 20. Przedstawia on wartość prędkości pożądanej wynikającej z wartości sygnału sterującego z zadajnika ruchu oraz wartość prędkości zadaną w kolejnych makrokrokach. Łatwo można zaobserwować wspomniany mechanizm ograniczenia przyspieszenia, gdy różnica między aktualną prędkością a pożądaną jest większa od ograniczenia na przyspieszenie - generator obliczając trajektorie uwzględnia ograniczenie, przez co potrzeba kilku makrokroków, by osiągnąć

pożądaną prędkość.



Rysunek 20: Przyrost prędkości ruchu w zależności od wartości sygnału sterującego

Ograniczenie przyspieszenia na pojedynczej osi ruchu nie jest jedyną modyfikacją sygnału sterującego realizowaną przez metody generatora bazowego. Próba wykorzystania implementowanego rozwiązania na rzeczywistym manipulatorze wykazała, że operator systemu robotycznego w pobliżu punktu docelowego potrzebuje większej precyzji w poruszaniu końcówką manipulatora niż w sytuacji, gdy przemieszcza końcówkę pomiędzy dwoma odległymi węzłami, kiedy z kolei istotne jest możliwie krótki czas ruchu. Dlatego wprowadzona zostałą nieliniowa zależność między wartością sygnału sterującego otrzymywanego z zadajnika, a wartością przekazywaną do metod konkretnych generatorów, na podstawie której generują one kolejny krok. Wprowadzona zależność ma postać:

$$x` = x^3$$
$$y` = y^3 \qquad (5)$$
$$z` = z^3$$

Takie przekształcenie wartości sterującej zapewnia większą precyzję ruchu dla wartości sygnału sterującego bliskich 0, tj. dla niewielkich wychyleń zadajnika. Dla dużych wychyleń przekształcenie nieznacznie tylko zmniejsza wartość sygnału sterującego. Przy odpowiednim doborze mnożników w pliku konfiguracyjnym mechanizm ten zapewnia szybki ruch dla dużych wychyleń zadajnika, zapewniając jednocześnie precyzję ruchu dla niedużych wychyleń.

Klasa *wii* deklaruje również szereg metod abstrakcyjnych, które muszą być zaimplementowane w klasie rzeczywistego generatora.

Metoda	**Opis**
bool first_step()	deklaracja odziedziczona z klasy nadrzędnej *ecp_mp::generator::generator* - metoda ta powinna zainicjalizować komunikację z manipulatorem i wygenerować pierwszy krok
void preset_position()	ta metoda wołana jest w *next_step* przed metodą *calculate_change()* - powinna odpowiednio wypełnić tablice pomocnicze zdefiniowane w klasie *wii*, z których tamta metoda korzysta
void set_position()	na podstawie wartości tablic pomocniczych ta metoda powinna wyznaczyć współrzędne następnego kroku zgodnie z algorytmem konkretnego generatora, a następnie wypełnić odpowiednie struktury sterujące manipulatora

Tablica 18: Metody abstrakcyjne klasy wii

Odpowiednia implementacja powyższych metod wystarcza, by stworzyć generator trajektorii poruszający końcówką manipulatora w wybrany sposób. W ramach pracy zaimplementowałem trzy generatory trajektorii, a algorytmy realizowane przez nie zawierają się właśnie w konkretnych implementacjach wspomnianych metod.

Generowanie kolejnego kroku Utworzenie nowego generatora wymaga od programisty implementacji trzech metod, wymienionych w tabeli 18. Większość logiki generowania

kolejnego kroku zaszyta jest w metodach odziedziczonych z generatora bazowego. Wywołanie przez proces ECP metody Move() generatora realizuje algorytm generowania ruchu, który wygląda następująco:

1. Metoda first_step() konkretnego generatora inicjalizuje ruch.
2. Wywoływana w pętli metoda next_step() generatora bazowego:
   (a) Sprawdza, czy został od operatora systemu otrzymany rozkaz zakończenia ruchu - jeżeli tak, ustawiana jest odpowiednia flaga.
   (b) Pobiera z czujnika wirtualnego wartość sygnału sterującego i wprowadza wspomnianą wcześniej nieliniowość.
   (c) Inicjalizuje tablice pomocnicze za pomocą metody preset_position() konkretnego generatora.
   (d) Określa oś ruchu za pomocą metody get_axis().
   (e) Wyznacza przyrosty prędkości na wszystkich osiach w danym kroku za pomocą metody calculate_change().
   (f) Jeżeli operator zażądał przerwania ruchu sprawdza, czy ruch na poszczególnych osiach został wykonany - jeżeli tak, ruch się kończy, a sterowanie wraca do procesu ECP.
   (g) Przyrosty na poszczególnych osiach wpisywane są do odpowiednich struktur sterujących robotem za pomocą metody set_position() konkretnego generatora.

