POLITECHNIKA WARSZAWSKA

Wydział Elektroniki i Technik Informacyjnych

Instytut Systemów Elektronicznych

Maciej Gąbka

nr albumu: 198404

Praca dyplomowa magisterska

Sterowanie robotem - zawodnikiem rozgrywek RoboCup

Praca wykonana pod kierunkiem

prof. dr hab. Jarosława Arabasa

Warszawa 2011

Tytuł: Sterowanie robotem - zawodnikiem rozgrywek RoboCup

Celem pracy było przygotowanie środowiska symulacyjnego odpowiadającego realiom ligi małych robotów w rozgrywkach *RoboCup* oraz opracowanie algorytmu nawigacji do wyznaczania bezkolizyjnej ścieżki w silnie dynamicznym środowisku. W pracy zaprezentowane zostały opracowane modele robotów oraz wyjaśnione zasady ich tworzenia. Zamieszczono także przegląd algorytmów unikania kolizji. W dalszej części pracy skupiono się na opracowaniu algorytmu sterującego zawodnikami. Zaimplementowano algorytmy pozwalające na wykonywanie robotowi prostych zachowań, niezbędnych do gry w piłkę nożną, takich jak: podawanie piłki, wychodzenie na pozycję do strzału, strzelanie na bramkę, jazda do zadanego celu. Koordynację działań zawodników osiągnięto wzorując się częściowo na architekturze `STP` szerzej opisanej w pracy.

Title: Simulation environment and collision avoidance algorithm for football playing mobile robot

The aim of this thesis was to develop simulation environment capable of modelling RoboCup League. Second goal was to implement a collision avoidance algorithm and to use the simulation environment to test it in the RoboCup match.

The Gazebo simulator, with it's basic features described, was chosen to design RoboCup environment. After that, the created models (ball, field and robots) were presented.

Furthermore, the survey of collision avoidance methods was made. For this thesis purposes, Curvature Velocity Method (CVM) was selected as the solution to be implemented. Afther that, a set of tests and modifications were made to increase method's performance in the dynamic RoboCup League environment.

Spis treści

1 Wstęp **5**

1.1 Zakres pracy 5

1.2 Cel pracy 8

2 Liga RoboCup **10**

2.1 Opis projektu RoboCup 10

2.2 Szczegółowe omówienie ligi Small-size (F180) 12

2.2.1 Zasady 12

2.2.2 Schemat komunikacji 14

2.3 Budowa robota w *Small-size League* 15

3 Player/Stage/Gazebo **17**

3.1 Koncepcja 17

3.2 Architektura symulatora 18

3.2.1 Gazebo 19

3.3 Modelowanie obiektów w Gazebo 20

3.3.1 Zasady modelowania w Gazebo 0.10 20

3.3.2 Realizacja środowiska Ligi RoboCup 26

4 Sterowanie modelem robota z *Small-size League* **30**

4.1 Omówienie omnikierunkowej bazy jezdnej 30

4.1.1 Opis położenia kół 31

4.1.2 Opis kinematyki oraz dynamiki bazy 32

4.2 Obliczanie profilu prędkości liniowej robota 33

4.3 Dryblowanie z piłką 35

5 Algorytmy unikania kolizji **38**
5.1 Krótki przegląd algorytmów unikania kolizji 38
5.1.1 Algorytm Bug 38
5.1.2 Algorytm VHF 40
5.1.3 Technika dynamicznego okna 42
5.1.4 Algorytmy pól potencjałowych 42
5.1.5 Algorytm CVM (`Curvature Velocity Method`) 45
5.2 Zasada działania CVM 46
5.3 Algorytm RRT (`Rapidly-Exploring Random Tree`) 52
5.3.1 Modyfikacje algorytmu, czyli przejście z `RRT` do `ERRT` 54
5.4 Zalety algorytmu RRT w stosunku do CVM 56

6 Testy zaimplementowanej wersji algorytmu `RRT` **57**
6.1 Opis implementacji algorytmu `RRT` 57
6.1.1 Parametry zastosowanego algorytmu 61
6.2 Aplikacja analizująca działanie algorytmu 61
6.3 Opis środowisk testowych 62
6.4 Wyniki eksperymentów w środowisku statycznym 64
6.5 Wyniki eksperymentów środowisku dynamicznym 68

7 Realizacja złożonych funkcji przez robota **73**
7.1 Opis algorytmu sterujacego zawodnikiem 74
7.1.1 Omówienie warstwy `Play` 76
7.1.2 Omówienie warstwy `Tactics` 77
7.1.3 Realizacja `Skills` 78
7.1.4 Opis zrealizowanej wersji algorytmu 79
7.2 Testy warstw `tactics` oraz `skills` 80
7.3 Nawigacja w dynamicznym środowisku 80
7.3.1 Wyniki testu nawigacji 81
7.4 Strzelanie po podaniu 83
7.4.1 Wyniki testu 83

8 Podsumowanie **84**

A Zawartość płyty CD **85**

B Instrukcja instalacji Gazebo **86**

C Szczegóły eksperymentów **87**

Rozdział 1

Wstęp

Szybki postęp technologiczny zapoczątkowany w XX wieku umożliwił rozwój nowych dziedzin nauki, między innymi robotyki. Obecnie maszyny mogą zastępować ludzi przy wykonywaniu wielu rozmaitych czynności. Innowacyjne rozwiązania stwarzają jednak potrzebę opracowania wymagającego środowiska testowego. Jedną z takich inicjatyw, mających na celu wspieranie rozwoju i popularyzację robotyki są rozgrywki RoboCup. W obrębie przedsięwzięcia wyróżnionych jest kilka lig, różniących się zasadami oraz typem robotów biorących udział w zawodach. Głównym celem przyświecającym organizatorom jest stworzenie do 2050 roku drużyny robotów zdolnej wygrać z ówczesnymi mistrzami świata. Liga RoboCup budzi zainteresowanie wielu ludzi na całym świecie, prace z nią związane są prowadzone w wielu środowiskach akademickich na całym świecie. W Polsce nie została jeszcze stworzona drużyna, która wystartowałaby w tych rozgrywkach. Rozwiązanie problemu stawia przed uczestnikami wyzwania konstrukcyjne oraz algorytmiczne. Niniejsza praca skupiona jest głównie na rozwiązaniach algorytmicznych. Omówiony zostanie problem sterowania zawodnikiem, wyznaczania bezkolizyjnej ścieżki do celu oraz realizacji prostych zachowań niezbędnych podczas rozgrywek.

1.1 Zakres pracy

Niniejsza praca jest niejako kontynuacją działań podjętych podczas realizacji pracy inżynierskiej. Tematyka w niej poruszona dotyczy na wstępie samych rozgrywek RoboCup. Jak już napisano we wstępie przedsięwzięcie dotyczy głównie rozgrywek piłkarskich, w których udział biorą roboty. Jednak ze względu na dynamiczny roz-

wój samej robotyki do projektu dołączane są kolejne tematy. Obecnie podział jest następujący:

- *RoboCupSoccer*, czyli rozgrywki piłkarskie autonomicznych robotów,
- *RoboCupRescue*, obejmujący działania robotów w sytuacjach kryzysowych, czy niebezpiecznych dla ludzi,
- *RoboCup@Home*, skupiający się na robotyce mającej na celu pomaganie człowiekowi w codziennych czynnościach,
- *RoboCupJunior* popularyzujący robotykę wśród młodzieży.

Same rozgrywki piłkarskie *RoboCupSoccer* także są podzielone na kilka lig, tutaj głównym kryterium jest konstrukcja mechaniczna zawodników. Więcej informacji na ten temat zostanie zaprezentowanych w rozdziale 2. W ramach samej pracy skupiono się na lidze małych robotów (*Small-size League*), ponieważ już wcześniej prowadzone były prace w tym kierunku na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej.

W dalszej części pracy opisano nowe testowe środowisko symulacyjne oraz wprowadzone poprawki modyfikujące pracę symulatora. Z racji, iż wcześniejsze prace prowadzono na symulatorze *Player/Stage/Gazebo* zdecydowano się na dalsze prace na tej platformie. Opracowane zostały nowe modele zawodników, wzorowane na rzeczywistych, holonomicznych biorących udział w rozgrywkach. Opracowano sterownik (element symulatora) umożliwiający: kontrolę nad prędkością robota (wyposażono go dodatkowo w regulator PID), prowadzenie piłki jak i oddawanie strzału. W stosunku do poprzednich prac zdecydowano się także na implementację oraz poddanie testom nowego algorytmu nawigacji. Rozwiązanie to zostało przetestowane w takich samych warunkach jak poprzednie. Dokonano także analizy i porównania otrzymanych wyników. Zaprezentowano także inne, stosowane w robotyce metody unikania kolizji. Jeden z nich także został zaimplementowany, w celu rozwiązywania mniej złożonych zadań, takich jak odjechanie od przeszkody w momencie wystąpienia kolizji.

W pracy zostało przedstawione także jedno z podejść, powszechnie stosowane do koordynacji i planowania działań zawodników (`STP - Skill Tactics Play` [11]). Zakłada ono planowanie zachowań drużyny na 3 poziomach. Każdy odnosi się do innej warstwy abstrakcji.

1. Poziomem najwyżej w hierarchii jest `Play`. Przez to pojęcie rozumiany jest plan gry dla całej drużyny, uwzględniana jest tutaj koordynacja poczynań pomiędzy zawodnikami. Plan zakłada przydział zawodnika do określonej roli. W oryginalnym rozwiązaniu role przydzielane są dynamicznie w zależności od sytuacji na boisku. W obrębie danego planu zawodnik wykonuje swoją rolę (sekwencję kilku `Tactics`), aż do momentu zakończenia danego planu lub wyznaczenia kolejnego.

2. Przez `Tactics` rozumiany jest plan działań dla jednej roli. W taktyce zamknięte jest wykonywanie przez robota pojedynczej złożonej akcji. Przykładem może być strzał na bramkę. Robot dostaje polecenie oddania strzału na bramkę, zatem plan jego poczynań ma doprowadzić do sytuacji, w której osiągnie on pozycję umożliwiającą oddanie strzału na bramkę z zadanym powodzeniem. Plan na szczeblu pojedynczego robota jest wykonywany do momentu zmiany planu gry całej drużyny (ponownego przydziału robotów do ról). Przykładowe `Tactics`:

   - strzał na bramkę,
   - podanie piłki,
   - odebranie podania,
   - blokowanie innego robota,
   - wyjście na pozycję,
   - bronienie pozycji,
   - dryblowanie z piłką.

   Taktyka determinuje skończony automat stanów, w którym elementami są `Skills`. Wykonanie bieżącej taktyki sprowadza się do przechodzenia pomiędzy kolejnymi zachowaniami robota w obrębie danego automatu.

3. Najniżej w hierarchii znajduje się pojęcie `Skills` odnosi się ono do konkretnych umiejętności robota, takich jak:

   - przechwycenie piłki,
   - oddanie strzału,

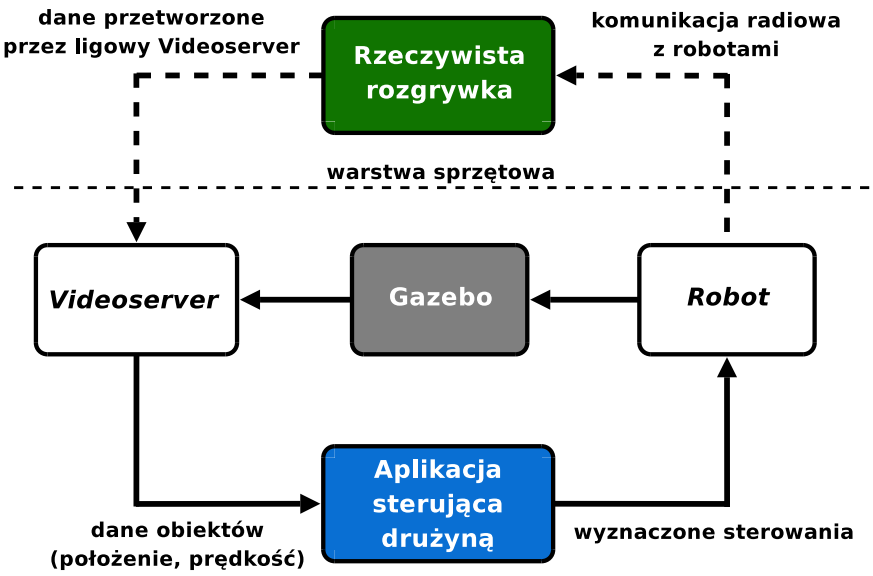
- doprowadzenie piłki do celu,
- przemieszczenie robota do celu,
- podążanie za innym robotem.

Na tym szczeblu nie występuje koordynacja w obrębie drużyny. Zachowanie robota zmieniane jest w każdym kroku gry. Z każdego `Skill`, w każdym momencie określone musi być przejście albo do nowego zadania bądź kontynuowanie tego samego zadania. Przykładowo jeśli zlecimy robotowi przemieszczenie z piłką do celu i piłka odskoczy robotowi, to powinien do niej podjechać, przechwycić ją, a następnie ponownie prowadzić do zadanego celu (należy jednak cały czas pamiętać, że jeśli nastąpi dobra okazja do strzału na bramkę to należy z niej skorzysta.

W ramach pracy w pełni zrealizowane zostały dwie warstwy z oryginalnej hierarchii opisanej powyżej (`Skills` oraz `Tactics`). Aplikację projektowano w taki sposób, aby pozostawić miejsce na implementację warstwy `Play`, której nie dokończono. Użyteczność algorytmu sprawdzono podczas przeprowadzonych eksperymentów, wzorowanych na rzeczywistych zadaniach eliminacyjnych stosowanych w rozgrywkach RoboCup w ostatnich latach.

1.2 Cel pracy

Za główny cel niniejszej pracy postawiono przetestowanie użyteczności i skuteczności architektury `STP`. Aby jednak zrealizować to zadanie konieczne było stworzenie środowiska symulacyjnego, umożliwiającego modelowanie rozgrywki *Small-size League*. Przy opracowywaniu modelu środowiska starano się w jak największym stopniu odzwierciedlić realia ligi *Small-size League*. Po drugie środowisko symulacyjne miało być na tyle elastyczne, aby w przyszłości umożliwiało testowanie różnorodnych rozwiązań sterowania drużyną. Zachowano schemat przepływu informacji z pracy inżynierskiej. Został on zamieszczony na rysunku 1.1. Stosowany w rozgrywkach *Small-size League* `videoserwer` został ujęty w osobną warstwę aplikacji, komunikująca się bezpośrednio z symulatorem. Osobną warstwę stanowi także część aplikacji odpowiedzialna za sterowanie robotem.



Rysunek 1.1: Komunikacja pomiędzy warstwami aplikacji.

Główną motywacją takiej architektury było umożliwienie prostego przystosowania aplikacji do sterowania rzeczywistym robotem pobierającym dane z zewnętrznego serwera. Ponieważ w rozgrywkach biorą udział roboty holonomiczne, zdecydowano się na odejście od modelu robota o napędzie różnicowym stosowanego w pracy inżynierskiej (wzorowanego na *HMT*) i opracowanie nowego wzorowanego na rzeczywistych zawodnikach. Kolejnym celem pracy wynikającym bezpośrednio ze zmiany modelu zawodnika, było zaimplementowanie i przetestowanie algorytmu nawigacji robota w dynamicznym środowisku. Zdecydowano się na algorytm `RRT`, postanowiono także porównać jego skuteczność z wcześniej stosowanym `CVM`.

Rozdział 2

Liga RoboCup

2.1 Opis projektu RoboCup

Projekt RoboCup, jego idea jak i historia szerzej zostały opisane w pracy inżynierskiej [15], więcej informacji na temat mistrzostw można także znaleźć na oficjalnej stronie projektu `http://www.robocup.org`. W niniejszej pracy problematyka rozgrywek robotów w piłkę nożną zostanie przedstawiona jedynie skrótowo ze szczególnym uwzględnieniem budowy robota wykorzystywanego w lidze na której wzorowano się podczas prac. Jak już przytoczono we wstępie, głównym celem przedsięwzięcia jest stworzenie do 2050 roku drużyny w pełni autonomicznych robotów humanoidalnych zdolnych wygrać rozgrywkę z aktualnymi mistrzami świata. Obecnie projekt został rozszerzony o nowe obszary zastosowań robotów, tematyka została pogrupowana następująco:

- *RoboCupSoccer*, czyli rozgrywki piłkarskie autonomicznych robotów,
- *RoboCupRescue*, obejmujący działania robotów w sytuacjach kryzysowych, czy niebezpiecznych dla ludzi,
- *RoboCup@Home*, skupiający się na robotyce mającej na celu pomaganie człowiekowi w codziennych czynnościach,
- *RoboCupJunior* popularyzujący robotykę wśród młodzieży.

W ramach niniejszej pracy skupiono się na *RoboCupSoccer*. Kluczową rolę w realizacji postawionego zadania pełni konstrukcja zarówno mechaniczna jak i elektroniczna

robota. Zawodnik powinien być wyposażony w odpowiedni zestaw czujników umożliwiających osiągnięcie pełnej autonomiczności. Osobnym problemem jest opracowanie funkcjonalnego oprogramowania umożliwiającego koordynację działań wielu robotów. Projekt jest realizowany nieprzerwanie od września 1993 roku, a mistrzostwa odbywają się regularnie w różnych miejscach na świecie. Rozgrywki toczone są w kilku niezależnych od siebie ligach. Aktualnie wyróżnione zostały następujące ligi:

- liga symulacyjna
- *Small-size League*
- *Middle-size League*
- *Standard Platform League*
- *Humanoid League*

Liga symulacyjna jest pewnego rodzaju grą, w której uczestniczące drużyny implementują program decydujący o zachowaniu zawodników. Jest ona najstarszą z lig, towarzyszy przedsięwzięciu od samego początku jego istnienia. Zachowanie robotów jest symulowane za pomocą programu zwanego *RoboCup Soccer Simulator*.

Kolejną z lig jest *Small-Size League*. W rozgrywkach tej ligi drużyna składa się maksymalnie z pięciu niewielkich robotów, takich jak widoczne na fotografii 2.1. Roboty nie są jednak w pełni autonomiczne, ponieważ nie posiadają własnych sensorów wizyjnych. Algorytm sterujący czerpie informację o położeniu piłki oraz robotów globalnego systemu wizyjnego *SSL-Vision* składającego się z szeregu kamer umieszczonych nad boiskiem. Każde z boisk na którym toczone są rozgrywki jest wyposażone w ten system. Drużyna odbiera już przetworzone przez system informacje o położeniu i prędkościach robotów oraz piłki. Lidze tej został poświęcony w całości paragraf 2.2.

Middle-size League to rozgrywki w pełni autonomicznych robotów. W przeciwieństwie do poprzedniej ligi, globalny system wizyjny jest całkowicie zakazany. Każdy robot jest wyposażony w osobny zestaw czujników wizyjnych. Zawodnicy niezależnie muszą reagować na zmieniającą się sytuację na planszy oraz wspólnie dążyć do wypracowania wspólnej strategii w zależności od zachowań innych graczy.

W projekcie wyróżnione są także ligi *Standard Platform League* , czyli rozgrywki piesków *Aibo* konstruowanych przez firmę *Sony* oraz liga robotów humanoidalnych.

Rysunek 2.1: Roboty biorące udział w *Small-Size League*
(źródło: `www.robocup.org`)

W tej ostatniej biorę udział roboty przypominające swoją budową ludzi czyli posiadać korpus, nogi, ręce oraz głowę.

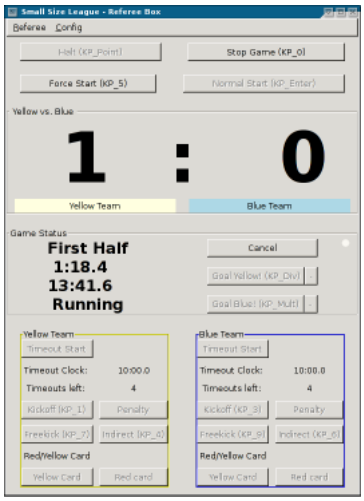
2.2 Szczegółowe omówienie ligi Small-size (F180)

2.2.1 Zasady

Corocznie przed nowymi mistrzostwami wydawany jest zmodyfikowany regulamin, który będzie na nich obowiązywać. Zmiany nie są zazwyczaj daleko idące, modyfikacje dotyczą przykładowo nowego rozmiaru boiska. W poprzednich latach dopuszczono możliwość stosowania przez drużyny własnego globalnego systemu udostępniającego informację o położeniu i prędkości zawodników. Jednak od 2010 roku sprawę systemu wizyjnego usystematyzowano, stworzono system o nazwie *SSL-Vision*, który jest obowiązkowym w lidze. Całkowity rozmiar boiska, wliczając w to pola autowe jest równy 7.4 [m] na 5.4 [m], natomiast sama plansza na której toczona jest rozgrywka ma wymiary 6.05 [m] na 4.05 [m]. Powierzchnia jest równa i wyłożona zielonym dywanem lub wykładziną. Podczas meczu wykorzystywana jest standardowa piłka do golfa koloru pomarańczowego. Jej promień jest równy 43 [mm], a masa 46 [g]. Uczestniczące drużyny mogą składać się z maksymalnie pięciu robotów, jeden z nich może zostać oddelegowany do pełnienia funkcji bramkarza, jednak powinno to zostać zgłoszone przed rozpoczęciem meczu. W trakcie rozgrywki roboty mogą być wymieniane na nowe, tak samo jak w rzeczywistym meczu, jednak sytuacja taka musi być zgłoszona wcześniej arbitrowi. Konstrukcja mechaniczna robota nie jest mocno ograniczona, głównym zalecenie dotyczy wymiarów. Robot powinien zmieścić się w walcu

o średnicy 18 [cm] oraz wysokości 15 [cm]. Dodatkowo może on być wyposażony w urządzenie do prowadzenia piłki. Tutaj jednak istnieją pewne ograniczenia dotyczące samej budowy oraz stosowania w czasie gry. Zawodnik nie może pokonać z piłką odległości większej niż 50 [cm]. Po przejechaniu takiego dystansu powinien przestać powinien zdecydować się na podanie jej innemu zawodnikowi, oddanie strzału na bramkę lub kopnąć przed siebie i dalej z nia dryblować. Niedostosowanie się do tych zasad powoduje sygnalizowanie przez arbitra przewinienia i wykonanie rzutu wolnego przez drużynę przeciwną. Konstrukcja urządzenia do dryblowania nie powinna także uniemożliwiać kontaktu z piłką zawodnikowi z drużyny przeciwnej.

Rozgrywka jest całkowicie kontrolowana przez arbitra, który czuwa nad tym, aby regulamin był przestrzegany. Do jego zadań należy sygnalizowanie przewinień, zdobytych bramek oraz innych typowych sytuacji na boisku. Sędzia ma prawo zmienić swoją decyzje po konsultacjach z asystentem. Arbiter komunikuje się z zawodnikami za pomocą programu, którego interfejs użytkownika zamieszczono na rysunku 2.2 .

