Spis treści

1 Wstęp 1
1.1 Krótki opis ćwiczeń laboratoryjnych 1
1.2 Symulator CoppeliaSim 2
2 Budowa i sterowanie prostego robota o napędzie różnicowym 4
2.1 Bodowa robota 4
2.2 Podstawowe sterowanie robotem 10
2.3 Rysowanie trajektorii robota z wykorzystaniem cyklicznego bufora położenia robota 13
2.4 Scenariusz ćwiczenia 15
2.5 Pytania sprawdzające 16
3 Behawioralne sterowanie robotem w zadaniu omijania przeszkód 17
3.1 Przygotowanie symulacji 17
3.2 Algorytm Braitenberga 17
3.3 Metoda przełączanych zachowań 19
3.4 Rysowanie trajektorii robota z wykorzystaniem tekstur 20
3.5 Scenariusz ćwiczenia 23
3.6 Pytania sprawdzające 23
4 Zaawansowane sterowanie robotem na przykładzie zadania śledzenia linii 25
4.1 Sterowanie typu włącz-wyłącz 25
4.2 Sterowanie proporcjonalne z przełączaniem 26
4.3 Regulator PID 27
4.4 Szczególne uwarunkowania zadania 28
4.5 Wizualizacja parametrów robota 28
4.6 Scenariusz ćwiczenia 30
4.7 Pytania sprawdzające 30
5 Podstawowe sterowanie maszyną kroczącą z poziomu języka C++ 33
5.1 Model robota 33
5.2 Programowanie z poziomu języka wysokiego poziomu C++ 33
5.3 Biblioteka i klasa opisująca model robota w języku C++ 35
5.4 Szablon programu 38
5.5 Scenariusz ćwiczenia 39

5.6 Pytania sprawdzające ... 39
6 Wykorzystanie czujników odległości do omijania przeszkód przez maszynę kroczącą ... 41
6.1 Model robota i środowiska ... 41
6.2 Biblioteka i klasa opisująca model robota w języku C++ ... 42
6.3 Szablon programu ... 43
6.4 Scenariusz ćwiczenia ... 44
6.5 Pytania sprawdzające ... 45
7 Planowanie drogi w labiryncie ... 47
7.1 Model robota i środowiska ... 47
7.2 Biblioteka i klasa opisująca model robota w języku C++ ... 48
7.3 Szablon programu ... 48
7.4 Planowanie ścieżki ... 48
7.5 Scenariusz ćwiczenia ... 48
7.6 Pytania sprawdzające ... 48

1 Wstęp

Niniejszy podręcznik to zbiór instrukcji do ćwiczeń zdalnych dla przedmiotu Robotyka mobilna, ale instrukcje mogą być także wykorzystywane na zajęciach z innych przedmiotów o zbliżonej tematyce.

W ramach zajęć laboratoryjnych do wykonania jest 6 ćwiczeń o rosnącym stopniu trudności. Ich celem jest zapoznanie użytkownika z podstawami środowiska symulacyjnego CoppeliaSim oraz wybranymi zagadnieniami z robotyki mobilnej. Trzy pierwsze ćwiczenia dotyczą robotów kołowych z najbardziej rozpowszechnionym wśród takich robotów napędem różnicowym, trzy kolejne – sześcionożnego robota kroczącego. W ramach trzech pierwszych ćwiczeń, roboty będą oprogramowywane wyłącznie z użyciem środowiska CoppeliaSim i wykorzystywanego przez nie języka programowania Lua. Trzy kolejne ćwiczenia będą realizowane z poziomu języka C++, co będzie wymagało dodatkowych narzędzi i plików konfiguracyjnych oraz współpracy na styku języków Lua i C++.

1.1 Krótki opis ćwiczeń laboratoryjnych

W pierwszym ćwiczeniu użytkownik buduje, a następnie oprogramowuje prostego robota, który realizuje elementarne zadania ruchowe. Użytkownik odczytuje także położenie robota oraz dokonuje wizualizacji jego trajektorii. Na podstawie analizy zastosowanych sterowań i przejechanej odległości użytkownik tworzy proste funkcje wysterowujące robota, dzięki którym robot jest w stanie w przybliżeniu zrealizować zadaną trajektorię.

Drugie ćwiczenie dotyczy zagadnienia sterowania behawioralnego. Wykorzystana w nim zostanie gotowa scena oraz robot z czujnikami ultradźwiękowymi z biblioteki CoppeliaSim. Zarówno scena, jak i robot będą wymagały nieznacznych modyfikacji w trakcie ćwiczenia. Trajektoria robota zostanie zwizualizowana w odmienny sposób niż w ćwiczeniu pierwszym, a ostatecznym celem tego ćwiczenia będzie takie wysterowanie robota, aby odwiedził on wszystkie dostępne miejsca w swoim otoczeniu.