Warto wspomnieć, że o ile zadawanie ruchu możliwe jest tylko dla pojedynczej osi, to ruch manipulatora może odbywać się jednocześnie w kilku osiach. Powodem tego jest ograniczenie na maksymalne przyspieszenie ruchu na pojedynczej osi. Gdy operator zmienia oś ruchu, pierwotna oś dostaje sygnał sterujący równy 0, a nowa oś sygnał sterujący z zadajnika. Ograniczenie na przyspieszenie i wprowadzana w ten sposób bezwładność uniemożliwia jednak natychmiastowe zatrzymanie ruchu w wybranej osi, więc przez kilka kolejnych kroków prędkość ruchu na osi pierwotnej będzie niezerowa. W tym czasie aktualny sygnał sterujący przekazywany jest na nową oś ruchu, wskutek czego ruch następuje jednocześnie na dwóch osiach. Przy odpowiednio niskich ograniczeniach przyspieszenia opisaną sytuację można rozszerzyć na wszystkie osie ruchu.

5.5.2 Ruch w przestrzeni stawów

Najprostszym generatorem jest generator ruchu w przestrzeni zmiennych konfiguracyjnych robota. Zmiennymi sterującymi przekazywanymi do robota są przyrosty kątów

skręcenia w przypadku par kinematycznych obrotowych lub przyrosty przesunięcia w przypadku par kinematycznych przesuwnych w pojedynczym makrokroku, czyli tak naprawdę prędkości. Ruch odbywa się z uwzględnieniem ograniczeń na maksymalny przyrost prędkości.

Struktura MRROC++ umożliwia sterowanie robotem w tzw. trybie relative, w który zamiast pozycji docelowych zadawane są przyrosty poszczególnych współrzędnych. Upraszcza to znacznie implementację generatora, gdyż sterowanie robotem ogranicza się do przepisania do struktur sterujących robotem przyrostów odebranych od operatora i wykonania instrukcji ruchu.

Pozycja końcówki manipulatora reprezentowana jest jako wektor 7 liczb reprezentujących pozycję w przestrzeni współrzędnych wewnętrznych, tj. przesunięcie na torze jezdnym lub skręcenie poszczególnych stawów. Kolejna wartość sterująca ruchu na pojedynczej osi wyznaczona może być z zależności:

$$x` = x + \Delta x \quad (6)$$

gdzie:

x – aktualna wartość współrzędnej

Δx – wartość sygnału sterującego

$x`$ – wartość współrzędnej zadana w kolejnym kroku

Ponieważ struktura MRROC++ umożliwia zadawanie ruchu w przestrzeni stawów w trybie względnym, takie przekształcenie jest zbędne, a ruch zadawany jest poprzez przepisywanie wartości sygnału sterującego, a więc zadanego przyrostu, do struktur sterujących robotem.

Generator ruchu w przestrzeni zmiennych konfiguracyjnych został zaimplementowany w klasie *wii_joint*. Metoda first_step() tego generatora konfiguruje sposób komunikacji z robotem. Reprezentacja pozycji docelowej ustawiana jest na lib::JOINT, tj. przestrzeń stawów robota, a tryb ruchu na lib::RELATIVE, tj. ruch względny. Metoda set_position(), wołana w metodzie next_step() klasy bazowej, przepisuje wartości z tablicy nextChange do struktur sterujących robotem.

Sterowanie robotem przy pomocy tego generatora nie jest zbyt intuicyjne. Ruch manipulatorem do wybranej pozycji wymaga tak naprawdę od operatora wyznaczenia w głowie przybliżonego rozwiązania odwrotnego zadania kinematyki w celu określenia docelowych położeń stawów manipulatora, a następnie przemieszczenia manipulatora do wyznaczonej pozycji. Kierunki ruchu końcówki są inne w różnych położeniach manipulatora, więc sterowanie robotem tym sposobem wymaga od operatora dużo doświadczenia.