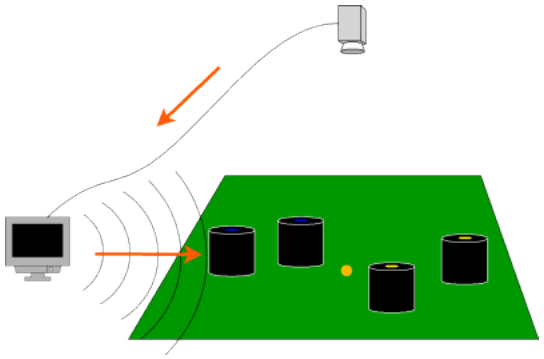


Rysunek 2.2: Program wykorzystywany przez sędziego w *Small Size League* (źródło: `www.robocup.org`)

Wszystkie akcje sędziego wprowadzone do tego programu są udostępniane specjalnym protokołem zawodnikom, dzięki temu rozgrywka nie wymaga ludzkiej ingerencji.

2.2.2 Schemat komunikacji

Kluczową sprawą podczas rzeczywistej rozgrywki jest dostęp do informacji o położeniu piłki i pozostałych robotów. Jak już wspomniano wcześniej organizacja udostępnia drużynom system *SSL-Vision*, z którego drużyny są zobowiązane pobierać te informacje. Uproszczony schemat systemu wizyjnego widoczny jest na rysunku 2.3. Składa się on z kamery ustawionej centralnie nad boiskiem oraz specjalnej aplikacji *SSL-Vision*[1]. Obecnie w skład systemu może wchodzić kilka kamer. Obraz zarejestrowany przez kamery jest przesyłany do komputera, gdzie jest przetwarzany przez *SSL-Vision* w czasie rzeczywistym. Program dostarcza informację o położeniu oraz prędkościach robotów i piłki. Dane te następnie są wykorzystywane przez rywalizujące ze sobą drużyny na potrzeby ich algorytmów sterujących zawodnikami. Algorytm sterujący jako wynik swojego działania powinien zwracać kierunek i prędkość zawodników. Ta ostateczna informacja jest wysyłana droga radiową do zawodnika.



Rysunek 2.3: Schemat komunikacji w *Small Size League*
(źródło: `www.robocup.org`)

SSL-Vision może rozpoznawać nie tylko położenie poszczególnych robotów, ale także ich orientację na płaszczyźnie. Jednak do tego celu niezbędne jest zastosowanie specjalnych znaczników. Każdej z drużyn przed rozpoczęciem rozgrywki zostaje przypisana para kolorów, jakie zawiera znacznik. Dzięki zastosowaniu dwóch kolorów możliwe jest nie tylko rozróżnianie robotów z różnych drużyn, ale także określanie ich orientacji.

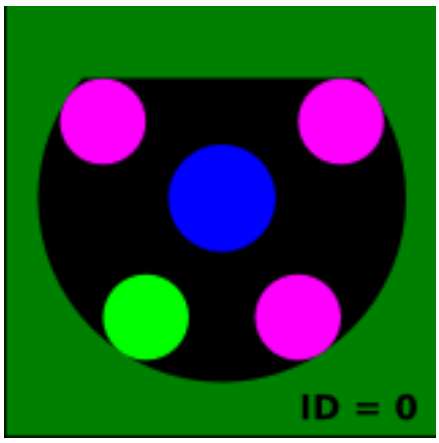
[1]Do pobrania z `http://code.google.com/p/ssl-vision`

(a) (b) (c)

Rysunek 2.4: Popularny model robota wykorzystywany w lidze *F180* (źródło: `www.robocup.org`)

2.3 Budowa robota w *Small-size League*

Oficjalne zasady ligi nie zobowiązują uczestników do używania konkretnych modeli robotów. Określone są jedynie rygory dotyczące maksymalnych wymiarów robota, oraz sposoby prowadzenia piłki. Obserwując kolejne mistrzostwa, łatwo zauważyć, że wśród zgłaszanych drużyn dominuje jedna konstrukcja mechaniczna. Została ona zaprezentowana na rysunkach 2.4. Podczas gry w piłkę nożną często wynika potrzeba zmiany orientacji w miejscu. W prezentowanym rozwiązaniu zdecydowano się na omnikierunkową bazę jezdną. Składa się ona z trzech kół szwedzkich, w tym dwóch niezależnie napędzanych. Koło szwedzkie posiada taką zaletę, iż dodatkowo poza obrotem wokół własnej osi umożliwia obrót wokół punktu styczności koła z podłożem oraz wokół osi rolek umieszczonych na kole. Dzięki zastosowaniu takiego rozwiązania uzyskano w pełni holonomiczną budowę robota. Robot biorący udział w rozgrywkach musi być zdolny do prowadzenia piłki. Zastosowana konstrukcja jest wyposażona w urządzenie do dryblowania widoczne na rysunkach 2.4a oraz 2.4c. Zbudowane jest ono z walca nadającego piłce wsteczną rotację, przez co nie odbija się ona od robota, a także nie traci on nad nią kontroli w momencie hamowania lub obracania się.



Rysunek 2.5: Znacznik umożliwiający systemowi wizyjnemu identyfikację robotów (źródło: `www.robocup.org`)

W regulaminie rozgrywek dopuszczono do stosowania jedynie urządzenia do dryblowania

działające na piłkę siłą prostopadłą do podłoża rys. 2.4a (we wcześniejszych latach w użyciu były urządzenia, w których obracany walec był umieszczony pionowo).

Ostatnim ważnym elementem, w który musi być wyposażony robot, jest znacznik (rys. 2.5). Znajduje się on w takim miejscu, aby kamera umieszczona centralnie nad boiskiem mogła go zarejestrować (przykrywa robota od góry). Znaczniki umożliwiają systemowi wizyjnemu określenie, do której drużyny należy dany robot, a także poprawne rozpoznanie jego pozycji, orientacji oraz prędkości na boisku.

Rozdział 3

Player/Stage/Gazebo

Symulator `Player/Stage/Gazebo` został już opisany w pracy inżynierskiej [15] jednak z uwagi na jego ważną rolę i zmiany jakie zdecydowano się wprowadzić w symulowanym środowisku podczas rozwijania aplikacji zdecydowano się na krótkie przypomnienie koncepcji projektu.

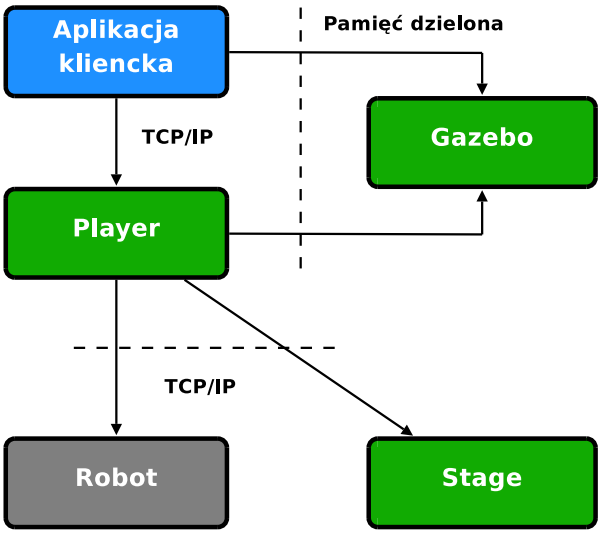
3.1 Koncepcja

Oprogramowanie składa się z trzech niezależnych aplikacji *Player*, *Stage* oraz *Gazebo*. *Gazebo* oraz *Stage* są środowiskami symulacyjnymi. Z tą różnicą, że *Stage* jest środowiskiem dwuwymiarowym, dedykowanym do symulowanie dużych populacji robotów mobilnych. Natomiast *Gazebo* zapewnia pełną trójwymiarową symulację, uwzględniając również oddziaływania fizyczne pomiędzy stosowanymi obiektami. W trakcie realizacji niniejszej pracy wydana została pierwsza stabilna wersja oprogramowanie dostępna pod adresem `http://gazebosim.org`. Wcześniej wszelkie dane dotyczące wszystkich aplikacji wchodzących w skład projektu dostępne były na stronie `http://playerstage.sourceforge.net`. Z uwagi na trudności związane z adaptacją symulatora zdecydowano się na kontynuację prac na wersji pobranej ze starego repozytorium. Ostatnia aplikacja wchodząca w skład projektu, czyli *Player* jest serwerem sieciowym służącym do sterowania rzeczywistymi robotami. Dostarcza prosty interfejs, wspierający komunikację z wieloma typami powszechnie stosowanych czujników i aktuatorów. Oprogramowanie jest kompatybilne z systemami Linux, Solaris, *BSD oraz Mac OSX. W niniejszej pracy zdecydowano się na wykorzystanie jedynie symulatora *Gazebo*.

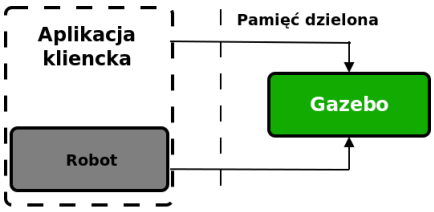
3.2 Architektura symulatora

Wymiana informacji pomiędzy elementami środowiska symulacyjnego *Player/Stage/Gazebo* została przedstawiona na rysunku 7.2. Po zapoznaniu się z nim , można lepiej zrozumieć rolę programu *Player*. Pełni on funkcję pośrednika pomiędzy aplikacją klienta, stworzona przez użytkownika a rzeczywistym robotem lub symulatorami modelującymi jego zachowanie w tym przypadku (*Stage* lub *Gazebo*). Komunikacja pomiędzy *Player-em*, robotem oraz aplikacją kliencką realizowana jest za pomocą protokołu TCP/IP. Dzięki takiej architekturze z punktu widzenia klienta nie istotne jest, czy interfejsy udostępnione przez *Player-a* sterują rzeczywistym robotem, czy symulowanym odpowiednikiem.

Z samym symulatorem, zarówno *Stage* jak i *Gazebo* można komunikować się poprzez pamięć współdzieloną (rysunek 3.2), bez wykorzystywania aplikacji *Player*. Rozwiązanie to jest preferowane w sytuacji gdy korzystanie z *Playera* nie jest uzasadnione, lub kiedy użytkownik dokonał modyfikacji działania symulatora, których nie implementuje *Player*. Aplikacja *Gazebo* udostępnia w tym celu bibliotekę *libgazebo*, dostarczającą zestaw funkcji do komunikacji. Z tego właśnie rozwiązania zdecydowano się skorzystać w niniejszej pracy. Po pierwsze, dlatego iż zmniejsza to nakład niezbędnych prac, po drugie iż nie posiadano rzeczywistego modelu zawodnika. Wprowadzone poprawki podczas opracowania środowiska testowego oraz przygotowanie własnego sterownika robota, także nasunęły to rozwiązanie.



Rysunek 3.1: Schemat komunikacji w środowisku *Player/Stage/Gazebo*



Rysunek 3.2: Zrealizowany schemat komunikacji w środowisku *Player/Stage/Gazebo*

Podsumowując, struktura pakietu oprogramowania *Player/Stage/Gazebo* umożliwia tworzenie aplikacji sterujących robotami w sposób, który zapewnia przenośność stworzonych programów i ich działanie zarówno na symulatorach robotów, jaki i na rzeczywistych obiektach.

3.2.1 Gazebo

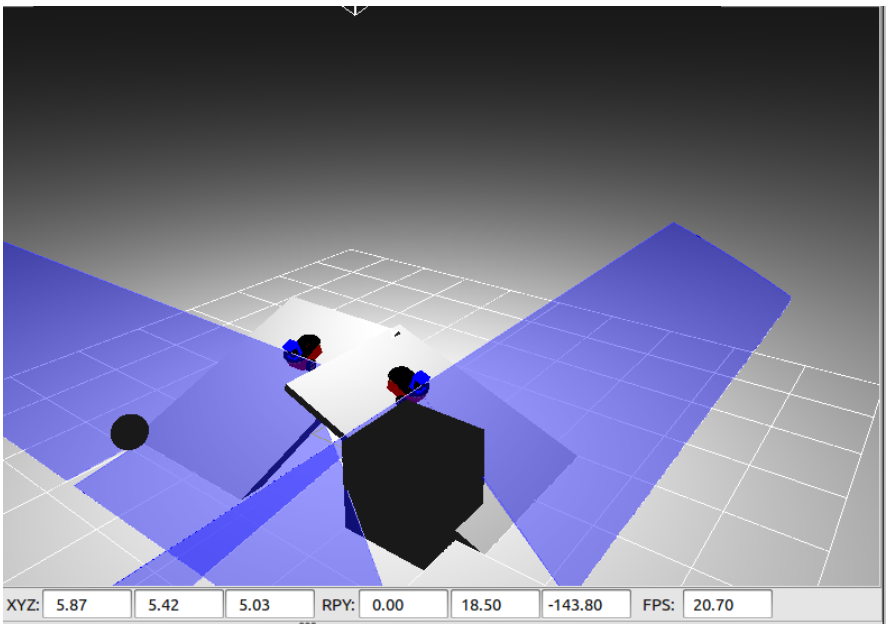
Symulator *Gazebo* jak wynika z wcześniejszego tekstu, umożliwia symulację grup robotów mobilnych. Symulowane środowisko jest w pełni trójwymiarowe, uwzględniona jest dynamika brył oraz obliczane rzeczywiste oddziaływania między komponentami symulowanego świata. Okupione jest to oczywiście większym niż w przypadku *Stage* zużyciem mocy obliczeniowej, dlatego symulowane populacje robotów nie powinny być zbyt liczne.

Trzy najistotniejsze komponenty wykorzystywane przez program *Gazebo* to:

- ODE[1], biblioteka odpowiadająca za symulację zależności fizycznych pomiędzy bryłami oraz detekcję kolizji,
- OGRE[2], silnik graficzny umożliwiający tworzenie wspomaganej sprzętowo grafiki 3D,
- FLTK[3], biblioteka dostarczająca przenośny interfejs użytkownika dla *Gazebo*.

W *Gazebo* możliwe jest symulowanie standardowych sensorów stosowanych w robotyce (m.in. czujniki odległości, kamery, GPS). Symulator wyposażony jest w

[1]Open Dynamics Engine, `http://www.ode.org`
[2]Object-Oriented Graphics Rendering Engine, `http://www.ogre3d.org/`
[3]Fast Light Toolkit, `http://www.fltk.org/`



Rysunek 3.3: *Gazebo* – okno główne (wersja 0.10)

gotowe modele popularnych robotów (jak np. *Pioneer2DX*, *Pioneer 2AT* oraz *SegwayRMP*). Dozwolone jest także tworzenie własnych modeli Pozwala ponadto na tworzenie własnych modeli, z możliwością konfigurowania ich właściwości fizycznych (takich jak np. masa, współczynnik tarcia, sztywność). Dzięki temu środowisko może w dużym stopniu odzwierciedlać rzeczywistość. Modele można wyposażyć w jeden ze zdefiniowanych kontrolerów umożliwiających sterowanie (zadawanie prędkości, czy odczyt danych z sensorów) lub opracować własny.

3.3 Modelowanie obiektów w Gazebo

Tworzenie symulowanego środowiska w *Gazebo* polega na opracowaniu opisu świata w pliku z rozszerzeniem *.world* za pomocą języka XML. W pliku powinny zostać zamieszczone parametry określające przebieg symulacji oraz informacje o występujących w danym świecie modelach.

3.3.1 Zasady modelowania w Gazebo 0.10

Forma pliku *.world* zawierającego opis symulowanego środowiska daje użytkownikowi możliwość konfiguracji wielu kluczowych elementów takich jak: konfiguracja

graficznego interfejsu użytkownika, zmianę sposobu renderowania sceny, wybór metody detekcji kolizji, zmianę wartości globalnych parametrów symulacji (m.in krok pracy symulatora). Elementy przykładowego pliku *.world* zostały zaprezentowane na listingach poniżej.

```
1        <?xml version="1.0"?>
2        <gazebo:world>
```

Bardzo istotnym elementem jest konfiguracja fizyki (ODE), interfejs umożliwia konfigurację globalnych parametrów takich jak:

```
3        <physics:ode>
4                <stepTime>0.001</stepTime>
5                <gravity>0 0 -9.8</gravity>
6                <erp>0.8</erp>
7                <cfm>0.05</cfm>
8                <stepType>quick</stepType>
9                <stepIters>25</stepIters>
10               <stepW>1.4</stepW>
11               <contactSurfaceLayer>0.007</contactSurfaceLayer>
12               <contactMaxCorrectingVel>100</contactMaxCorrectingVel>
13        </physics:ode>
```

Znaczenie poszczególnych parametrów jest następujące:

1. `stepTime` jest wyrażony w sekundach i określa czas pomiędzy kolejnymi aktualizacjami silnika `ODE`,

2. `gravity` jest wektorem określającym siłę grawitacji,

3. `erp` rozwija się na *error reduction parameter*, przyjmuje on wartości z przedziału od 0 do 1 i jest odpowiedzialny za redukcję błędów w połączeniach pomiędzy bryłami (problematyka zostanie szerzej poruszona na stronie 25); `erp` ustawione na 0 powoduje, że żadna dodatkowa siła nie jest przykładana do danej bryły, natomiast wartość 1 powoduje, że wszytskie błędy połączeń zostaną naprawione w kolejnym kroku symulatora,

4. `cfm` jest skrótem od *constraint force mixing* i odpowiada za sztywność ograniczeń występujących w kolizjach między bryłami, gdy wartość jest równa 0 ograniczenie wynikające z fizyki nie może zostać złamane, ustawienie parametru na wartość dodatnią powoduje, że ograniczenie staje się "miękkie", sprężyste,

5. `stepType` odpowiada za rodzaj używanej funkcji z `ODE` do detekcji kolizji, użytkownik może dokonać wyboru jednej z dwóch wartości:

   - *world*, używana jest wtedy funkcja `dWorldStep`, operująca na macierzy zawierającej wszystkie ograniczenia, złożoność obliczeniowa tej metody wynosi m^3, natomiast pamięciowa jest rzędu m^2, gdzie m określa ilość wierszy (ograniczeń) analizowanej macierzy,
   - *quick*, do detekcji kolizji stosowana jest metoda iteracyjna, w literaturze nazywana `SOR - Successive over-relaxation` należąca do rodziny metod Gaussa–Seidela, jej złożoność obliczeniowa jest rzędu $m * N$, a pamięciowa m, gdzie m ma znaczenie jak wyżej, natomiast N jest liczbą iteracji; dla dużych systemów metoda jest dużo bardziej wydajna jednak mniej dokładna, mogą także występować problemy z jej stabilnością; najprostszą metodą na poprawę stabilności jest zwiększanie `cfm`,

6. `stepIters` - ilość iteracji, gdy do detekcji kolizji wybrano metodę *quick*,

7. `stepW` określa czas relaksacji metody Gaussa–Seidela,

8. `contactSurfaceLayer` określa głębokość na jaką może wnikać bryła w podłoże, ustawienie parametru na małą wartość dodatnią zapobiega jitterowi w momencie, gdy ograniczenia są nieustannie tworzone i zrywane (na przykład podczas ruchu koła po podłożu),

9. `contactMaxCorrectingVel` jest maksymalną korektą prędkości jaka może wynikać z utworzonych tymczasowo kontaktów; domyślnie parametr przyjmuje nieskończoną wartość zmniejszanie jego wartości może zapobiec tworzeniu się zagnieżdżonych kontaktów.