Generator ten ma jednak kilka zastosowań, w których nie sprawdzają się pozostałe generatory. Podstawową jego zaletą jest to, iż ruch w przestrzeni zmiennych konfiguracyjnych pozbawiony jest punktów osobliwych. Pozwala to operatorowi wykorzystać generator *wii_joint* do przejścia przez punkty osobliwe pozostałych generatorów. Kolejną przewagą tego generatora nad innymi jest to, że operator ma wpływ na rzeczywistą konfigurację manipulatora, a nie na samo położenie końcówki robota, tak jak to się dzieje w przypadku generatorów ruchu rozwiązujących odwrotne zadanie kinematyki i wybierających jedną z dopuszczalnych konfiguraci robota. Przydać się to może w przypadku obecności obiektów w przestrzeni roboczej manipulatora, kiedy to zachodzi konieczność wyboru takiej konfiguracji manipulatora, kiedy oprócz osiągnięcia zadanej pozycji przez końcówkę trzeba również unikać kolizji z obiektem.

5.5.3 Ruch w układzie o orientacji końcówki

Kolejny generator umożliwia ruch w układzie końcówki, tj. przesuniecie wzdłuż i obrót wokół aktualnych osi końcówki manipulatora z uwzględnieniem ograniczeń na maksymalne przyrosty prędkości.

Implementacja tego generatora również wykorzystuje tryb relative ruchu, w którym zadawane są przyrosty wartości współrzędnych robota, a nie bezwzględne współrzędne docelowe. Zadawanymi przez operatora przyrostami wypełniany jest wektor współrzędnych kąt-oś, który określa zadane przesunięcie i obrót.

Generator ruchu w orientacji końcówki został zaimplementowany w klasie *wii_relative*. Metoda first_step() tego generatora konfiguruje sposób komunikacji z robotem. Współrzędne zadawania pozycji docelowej ustawiany jest na lib::FRAME, tj. macierz jednorodną, a tryb ruchu na lib::RELATIVE, tj. ruch względny. Metoda set_position(), wołana w metodzie next_step() klasy bazowej, wypełnia wektor współrzędnych kąt-oś wartościami z tablicy nextChange, a następnie wylicza na jego podstawie macierz jednorodną i przepisuje ją do struktur sterujących robota.

5.5.4 Ruch w układzie odniesienia układu bazowego

Najbardziej złożonym generatorem jest generator ruchu w układzie odniesienia umieszczonego w końcówce o orientacji układu bazowego, tj. ruch wzdłuż osi układu bazowego i obrót wokół nich. Przesunięcie początku układu odniesienia (a więc i osi obrotu) do punktu roboczego końcówki manipulatora powoduje jednak, że obrót wokół dowolnej osi nie skutkuje przemieszczeniem punktu roboczego końcówki.

Położenie końcówki w przestrzeni można przedstawić za pomocą wektora P przypiętego w punkcie roboczym końcówki i zorientowanego identycznie z końcówką. Wektor ten można w układzie bazowym B przedstawić za pomocą macierzy jednorodnej P, składają-

cej się z wektora translacji T (rozmiaru 3x1) i macierzy obrotu R układu A związanego z końcówką (rozmiaru 3x3) względem układu bazowego. Szerszy opis zagadnienia można znaleźć w [24].

Wektor translacji opisuje położenie punktu roboczego końcówki, a macierz obrotu orientację końcówki, tj. kąty obrotów wokół osi układu współrzędnych. W takiej reprezentacji przesunięcie wektora w przestrzeni można zdefiniować jako dodanie do macierzy jednorodnej wektora reprezentującego przesunięcie. Obrót zaś definiuje się jako mnożenie przez macierz jednorodną. Złożenie tych dwóch przekształceń pozwala opisać ruch końcówki w przestrzeni zmiennych bazowych.

Na podstawie sygnałów sterujących z zadajnika wyznaczane jest zadane przez operatora przemieszczenie końcówki - zmienne Δx, Δy, Δz - oraz przyrosty kątów obrotu wokół nich - $\Delta\alpha$, $\Delta\beta$, $\Delta\gamma$. Na ich podstawie można wyznaczyć macierz jednorodną reprezentującą zadany ruch.

Wyznaczenie macierzy jednorodnej reprezentującej zadane przez operatora oczekiwane położenie można uzyskać za pomocą wspomnianych wcześniej przekształceń. Wektor translacji to suma aktualnego wektora z wektorem reprezentującym zadane przesunięcie. Macierz obrotu to iloczyn macierzy reprezentującej aktualną orientację końcówki oraz macierzy reprezentującej zadany przez operatora obrót. Złożenie tych macierzy daje macierz jednorodną opisującą zadane przez operatora położenie docelowe.

Generator ruchu w układzie bazowym zaimplementowany został w klasie *wii_absolute*. Metoda first_step() tego generatora konfiguruje sposób komunikacji z robotem. Współrzędne zadawania pozycji docelowej ustawiany jest na lib::FRAME, tj. macierz jednorodną, a tryb ruchu na lib::ABSOLUTE, tj. ruch bezwzględny. Metoda set_position() wołana jest przez metodę next_step() klasy bazowej w każdym makrokroku i zawiera logikę wyznaczającą kolejny zestaw współrzędnych docelowych. W pierwszym kroku metoda ta pobiera ze struktury reprezentującej odpowiedź robota jego aktualne współrzędne i wypełnia nimi macierz jednorodną. Następnie na podstawie sygnału sterującego wyznacza macierz jednorodną reprezentującą zadany ruch. Na tej podstawie, zgodnie z opisanymi wcześniej przekształceniami, wyznacza macierz jednorodną reprezentującą położenie docelowe, która następnie jest przepisywana do struktur sterujących robota.

6 Obsługa zadania

Zadanie generacji trajektorii umożliwia poruszanie końcówką manipulatora jednocześnie na pojedynczej współrzędniej za pomocą kontrolera Wiimote i rejestrowanie wybranych pozycji osiągniętych przez manipulator. Ciąg zapamiętanych pozycji stanowić ma trajektorię dla generatorów pozycyjnych.

6.1 Trajektoria

Trajektoria to sekwencja punktów w przestrzeni roboczej manipulatora, nazywanych dalej węzłami. Każdy węzeł opisany jest zestawem współrzędnych w wybranym układzie odniesienia. Trajektoria posiada węzeł początkowy, węzeł końcowy oraz węzły pośrednie. Każdy węzeł poza końcowym posiada węzeł następny. Każdy węzeł poza początkowym posiada węzeł poprzedni.

Odtwarzanie trajektorii oznacza zrealizowanie przez końcówkę manipulatora ruchu pomiędzy kolejnymi węzłami trajektorii. Ścieżka, jaką przebywa końcówka pomiędzy węzłami jest zależna od generatora ruchu użytego do odtworzenia trajektorii.

Jeżeli trajektoria posiada co najmniej jeden węzeł, to można wyznaczyć tzw. węzeł aktualny trajektorii. W kontekście węzła aktualnego odbywają się wszelkie operacje modyfikujące trajektorię. Rola aktualnego szerzej opisana w części poświęconej trybowi zarządzania kategorią.

6.2 Uruchomienie zadania

Uruchomienie zadania obejmuje włączenie zadajnika, uruchomienie serwera komunikującego się z zadajnikiem oraz uruchomienie zadania *wii_teach* po stronie struktury MRROC++.

Włączenie zadajnika Zadajnik, który nie posiada aktywnego połączenia Bluetooth z innym urządzeniem, automatycznie się wyłącza. Aby go uruchomić, należy wcisnąć jednocześnie przyciski 1 i 2. Spowoduje to włączenie zadajnika na kilka sekund i przejście w tryb rozgłaszania 48-bitowego adres urządzenia, co sygnalizowane jest miganiem diod LED. Jeżeli w tym czasie nie zostanie nawiązane połączenie, zadajnik wyłączy się automatycznie.