Warto wspomnieć, że niektóre parametry można zmieniać dla poszczególnych modeli lub połączeń *joints*.
Ustawienia interfejsu: typ (dostępne tylko *fltk*), rozmiaru okna i jego pozycja początkowej:

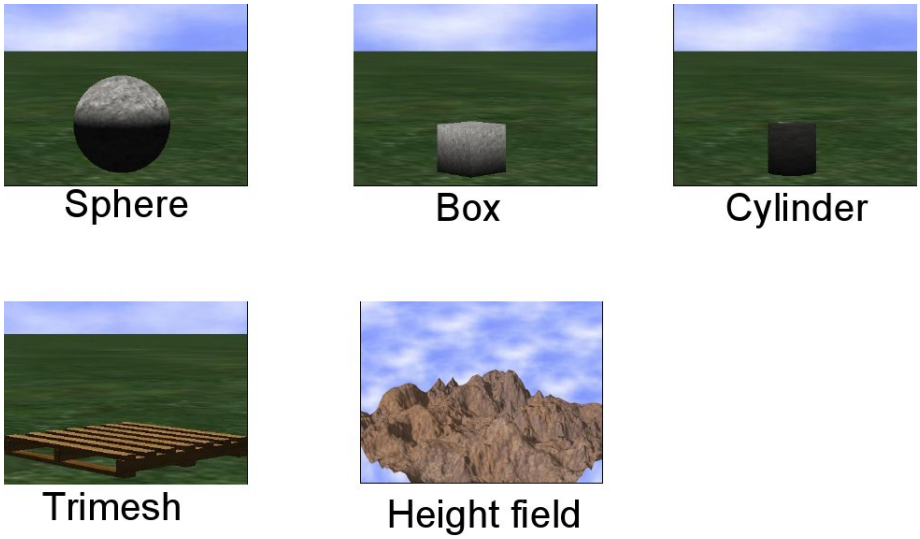
```
14      <rendering:gui>
15          <type>fltk</type>
16          <size>640 480</size>
17          <pos>0 0</pos>
```

```
18      </rendering:gui>
```

Określenie parametrów renderowanej sceny: techniki cieniowania, tekstury pokrywającej niebo (dostępne inne opcje, jak np. rodzaje oświetlenia, dodawanie mgły itp.):

```
19      <rendering:ogre>
20          <shadowTechnique>stencilAdditive</shadowTechnique>
21          <sky>
22                <material>Gazebo/CloudySky</material>
23          </sky>
24      </rendering:ogre>
25  </gazebo:world>
```

Do tak stworzonego pliku z opisem świata należy następnie dodać modele. Dodany obiekt musi zawierać co najmniej jeden element *body*, który jest złożony z elementów *geoms*. Sekcja opisująca przykładowy model piłki mogłaby wyglądać następująco:



Rysunek 3.4: Dostępne typy *geoms* – Gazebo 0.10 (źródło: [21])

```
1   <model:physical name="ball">
2       <static>false</static>
```

Na wstępie deklarowany jest model, którego parametr *static* ustawiono na *false*, co oznacza, że będzie on brał udział w symulacji fizycznej i może oddziaływać z innymi obiektami. Następnie należy zadeklarować "ciało" tworzonego obiektu:

```
3     <body:sphere name = "ball_body">
```

Wewnątrz *body*, które jest odpowiedzialne za dynamikę obiektu, należy umieścić elementy opisujące kształt obiektu (pozwalające tym samym na detekcję kolizji) – w tym przypadku zdefiniowano kulę o wymiarach i masie odpowiadającej piłce do golfa:

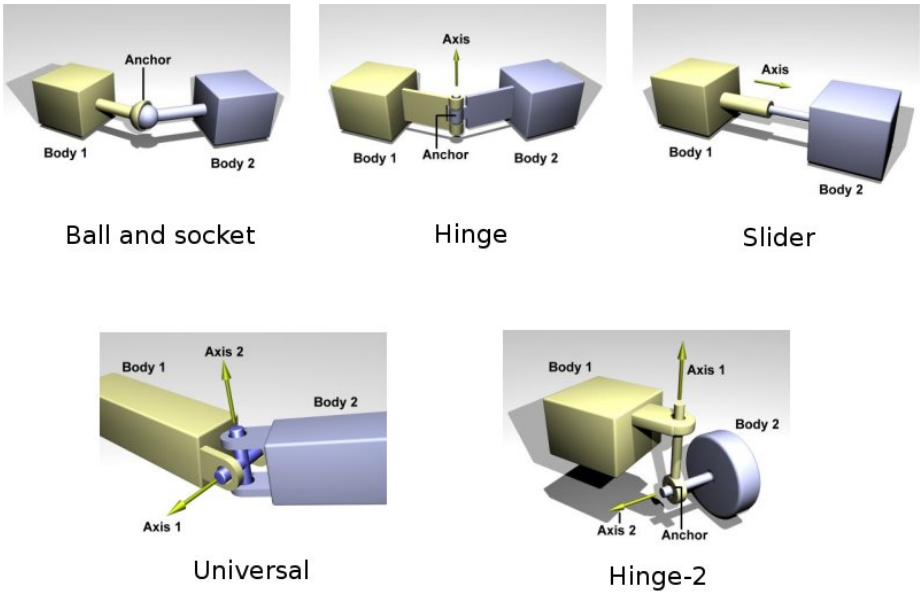
```
4         <geom:sphere name = "ball_geom">
5                 <xyz>0 0 0.01</xyz>
6                 <rpy>0 0 0 </rpy>
7                 <size>0.02</size>
8                 <mass>0.045</mass>
```

Sekcja *visual* bloku *geom* pozwala na przypisanie obiektowi wyglądu – może to być figura o dowolnym kształcie (stworzona np. w programie do grafiki 3D) i wybranej przez projektanta teksturze lub kolorze:

```
9                 <visual>
10                        <scale>0.02 0.02 0.02</scale>
11                        <size>0.02</size>
12                        <mesh>unit_sphere</mesh>
13                        <material>Gazebo/Orange</material>
14                <visual>
15        </geom:sphere>
16    </body:sphere>
17 </model:physical>
```

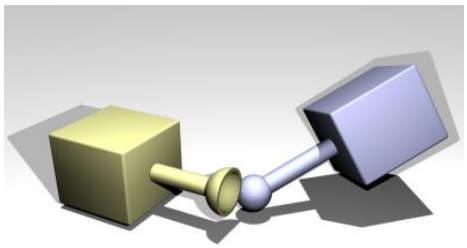
Połączenia pomiędzy bryłami

Tworząc modele, można korzystać z trzech podstawowych brył (kula, walec, prostopadłościan), a także z siatek *trimesh* oraz elementów *height field* (stworzonych do generowana terenu – por. rys. 3.4)). Żeby zamodelować robota, należy jeszcze stworzone bryły odpowiednio ze sobą powiązać. Do tego służą elementy typu *joint*, którymi można łączyć komponenty *body* (modelujące np. koła i podwozie robota). Stopnie swobody połączenia zależą od wybranego typu wiązania *joint* (dostępne typy przedstawiono na rys. 3.5). Obiekty połączone wiązaniem tworzą parę kinematyczną. *Gazebo* umożliwia nadawanie tak połączonym bryłom prędkości obrotowych względem siebie, co pozwala na modelowanie ruchomych elementów robotów.



Rysunek 3.5: Typy połączeń *joints* (Źródło: dokumentacja ODE)

Jak wspomniano wcześniej, w czasie realizowanej symulacji połączenia pomiędzy bryłami mogą ulec wypaczeniu, na skutek sił działających na model. Może to być tarcie, siła powodująca obrót bryły czy siły działające na model podczas kolizji. Sytuację taką przedstawiono na rysunku 3.6.



Rysunek 3.6: Zepsute połączenie (*joints*) (Źródło: dokumentacja ODE)

Błędy tego typu można redukować za pomocą parametru `erp` ustawianego globalnie lub dla każdego z połączeń indywidualnie.

Sterowniki modeli

Sterowanie zaprojektowanym modelem jest możliwe po uprzednim wyposażeniu go w sterownik (*controller*). Sterowniki implementowane są przez użytkownika w C++

jako klasa dziedzicząca po zdefiniowanym w *libgazebo* interfejsie. Klasa odpowiada za przetwarzanie danych z sensorów robota oraz pozwala na zadawanie prędkości bryłom połączonym wiązaniami. Komunikuje się on z programem sterującym za pośrednictwem interfejsów (bezpośrednio korzystając z *libgazebo* lub poprzez *Playera*). *Gazebo* oczywiście dostarcza gotowe sterowniki, pozwalające na kontrolowanie np. robota z napędem różnicowym, w wersji 0.10 dodano także przykładowy model robota holonomicznego zawarty w pliku *wizbot.world*. Aby z nich skorzystać, wystarczy przypisać wybrany sterownik do modelu robota, określić, które ze złączeń odpowiadają jego kołom i podać nazwę instancji interfejsu, za pomocą którego odbywać ma się komunikacja pomiędzy klientem a sterownikiem. W przypadku sterowania przemieszczeniem będzie to interfejs *position*, za pomocą którego możemy zadawać prędkości bryłom, ale *Gazebo* posiada też zaimplementowane interfejsy do sterowania innymi elementami np. do komunikacji z laserowymi czujnikami odległości, kamerą bądź chwytakiem manipulatora.

Listing 3.1: Przykład przypisanie sterownika do modelu

```
1 <model:physical name="pioneer_model">
2     <controller:pioneer2dx_position2d name="controller">
3         <leftJoint>left_hinge_joint</leftJoint>
4         <rightJoint>right_hinge_joint</rightJoint>
5         <interface:position name="position_iface"/>
6     </controller>
7 </model:physical>
```

3.3.2 Realizacja środowiska Ligi RoboCup

W celu realizacji niniejszej pracy stworzono dla symulatora *Gazebo* modele odpowiadające robotom biorącym udział w *Small-size League*. Boisko zostało stworzone z wykorzystaniem rzeczywistych wymiarów $5.4[m]x7.4[m]$, obowiązujących w lidze w 2011 roku [4]. Wszystkie linie boiska są także zgodne obowiązującymi w tych mistrzostwach. Margines na auty jest równy 675[mm]. Model piłki odpowiada piłce do golfa, która używana jest w oficjalnych rozgrywkach.

Wykonane modele robotów wzorowano na konstrukcji rzeczywistych robotów wykorzystywanych w *Small-size League*. Otrzymane modele zaprezentowane są na

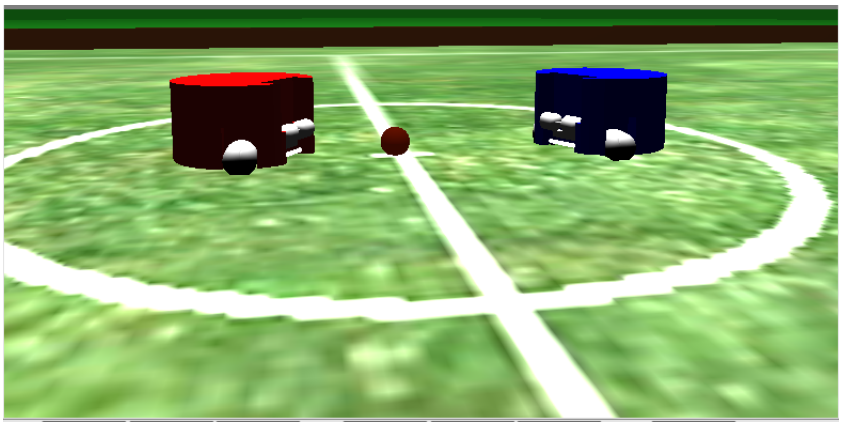
[4] `http://small-size.informatik.uni-bremen.de/_media/rules:ssl-rules-2011.pdf`

rysunku 3.7. Główna częścią robota jest walec o promieniu 6[cm] i wysokości 4[cm], umieszczony na trzech kołach rozmieszczonych symetrycznie na podstawie walca. Koła symulują zachowanie stosowanych w robotyce kół szwedzkich. Zachowanie takie osiągnięto poprzez dodanie nowego parametru do sekcji opisującej bryłę w symulatorze, określającego kierunek siły tarcia dla danej bryły:

Listing 3.2: Parametr określający kierunek siły tarcia

```
1          <fDir1>1 0 0</fDir1>
```

Powyższa wartość oznacza, że tarcie występuje jedynie w kierunku osi OX danej bryły. Należało także wprowadzić poprawkę w silniku fizycznym symulatora, tak aby `ODE` uwzględniało ten parametr. W strukturze opisującej punkt kontaktowy pomiędzy bryłami ustawiono flagę *dContactMu2*[5] w sytuacji kiedy dla bryły określony jest kierunek tarcia. Rozwiązanie takie umożliwiło ustawienie dla koła dużego tarcia w kierunku ruchu obrotowego oraz małego w kierunku prostopadłym, czyli osiągnięto zachowanie typowe dla kół szwedzkich.



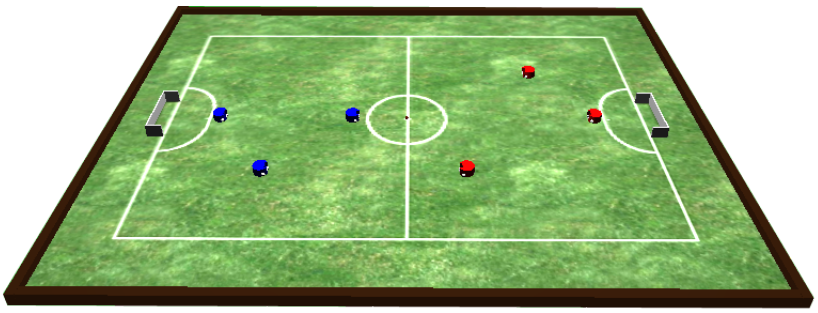
Rysunek 3.7: Opracowane modele robotów

Robot został dodatkowo wyposażony w *dribbler*[6] pomagający utrzymać kontrolę nad piłką. Mimo że *Gazebo* udostępnia sterownik dla napędu holonomicznego, sterowanie robotem wymagało napisania nowego kontrolera, ponieważ nie uwzględniał on obsługi *dribblera*, ani nie umożliwiał oddawania strzałów. Sterownik został zrealizowany w oparciu o teorię zawartą w rozdziale 4. Z punktu widzenia użytkownika sterowanie robotem odbywa się poprzez ustawianie odpowiedniej prędkości

[5] `http://www.ode.org/ode-latest-userguide.html`

[6] urządzenie opisano w par. 2.3

liniowej i kątowej. Sterownik przekształca je na odpowiednie prędkości obrotowe poszczególnych kół. Dodatkowo zaimplementowano programowy regulator `PID`, sterujący siłą przykładana do kół robota. Dzięki temu uzyskano lepszą dokładność sterowania robotem. Parametry regulatora początkowo dobrano posługując się metodą `Zieglera-Nicholsa`. Przy wyłączonych członach całkującym i różniczkującym analizowano wykresy rozkładu prędkości obrotowych kół robota w czasie przy ustalonej zadanej prędkości liniowej. Stopniowo zwiększano współczynnik wzmocnienia proporcjonalnego. Wyznaczono wartość wzmocnienia przy której układ zbliżył się do granicy stabilności (wzmocnienie krytyczne K_k) oraz wyznaczono okres drgań T_k. Wartości właściwych wzmocnień poszczególnych członów wyznaczono zgodnie z procedurą: $K = 0.6K_k$, $T_i = T_k$, $T_d = 0.12T_k$. Następnie eksperymentalnie dokonano ich korekty. W pliku opisującym model robota istnieje możliwość ręcznego modyfikowania nastaw regulatora. Kod źródłowy sterownika został zamieszczony na płycie CD razem z kodem źródłowym wykorzystywanej wersji symulatora.



Rysunek 3.8: Model boiska

W trakcie symulacji opracowanych modeli zauważono, że piłka raz wprawiona w ruch nie hamuje. Sytuacja taka była spowodowana brakiem tarcia tocznego. Zaimplementowano zatem kolejną modyfikację w symulatorze, tak aby dla każdej bryły istniała możliwość konfigurowania tego parametru (*angularDamping*) w pliku opisującym model.

Ponieważ korzystano z niestabilnej wersji oprogramowania (inne nie było jeszcze dostępne), pobranego bezpośrednio z repozytorium deweloperów, należało poprawić jeszcze kilka innych błędów symulatora, miedzy innymi niepoprawną pracę interfejsu

umożliwiającego komunikację pomiędzy symulatorem a aplikacją kliencką.

Rozdział 4

Sterowanie modelem robota z *Small-size League*

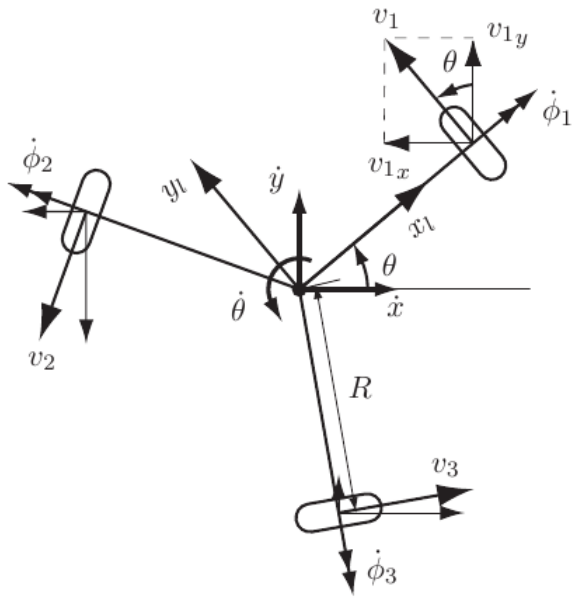
Budowę i napęd robota należy dobrać odpowiednio do postawionego zadania. W przypadku ligi *Small-size League*, najważniejszym elementem jest prostota i mobilność takiej jednostki, w pracy inżynierskiej [15] posługiwano się robotem modelem robota o napędzie różnicowym (dwa niezależnie napędzane koła). Decyzję o odwzorowaniu takiego robota w symulatorze motywowano wtedy próbą odzwierciedlenia rzeczywistego robota `HMT` [22] stworzonego w ramach koła robotyki `Bionik` działającego na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. W niniejszej pracy zrezygnowano jednak z tego modelu, ponieważ nie był wystarczająco funkcjonalny. Zdecydowano się natomiast na zbudowanie modelu wzorowanego na rzeczywistym zawodniku *Small-size League*.

4.1 Omówienie omnikierunkowej bazy jezdnej

Analogicznie jak podczas rzeczywistej rozgrywki, decydującym elementem jest budowa anatomiczna i zdolności motoryczne zawodników, tak samo podczas rozgrywek robotów istotna rolę odgrywa baza jezdna zawodników. W pracy inżynierskiej testom poddawany był robot o napędzie różnicowym. Baza ta posiadała jednak znaczące ograniczenia, które musiały zostać uwzględnione w algorytmach sterujących. Praktyczniejszą w użyciu jest baza omnikierunkowa. Korzystając z takiej bazy w wiekszości algorytmów robot może być traktowany jako punkt materialny.

4.1.1 Opis położenia kół

Baza omnikierunkowa składa się z umieszczonych symetrycznie co najmniej trzech kół szwedzkich tak jak zaprezentowano to na rysunku 4.1. Każde z kół posiada osobny napęd. Budowę koła omnikierunkowego przedstawiono na ilustracji 4.2. Koło



Rysunek 4.1: Rozkład kół w omnikierunkowej bazie jezdnej

szwedzkie posiada na swoim obwodzie zamontowane w odpowiedni sposób dodatkowe rolki. Umożliwiają one ruch koła w dowolnym kierunku, bez względu na to, jak koło jest zorientowane w przestrzeni. Dzięki temu umożliwia ruch robota w dowolnym kierunku, czyli należy do klasy robotów holonomicznych.

Rysunek 4.2: Konstrukcja przykładowego koła szwedzkiego

4.1.2 Opis kinematyki oraz dynamiki bazy

Podczas sterowania robotem istotnym problemem jest sposób w jaki prędkości i przyspieszenia obrotowe poszczególnych kół przekładają się na predkość/przyspieszenie kątowe i liniowe całego robota. Wszytskie poniższe obliczenia zostały wykonane przy założeniu, że koła nie ulegają poślizgowi, czyli że cały moment obrotowy śilników przekłada się na prędkość robota. Przyspieszenie liniowe i prędkość obrotowa środka masy takiego układu dane jest następującymi wzorami:

$$a = \frac{1}{M}(F1 + F2 + F3) \tag{4.1}$$

$$\dot{\omega} = \frac{R}{I}(f_1 + f_2 + f_3) \tag{4.2}$$

, gdzie f_i oznacza długość wektora siły przyłożonego do poszczególnego koła, a I jest momentem bezwładności. Przyspieszenie wzdłuż poszczególnych osi można obliczyć rozbijając siłę działającą na koło wzdłuż tychże osi, otrzymamy wtedy:

$$Ma_x = -f_1 sin\theta_1 - f_2 sin\theta_2 - f_3 sin\theta_3 \tag{4.3}$$

$$Ma_y = f_1 cos\theta_1 + f_2 cos\theta_2 + f_3 cos\theta_3 \tag{4.4}$$

Dla jednolitego cylindra moment bezwładności obliczany jest ze wzoru $I = \frac{1}{2}MR^2$, natomiast dla obręczy $I = MR^2$, gdzie R jest odpowiednio promieniem cylindra/obręczy natomiast M masą. Dla obiektów o rozkładzie masy pomiędzy obręczą, a cylindrem wprowadzany jest dodatkowy parametr α. Wzór przyjmuje wtedy postać: $I = \alpha MR^2$, gdzie $0 < \alpha < 1$. Używając zapisu macierzowego równania można przedstawić w postaci:

$$\begin{pmatrix} a_x \\ a_y \\ \dot{\omega} \end{pmatrix} = \begin{pmatrix} -sin\theta_1 & -sin\theta_2 & -sin\theta_3 \\ cos\theta_1 & cos\theta_2 & cos\theta_3 \\ \frac{MR}{I} & \frac{MR}{I} & \frac{MR}{I} \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} \tag{4.5}$$

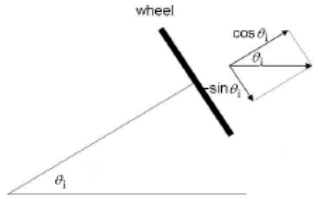
Podstawiając do powyższego wzoru $I = \alpha MR^2$ oraz zastępując $\dot{\omega}$ wyrażeniem $R\dot{\omega}$ otrzymujemy:

$$\begin{pmatrix} a_x \\ a_y \\ R\dot{\omega} \end{pmatrix} = \begin{pmatrix} -sin\theta_1 & -sin\theta_2 & -sin\theta_3 \\ cos\theta_1 & cos\theta_2 & cos\theta_3 \\ \frac{1}{\alpha} & \frac{1}{\alpha} & \frac{1}{\alpha} \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} \tag{4.6}$$

Macierz z powyższego równania o wymiarze $3x3$ zostanie oznaczona symbolem C_α.

Jednak najbardziej interesujący jest sposób w jaki prędkość obrotowa kół przekłada się na prędkość liniową robota. Załóżmy, że robot porusza się wzdłuż osi OX, zatem wektor prędkości $(v_x, v_y, R\omega)$ wygląda następująco $(1, 0, 0)$. Rozważmy jedno z kół, tak jak to przedstawiono na rysunku 4.3, dokonując rozkładu wektora prędkości na dwie składowe, jedną zgodną z ruchem obrotowym dużego koła, a drugą zgodną z ruchem małych, poprzecznych kółek otrzymujemy odpowiednio prędkości $v = -sin\theta$ $v_y = cos\theta$. Przy wyznaczaniu prędkośći koła przyjęto założenie, że prędkość dodatnia powoduje obrót w kierunku wyznaczonym przez kciuk prawej dłoni, gdy pokrywa się ona z osią silnika. Otrzymujemy zatem następujące powiązanie pomiędzy prędkościami robota, a prędkościami poszczególnych silników:

$$\begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} -sin\theta_1 & cos\theta_1 & 1 \\ -sin\theta_2 & cos\theta_2 & 1 \\ -sin\theta_3 & cos\theta_3 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ R\omega \end{pmatrix} \tag{4.7}$$

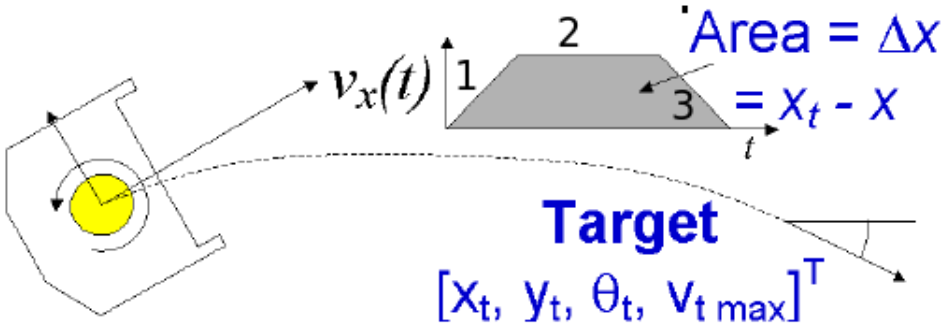


Rysunek 4.3: Prędkość liniowa dużego koła szwedzkiego i małych kółek, gdy robot porusza się wzdłuż osi OX

4.2 Obliczanie profilu prędkości liniowej robota

Znając już powiązanie pomiędzy prędkością obrotową kół a prędkością liniową i obrotową robota, ostatnim elementem jest wyznaczenie prędkości prowadzących robota do zadanego punktu. Należy przy tym uwzględnić takie parametry robota jak przyspieszenie i opóźnienie. W tym celu skorzystano z metody opisanej w [16] oraz [17]. Polega ona na dekompozycji dwuwymiarowego problemu sterowania robotem,

do dwóch problemów jednowymiarowych, tak jak to zaprezentowano na rysunku 4.4. Ruch robota w kierunku osi OX i w kierunku osi OY jest rozpatrywany osobno.