Uruchomienie serwera Proces serwera działa pod kontrolą systemu operacyjnego Linux. Uruchamia się go poleceniem:

./proxy <adres zadajnika>

Jedynym parametrem, który należy podać, jest 48-bitowy adres zadajnika Wiimote postaci XX:XX:XX:XX:XX:XX, gdzie X to cyfra w systemie szesnastkowym. Adres ten urządzenie nadaje w trybie rozgłaszania adresu. Aby go poznać, należy użyć dowolnego narzędzia wyświetlającego informacje o dostępnych urządzeniach Bluetooth. Dla systemu Linux jednym z takich narzędzi jest *hcitool*, służące do konfiguracji połączeń Bluetooth. Do wyświetlenia dostępnych urządzeń służy polecenie:

hcitool scan

Jeżeli zadajnik jest w trybie rozgłaszania, a podany adres jest jego adresem, nawiązane zostanie połączenie, diody zadajnika przestaną migać, a serwer rozpocznie proces kalibracji zadajnika. Należy postępować zgodnie z instrukcjami wyświelanymi w konsoli systemowej. Po zakończeniu procedury kalibracji serwer jest gotowy do udostępniania odczytów z zadajnika aplikacjom klienckim.

Uruchomienie zadania *wii_teach* Zadanie uruchamia się z poziomu procesu UI struktury MRROC++. Po wczytaniu pliku konfiguracyjnego (opisanego w rozdziale 6.6) należy uruchomić proces EDP wybranego robota, a następnie proces MP.

Wciśnięcie przycisku START uruchomi zadanie. Użytkownik zostanie poproszony o wybranie pliku, z którego wczytana zostanie trajektoria i do którego będzie zapisywana. Od tego momentu sterowanie zadaniem odbywa się za pomocą zadajnika Wiimote.

6.3 Tryby pracy

Zadanie umożliwia użytkownikowi pracę w dwóch trybach - trybie zadawania ruchu i trybie zarządzania trajektorią. Pierwszy tryb służy do przemieszczania manipulatora do wybranych przez użytkownika położeń. Drugi tryb umożliwia odtwarzanie ruchu robota w dowolnym kierunku oraz modyfikację zapamiętanego ciągu położeń. Po uruchomieniu zadania znajduje się ono w trybie zarządzania kategorią. Miało to na celu zapobieżenie przypadkowym ruchom manipulatorem wskutek braku świadomości faktu, iż robot znajduje się w trybie ruchu. Do przełączenia się między trybami ruchu służy przycisk A.

6.4 Tryb zarządzania trajektorią

Tryb zarządzania kategorią sygnalizowany jest poprzez zapalenie wszystkich czterech diod zadajnika. W tym trybie operator ma możliwość modyfikacji trajektorii poprzez dodawanie, edycję i usuwanie punktów interpolacji. Operator może również trajektorię odtworzyć w dowolnym kierunku. Funkcje poszczególnych przycisków zadajnika przedstawiono poniżej.

lewo przemieszczenie końcówki manipulatora do poprzedniej pozycji trajektorii, jeżeli taka pozycja istnieje. Po wykonaniu ruchu, poprzednia pozycja stanie się węzłem aktualny.

prawo przemieszczenie końcówki manipulatora do następnej pozycji trajektorii, jeżeli taka pozycja istnieje. Po wykonaniu ruchu, następna pozycja stanie się węzłem aktualny.

góra przemieszczenie końcówki manipulatora do ostatniej pozycji trajektorii przechodząc przez wszystkie pośrednie położenia. Po wykonaniu ruchu, ostatnia pozycja stanie się węzłem aktualnym.

dół przemieszczenie końcówki manipulatora do pierwszej pozycji trajektorii przechodząc przez wszystkie pośrednie położenia. Po wykonaniu ruchu, pierwsza pozycja stanie się węzłem aktualnym.

plus dodanie aktualnego położenia końcówki manipulatora do trajektorii - aktualne położenie zostanie dodane po węźle aktualnym, stanie się również nowym węzłem aktualnym

minus usunięcie aktualnego węzła - Aktualnym węzłem staje się węzeł poprzedni, chyba że takiego węzła brak. W takiej sytuacji, jeżeli istnieje węzeł następny, on staje się węzłem aktualny.

home zastąpienie ostatnio osiągniętego położenia z trajektorii aktualnym położeniem, które staje się węzłem aktualnym

A przełączenie zadania w tryb zadawania ruchu

B zapisuje aktualną kategorię do pliku

Do odtwarzania ruchu zastosowany został generator smooth2 autorstwa Rafała Tulwina.

6.5 Tryb zadawania ruchu

Tryb zadawania ruchu sygnalizowany jest poprzez zapalenie jednej z diod zadajnika. Poszczególne diody oznaczają aktualnie wybrany generator ruchu:

1. generator ruchu w bazowym układzie współrzędnych *wii_absolute*

2. generator ruchu w układzie współrzędnych końcówki *wii_relative*

3. generator ruchu w układzie współrzędnych wewnętrznych *wii_joint*

W tym trybie operator przemieszcza końcówkę manipulatora w przestrzeni. Sygnałem sterującym jest kąt obrotu zadajnika wokół osi Y, a punktem zerowym jest położenie poziome zadajnika. Zadanie ruchu wymaga wybrania osi, w której odbywać się będzie ruch, poprzez przytrzymanie odpowiednich przycisków. Do czasu puszczenia przycisków odbywać się będzie ruch na podstawie wartości sygnału sterującego.

Funkcje poszczególnych przycisków zadajnika przedstawiono poniżej.

lewo, prawo, góra, dół, B służą do wyboru aktualnie modyfikowanej współrzędnej. Znaczenie poszczególnych kombinacji tych przycisków opisane jest w tabeli 19.

A przełączenie zadania w tryb zarządzania trajektorią

Przyciski	**wii_joint**	**wii_relative** i **wii_absolute**
lewo	Staw 1	x
góra	Staw 2	y
prawo	Staw 3	z
dół	Staw 4	-
lewo + B	Staw 5	obrót wokół x
góra + B	Staw 6	obrót wokół y
prawo + B	Staw 7	obrót wokół z
dół + B	Staw 8	-

Tablica 19: Wybór osi ruchu

6.6 Konfiguracja zadania

Konfiguracja zadania interaktywnego programowania robotów nie różni się od konfiguracji typowego zadania MRROC++. Niestandardowym elementem konfiguracji jest konieczność zdefiniowania ograniczeń na przyspieszenie na poszczególnych osiach każdego z dostępnych układów współrzędnych, w których wykonywany może być ruch. Dodatkowo zdefiniować należy mnożniki używane do określania zależności między wartością sygnału sterującego otrzymanego z zadajnika, a przyrostem zadanym na daną oś manipulatora.

Przykładowy plik konfiguracyjny przedstawia listing 7.

```
;konfiguracja interfejsu uzytkownika
[ui]
;wlaczenie menu robota irp6_on_track i wylaczenie pozostalych robotow
is_irp6ot_m_active=1
is_irp6p_m_active=0
is_conveyor_active=0
```

```
;wlaczenie menu procesow MP i ECP
is_mp_and_ecps_active=1

;konfiguracja procesu MP
[mp]
;nazwa programu uruchamianego w procesie MP
program_name=mp_c
;nazwa wezla, na ktorym zostanie uruchomiony
node_name=lenin

;konfiguracja procesu ECP robota irp6_on_track
[ecp_irp6ot_m]
;nazwa programu uruchamianego w procesie ECP
program_name=ecp_wii_teach
;nazwa wezla, na ktorym zostanie uruchomiony
node_name=lenin

;ograniczenia predkosci dla generatora wii_absolute
absolute_max_change_0=0.001
;...
absolute_max_change_7=0.001

;mnozniki dla generatora wii_absolute
absolute_multiplier_0=0.003
;...
absolute_multiplier_7=0.003

;ograniczenia predkosci dla generatora wii_joint
joint_max_change_0=0.001
;...
joint_max_change_7=0.001

;mnozniki dla generatora wii_joint
joint_multiplier_0=0.003
;...
joint_multiplier_7=0.001

;ograniczenia predkosci dla generatora wii_relative
relative_max_change_0=0.001
;...
relative_max_change_7=0.001

;mnozniki dla generatora wii_relative
relative_multiplier_0=0.003
;...
```

```
relative_multiplier_7=0.003

;konfiguracja procesu EDP robota irp6_on_track
[edp_irp6ot_m]
;nazwa wezla, na ktorym zostanie uruchomiony
node_name=robot1

;konfiguracja czujnika wirtualnego
[vsp_wiimote]
;adres IP wezla, na ktorym uruchomiony jest serwer Wiimote
wiimote_node_name=192.168.18.51
;numer portu, na ktorym serwer nasluchuje
wiimote_port=40666
```

Listing 7: Przykladowy plik konfiguracyjny zadania

7 Podsumowanie

Celem pracy magisterskiej było opracowanie narzędzia wspomagającego operatora w programowaniu robota usługowego. Istotnym aspektem programowania robotów usługowych jest generowanie trajektorii pozycyjnych, tak jak to miało miejsce w tradycyjnej robotyce przemysłowej. Powstałe w ramach niniejszej pracy rozwiązanie miało wspomóc operatora w procesie uczenia robota trajektorii.