Rysunek 4.4: Dekompozycja sterowania robotem w 2D na dwa niezależne zadania w 1D

Podejście to jest znane w robotyce pod nazwą trapezoidalnego profilu prędkości. Zaczynając od punktu w którym robot się znajduje przyspiesza on ze swoim stałym przyspieszeniem, aż do osiągnięcia maksymalnej prędkości, następnie zaczyna hamować, tak aby zatrzymać się w punkcie docelowym. W szczególnym przypadku profil prędkości może przybrać formę trójkąta. Prędkość obliczana jest według następujących reguł:

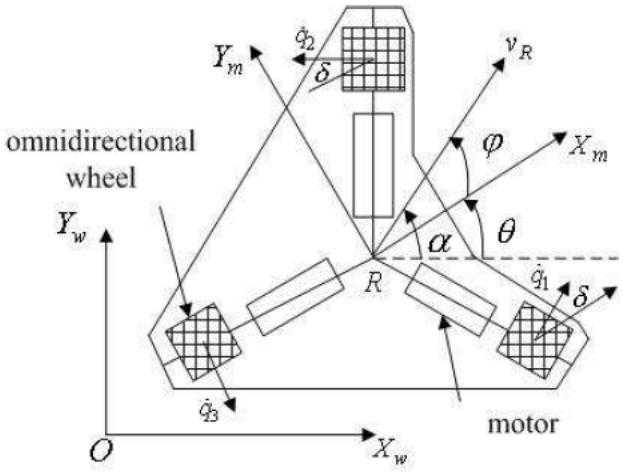
1. Jeśli bieżąca prędkość spowoduje oddalenie się robota od celu to wyhamuj do 0.
2. Jeśli robot poruszając się z bieżąca prędkością przejedzie cel to także należy zatrzymać go.
3. Gdy bieżąca prędkość przekracza maksymalną, wyhamuj do prędkości maksymalnej.

4. Oblicz trójkątny profil prędkości prowadzącej do celu.

5. Jeśli w obliczonym rozkładzie prędkość przekracza w jakimkolwiek momencie maksimum to należy obliczyć trapezoidalny profil.

4.3 Dryblowanie z piłką

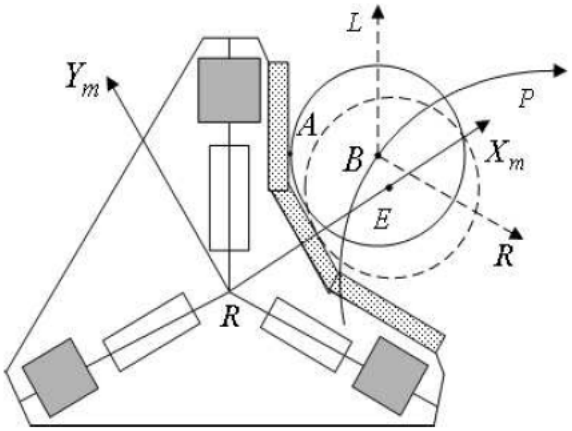
We wcześniejszym podrozdziale rozwiązano problem sterowania omnikierunkową bazą jezdną. Osobnym zadaniem jest wyznaczanie prędkości robota w sytuacji kiedy posiada on piłkę. Robot dryblujący z piłką nie może poruszać się z pełną dowolnością, tak aby nie stracić nad nią kontroli. Problem ten został dokładnie zaprezentowany w [14]. W publikacji co prawda skupiono się na modelu robota z rozgrywek *Middle-size League* jednak zaprezentowany algorytm zajduje zastosowanie także w lidze małych robotów. Na rysunku 4.5 przedstawiono bazę jezdną z zaznaczonym globalnym układem współrzędnych $[X_w; Y_w]$ oraz związanym ze sterowanym robotem $[X_m; Y_m]$, za pomocą wektora v_r zaznaczono prędkość liniową z jaką porusza się robot. Natomiast na rysunku 4.6 przedstawiona została sytuacja, gdy robot jest w



Rysunek 4.5: Omnikierunkowa baza jezdna stosowana w rozgrywkach średnich robotów. Źródło [14]

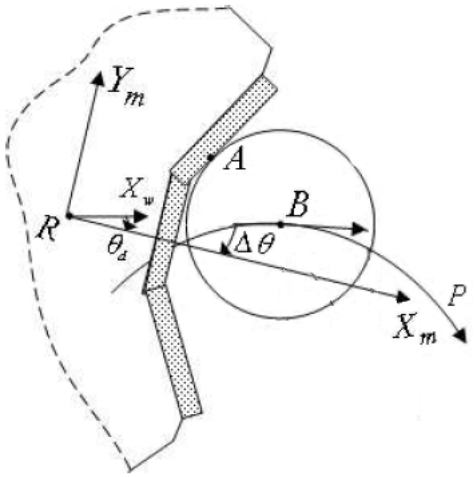
posiadaniu piłki. Krzywa P oznacza trajektorię po której porusza się zawodnik. Aby nie stracić piłki punkt E znajdujący się dokładnie na wprost robota w odległości L równej promieniowi piłki powinien pokrywać się z jej środkiem. Zadaniem algorytmu sterującego dryblującym robotem jest zatem, wyznaczenie prędkości obrotowej,

takiej aby środek piłki znajdował się w otoczeniu punkt E. W przypadku robotów z ligi *Small-size League* $L = r_{ball} + r_{robot} + w_{dribbler}$. Na rysunku 4.7 przedstawiono



Rysunek 4.6: Pozycja piłki w układzie współrzędnych związanym z robotem. Źródło [14]

robota wykonującego obrót z piłką. W momencie kiedy robot wraz z piłką poruszają się po łuku c, piłka posiada pewne przyspieszenie dośrodkowe, $a_b = cv_e$ powodujące jej odchylenie od punktu E. Na rysunku symbolem $\Delta\Theta$ zaznaczono odchylenie kątowe, wynikające właśnie z działania siły dośrodkowej, które należy minimalizować, aby robot nie stracił kontroli nad piłką. Odchylenie to można zamodelować jako: $\Delta\Theta = k_\Theta c v_d^2$, gdzie k_Θ jest dobranym empirycznie parametrem. W przypadku idealnym rotacja robota wykonującego manewr powinna być równa: $\Theta_d = \Theta_P + \Delta\Theta$, gdzie Θ_P jest nachyleniem stycznej do trajektorii po której porusza się robot, w punkcie będącym rzutem prostokątnym E na ścieżkę. Do regulacji rotacji robota posiadającego piłkę w publikacji [14] wykorzystano regulator `PD`. Prędkość obrotowa obliczana jest zatem następująco: $\omega = k_p(\Theta_d - \Theta) + k_d(\dot{\Theta}_d - \dot{\Theta})$. Parametry k_p oraz k_d są dobierane empirycznie.



Rysunek 4.7: Odchylenie piłki od idealnego położenia podcza manewrowania. Źródło [14]

Rozdział 5

Algorytmy unikania kolizji

W rozdziale zostanie szerzej zaprezentowany jeden z algorytmów unikania kolizji jakim jest RRT(`Rapidly-Exploring Random Tree`). Poruszona zostanie także kwestia powodów, dla których wybrano tę, a nie inną metodę. Uzasadnione zostanie odejście od algorytmu opracowanego w ramach pracy inżynierskiej (CVM `Curvature Velocity Method`). Ponadto opisane zostaną inne metody planowania ścieżki oraz omówione zostaną ich właściwości. Na tej podsatwie zostanie uzasadniony wybór algorytmu RRT.

5.1 Krótki przegląd algorytmów unikania kolizji

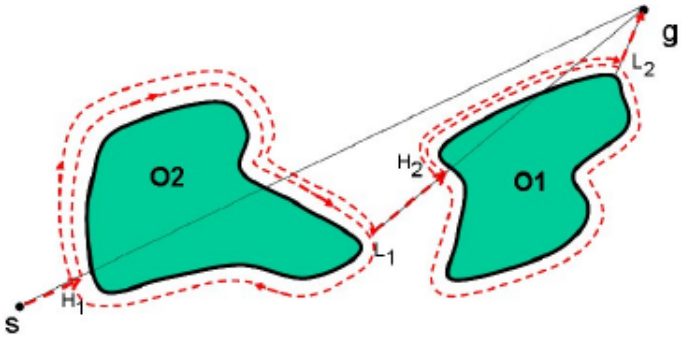
Jednym z podstawowych problemów podczas poruszania się każdej jednostki mobilnej jest wyznaczenie bezkolizijnej ścieżki prowadzącej do celu. Współczesna robotyka zna wiele algorytmów rozwiązujących z mniejszym bądź większym sukcesem to zadanie. Użycie wielu z nich jest jednak w pełni uzasadnione tylko w szczególnych okolicznościach.Poniżej zostaną zaprezentowane najbardziej znane algorytmy unikania kolizji (omówione także w pracy inżynierskiej [15]

5.1.1 Algorytm Bug

Najprostszym algorytmem unikania kolizji jest algorytm *Bug*, naśladujący zachowanie pluskwy. Gdy robot, podążając do punktu docelowego napotyka przeszkodę, okrąża ją całkowicie i zapamiętuje położenie w którym znajdował się najbliżej celu. Następnie ponownie podczas powtórnego okrążania przeszkody robot dąży do

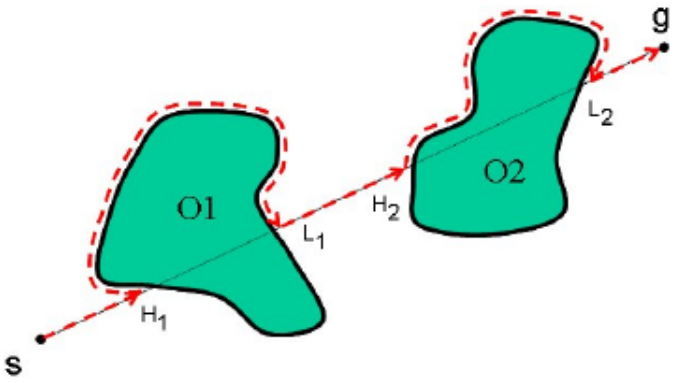
uprzednio zapamiętanego punktu po czym odrywa się od przeszkody i zaczyna podążać w stronę celu. Ścieżka wyznaczona za pomocą algorytmu została zaprezentowana na rysunku 5.1. Robot wykorzystujący ten algorytm powinien być wyposażony w dwa rodzaje czujników:

- czujnik celu – wskazujący kierunek do celu oraz umożliwiający pomiar odległości do celu
- czujnik lokalnej widoczności – umożliwiający podążanie wzdłuż konturu przeszkody



Rysunek 5.1: Bezkolizyjna ścieżka wyznaczona przez algorytm Bug

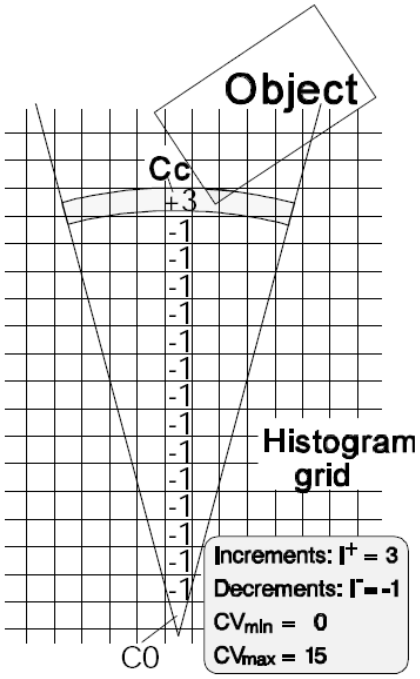
Ścieżka wyznaczona przez ten algorytm jest daleka od optymalnej. Robot niezależnie od wybranego kierunku omijania przeszkody musi ją okrążyć całkowicie. W pewnych sytuacjach całkowite okrążenie przeszkody jest niemożliwe, np jesli przeszkodą jest ściana. W takim przypadku algorytm zawodzi całkowicie. Efektywność algorytmu można poprawić sprawdzając podczas okrążania przeszkody czy punkt w którym aktualnie znajduje się robot jest poszukiwanym punktem położonym najbliżej celu. Przykładowa ścieżka wyznaczona za pomocą zmodyfikowanej wersji algorytmu jest widoczna na rysunku 5.2. Robot otacza przeszkodę w wybranym wcześniej kierunku i odłącza się od niej w momencie przecięcia prostej łączącej punkt startowy z punktem docelowym. Uzyskana w ten sposób ścieżka nadal zależy od wybranego a priori kierunku jazdy. Obie zaprezentowane wersje algorytmu Bug nie uwzględniaja ograniczeń wynikających z kinematyki robota, a w szczególności faktu, że rozpatrywany robot może być nieholonomiczny.



Rysunek 5.2: Zasada działania algorytmu Bug w wersji rozszerzonej

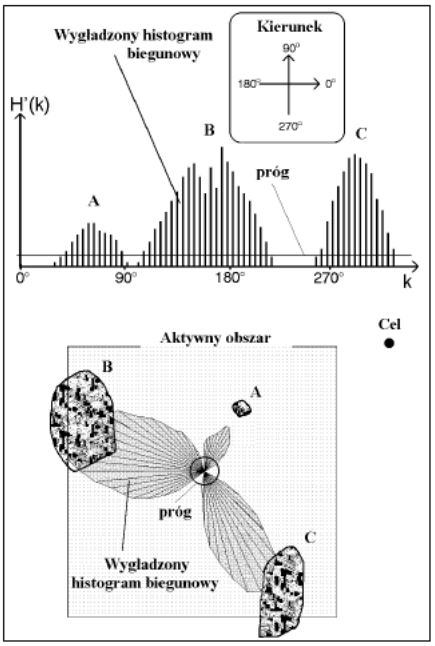
5.1.2 Algorytm VHF

Pełna anglojęzyczna nazwa metody brzmi *Vector Field Histogram*, na język polski można ją przetłumaczyć jako algorytm histogramu pola wektorowego. Należy ona do grupy metod, które w czasie rzeczywistym pozwalają na jednoczesne wykrywanie i omijanie przeszkód oraz kierowanie robota na cel. Algorytm ten został szczegółowo opisany w pracy inżynierskiej [15], więcej informacji można też znaleźć w publikacjach autorów, [8] oraz [9]. W skrócie jego zasada działania opiera się na konstrukcji dwuwymiarowego opisu świata w postaci siatki. Każdemu elementowi siatki przypisany jest poziom ufności oddający prawdopodobieństwo z jakim w danym położeniu może pojawić się przeszkoda (rysunek 5.3).



Rysunek 5.3: Zasada tworzenia dwuwymiarowego histogramu (na podstawie [9])

Tak skonstruowana mapa redukowana jest do jednowymiarowego histogramu biegunowego (rysunek 5.4), który dodatkowo wygładzany jest przez filtr dolnoprzepustowy. W ten sposób otrzymuje się informację o poziomie ufności wystąpienia przeszkody poruszając się w danym kierunku (świadomomie rezygnuje się z informacji o odległości od przeszkody). Na podstawie tego histogramu wyznaczane są sterowania dla robota. Histogram jest analizowany w poszukowaniu minimów lokalnych, wszystkie doliny, których poziom ufności znajduję się poniżej ustalonego umownie progu są rozważane przy wyznaczaniu kierunku jazdy robota. Jeśli minimów jest kilka wybierane jest to, którego kierunek prowadzi najbliżej do celu. Do poprawnego działania algorytm potrzebuje informacji o rozłożeniu przeszkód w otoczeniu robota, w oryginalnej implementacji była ona pozyskiwana z sensorów ultradźwiękowych, można je zastąpić z powodzeniem czujnikami laserowymi. Obraz video z kamery umieszczonej nad eksplorowanym światem także dostarcza tej informacji (jak w rozgrywkach *Small-size League*).



Rysunek 5.4: Histogram biegunowy dla przykładowego rozmieszczenia przeszkód (na podstawie [20])

5.1.3 Technika dynamicznego okna

Popularną, stosowaną w robotyce metodą unikania kolizji jest także technika dynamicznego okna. Algorytm ten analizuje jedynie możliwe do osiągnięcia w danej sytuacji prędkości. Więcej informacji na temat algorytmu można znaleźć w publikacji [10] lub w pracy inżynierskiej [15]. Ujmując w jedym zdaniu algorytm polega na przeszukiwaniu zbioru dopuszczalnych prędkości liniowych i kątowych, tak aby maksymalizować funkcję celu. Sposób konstrukcji funkcji celu gwarantuje, że wybrane prędkości będą prowadzić robota w zamierzonym kierunku oraz, że nie natrafi na przeszkodę.

5.1.4 Algorytmy pól potencjałowych

Algorytm w wersji podstawowej

Kolejne podejście do problemu nawigacji robota mobilnego zaczerpnięto z fizyki. Problem znalezienia bezkolizyjnej ścieżki został sprowadzony do problemu konstrukcji funkcji opisującej rozkład energii w danym środowisku. W takim układzie dotarcie

do celu jest równoważne ze znalezieniem minimum funkcji opisującej rozkład sztucznego pola potencjałowego. Robot podąża w kierunku ujemnego gradientu tej fuknkcji, mamy zatem $c(t) = -\nabla U(c(t))$. W klasycznym podejściu sztuczne pole potencjałowe tworzy się w ten sposób, że przeszkody są źródłem ujemnego pola(odpychającego), natomiast cel emituje pole dodatnie. Ujęte jest to w następujących wzorach:

$$U(q) = U_{att}(q) + U_{rep}(q) \tag{5.1}$$

$$U_{att}(q) = \begin{cases} \frac{1}{2}\xi d^2 & \text{dla } d \leqq d^*_{goal} \\ \xi d^*_{goal} d - \frac{1}{2}(d^*_{goal})^2 & \text{dla } d \geqq d^*_{goal} \end{cases} \tag{5.2}$$

, gdzie $d = d(q, q_{goal})$ jest odległością danego położenia od celu, ε jest stałą określającą poziom przyciągania do celu, natomiast d^*_{goal} określa próg, po którym funkcja z kwadratowej przechodzi w trójkątną. Oddziaływanie przeszkód opisane jest następującym wzorem:

$$U_{rep}(q) = \begin{cases} \frac{1}{2}\eta(\frac{1}{D(q)} - \frac{1}{Q^*})^2 & \text{dla } D(q) \leqslant Q^* \\ 0 & \text{dla } D(q) > Q^* \end{cases} \tag{5.3}$$

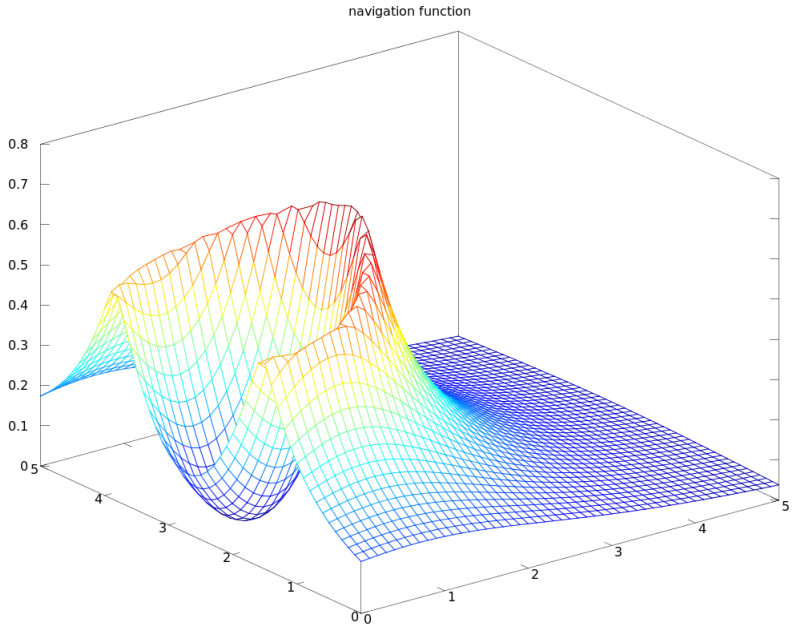
,gdzie Q^* jest zasięgiem pola odpychającego przeszkody, natomiast η odpowiada za siłę tego pola. Kierunek w którym podążać ma robot wyznaczany jest ze wzoru:

$$-\nabla U(q) = -\nabla U_{att}(q) - \nabla U_{rep}(q) \tag{5.4}$$

Powyższe podejście w stosunkowo prosty sposób umożliwa wyznaczenie kierunku bezkolizyjnej ścieżki do celu, jednak posiada pewną dość istotną wadę, mianowicie w pewnych, szczególnych sytuacjach robot może utknąć w minimum lokalnym. Sposób w jaki konstruowana jest funkcja celu w żaden sposób nie wyklucza występowania minimów lokalnych. W przypadku, gdy sterowany robot utknie w lokalnym minimum, konieczna jest reakacja ze strony wyższej warstwy nawigującej maszyną, przykładowo można wykorzystać algorytm błądzenia losowego.

Funkcja nawigacji

Istnieje jednak pewna szczególna funkcja, która posiada tylko globalne minimum, została ona zdefiniowana w publikacjach [18],[19]. Przykładowy wykres funkcji nawigacji jest zamieszczony poniżej (rysunek 5.5). Zasięg oddziaływania minimum globalnego zależy od kilku istotnych parametrów, które zawarte są we wzorze opisującym rozkład sztucznego potencjału. Dla bieżącego położenia robota q funkcja



Rysunek 5.5: Funkcja nawigacji dla $\kappa = 10$

nawigacji zdefiniowana jest następująco:

$$\phi = \frac{(d(q, q_{goal}))^2}{(\lambda\beta(q) + d(q, q_{goal})^{2\kappa})^{\frac{1}{\kappa}}} \tag{5.5}$$

gdzie $\beta(q)$ jest iloczynem funkcji odpychających zdefiniowanych dla wszytskich przeszkód:

$$\beta \triangleq \prod_{i=0}^{N} \beta_i(q) \tag{5.6}$$

Natomiast dla każdej przeszkody($i > 0$) funkcja określona jest następująco:

$$\beta_i(q) = (d(q, q_i))^2 - r_i^2 \text{ dla i} = 1...\text{N, gdzie N jest liczbą przeszkód} \tag{5.7}$$

Powyższa funkcję definiuje się dla każdej z przeszkód o promieniu r_i, której środek znajduje się w punkcie q_i. Jak łatwo zauważyć, funkcja ta przyjmuje ujemne wartości wewnątrz okręgu opisującego przeszkodę, natomiast dodatnie na zewnątrz okęgu. Dodatkowo definiuje się funkcję $\beta_0(q) = -(d(q, q_0))^2 + r_0^2$, w której parametry r_0 oraz q_0 oznaczają odpowiednio promień świata w którym porusza się robot oraz środek tego świata. Wzór 5.5 posiada dwa istotne parametry, λ ogranicza przeciwdziedzinę do przedziału $[0, 1]$, gdzie wartość 0 jest tożsama z osiągnięciem celu, natomiast 1

jest osiągna na brzegu każdej z przeszkód. Drugi parametr κ odpowiednio dobrany powoduje, że blisko celu wykres funkcji ϕ przybiera kształt misy. Zwiększanie parametru κ powoduje przesuwanie globalnego minimum w kierunku położenia punktu docelowego, zwiększa zatem oddziaływanie przyciągającego pola emitowanego przez punkt docelowy.