Jednym z elementów rozwiązania miał być zadajnik ruchu. W ramach pracy przeprowadzona została analiza dostępnych na rynku gier zadajników pod kątem spełnienia wymagań stawianych zadajnikowi ruchu systemu robotycznego. Po analizie dostępnych na rynku gier rozwiązań, do realizacji zadania został wybrany bezprzewodowy zadajnik Wiimote firmy Nintendo, wyposażony w trójosiowy akcelerometr. Zaimplementowany został sterownik zadajnika dla struktury programowej MRROC++.

Podjęta zostałą próba wyznaczenia orientacji zadajnika na podstawie rozkładu sił grawitacji na poszczególne osie akcelerometru i wykorzystanie tej informacji do sterowania końcówką robota Irp-6. Odczyty z akcelerometru pozwalają jednak na ustalenie obrotu wokół tylko dwóch z trzech osi, a ustalenie orientacji w przestrzeni wymagałoby bardziej zaawansowanego narzędzia wyposażonego w żyroskop lub trójosiowy kompas. Uzyskana informacja jest jednak wystarczająca do płynnego zadawania ruchu na płaszczyźnie, dzięki czemu zadajnik może znaleźć zastosowanie w sterowaniu robotami mobilnymi. Odczyty z akcelerometru mogłyby również zostać wykorzystane do sterowania manipulatorem lub zadaniem przy pomocy gestów.

Kolejnym krokiem było opracowanie metody zadawania ruchu na podstawie odczytów z zadajnika. Opracowane zostały trzy generatory trajektorii, umożliwiające ruch manipulatorem w dwóch zewnętrznych układach odniesienia oraz w układzie współrzędnych wewnętrznych. Pozwoliły one operatorowi na wygodne poruszanie końcówką manipulatora w przestrzeni roboczej. Dodatkowo generator ruchu w układzie współrzędnych wewnętrznych umożliwił przechodzenie przez punkty osobliwe innych reprezentacji. Dalszy rozwój generatorów ruchu sterowanych sygnałem z zadajnika powinien obejmować implementację generatorów ruchu chwytakiem trójpalczastym, zamontowanym na robocie Irp-6 na stanowisku laboratoryjnym.

Sygnałem sterującym dla ruchu był kąt obrotu zadajnika wokół jednej z osi, co dało operatorowi możliwość płynnej regulacji prędkości ruchu pełnym jej zakresie. Wprowadzona dodatkowa transformacja sygnału pozwoliła na zwiększenie precyzji zadawania ruchu przy małych prędkościach.

Możliwość swobodnego zadawania ruchu robotowi przy pomocy zadajnika została wykorzystana w zaimplementowanym w ramach pracy zadaniu uczenia robota trajektorii. Do sterowania przebiegiem wykonania zadania zostały wykorzystane przyciski zadajnika

ruchu. Umożliwiło to operatorowi robota realizację procedury uczenia przy użyciu samego zadajnika. Ilość wejść zadajnika okazała się być wystarczająca do obsługi zadania uczenia, spełniającego postawione przed nim wymagania funkcjonalne.

Zaimplementowane rozwiązanie umożliwia sygnalizowanie operatorowi sytuacji wyjątkowych poprzez wprowadzenie zadajnika w wibracje. Pozostałe komunikaty systemowe wyświetlane są tylko w graficznej konsoli sterowniczej uruchomionej na komputerze, co czyni je trudno dostępnymi dla operatora poruszającego się wokół przestrzeni roboczej manipulatora. Zadajnik wyposażony jest w głośnik, który możnaby wykorzystać do odtwarzania komunikatów przetworzonych przez syntezator mowy. Nie udało się tego zrealizować w ramach niniejszej pracy ze względu na niepełną na dzień dzisiejszy specyfikację komunikacji z zadajnikiem. Uzyskana metodami inżynierii wstecznej specyfikacja jest jednak cały czas rozwijana, więc w niedalekiej przyszłości implementacja wspomnianego rozwiązania powinna być możliwa.