Wykres funkcji nawigacji przedstawiony na rysunku 5.5 został sporządzony dla następujących parametrów(wszystkie jednostki wyrażone są w metrach):

1. eksplorowany świat ograniczony jest okręgiem o środku w punkcie $q_0(2.7; 3.7)$ i promieniu $r_0 = 7.4$,
2. $\lambda = 0.2$,
3. $\kappa = 10$,
4. przeszkody umiejscowione są w punktach $(2; 2), (3; 3)(4; 4), (3.5; 3.5)$ i mają stały promień, odpowiadający modelowi robota zastosowanemu podczas doświadczeń $r_i = 0.14$,
5. punkt docelowy ma współrzędne $q_g(2.7; 0.675)$,
6. funkcja nawigacji została wykreślona z krokiem 0.1.

5.1.5 Algorytm CVM (`Curvature Velocity Method`)

W pracy inżynierskiej [15] jako docelowy algorytm unikania kolizji wybrany został właśnie `CVM`. Metoda krzywizn i prędkości (ang. *Curvature Velocity Method*) została zaproponowana przez R. Simmonsa w [6], należy do grupy metod bazujących na przeszukiwaniu przestrzeni prędkości, a jej działanie jest zbliżone do techniki dynamicznego okna zaprezentowanej w 5.1.3.

Spośród wszystkich par (v, ω) wybierana jest taka, która osiąga największą wartość funkcji celu. Podejście takie umożliwia uwzględnienie przede wszystkim ograniczeń kinematycznych, ale także i dynamicznych. Wyznaczone przez algorytm sterowanie na następny krok definiuje łuk $c = \frac{\omega}{v}$, po którym będzie poruszał się robot do chwili wyznaczenia kolejnego sterowania.

Funkcja celu została tak skonstruowana, aby preferować prędkości, które prowadzą do celu, ale jednocześnie nie powodują kolizji. Ponadto dołożone zostało kryterium na maksymalizację prędkości liniowej robota. Funkcja celu jest zatem sumą

ważoną trzech składowych:

$$F(v,\omega) = \alpha_1\mathcal{V}(v,\omega) + \alpha_2\mathcal{D}(v,\omega) + \alpha_3\mathcal{G}(v,\omega)) \tag{5.8}$$

gdzie:

- $\alpha_1, \alpha_2, \alpha_3$ współczynniki wagowe,
- funkcja $\mathcal{V}(v,\omega)$ określa stosunek między ocenianą prędkością liniową a maksymalną,
- funkcja $\mathcal{D}(v,\omega)$ określa odległość od najbliższej przeszkody na którą napotka robot poruszający się po trajektorii wyznaczonej przez (v,ω),
- funkcja $\mathcal{G}(v,\omega)$ odpowiada za kierowanie się robota na cel.

Zachowaniem robota można sterować poprzez zmianę wartości współczynników wagowych. W skrajnych przypadkach, gdy któryś ze współczynników zostanie wyzerowany, traci on całkowicie wpływ na trajektorię po której porusza się robot. Szczegółowy opis poszczególnych składowych, jak i samego działania metody został zaprezentowany w paragrafie 5.2.

5.2 Zasada działania CVM

Omawianie mechanizmu wyboru najlepszego sterowania z wykorzystaniem algorytmu *CVM* należy rozpocząć od przypomnienia funkcji celu (5.8)

$$F(v,\omega) = \alpha_1\mathcal{V}(v,\omega) + \alpha_2\mathcal{D}(v,\omega) + \alpha_3\mathcal{G}(v,\omega))$$

W powyższym równaniu składowe odpowiadające za kierowanie na cel oraz za maksymalizację prędkości liniowej zostały zdefiniowane następująco:

$$\mathcal{V}(v,\omega) = \frac{v}{V_{max}} \tag{5.9}$$

$$\mathcal{G}(v,\omega) = 1 - \frac{|\theta_{cel} - T_{alg}\omega|}{\pi} \tag{5.10}$$

gdzie:

- V_{max} – maksymalna prędkość liniowa z jaką może poruszać się robot

- T_{alg} – czas na jaki wystawiane jest obliczone sterowanie
- θ_{cel} – orientacja do celu w układzie współrzędnych robota

Natomiast zdefiniowanie składowej odpowiedzialnej za omijanie przeszkód wymaga określenia pewnych dodatkowych funkcji wykorzystywanych w algorytmie. Należy dokonać przekształcenia opisu przeszkód we współrzędnych kartezjańskich do przestrzeni prędkości. Dokonywane jest to za pomocą odległości do punktu p, w którym wystąpi kolizja z przeszkodą o, jeśli robot będzie poruszał się po trajektorii $c = \frac{\omega}{v}$. Wartość ta jest oznaczana jako $d_c((0,0), p)$, gdzie p jest punktem zderzenia z przeszkodą. Następnie należy zdefiniować funkcję odległości robota od przeszkody:

$$d_v(v, \omega, p) = \begin{cases} d_c((0,0), p_i & \text{dla } v \neq 0 \\ \infty & \text{dla } v = 0 \end{cases} \tag{5.11}$$

Dla danego zbioru przeszkód O zawsze wybierana jest odległość do najbliższej przeszkody leżącej na trajektorii. Zatem funkcję dla danego zbioru przeszkód O można zapisać następująco:

$$D_v(v, \omega, O) = \inf_{o \in O} d_v(v, \omega, o) \tag{5.12}$$

W rzeczywistości jednak informacja o położeniu przeszkód jest ograniczona do pewnego obszaru, który może wynikać z zasięgu czujników w jakie wyposażony jest robot, bądź ze sztucznego ograniczenia zasięgu działania algorytmu w przypadku, gdy możliwy jest dostęp do globalnej informacji o położeniu przeszkód. Zasięg algorytmu będzie w dalszej część pracy oznaczany jako L. Funkcja (5.12) jest zatem ograniczona od góry w sposób następujący:

$$D_{ogr}(v, \omega, P) = \min(L, D(v, \omega, P)) \tag{5.13}$$

Po uprzednim zdefiniowaniu niezbędnych funkcji można podać postać funkcyjną składowej funkcji celu odpowiadającej za unikanie kolizji:

$$\mathcal{D}(v, \omega) = \frac{D_{ogr}}{L} \tag{5.14}$$

Aby sposób obliczania odległości do przeszkody był możliwie jak najprostszy, każda przeszkoda jest reprezentowana w algorytmie jako okrąg o promieniu r (wynikającym z jej rozmiarów) i współrzędnych środka (x_0, y_0). Można zatem dość łatwo wyznaczyć odległość, jaką przebędzie robot poruszający się po łuku wyznaczonym

przez parę (v, ω) do momentu przecięcia z okręgiem. Zostało to zobrazowane na rysunku 5.6.

Dla robota o początkowej orientacji zgodnej z osią OY, poruszającego się po trajektorii o krzywiźnie c przecinającej okrąg opisujący przeszkodę w punkcie $P(x, y)$ prawdziwe są następujące wzory:

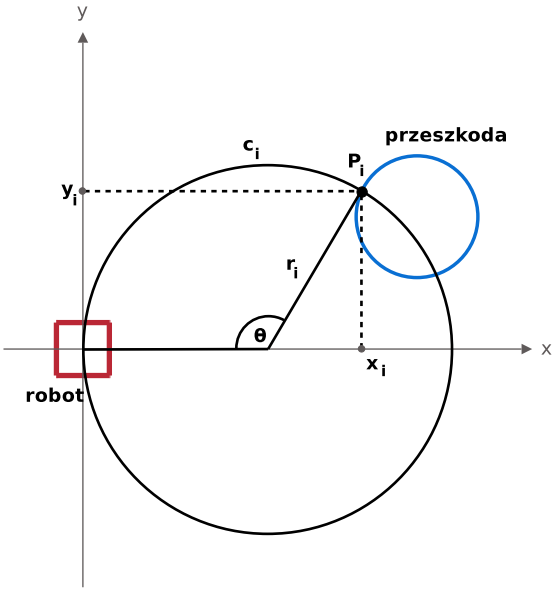
$$\theta = \begin{cases} \text{atan2}(y, x - \frac{1}{c}) & \text{dla } c < 0 \\ \pi - \text{atan2}(y, x - \frac{1}{c}) & \text{dla } c > 0 \end{cases} \tag{5.15}$$

$$d_c((0,0), P) = \begin{cases} y & \text{dla } c = 0 \\ |\frac{1}{c}|\theta & \text{dla } c \neq 0 \end{cases} \tag{5.16}$$

Zastosowana we wzorze (5.15) funkcja *atan2* zwraca wartości z przedziału $[-\pi; \pi]$, zatem uwzględnia w której ćwiartce leży kąt:

$$\text{atan2}(y, x) = \begin{cases} \arctan \frac{|y|}{|x|}\text{sgn}(y) & \text{dla } x > 0 \\ \frac{\pi}{2}\text{sgn}(y) & \text{dla } x = 0 \\ \pi - \arctan \frac{|y|}{|x|}\text{sgn}(y) & \text{dla } x < 0 \end{cases} \tag{5.17}$$

Jednakże obliczanie odległości od przeszkody dla każdego sterowania (v, ω) w sposób omówiony powyżej jest czasochłonne. Stąd konieczne jest kolejne uproszczenie. Łatwo zauważyć, że z punktu $(0, 0)$ w którym znajduje się robot można



Rysunek 5.6: Obliczanie odległości od przeszkody

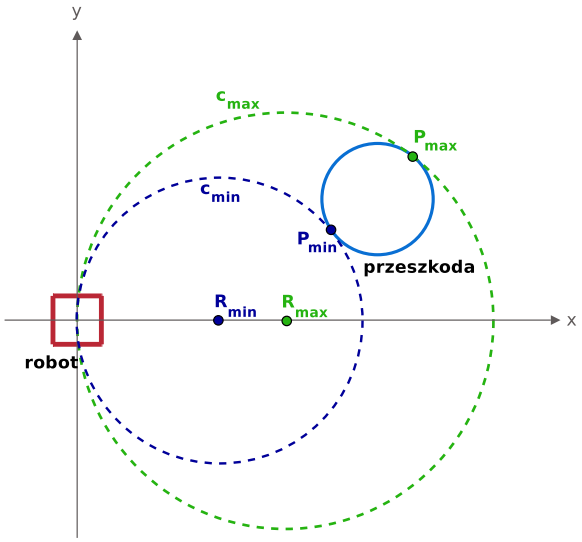
poprowadzić dwa łuki styczne do okręgu opisującego przeszkodę p. Zatem na zewnątrz krzywych stycznych odległość $d_c((0,0),p)$ jest nieskończona. Natomiast dla krzywizn zawierających się pomiędzy łukami stycznymi można dokonać przybliżenia wartością stałą. Omawiana sytuacja została zobrazowana na rysunku 5.7. Aby wyznaczyć okręgi styczne do okręgu opisującego przeszkodę należy znaleźć takie r dla którego poniższy układ równań ma dokładnie jedno rozwiązanie:

$$\begin{cases} (x-x_0)^2+(y-y_0)^2 = {r_0}^2 \\ (x-r)^2+y^2 = r^2 \end{cases} \tag{5.18}$$

Gdzie (x_0, y_0) jest środkiem okręgu opisującego przeszkodę, natomiast r_0 jego promieniem. W wyniku takiego postępowania można otrzymać punkty styczności P_{max} oraz P_{min} oraz odpowiadające im krzywizny c_{min} oraz c_{max}. Ostatecznie można już dokonać przybliżenia odległości po łuku od rozpatrywanej przeszkody p w następujący sposób:

$$d_v(v,\omega,p) = \begin{cases} \min(d_c((0,0),P_{max}), d_c((0,0),P_{min})) & c_{min} \leqslant c \leqslant c_{max} \\ \infty & \text{w p.p.} \end{cases} \tag{5.19}$$

Zaprezentowane rozumowanie prowadzi do sytuacji, w której zbiorowi przeszkód O odpowiada zbiór przedziałów krzywizn o stałej odległości od przeszkody. W dalszej części pracy przedział będzie rozumiany jako trójka liczb postaci $([c_1, c_2], d_{1,2})$, gdzie c_1,c_2 to krzywizny okręgów wyznaczających ten przedział, natomiast $d_{1,2}$ jest odległością zdefiniowaną równaniem (5.19).



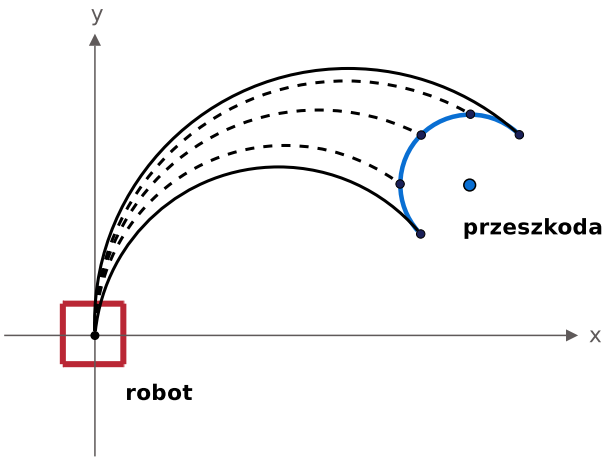
Rysunek 5.7: Zasada wyznaczania przedziału (c_{min},c_{max})

Ze zbioru przedziałów należy następnie utworzyć listę przedziałów $\mathbb{S}$, tak, aby zawierała przedziały s rozłączne o odległości do najbliższej przeszkody (zgodnie z równaniem (5.12)). W oryginalnej pracy na temat *CVM* [6] zaproponowany został algorytm (5.1) realizujący to zadanie.

Algorytm 5.1 tworzy listę rozłącznych przedziałów $\mathbb{S}$

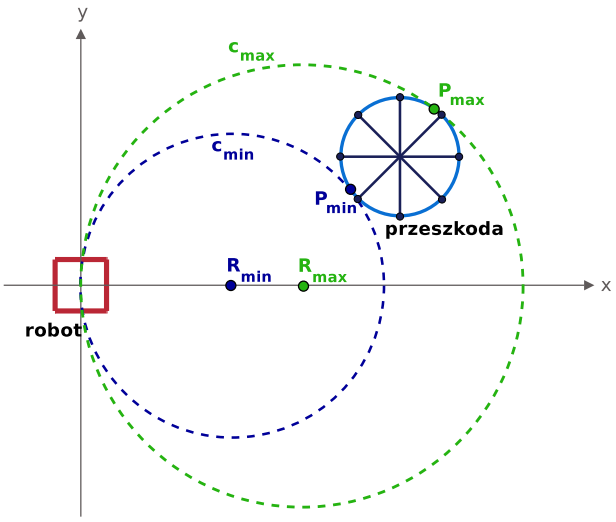
$\mathbb{S} := ((-\infty; \infty), L)$
wybierz kolejny nie dodany przedział $([c_{min}; c_{max}], d)$
for all $s \in \mathbb{S}$ **do**
  if zbiory są rozłączne **then**
    nic nie rób
  else if s zawiera się w $([c_{min}; c_{max}], d)$ **then**
    ustaw odległość $s.d = min(s.d, d)$
  else if s zawiera $([c_{min}; c_{max}], d)$ **then**
    if $d < s.d$ **then**
      podziel s na trzy przedziały:
      $([s.c_{min}; c_{min}], s.d)$, $([c_{min}; c_{max}], d)$, $([c_{max}; s.c_{max}], s.d)$
    else
      nic nie rób
  else if s częściowo zawiera $([c_{min}; c_{max}], d)$ **then**
    if $d < s.d$ **then**
      podziel s na dwa przedziały
      if $c_{max} > s.c_{max}$ **then**
        podziel na przedziały $([s.c_{min}; c_{min}], s.d)$ $([c_{min}; s.c_{max}], d)$
      else
        podziel przedziały $([s.c_{min}; c_{max}], d)$ $([c_{max}; s.c_{max}], s.d)$
    else
      nic nie rób

W zależności od rozmieszczenia przeszkód w środowisku przedziały wyznaczone przez krzywizny c_{min} oraz c_{max} charakteryzują się różną szerokością. Często przybliżenie zbioru długości krzywych należących do $[c_{min}, c_{max}]$ jedną stałą wartością jest niewystarczające. Zazwyczaj przedział ten zawiera zbiór łuków których odległość od przeszkody jest istotnie mniejsza niż ta wynikająca ze wzoru (5.19). Sytuacja taka została zilustrowana na rysunku 5.8.



Rysunek 5.8: Ukazanie sensu zastosowania segmentacji

Jednym z możliwych rozwiązań powyższego problemu jest podzielenie przedziału $[c_{min}, c_{max}]$ na kilka podprzedziałów i do każdego z nich zastosowanie równania (5.19) w celu wyznaczenia odległości od przeszkody. W pracy przyjęto rozwiązanie analogiczne jak opisane w [7]. Pierwszym etapem jest wyznaczenie punktu leżącego na okręgu reprezentującym przeszkodę położonego najbliżej robota zgodnie z metryką euklidesową. Następnie okrąg jest dzielony symetrycznie na k segmentów. Postępowanie takie zostało przedstawione na rysunku 5.9. Dla każdych dwóch sąsiadujących ze sobą punktów leżących pomiędzy punktami styczności P_{min} oraz P_{max} wyznaczane są krzywizny c_1 oraz c_2 i tworzony jest przedział. Jako odległość przyjmowana jest krótsza z odległości zgodnie ze wzorem (5.19). Tak otrzymane przedziały są dodawane do listy za pomocą algorytmu 5.1.



Rysunek 5.9: Efekt podziału okręgu na części

5.3 Algorytm RRT (Rapidly-Exploring Random Tree)

Kolejnym zupełnie odmiennym podejściem w planowaniu bezkolizyjnej ścieżki jest losowe przeszukiwanie dopuszczalnej przestrzeni położeń robota. Szczegółowe informacje na jego temat można znaleźć w [2] oraz w [3]. Algorytm RRT jest wydajnym algorytmem, który w czasie rzeczywistym może wyznaczyć drogę do celu, jednak nie jest ona optymalna pod względem dynamiki, kinematyki ani długości. Podstawową strukturą danych na której operuje algorytm jest drzewo. Jako korzeń przyjmowany jest stan reprezentujący położenie robota w momencie uruchomienia algorytmu. Budowa drzewa polega na dodawaniu kolejnych węzłów do drzewa w kierunku punktu obranego w danym kroku jako docelowy. Tymczasowy stan docelowy generowany jest w następujący sposób:

Algorytm 5.2 Funkcja obliczająca stan docelowy

```
function chooseTarget(goal:state) state;
p = uniformRandom in[0.0...1.0]
if 0.0 < p < goalProb then
  return goal;
else if goalProb < p < 1.0 then
  return RandomState()
```

Na listingu 5.2 użyta została funkcja `RandomState()` zwracająca losowy punkt ze zbioru wszystkich możliwych położeń robota. W najprostszej realizacji może ona zwracać współrzędne punktu dobierane z rozkładem jednostajnym z zadanego przedziału. W każdej iteracji algorytmu tymczasowy stan docelowy obliczany jest na nowo. Aktualne drzewo, na którym operuje algorytm przeszukiwane jest w celu znalezienia węzła `nearest` znajdującego się najbliżej tego celu. Węzeł `nearest` rozszerzany jest w kierunku tegóż celu za pomocą funckji `extend`. Operacja ta w najprostszym ujęciu polega na stworzeniu nowego węzła przesuniętego względem `nearest` o odcinek `rrtStep` w kierunku tymczasowego celu. Istotne jest, aby sprawdzić czy nowo wygenerowany stan znajduje się w dopuszczalnej przestrzeniu stanów $. W najprostszym wariancie można zastosować heurystykę sprawdzającą jedynie czy położenie to nie koliduje z żadną z przeszkód znajdujących się w danej sytuacji oraz czy robot jest w stanie do niego dotrzeć. Bardziej zaawansowane heurystyki powinny sprawdzać czy na przykład podczas przemieszczania się do nowej pozycji nie nastąpi kolizja z dynamiczną przeszkodą. W ogólnym ujęciu algorytm `RRT` wygląda zatem następująco:

Algorytm 5.3 Ogólna zasada algorytmu `RRT`

```
zbuduj model eksplorowanego świata ↦ env:environment;
zainicjuj stan początkowy ↦ initState:state;
zainicjuj stan końcowy ↦ goalState:state;

while goalState.distance(nextState) > minimalDistance do
  wybierz tymczasowy cel: target = chooseTarget(goalState)
  nearestState =  nearest(tree, target)
  extendedState = extend(env, nearest, target)
  if extendedState != NULL then
    addNode(tree, extended)
return tree;
```

W opisie algorytmu 5.3 użyte zostały nie omówione jeszcze funkcje, `distance` zwracająca odległość w sensie euklidesowym między dwoma węzłami drzewa, `addNode` dodająca węzeł do bieżącego drzewa. Parametr `rrtStep` jest natomiast wyrażony w metrach. Tak zaimplementowana wersja algorytmu może być zastosowana jako skuteczne narzędzie do bezkolizyjnej nawigacji robotem, jednak jego wydajność można poprawić poprzez modyfikacje opisane w kolejnym podrozdziale.

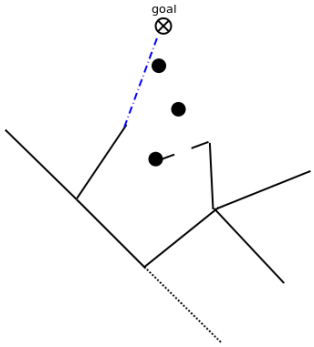
5.3.1 Modyfikacje algorytmu, czyli przejście z `RRT` do `ERRT`

Omawiany algorytm można zmodyfikować pod kątem szybkości działania. Po pierwsze elementy drzewa można przechowywać w strukturach ułatwiających przeszukiwanie drzewa, w literaturze stosowane są w tym celu `KD-drzewa`. Kolejną modyfikacją jest zapamiętywanie losowo wybranych węzłów ze ścieżki znalezionej w poprzednim uruchomieniu algorytmu. W takim wariancie tymczasowy punkt docelowy obierany jest następująco:

Algorytm 5.4 Zmodyfikowana funkcja obliczająca stan docelowy

```
function chooseTargetExt(goal:state) state;
p = uniformRandom in[0.0...1.0]
i = uniformRandom in[0...number_of_waypoints − 1]
if 0.0 < p < goalProb then
  return goal;
else if goalProb < p < goalProb + wayPointProb then
  return WayPoints[i]
else if goalProb + wayPointProb < p < 1.0 then
  return RandomState()
```

W publikacji [2] prawdopodobieństwo podążania do właściwego celu ostatecznie ustalane zostało na poziomie 0.1, natomiast `wayPointProb` wynosiło 0.7. Kolejną



Rysunek 5.10: `RRT` z zapamiętanymi węzłami z porzedniego uruchomiena. Linią kropkowaną zaznaczono gałąź dodaną w kierunku losowego punktu, linią przerywaną w kierunku jednego z elementów z poprzedniej ścieżki, a linią niebiską w kierunku celu.

modyfikacją jest zmiana sposobu obliczania odległości, standardowo w algorytmie wykorzystywana jest odległość pomiędzy liściem drzewa a punktem docelowym. Aby uniknąć nadmiernego rozrastania drzewa, odległość może uwzględniać dodatkowo odległość liścia od korzenia pomnożoną przez określony współczynnik wagowy.