Wspomniane elementy - sterownik zadajnika, generatory ruchu i zadanie uczenia - złożyły się na rozwiązanie wspomagające programistę w procesie programowania robota i zwiększające jego efektywność.

Literatura

[1] R. Tulwin. Trajectory generation in MRROC++ applications. Master's thesis, WEiTI, 2010.

[2] G. Kost. *Programowanie robotów przemysłowych*. Wydawnictwo Politechniki Śląskiej, 2000.

[3] C. Zieliński and W. Szynkiewicz. MRROC++ system for irp-6 robot. Technical Report nr 99-20, IAIS, May Warsaw, 1999. System MRROC++ dla robota IRp-6 na torze jezdnym.

[4] KUKA Robot Group. *KUKA.Sim - Quick Guide*. KUKA Robot Group, 2007.

[5] Y.J. Kanayama and C.T. Wu. It's time to make mobile robots programmable. *IEEE International Conference on Robotics and Automation*, 2000.

[6] X. Dai, G. Hager, and J. Peterson. Specifying behavior in C++. *IEEE International Conference on Robotics and Automation*, 2002.

[7] D.S. Blank, D. Kumar, L. Meeden, and H. Yanco. Pyro - python robotics. http://www.pyrorobotics.org/, 2010.

[8] M. Kisiel. Język opisu i realizacja zadań przez automat skończony w systemie MRROC++ (In Polish). Master's thesis, WEiTI, Warsaw, 2008.

[9] B. MacDonald, D. Yuen, S. Wong, E. Woo, and R. Gronlund. Robot programming environments.

[10] KUKA Robot Group. *KR C2 sr - Specification*. KUKA Robot Group, 2007.

[11] M. Hwan Choi and S. Joo Kim. The force/moment direction sensor and it's use for human-robot interface in robot teaching. *IEEE International Conference on Robotics and Automation*, 2000.

[12] M. Guillemot Y. Xu and T. Nishida. An experiment study of gesture-based human-robot interface. *IEEE/ICME International Conference on Complex Medical Engineering*, 2007.

[13] Pedro Neto and J. Norberto Pires. Accelerometer-based control of an industrial robotic arm. *The 18th IEEE International Symposium on Robot and Human Interactive Communication*, 2009.

[14] G. Kost and J. Świder. *Programowanie robotów on-line.* Wydawnictwo Politechniki Śląskiej, 2008.

[15] C. Zieliński, W. Szynkiewicz, and T. Zielińska. System MRROC++. Technical report, IAiIS, may 1999.

[16] C. Zieliński. Implementation of control systems for autonomous robots. *6th International Conference on Control, Automation, Robotics and Vision*, 2000.

[17] C. Zieliński, W. Szynkiewicz, and T. Winiarski. Applications of MRROC++ robot programming framework. *Fifth International Workshop on Robot Motion and Control*, 2005.

[18] C. Zieliński, W. Szynkiewicz, T. Winiarski, M. Staniak, W. Czajewski, and T. Kornuta. Rubik's cube as a benchmark validating MRROC++ as an implementation tool for service robot control systems. *Industrial Robot: An International Journal*, 34(5):368–375, 2007.

[19] Tomasz Bem. Robot playing checkers. Master's thesis, WEiTI, Warsaw, 2009.

[20] J. Kuryło. Graficzna konsola sterownicza systemu MRROC++ stworzona w oparciu o platformę Java (In Polish). Master's thesis, WEiTI, Warsaw, 2008.

[21] M.Żbikowski. Graficzne środowisko symulacyjne systemu wielomanipulatorowego (In Polish). Master's thesis, WEiTI, Warsaw, 2009.

[22] M. Casamassina. Revolution's Horsepower. *Imagine Games Network*, 2006.

[23] H. Wisniowski. Analog Devices And Nintendo Collaboration Drives Video Game Innovation With iMEMS Motion Signal Processing Technology. *Analog Devices, Inc.*, 2006.

[24] J. Craig. *Wprowadzenie do robotyki - mechanika i sterowanie.* WNT, 1993.