5.4 Zalety algorytmu RRT w stosunku do CVM

Podczas kontynuacji prac nad rozgrywkami RoboCup zdecydowano się na zmianę algorytmu wyznaczającego bezkolizyjną ścieżkę. Zmiana została wymuszona decyzją o zmianie modelu sterowanego robota. Bazę jezdną o napędzie różnicowym zastąpiono holonomiczną. Algorytm *CVM* jest algorytmem dedykowanym do robotów o napędzie różnicowym, doskonale można ująć w nim ograniczenia na dopuszczalne prędkości. Jednak sterowanie za jego pomocą robotem holonomicznym nie pozwala na pełne wykorzystanie możliwości robota. Dodatkowo jak zostało udowodnione w pracy inżynierskiej [15], *CVM* w przypadku środowisk z przeszkodami dynamicznymi osiągał skuteczność na poziomie 80%. Kolizje występowały głównie w sytuacjach, kiedy przeszkoda uderzała w robota z boku lub z tyłu. Wynikało to z faktu, że trajektoria ruchu takiej przeszkody znajdowała się poniżej osi *OY* w układzie współrzędnych związanym ze sterowanym robotem. Przeszkody takiego typu nie były uwzględnianie przez algorytm. *RRT* pozwala na uwzględnianie przeszkód rozsianych po całej planszy. Dodatkowo umożliwia poruszanie się po bardziej złożonych trajektoriach. Kolejną zaletą *RRT* jest prostota w ograniczaniu przestrzeni z której losowane są tymczasowe punkty docelowe. Dzięki temu w prosty sposób można ograniczyć poruszanie się robota tylko do wnętrza boiska, w przypadku *CVM* istniała konieczność modelowania bandy boiska jako zbioru okręgów, mnożyło to ilość obiektów z jakimi należało detekować kolizje. Kolejnym problemem *CVM* jest nawigacja do celu znajdującego się za robotem, w pracy inżynierskiej wprowadzono modyfikację, zmieniającą parametr odpowiedzialny za kierowanie robota na cel, uzależniając go od orientacji robota względem celu. Intencją było wymuszenie obrotu robota w kierunku celu, w sytuacji gdy orientacja do celu jest duża. Modyfikacja poprawiła skuteczność jednak, nawet dla najlepszego zestawu wag wynosiła ona 80%. W przypadku bazy holonomicznej obracanie się w kierunku punktu docelowego nie jest konieczne, stąd takie sterowanie nie jest optymalne.

Rozdział 6

Testy zaimplementowanej wersji algorytmu RRT

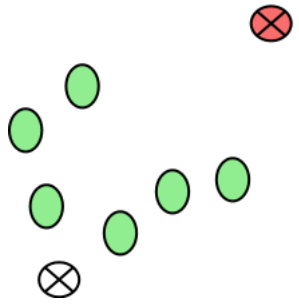
6.1 Opis implementacji algorytmu RRT

W trakcie testów wersji podstawowej algorytmu *RRT* opisanej w 5.3 zdecydowano się na dodatkowe modyfikacje algorytmu, część zaczerpnięto z 5.3.1 dodano też własne modyfikacje, które miały na celu poprawienie wydajności algorytmu i skrócenie czasu obliczeń. Zdecydowano się na implementację algorytmu w wersji z zapamiętywaniem części węzłów ze ścieżki wyznaczonej w poprzednim uruchomieniu algorytmu (pod warunkiem, że punkt docelowy nie uległ zmianie) i traktowanie ich jako tak zwanych `way points`. Jednak z uwagi na specyfikę eksplorowanego środowiska (brak wąskich korytarzy, przeszkody są wypukłe) zdecydowano się na eliminowanie z zapamiętanej ścieżki węzłów których odległość między sąsiadami jest mniejsza niż $5cm$. Zrezygnowano z przechowywania węzłów w postaci `KD-drzewa`.

Dodatkowo wprowadzono ograniczenie na maksymalną liczbę węzłów algorytmu. Sytuacja na planszy podczas rozgrywki szybko ulega zmianie, stąd wyznaczanie kompletnej ścieżki do celu pozbawione jest sensu, jeśli ma to być czasochłonne. Przy ponownym uruchomieniu algorytmu sytuacja może wyglądać inaczej i algorytm szybciej wyznaczy ścieżkę. Jeśli uruchomienie algorytmu nie zakończyło się wyznaczeniem kompletnej ścieżki do celu jako punkt docelowy dla robota wybierany jest węzeł znajdujący się najbliżej celu.

Zauważono także, że przy małym prawdopodobieństwie budowy ścieżki do celu,

drzewo może ulegać nadmiernemu rozbudowaniu przy samym korzeniu. Ma to miejsce w sytuacji, gdy tymczasowe punkty losowane są z otoczenia korzenia, kolejne wylosowane węzły mogą powodować poszerzenie gałęzi, a tak rozbudowane gałęzie mogą znajdować się w podobnej odległości od celu. W efekcie algorytm przez dłuższy czas może próbować wyznaczać kilka ścieżek prowadzących do celu, co nie jest pożądane. Sytuację taką dla dwóch gałęzi przedstawiono na rysunku 6.1. W



Rysunek 6.1: Drzewo wyznaczające ścieżkę do celu.

zaimplementowanej wersji ograniczono liczbę wezłów potomnych korzenia do 4.

Podobnie jak w pracy inżynierskiej [15] zdecydowano się na naiwną predykcję położenia dynamicznych przeszkód. Ponieważ baza jezdna robota zbliżona jest do okręgu w algorytmie jest ona analizowana jako okrąg o środku znajdującym się w danym położeniu robota (przeszkody) i promieniu $r = 2 * (r_{base} + l_{dribbler})$, gdzie r_{base} jest promieniem bazy jezdnej robota, a $l_{dribbler}$ długością dribblera. Do zbioru rzeczywistych przeszkód dodawne są przeszkody, których środek przesunięty jest o drogę jaką może pokonać dana przeszkoda w czasie trwania jednego uruchomienia algorytmu. Podczas planowania ścieżki promienie przeszkód powiększane są dodatkowo o margines bezpieczeństwa, ma on za zadanie wyeliminowanie ewentualnych kolizji wynikających z poślizgu robota podczas hamowania, czy opóźnienia przy zmianie kierunku jazdy. Główną zaletą RRT jest jego szybkość, stąd przewidywanie położenia przeszkody na 1 krok w przód okazało się wystarczające dla zastosowanych środowisk testowych. W przypadku CVM konieczne było dodawanie "wirtualnych" przeszkód symulujących położenie robotów za 15, 30 oraz 45 kroków symulatora.

W zaimplementowanej formie algorytm ma następujący przebieg:

Algorytm 6.1 Wersja RRT z wprowadzonymi modyfikacjami

ogranicz maksymalną przestrzeń do rozmiarów boiska
if punkt docelowy jest poza dopuszczalnymi współrzędnymi **then**
  zwróć błąd *waypoints*
if w poprzednim uruchomieniu znaleziono ścieżkę do celu **then**
  zainicjuj listę *waypoints*
dodaj statyczne oraz dynamiczne przeszkody;
if robot uległ kolizji (nie są brane pod uwagę przewidywane położenia przeszkód) **then**
  tak zwróć błąd;
if robot dojechał do celu **then**
  tak zwróć sukces;
if punkt docelowy jest w obrębie przeszkody **then**
  zwróć błąd;
if punkt docelowy jest bezpośrednio osiągalny **then**
  zwróć go jako wynik działania algorytmu;
while `goalState.distance(nextState)` > `minimalDistance` && `treeSize` ¡ `maxNodeNumber` **do**
  wybierz tymczasowy cel: `tmp_target` = `chooseTarget(goalState)`
  nearestState = `nearest(tree, target)`
  `extendedState` = `extend(env, nearest, target)`
  if `extendedState` ! = `NULL` **then**
    odblokuj ustawianie `tmp_target` jako `goalState`
    `addNode(tree, extended)`
  else if `tmp_target` należy do zbioru *waypoints* **then**
    usuń punkt ze zbioru *waypoints*
  else if `tmp_target` jest właściwym punktem docelowym **then**
    zablokuj ustawianie `tmp_target` jako `goalState`
if `nearestState.distance(goalState)` > `minimalDistance` **then**
  return węzeł znajdujący się najbliżej celu
else
  zapamiętaj ścieżkę do celu
  return węzeł bezpośrednio osiągalny znajdujący się najbliżej celu;

W opisie algorytmu zastosowano fukcje `chooseTarget` oraz `extend`, których działanie również zostało lekko zmodyfikowane w stosunku do oryginalnej wersji algorytmu. Ich działanie zostało przedstawione poniżej:

Algorytm 6.2 funckcja `chooseTarget` wyznaczająca tymczasowy punkt docelowy

`function` *chooseTarget*(*goal:state*) *state*;
p = `uniformRandom in`[0.0...1.0]
if wyznaczono ścieżkę w poprzednim uruchomieniu algorytmu **then**
  if $0.0 < p < goalProb$ **then**
    if wybór właściwego punktu docelowego jest zablokowany **then**
      return `randomPoint`;
    return właściwy punkt docelowy
  else if $p < goalProb + wayPointProb$ **then**
    i = `uniformRandom in`[0...$number\_of\_waypoints - 1$]
    return *WayPoints[i]*
  else
    return *RandomState()*
else
  if $0.0 < p < goalProb$ `AND` wybór właściwego punktu docelowego nie jest zablokowany **then**
    return `goal`;
  else
    return *RandomState()*

Algorytm 6.3 funckcja `extend(nearest,tmp_target)` rozszerzejąca drzewo w kierunku zadanego punktu

podczas detekcji kolizji przeszkody powiększane są o margines bezpieczeństwa oraz uwzględniane są dodatkowe przeszkody wynikające z ruchu robotów
if punkt docelowy leży w obrębie przeszkody **then**
  return `NULL`
wyznacz nowy węzeł (`newNode`) odległy od `nearest` w kierunku `tmp_target`
if wyznaczony punkt leży w obrębie przeszkody **then**
  return `NULL`
return `newNode`

6.1.1 Parametry zastosowanego algorytmu

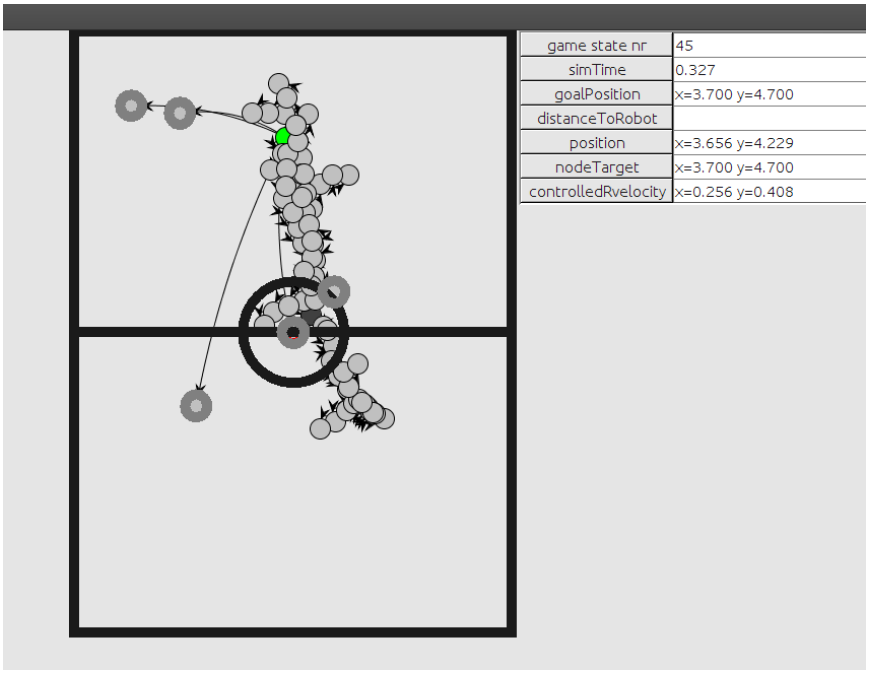
Zaimplementowowaną wersję algorytmu RRT można w łatwy sposób konfigurować poprzez zmiane kluczowych parametrów takich jak:

1. włączanie/wyłączanie predykcji ruchu przeszkod,
2. prawdopodobieństwo losowania tymczasowego punktu docelowego należącego do wayPoints (*goalProb*) lub jak punkt właściwy punkt docelowy *wayPoint-Prob*,
3. rozmiary dopuszczalnej przestrzeni z jakiej losowany jest tymczasowy cel, wynikajace z rozmiaru swiata lub innych ograniczeń,
4. ograniczenia na przestrzeń przeszukiwań wynikające z zaistniałej sytuacji na planszy, bądź z rodzaju zadania,
5. odległość po przekroczeniu której uznajemy, że robot dojechał do celu,
6. ścieżka z poprzedniego wywołania algorytmu, przekazywana tylko wtedy, gdy punkt docelowy nie zmienił się,
7. maksymalna liczba węzłów drzewa RRT,
8. maksymalna liczba węzłów doczepionych do korzenia,
9. odległość między wayPoints; z przekazanej ścieżki scalane są węzły między którymi odległość jest mniejsza niż wartość parametru,
10. *rrtStep* odległość o jaką rozszerzany jest węzeł leżący najbliżej tymczasowego celu w jego kierunku,

6.2 Aplikacja analizująca działanie algorytmu

W trakcie implementacji algorytmu RRT wyniknęła konieczność napisania programu umożliwiającego sprawdzenie poprawności działania, w tym celu powstała aplikacja RRT_debug. Umożliwia ona wczytanie serii drzew zapisanych w formacie xml. Program został napisany w języku Java, a do reprezentacji węzłów drzewa wykorzystano bibliotekę JUNG-Java Universal Network/Graph Framework. Aplikacja umożliwia

śledzenie konstruowanych przez RRT drzew. Ilustracja 6.2 zawiera przykładowy zrzut ekranu z uruchomionej aplikacji. Program umożliwia wyświetlanie dodatkowych informacji o zaznaczonym węźle drzewa, oraz wykonywanie zbliżenia celem podglądu szczegółowo wybranej części drzewa. W prawym górnym rogu wyświetlane są infomrmacje o czasie symulacji w jakim zostało stworzone drzewo, pozycji docelowej drzewa, tymczasowej pozycji w kierunku której rozszerzano drzewo przy dodawaniu zaznaczonego węzła, współrzędnych zaznaczonego węzła na planszy oraz prędkości sterowanego robota.



Rysunek 6.2: Aplikacja RRT_debug.

6.3 Opis środowisk testowych

Algorytm RRT poddano wstępnym testom w aplikacji RRT_debug. Po sprawdzeniu poprawności działania zdecydowano się na porównanie algorytmu pod względem skuteczności z metodą CVM opracowaną w pracy inżynierskiej [15]. W tym celu zastosowano ten sam zestaw środowisk testowych (załącznik C strona 88). W związku ze zmianą wymiarów boiska, sytuacje zaczerpnięte z pracy inżynierskiej zostały przeskalowane, odpowiednio wydłużony został także maksymalny czas. Dla środowisk dynamicznych wyniósł on 12.47$[s]$, a dla statycznych 8.73$[s]$. Po przeskalowaniu

droga do piłki nie przekraczała $3[m]$, maksymalną prędkość ograniczono do $1[\frac{m}{s}]$. Jadąc prosto do celu z maksymalną prędkością robot powinien przejechać dystans w maksymalnie $3[s]$. Testy przeprowadzono na 10 środowiskach statycznych oraz na 10 środowiskach dynamicznych. W przypadku eksperymentów w środowisku statycznym dokonano pewnego uproszczenia. Zrezygnowano z rozróżnienia eksperymentów na sytuacje, gdy robot zwrócony jest w stronę celu lub przeciwnie. Metoda opracowana w pracy inżynierskiej była zdecydowanie mniej skuteczna w sytuacjach, gdy cel znajdował się za robotem. Wynikało to ze specyfiki samego algorytmu, jak i rodzaju zastosowanej bazy jezdnej (ograniczenia wynikające z więzów nieholonomicznych). Algorytm RRT jest niewrażliwy na położenie punktu docelowego względem sterowanego robota, zastosowana baza jezdna także pozwala na poruszanie się w dowolnym kierunku. Zadaniem sterowanego robota było dojechanie do piłki, tak aby uniknąć kolizji z innymi robotami, bandą boiska oraz bramkami. Aby warunku eksperymentu były zbliżone do tych jakim poddano algorytm CVM zdecydowano się wyłączyć regulator PID w sterowniku. Zdecydowano się na zbadanie działania algorytmu dla różnych prawdopodobieństw wyboru tymczasowego punktu docelowego. Na każdej planszy algorytm uruchomiono 20-krotnie dla różnego zestawu prawdopodobieństw $goalProb$, $wayPointProb$. Przetestowano wszystkie możliwe warianty prawdopodobieństw z krokiem 0.1 których jest dokładnie 65. Ponumerowane zestawy współczynników zamieszczone są w załączniku C na stronie 88. Pomiarom zostały poddanne następujące wielkości:

1. procent eksperymentów zakończonych sukcesem dla każdego zestawu wag,
2. czas dojazdu robota do celu, przy czym uwzględnione zostały jedynie sytuacje w których robot zakończył eksperyment sukcesem,
3. średni czas obliczeń jednego uruchomienia algorytmu, z pominięciem sytuacji w których cel był bezpośrednio osiągalny,
4. maksymalna długość wyznaczonej ścieżki w czasie trwania jednego eksperymentu, dla testowanego zestawy prawdopodobieństw,
5. maksymalna liczba węzłów drzewa wyznaczana w analogiczny sposób jak powyżej.

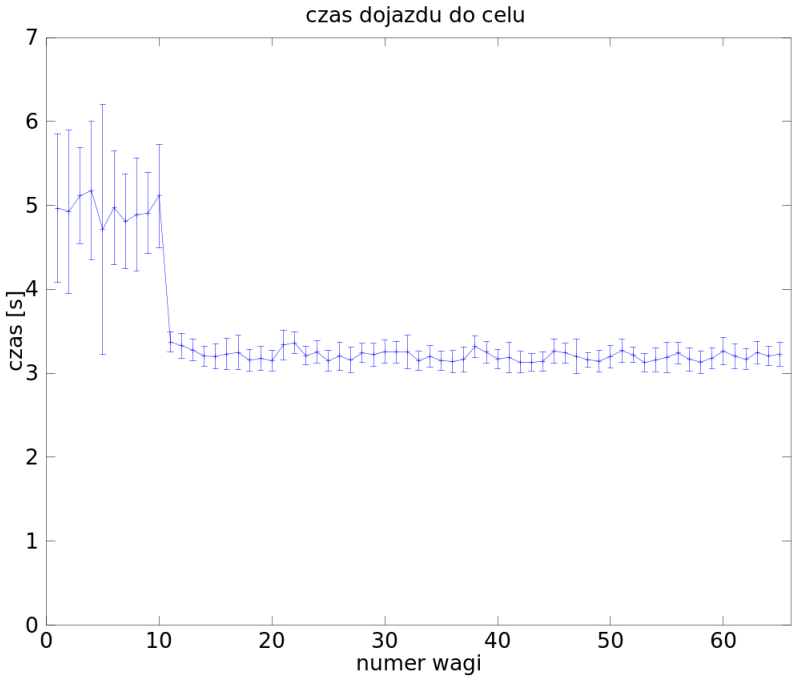
Dodatkowo zamieszczony został wykres przedstawiający ile razy dla danego zestawu wag i konkretnej planszy choć raz znaleziono ścieżkę do celu.

W trakcie przeprowadzanych eksperymentów krok, z jakim pracował symulator został ustawiony na $0.001[s]$, przy takiej wartości `real-time factor` kształtował się w okolicach $0.9 - 1.0$ na maszynie wyposażonej w procesor `Intel i5-2520M` z zegarem $2.5[GHz]$. W przypadku środowisk statycznych daje to maksymalny czas pełnego eksperymentu na poziomie 31 godzin, w przypadku środowisk dynamicznych górne ograniczenie wyniosło 45 godzin.

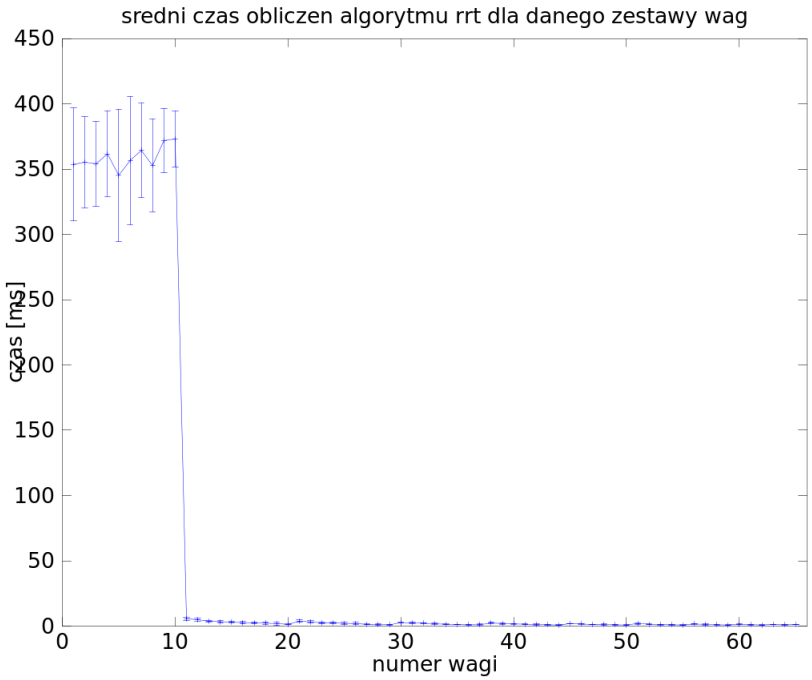
6.4 Wyniki eksperymentów w środowisku statycznym

Poniższe wykresy prezentują wyniki otrzymane dla środowisk statycznych. Można zauważyć, że dla współczynników poniżej numeru 11, czyli gdy prawdopodobieństwo budowania drzewa do celu jest równe 0, algorytm przy odpowiednio długim uruchomieniu także potrafi wyznaczyć ścieżkę do celu, jednak czas obliczeń samego algorytmu jest długi, rzędu $350[ms]$, a czas dojazdu jest na poziomie $5[s]$. Przy takich parametrach, znalezienie ścieżki do celu wymaga zbudowania drzewa zawiarającego w granicach 800 węzłów. Dla większości zestawów wag, populacja wyników jest skupiona przy wartości 10 co oznacza 100% skuteczność, dla porównania na wykresie 6.3, jako ostatni współczynnik wagowy zamieszczono najlepszy wynik uzyskany przez algorytm `CVM`. Drugi kwarty jest równy 9.47, co także oznacza wysoką skuteczność, jednak nieco niższą niż `RRT`. na wykresach można jeszcze zauważyć, że prawdopodobieństwo `wayPointProb` nie ma praktycznie wpływu na czas obliczeń, ani na czas dojazdu do celu. Natomiast jego wzrost pociąga za sobą zmniejszenie rozmiaru budowanego drzewa jak i samej ścieżki do celu.

Rysunek 6.3: Liczba eksperymentów zakończonych sukcesem. Symbolem *o* oznaczone są wartości których odchylenie od populacji przekracza trzykrotnie rozstęp ćwiartkowy (IQR), natomiast symbolem + wartości pomiędzy 1.5IQR, a 3IQR



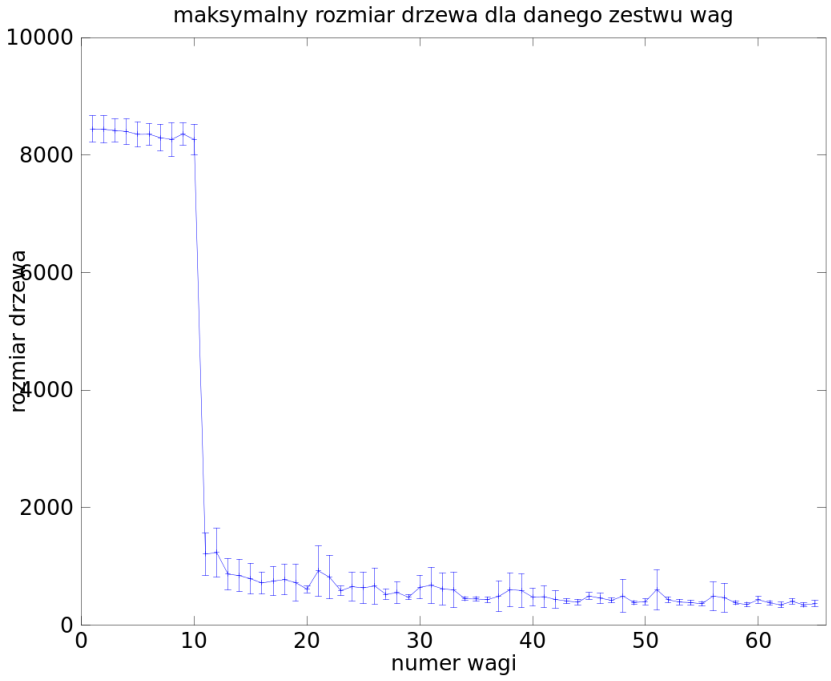
Rysunek 6.4: Czas dojazdu do celu.



Rysunek 6.5: Czas jednego uruchomienia RRT .

Rysunek 6.6: Liczba znaleźionych ścieżek prowadzących do celu.

Rysunek 6.7: Maksymalny rozmiar ścieżki prowadzącej do celu.



Rysunek 6.8: Maksymalny rozmiar drzewa znalezionego przez RRT.

zamiescic przykładowe trajektorie w srodowisku statycznym

6.5 Wyniki eksperymentów środowisku dynamicznym

Rezultaty otrzymane podczas testów z ruchomymi przeszkodami zostały zaprezentowane poniżej. Na rysunku 6.9 ostatnia waga o numerze 66 odpowiada najlepszemu wynikowi dla algorytmu CVM. Można zauważyć, że liczba sukcesów dla wag poniżej numeru 11 jest wyższa niż w przypadku eksperymentów statycznych. Jest to spowodowane tym, że po pewnym czasie piłka staje się bezpośrednio osiągalna i algorytm unikania kolizji nie jest konieczny. Podobnie jak w przypadku eksperymentów statycznych czas obliczeń w tym przypadku jest wysoki rzędu 320$[ms]$, a czas dojazdu przekracza 6$[s]$. Najlepsze wyniki pod względem liczby sukcesów otrzymano dla następujących wag:

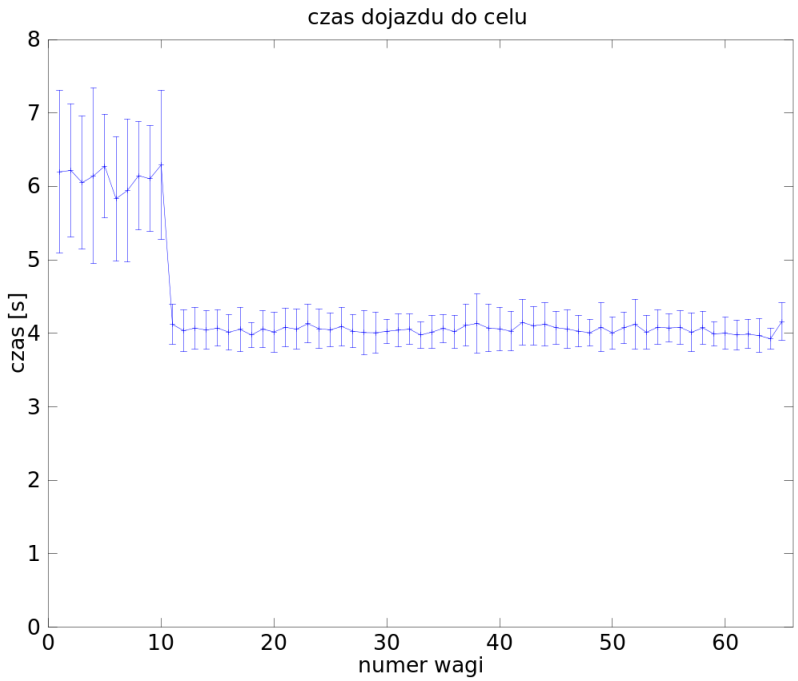
- $goalProb = 0.1$, $wayPointProb = 0$
- $goalProb = 0.1$, $wayPointProb = 0.4$

- $goalProb = 0.4$, $wayPointProb = 0.4$
- $goalProb = 0.7$, $wayPointProb = 0.3$
- $goalProb = 0.7$, $wayPointProb = 0.3$

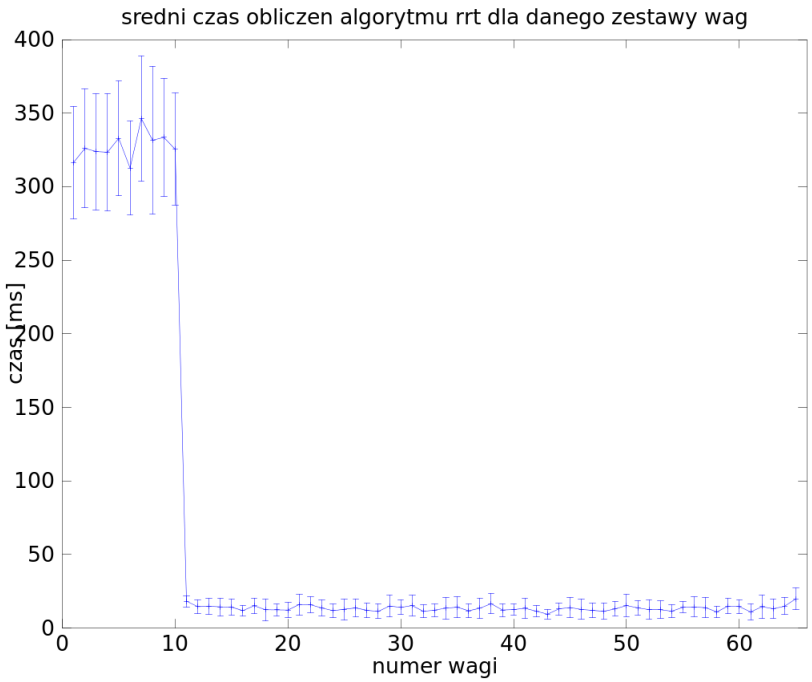
Drugi kwartyl przyjmuje wtedy wartość w okolicach 9.5, co oznacza, że ponad połowa eksperymentów zakończyła się 9 lub 10 razy sukcesem. Dla porównania w przypadku algorytmu CVM kwartyl rzędu drugiego ma wartość 8.

Rysunek 6.9: Liczba eksperymentów zakończonych sukcesem. Symbolem *o* oznaczone są wartości których odchylenie od populacji przekracza trzykrotnie rozstęp ćwiartkowy (IQR), natomiast symbolem + wartości pomiędzy 1.5IQR, a 3IQR

zamiescic przykładowe trajektorie w środowisku dynamicznym



Rysunek 6.10: Czas dojazdu do celu.



Rysunek 6.11: Czas jednego uruchomienia RRT .

Rysunek 6.12: Liczba znaleźionych ścieżek prowadzących do celu.

Rysunek 6.13: Maksymalny rozmiar ścieżki prowadzącej do celu.

Rysunek 6.14: Maksymalny rozmiar drzewa znalezionego przez RRT.

Rozdział 7

Realizacja złożonych funkcji przez robota

Problem sterowania i koorydynacji działań w obrębie drużyny robotów nie jest zadaniem trywialnym. Po pierwsze środowisko w którym znajdują się roboty jest dynamiczne, odrębym problemem jest analizowanie na bieżąco działań drużyny przeciwnej. Przed takim algorytmem stawiane są następujące oczekiwania:

- koordynacja działań w obrębie drużyny mająca za zadanie osiągnięcie długofalowych celów (oddanie strzału na bramkę, a w efekcie zwycięstwo),
- reagowanie w czasie rzeczywistym (*on-line*) na zachowania drużyny przeciwnej,
- budowa modelu dynamicznego świata na podstawie niepewnej informacji z czujników,
- modułowa architektura, umożliwiająca łatwą konfigurację i adaptację do bieżącej sytuacji.

Przed przystąpieniem do prac nad aplikacją sterującą drużyną robotów dokonano przeglądu stosowanych rozwiązań przez uczestników mistrzostw. Okazało się, że wstępnie opracowana architektura jest zbieżna z już istniejącym modelem nazwanym `STP - Skill Tactics Play` szerzej opisanym w publikacji `STP: Skills, tactics and plays for multi-robot control in adversarial environments` [11] stworzonym przez grupę naukowców z Carnegie Mellon University. Rozwiązanie to było z powodzeniem stosowane przez drużynę `CMD Dragons` w kilku mistrzostwachRoboCup. Architektura `STP` rozwiązuje problem koordynacji i planowania działań w

obrębie drużyny. Głównym założeniem jest generowanie planów na trzech poziomach:

1. `Play`, czyli strategia działań dla całej drużyny, przydzielająca robotom wykonywanie kolejnych `Tactics`,
2. `Tactic`, zawierający plan zachowań pojedynczego zawodnika, zapisany w formie maszyny stanów `SSM` (`skill state machine`),
3. `Skills`, obejmujący sterowanie robotem zmierzające do wykonania pojedynczej akcji (podążanie do wyznaczonego punkty, przechwycenie piłki etc.).

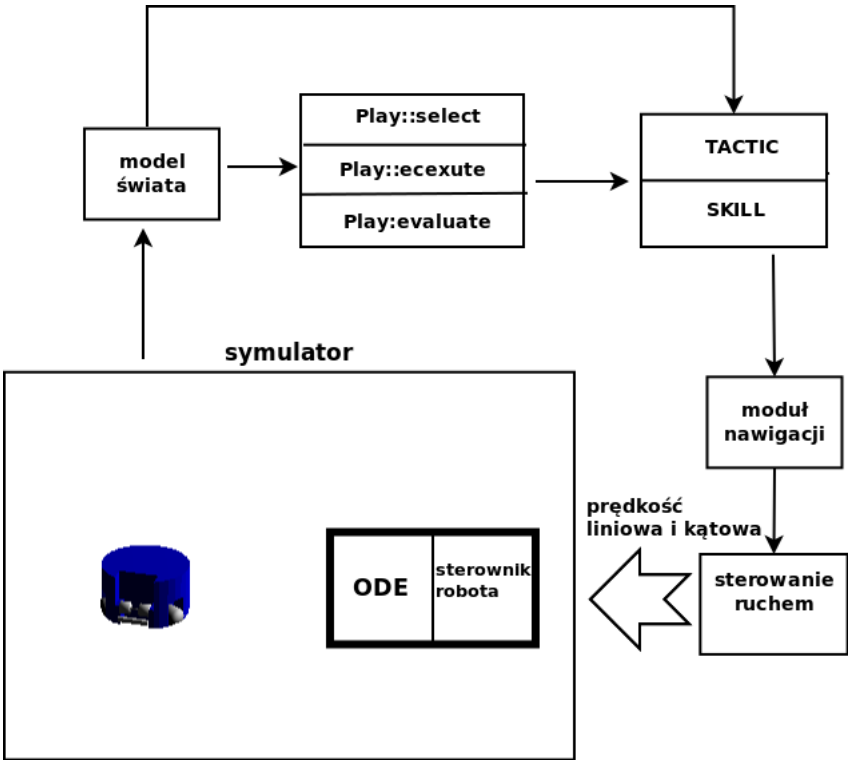
Każdy

7.1 Opis algorytmu sterujacego zawodnikiem

Nazwa `STP` odnosi się bezpośrednio do podejścia rozwiązującego problem planowania działań zawodników w obrębie drużyny. Jak sam skrót wskazuje, podejście zakłada planowanie zachowań na kilku poziomach. Hierarchia ma za zadanie ułatwić adaptację i paramteryzowanie poszczególnych działań. Realizacja architektury wymaga jednak istnienia następujących modułów nie związanych bezpośrednio z `STP`:

1. zawierającego informacje o świecie,
2. umożliwiającego ocenę sytuacji na planszy (wyznaczającego atrakcyjności zachowań, punktów docelowych etc),
3. planowania; moduł ten powienien być odpowiedzialny za koordynację działań drużyny (tutaj właściwie realizowane jest `STP`),
4. modułu odpowiedzialnego za nawigację robota; moduł ten powinien być odpowiedzialny za tworzenie bezkolizyjnej ścieżki prowadzącej do zadanego celu,
5. moduł sterowania ruchem robota; moduł ten powinien wyznaczyć optymalne prędkości prowadzące do zadanego punktu (problem bezwładności robota, wyhamowanie przed punktem docelowym etc.),
6. moduł bezpośrednio odpowiedzialny za sterowanie warstwą fizyczną robota (zadawanie prędkości liniowej i kątowej, uruchamianie urządzenia do prowadzenia piłki –*dribbler-a*, kopnięcie piłki).

Wszystkie warstwy algorytmu oraz przepływ informacji zostały zobrazowane na rysunku 7.1. W trakcie niniejszej pracy moduł nawigacji został zrealizowany w oparciu o opisany w paragrafie 5.3 algorytm `RRT`, sterowanie ruchem robota polega na obliczaniu trapezoidalnego rozkładu prędkości opisanego w 4.2, natomiast kontrolę nad warstwą fizyczną robota sprawuje zaimplementowany w ramach pracy magisterskiej i omówiony w 3.3.2 sterownik modelu robota włączony do symulatora. Model świata zrealizowano w najprostszej formie, poprzez udostępnianie jedynie informacji o prędkości i położeniu wszytskich elementów na boisku, w bieżącym momencie.



Rysunek 7.1: Architektura aplikacji sterującej drużyną.

Zasada działania podejścia `STP` została zaprezentowana na listingu poniżej:

Algorytm 7.1 Główna pętla planowania `STP`

pobierz dane z czujników
zaktualizuj model świata
na podstawie aktualnej sytuacji na planszy wyznacz bieżącą strategię `P`
for numer robota $i = 1 \rightarrow N$ **do**
  $(T_i, Tparams_i) \leftarrow$ WyznaczTaktykę(P, i)
  $(SSM_i, Sparams_i) \leftarrow$ WykonajTaktykę$(T_i, TParams_i)$
  if T_i jest nową taktyką **then**
    $S_i \leftarrow SSM_i(0)$
  wykonaj kolejne zadanie z automatu SSM_i i przejdź do kolejnego stanu S'
  wyznacz sterowanie dla robota
  wyślij odpowiednią komendę do robota

7.1.1 Omówienie warstwy `Play`

Poziomem najwyżej w hierachii jest `Play`. Przez to pojęcie rozumiany jest plan gry dla całej drużyny, uwzględniana jest tutaj koordynacja poczynań pomiędzy zawodnikami. Plan zakłada przydział roli dla każdego zawodnika. W obrębie danego planu zawodnik wykonuje swoją rolę, aż do momentu zakończenia danego planu lub wyznaczenia kolejnego. Na tym poziomie określane są parametry używane przy realizacji taktyk. Dodatkowo każdy poziom poniżej `Play` może określać nowe parametry wykorzystywane przy wykonywaniu zadań na niższym poziomie. Wykonywanie każdego planu `Play` wymaga spełnienia określonych predykatów, np. czy mamy rozpoczęcie gry z autu, czy wystąpił rzut rożny. Poniżej zamieszczony został przykładowy plan atakujący dla drużyny zaczerpnięty z oryginalnej publikacji:

Listing 7.1: Przykładowy plan atakujący

```
1        APPLICABLE: offense
2        DONE: aborted, !offense
3
4        ROLE1
5          shoot A
6          none
7        ROLE2
```

```
8           defend point {-1400 250} 0 700
9           none
10        ROLE3
11          defend lane {B 0 -200} {B 1175 -200}
12          none
13        ROLE4
14          defend point {-1400 -250} 0 1400
15          none
```

Na listingu wyszczególnione są warunki konieczne do realizacji danego planu, oraz warunki powodujące jego zakończenie. W tym przypadku są to proste wyrażenia, mogą one przybierać bardziej złożone formy logiczne. Same predykaty mogą także być bardziej wyrafinowane, na przykład czy piłka znajduje się na aucie, czy drużynie przydzielono rzut rożny etc. Plan jest przewidziany dla czterech ról. Zawodnik do którego przydzielona jest rola pierwsza, ma za zadanie oddać strzał. Natomiast pozostali zawodnicy mają zadanie bronić punktu lub obszaru. Każda ktrola składa się w tym przypadku tylko z jednej taktyki.

7.1.2 Omówienie warstwy `Tactics`

Przez `Tactics` rozumiany jest plan działań dla jednej roli. Można rozumieć to jako plan działań na szczeblu robota prowadzący do osiągnięcia pożądanego w danej sytuacji efektu. Przykładem może być strzał na bramkę. Robot dostaje polecenie oddania strzału na bramkę, zatem plan jego poczynań ma doprowadzić do sytuacji, w której osiągnie pozycję umożliwiającą strzał na bramkę z zadanym powodzeniem. W oryginalnej wersji algorytmu zbiór taktyk podzielony został na aktywne (wymagające posiadania piłki) oraz nieaktywne, czyli nie wymagające kontaktu z piłką. W danym momencie rozgrywki tylko jeden robot może mieć przydzieloną aktywną taktykę. Z chwilą zakończenia wykonywanej ostatnio taktyki robot rozpoczyna wykonywanie kolejnej, nałożonej na niego przez warstwę `Play`. Jak już wspomniano wcześniej taktyka zapisana jest w postaci maszyny stanów `SSM` (`skill state machine`). Wykonując daną taktykę robotowi zlecane jest wykonanie kolejnych zachowań Zachowania mogą być stosowane w różnych taktykach, jednak ich wykonanie może przynosić odmienne efekty. Przykładem może być oddanie strzału w przypadku podania i w przypadku próby zdobycia gola. Plan na szczeblu pojedynczego robota (sekwencja taktyk) jest wykonywany do momentu zmiany planu gry całej druzyny.

Przykładowe plany działań dla robotów:

- strzał na bramkę,
- podanie piłki,
- odebranie podania,
- blokowanie innego robota,
- wyjście na pozycję,
- bronienie pozycji,
- dryblowanie z piłką.

7.1.3 Realizacja `Skills`

Pojęcie `Skills` odnosi się do konkretnych umiejętności robota. Każdy `skill` traktowany jest jako osobne zachowanie. Rozróżnić można następujące, podstawowe umiejętności robota:

- doprowadzenie piłki do celu,
- przemieszczenie robota do celu,
- podążanie za innym robotem,
- obrót z piłką,
- oddanie strzału.

Z pojedynczych zachowań budowana jest wspomniana wcześniej maszyna stanów, tworząca taktykę. Na tym szczeblu zachowanie robota zmieniane jest w każdym kroku gry. Z każdego zadania, w każdym momencie określone musi być przejście do nowego zadania. Możliwe jest także kontynuowanie tego samego zadania, lub ponowne jego wykonanie. Przykładowo jeśli zlecimy robotowi przemieszczenie do celu z piłką i piłka odskoczy robotowi, to powinien do niej podjechać i ponownie prowadzić lub jeśli nastąpi dobra okazja do strzału wykorzystać ją. Realizacja pojedynczego zadania składa się z trzech etapów:

- pobrania informacji z modelu świata,
- przetworzenia stanu na planszy i utworzenia polecenia dla sterownika robota,
- wyznaczenia kolejnego zachowania do realizacji.

Przejście do kolejnego zachowania jest uwarunkowane parametrami narzuconymi przez aktualnie wykonywana taktykę. Poniżej zamieszczono przykładowe zachowanie zaczerpnięte z publikacji [11].

Algorytm 7.2 Zadanie dojechania z piłką do celu oryginalnej publikacji na temat STP

if (SSM_i=MoveBall AND ball_on_front AND can_kick AND shot_is_good) **then**
  Transition(Kick)
if ball_on_front AND ball_is_visible) **then**
  Transition(GoToBALL)
command generation
calculate target
navigate to target

7.1.4 Opis zrealizowanej wersji algorytmu

W ramach niniejszej pracy zrealizowano dwie warstwy z oryginalnej publikacji. Największy nacisk położono na realizacje prostych zachowań. Stworzona aplikacja udostępnia następujące zachowania:

- poruszanie z piłką do celu,
- przejęcie piłki,
- obrót z piłką,
- oddanie strzału,
- poruszanie do wyznaczonego celu,
- odjechanie od przeszkód na zadaną odległość w danym kierunku.
- zatrzymanie robota

Zaimplementowane zostały także następujące taktyki:

- odebranie podania,
- podanie piłki do gracza,
- dryblowanie z piłką do zadnego punktu,
- wyjście z piłką na pozycję do strzału,
- wyjście na pozycję do oddania strzału.

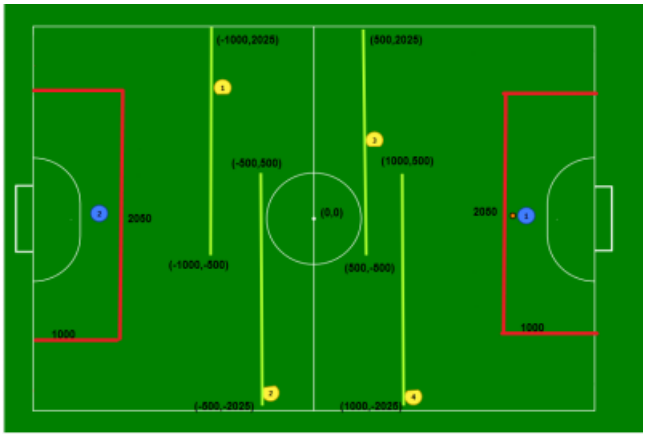
Stworzono też prostą warstwę `Play`, składającą się jedynie z planów służących wykonaniu zaplanowanych testów. Jednen z planów przydziela każdemu z robotów taktykę realizująca podążanie do zadanego punktu i przejęcie piłki jeśli to możliwe. Drugi plan przydziela jednemu z zawodników taktykę wykonania podania, a pozostałym wyjście na pozycję, odebranie podania i oddanie strzału na bramkę.

7.2 Testy warstw `tactics` oraz `skills`

Kolejnym etapem jakim poddano aplikację było przetestowanie zaimplementowanej warstwy wzorowanej na architekturze `STP`. Każda zgłoszona do rozgrywek drużyna, przed przystąpieniem do turnieju głównego musi przejść eliminacje sprawdzające jej poziom. Jako zadania testowe postanowiono wybrać niektóre rzeczywiste zadania eliminacyjne, przez jakie musiały przejść drużyny w ostatnich latach. Więcej informacji na temat eliminacji można znaleźć na stronie projektu [1].

7.3 Nawigacja w dynamicznym środowisku

Pierwsze zadanie pochodzi z eliminacji do mistrzostw w 2011 roku. Celem próby jest sprawdzenie zdolności robotów do bezpiecznego poruszania się w dynamicznym środowisku. Poniżej zamieszczono rysunek przedstawiający środowisko testowe. Znajduje się na nim 6 robotów pełniących role przeszkód. Dwa spośród nich są nieruchome, a pozostałe cztery poruszają się wzdłuż zaznaczonej kolorem żółtym linii prostej. Zadaniem robota jest poruszanie się pomiędzy dwoma statycznymi przeszkodami, każdorazowo mijając od strony bramki. Zasady eksperymentu są następujące:



Rysunek 7.2: Plan środowiska testowego.

1. Liczba startujących robotów jest ograniczona do trzech.
2. Uczestniczące roboty muszą poruszać się pomiędzy dwoma nieruchomymi przeszkodami.
3. Każdorazowo kiedy robot dotknie przeszkody otrzymuje punkt ujemny.
4. Każdy uczestnik, który pokona z powodzeniem trasę otrzymuje punkt.
5. Robot, który wykona okrążenie z piłką otrzymuje dodatkowo 2 punkty.
6. Test trwa 2 minuty.

Ponieważ nie znaleziono iformacji, jak należy się zachować w momencie, gdy robot prowadzący piłkę straci nad nią kontrolę i piłka znajdzie się na aucie, zdecydowano się na przeprowadzenie eksperymentu w dwóch wariantch. W pierwszym z nich, jeśli piłka znalazła się na aucie pozostawała tam, aż do momentu ukończenia eksperymentu. W drugim wariancie w sytuacji, gdy piłka opuściła boisko była ustawiana ponownie na środku boiska. Dla każdego z wariantów eksperyment przeprowadzono 20 razy. Wyniki zamieszczono w kolejnym paragrafie.

7.3.1 Wyniki testu nawigacji

Po wstępnej analizie zachowania zawodnika podczas realizacji eksperymentu, zdecydowano się na drobne modyfikacje algorytmy unikania kolizji. Zastosowana w 6

prosta predykcja położenia przeszkody okazała sie niewystarczająca. Dokonano modyfikacji polegającej na zapamiętywaniu prędkości przeszkody z ostatnich trzech kroków algorytmu. Dla każdego zestawu prędkości dokonywano predykcji położenia przeszkody w następnym kroku algorytmy i za 5 kroków. Dodatkowo zdecydowano się uzależnić promień przeszkody od odległości od sterowanego robota. Im przeszkoda znajdowała się dalej tym większy był promień ją opisujący. Podejście to miało na celu zmuszenie robota do wcześniejszego wykonania manewru unikania kolizji. Dodatkowo, aby zmusić robota do okrążania statycznych przeszkód wprowadzono zmieniające się w zależności od położenia robota ograniczenie na przestrzeń przeszukiwaną przez algorytm `RRT`.

Rezultaty uczestników eliminacji

Poniżej zamieszczone zostały wyniki zadania, jakie zdobyły drużyny startujące w mistrzostwach. Niestety nie znaleziono informacji bardziej szczegółowych, dotyczących na przykład średniego czasu przejazdu okrążenia, bądź liczby robotów biorących udział w zadaniu.

Tabela 7.1: Wyniki zadania otrzymane w 2011 roku.

Nazwa drużyny	*wynik*
Skuba	35
MRL	29
Thunder bots	25

Wyniki zaimplementowanego algorytmu wariant 1

Tabela 7.2: Otrzymane wyniki w wariancie pozostawiającym piłkę na aucie.

Parametr	wartość oczekiwana	wariancja
suma punktów	21.250	4.0234
liczba dodatnich punktów	24.4	3.4409
okrążenia z piłką	3.5500	2.0851
czas przejazdu	14.021	2.3811

Wyniki zaimplementowanego algorytmu wariant 2

Tabela 7.3: Otrzymane wyniki w wariancie ustawiającym piłkę na środku boiska po wyjściu na aut.

Parametr	wartość oczekiwana	wariancja
suma punktów	21.1	6.8695
liczba dodatnich punktów	24.8	3.043
okrążenia z piłką	4.25	0.99373
czas przejazdu	13.452	1.7945

7.4 Strzelanie po podaniu

Kolejne zadanie pochodzi z 2009 roku. W zadaniu uczestniczy od 2 do 3 robotów z jednej drużyny. Ich zadaniem jest zdobycie jak największej ilości goli w przeciągu 120 sekund. Punkty przyznawane są następująco:

1. drużyna zdobywa 1 punkt w momencie gdy przed poprawnym strzałem na bramkę dwa roboty dotkną piłki (wykonane zostanie np. jedno podanie).
2. drużyna zdobywa 2 punkty gdy przed oddanym strzałem 3 roboty dotkną piłki.

Zasady eksperymentu zamieszczono poniżej:

1. Przed startem wszytkie roboty muszą być umieszczone w odległości nie przekraczającej $1[m]$ od własnej linii bramkowej,
2. Zawodnicy drużyny przeciwnej pełnią rolę statycznych przeszkód,
3. Przy każdym starcie piłka jest umieszczana w jednym z narożników, na własnej połowie grającej drużyny,
4. Gol może być zdobyty tylko w sytuacji, gdy robot znajduje się na połowie przeciwnika,
5. Po zdobytej bramce piłka ponownie wraca do jednego z narożników.

7.4.1 Wyniki testu

Rezultaty uczestników eliminacji

Rozdział 8

Podsumowanie

Dodatek A

Zawartość płyty CD

Dołączona do pracy płyta CD zawiera następujące elementy:

- pliki źródłowe symulatora *Gazebo* wraz z naniesionymi poprawkami opisanymi w par. 3.3.2 – folder `gazebo`,
- modele elementów środowiska *RoboCup* do symulatora *Gazebo* – folder `modele`,
- kod źródłowy aplikacji sterującej robotem (projekt w środowisku Eclipse[1] dla C++) wraz z dokumentacją – folder `aplikacja_sterujaca`,
- filmy oraz zrzuty ekranu prezentujące wykonane modele oraz działanie aplikacji sterującej – folder `media`,
- plik `praca_magisterska.pdf` – wersja elektroniczna niniejszego dokumentu.

[1] Środowisko jest dostępne pod adresem `www.eclipse.org`.

Dodatek B

Instrukcja instalacji Gazebo

poprawic opis instalacji gazebo

Instrukcję instalacji można znaleźć w poradniku dostępnym pod adresem `http://playerstage.sourceforge.net/doc/Gazebo-manual-svn-html/install.html`. W razie problemów można skorzystać także z opisu dostępnego pod adresem `http://www.irobotics.org/gazebo08.f8.html`. Do instalacji należy użyć wersji *Gazebo* dostępnej na płycie CD dołączonej do pracy (jest to wersja 6782 dostępna poprzez repozytorium SVN, zawiera jednak poprawki opisane w par. 3.3.2). Do poprawnej pracy *Gazebo* wymagana jest obecność dodatkowych aplikacji. Najważniejsze z nich wykorzystano w niniejszej pracy w nastepujących wersjach:

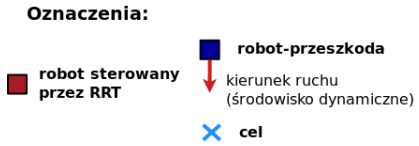
zmienic numery rewizji zastosowanych aplikacji

- Player (wymagany do kompilacji Gazebo) – pobrany z repozytorium SVN w wersji 6350,
- ODE pobrane z oficjalnego repozytorium SVN, wersja 1451,
- OGRE w wersji 1.4.7,
- pozostałe wymagane aplikacje zostały zainstalowane w wersjach zgodnych z opisem w podręczniku instalacji.

Dodatek C

Szczegóły eksperymentów

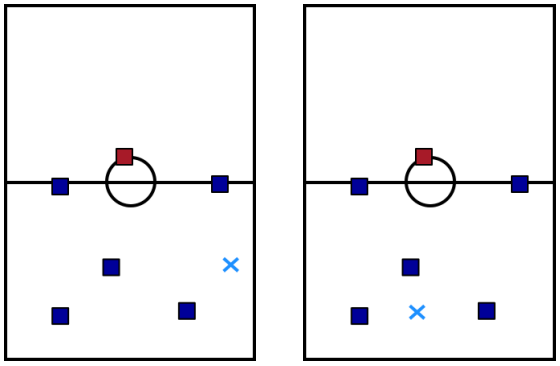
Dodatek zawiera poglądowe rysunki przedstawiające środowiska testowe, na których przeprowadzane były eksperymenty. Na każdym z nich zaznaczono inne roboty nie podlegające sterowaniu przez testowany algorytm oraz początkowe położenie i orientację sterowanego robota. W przypadku eksperymentów w środowisku dynamicznym zaznaczono także kierunek i zwrot prędkości z jaka poruszały się przeszkody. Poniżej zamieszczono legendę objaśniającą znaczenie użytych symboli.



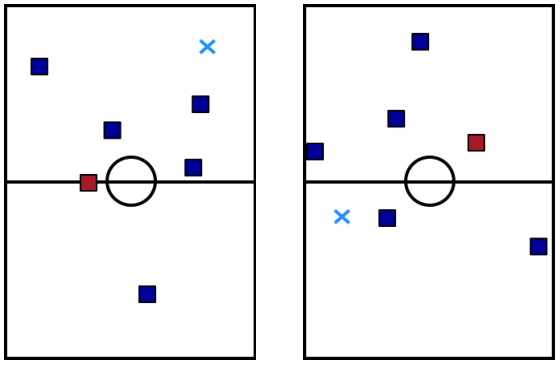
W drugiej części dodatku zamieszczono tabelę z zastosowanymi podczas eksperymentów z użyciem algorytmu *CVM* zestawami wag. Ponieważ każda ze składowych funkcji celu (5.8) algorytmu zwraca wartość z przedziału $[0; 1]$ zdecydowano się na przetestowanie takich trójek liczb z tego przedziału, które sumują się do jedności.

Środowiska testowe

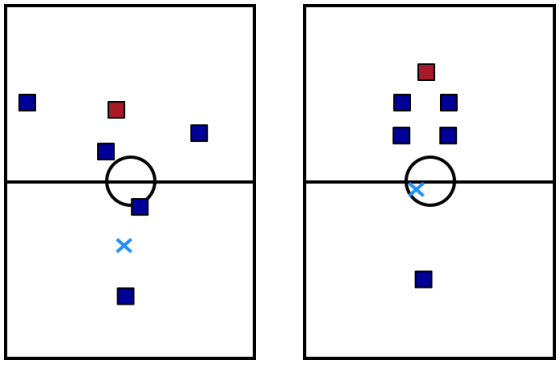
statyczne:



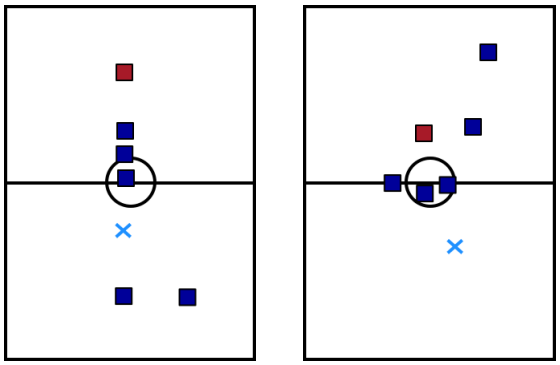
(a) (b)



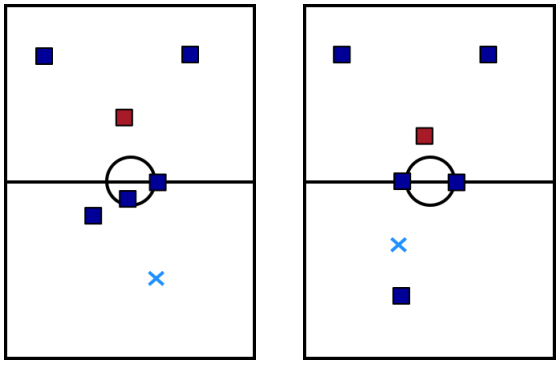
(c) (d)



(e) (f)

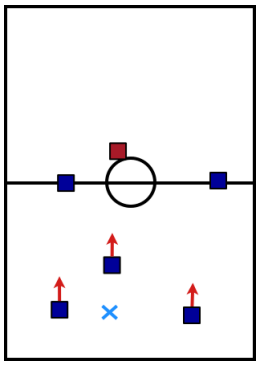


(g) (h)

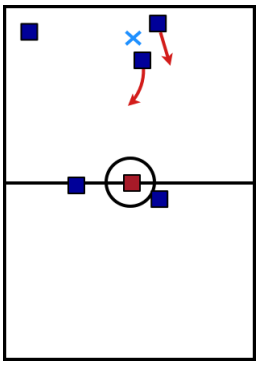


(i) (j)

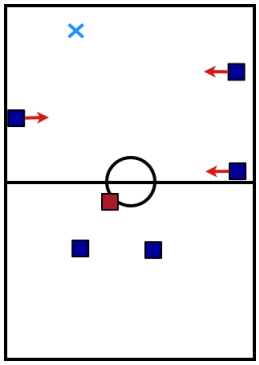
dynamiczne:



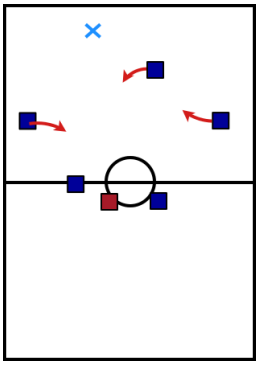
(k)



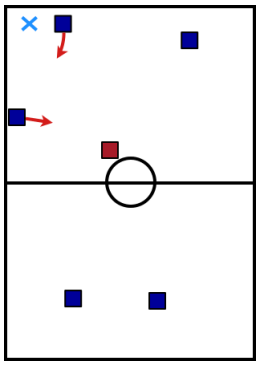
(l)



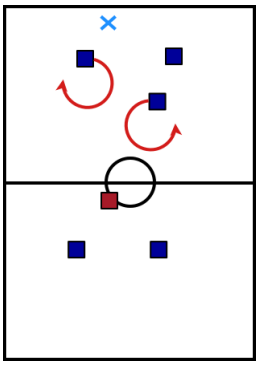
(m)



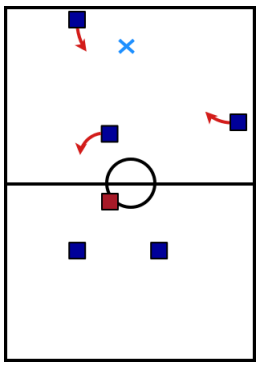
(n)



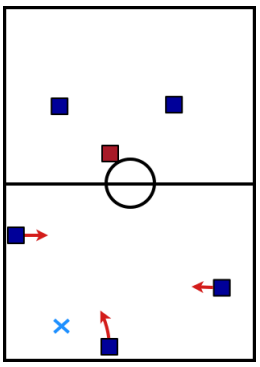
(o)



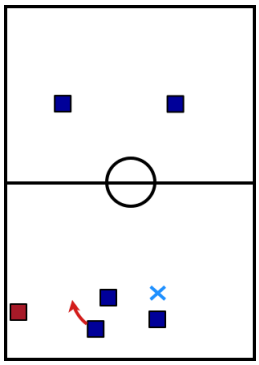
(p)



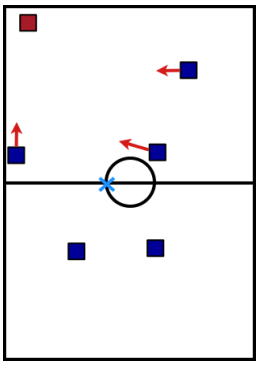
(q)



(r)



(s)



(t)

Zestawy wag używane w eksperymentach

Zestaw wag	$goalProb$	$wayPointProb$
1	0.0	0.0
2	0.0	0.2
3	0.0	0.3
4	0.0	0.4
5	0.0	0.5
6	0.0	0.6
7	0.0	0.7
8	0.0	0.8
9	0.0	0.9
10	0.0	1.0
11	0.1	0.0
12	0.1	0.1
13	0.1	0.2
14	0.1	0.3
15	0.1	0.4
16	0.1	0.5
17	0.1	0.6
18	0.1	0.7
19	0.1	0.8
20	0.1	0.9
21	0.2	0.0
22	0.2	0.1
23	0.2	0.2
24	0.2	0.3
25	0.2	0.4
26	0.2	0.5
27	0.2	0.6
28	0.2	0.7
29	0.2	0.8
30	0.3	0.0
31	0.3	0.1
32	0.3	0.2
33	0.3	0.3

Zestaw wag	$goalProb$	$wayPointProb$
34	0.3	0.4
35	0.3	0.5
36	0.3	0.6
37	0.3	0.7
38	0.4	0.0
39	0.4	0.1
40	0.4	0.2
41	0.4	0.3
42	0.4	0.4
43	0.4	0.5
44	0.4	0.6
45	0.5	0.0
46	0.5	0.1
47	0.5	0.2
48	0.5	0.3
49	0.5	0.4
50	0.5	0.5
51	0.6	0.1
52	0.6	0.2
53	0.6	0.3
54	0.6	0.4
55	0.7	0.0
56	0.7	0.1
57	0.7	0.2
58	0.7	0.3
59	0.8	0.0
60	0.8	0.1
61	0.8	0.2
62	0.8	0.2
63	0.9	0.0
64	0.9	0.1
65	1.0	0.0

Bibliografia

[1] Oficjalna strona Ligi *RoboCup* dostępna pod adresem:
`www.robocup.org`

[2] J. Bruce, M. Veloso: *Real-Time Randomized Path Planning for Robot Navigation*. Carnegie Mellon University, 2002.

[3] J.Kim J.M. Esposito, V. Kumar: *An RRT-Based algorithm for testing and validating multi-robot controllers*. University of Pennsylvania, US Naval Academy, 2005.

[4] I. Dulęba: *Metody i algorytmy planowania ruchu robotów mobilnych i manipulacyjnych*. Warszawa, Akademicka Oficyna Wydawnicza EXIT, 2001.

[5] T. Quasn, L.Pyeatt, J.Moore: *Curvature-Velocity Method for Differentially Steered Robots*. AI Robotics Lab Computer Science Department, Texas Tech University, 2003.

[6] R. Simmons: *The Curvature-Velocity Method for Local Obstacle Avoidance*. School of Computer Science, Carnegie Mellon University, 1996.

[7] M. Majchrowski: *Algorytm unikania kolizji przez robota mobilnego bazujący na przeszukiwaniu przestrzeni prędkości*. Praca Magisterska, Politechnika Warszawska, 2006.

[8] J. Borenstein, Y. Koren: *The vector field histogram – fast obstacle avoidance for mobile robots*. IEEE Transaction on Robotics and Automation, 1991.

[9] J. Borenstein,Y. Koren: *Histogramic in-motion mapping for mobile robot obstacle avoidance*. Department of Mechanical Engineering and Applied Mechanics, The University of Michigan, 1991.

[10] D. Fox, W. Burgard, S. Thrun: *The Dynamic Window Approach to Collision Avoidance*. Department of Computer Science, University of Bonn; Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1991.

[11] B.Browning, J.Bruce, M.Bowling, M.Veloso: *STP: Skills, tactics and plays for multi-robot control in adversarial environments*. Carnegie Mellon University, Pittsburgh. 2004.

[12] R.Rojas, A.Gloye Föorster: *Holonomic Control of a robot with an omnidirectional drive*. Freie Universität Berlin, 2006.

[13] R.Rojas, A.Gloye Föorster: *Design of an omnidirectional universal mobile platform*. Eindhoven University of Technology, 2005.

[14] X. Li, M.Wang, A.Zell: *Dribbling control of omnidirectional soccer robots*. In Proceedings of 2007 IEEE International Conference on Robotics and Automation (ICRA'07).

[15] M.Gąbka, K.Muszyński Praca dyplomowa inzynierska *Środowisko symulacyjne i algorytm unikania kolizji robota mobilnego grającego w piłkę nożną*. Politechnika Warszawska, 2008.

[16] J.Bruce, M.Bowling, B.Browning, M.Veloso: *Multi-Robot Team Response to a Multi-Robot Opponent Team*. Carnegie Mellon University, Pittsburgh, 2002.

[17] J.Bruce, M.Veloso: *Real-time multi-robot motion planning with safe dynamics*. Carnegie Mellon University, Pittsburgh, 2006.

[18] D. E. Koditschek, and E.Rimon: *Robot navigation functions on manifolds with boundary*. Advances in Applied Mathematics Volume 11 Issue 4, Dec. 1990.

[19] D. E. Koditschek, and E.Rimon: *Exact robot navigation using artificial potential functions*. IEEE Transactions on Robotics and Automation. Vol. 8, no. 5, pp. 501-518. Oct. 1992

[20] C. Zieliński, W. Szynkiewicz: *Konspekt do wykładu: Inteligentne Systemy robotyczne*. Politechnika Warszawska, 2008.

[21] N. Koenig: *Gazebo. The Instant Expert's Guide*. Player Summer School on Cognitive Robotics, Monachium 2007.

[22] M.Hamada *Układ sterowania autonomicznym robotem mobilnym.* Praca magisterska, Politechnika Warszawska, 2007.

Todo list

zamiescic przykładowe trajektorie w srodowisku statycznym 68
zamiescic przykładowe trajektorie w srodowisku dynamicznym 69
poprawic opis instalacji gazebo 86
zmienic numery rewizji zastosowanych aplikacji 86