Trzecie ćwiczenie zakłada wykorzystanie gotowego modelu robota o napędzie różnicowym wyposażonego w czujniki wizyjne, dostępnego w bibliotece środowiska CoppeliaSim. Jest to typowy robot do zadania śledzenia linii. Zadaniem użytkownika jest takie zaprogramowanie robota, aby jak najszybciej i jak najpłynniej pokonał on wyznaczoną trasę w postaci linii na podłodze. Użytkownik najpierw zbada prosty algorytm typu włącz-wyłącz, a następnie przejdzie ro realizacji regulatora proporcjonalnego oraz typowego regulatora PID. Dobór optymalnych nastaw tego ostatniego jest istotną częścią tego ćwiczenia.

Ćwiczenie czwarte pokazuje sposób wysterowania robota kroczącego o 18 stopniach swobody z poziomu aplikacji języka C++. Takiego robota nie da się zrobić z podstawowych elementów dostępnych w ramach symulatora CoppeliaSim – konieczne jest użycie

zewnętrznych narzędzi do modelowania 3D i import stworzonych tam komponentów. Ze względu na złożoność i czasochłonność całego procesu, w ramach ćwiczenia nie jest rozpatrywana budowa maszyny, a jedynie jej wykorzystanie w aplikacji do sterowania robotem zaimplementowanej w języku C++. Dla aplikacji w języku C++ została zbudowana biblioteka pozwalająca na sterowanie maszyną z wykorzystaniem klas. W ćwiczeniu tym podstawowym celem jest zapoznanie się ze sterowaniem robotem w symulatorze CoppeliaSim. Do realizacji przewidziano sterowanie niskopoziomowe i zadawanie określonych kątów dla przegubów nóg robota, sterowanie nogą przez zadawanie pozycji jej końcówki oraz takie wysterowanie nóg robota, by poruszał się on chodem trójpodporowym oraz innym wybranym przez wykonującego ćwiczenie.

Piąte ćwiczenie dotyczy sterowania robotem kroczącym poprzez klawisze na klawiaturze komputera, a także implementacji aplikacji realizującej pracę autonomiczną maszyny. Po wyznaczonym obszarze robot powinien poruszać się tak, by omijając przeszkody, znaleźć wyjście i opuścić ten obszar. W ćwiczeniu musi zostać zaimplementowana autonomia robota, a w szczególności algorytm omijania przeszkód oraz znajdowania wyjścia z obszaru otoczonego ścianami.

Ostatnie, szóste ćwiczenie to poruszanie się robota po labiryncie. W ramach tego ćwiczenia należy zaimplementować aplikację języka C++, tak aby realizowała ona sterowanie robotem umożliwiające swobodne poruszanie się po labiryncie. W labiryncie robot powinien znaleźć wyjście i przez nie przejść opuszczając labirynt. Osoba realizująca ćwiczenie ma za zadanie wybrać odpowiedni algorytm poszukiwania wyjścia z labiryntu oraz go zaimplementować.

Do każdego ćwiczenia, za wyjątkiem pierwszego, ze względu na jego podstawowy charakter, dołączone są dodatkowe materiały w postaci scen dla symulatora CoppeliaSim i/lub dodatkowego oprogramowania.

1.2 Symulator CoppeliaSim

CoppeliaSim to środowisko pozwalające na symulację niemal dowolnych zagadnień z dziedziny robotyki wraz ze zintegrowanym środowiskiem programistycznym. Oparte jest ono na rozproszonej architekturze sterowania: każdym obiektem/modelem można sterować niezależnie za pomocą wbudowanego skryptu, wtyczki, węzła ROS, zdalnego klienta API lub niestandardowego rozwiązania. W CoppeliaSim można programować i symulować wiele niezależnych i zróżnicowanych robotów jednocześnie. Oprogramowanie można implementować w językach C/C++, Python, Java, Lua, Matlab lub Octave.

CoppeliaSim dedykowana jest do szybkiego opracowywania algorytmów, symulacji automatyzacji fabryk, szybkiego prototypowania i weryfikacji, edukacji związanej z robotyką, zdalnego monitorowania, podwójnej kontroli bezpieczeństwa, jako cyfrowy bliźniak i wielu innych zastosowań.

CoppeliaSim oferuje 5 silników fizycznych (MuJoCo, Bullet Physics, ODE, Newton i Vortex Dynamics) do szybkich obliczeń dynamicznych, symulacji fizyki rzeczywistego świata i

interakcji obiektów (kolizje, chwytanie obiektów itp.). Umożliwia realistyczne i dokładne symulowanie czujników zbliżeniowych i wizyjnych. W CoppeliaSim można zbudować wszystko, od sensorów lub aktuatorów do całych systemów robotycznych, łącząc podstawowe obiekty i ich funkcjonalności za pomocą osadzonych skryptów.

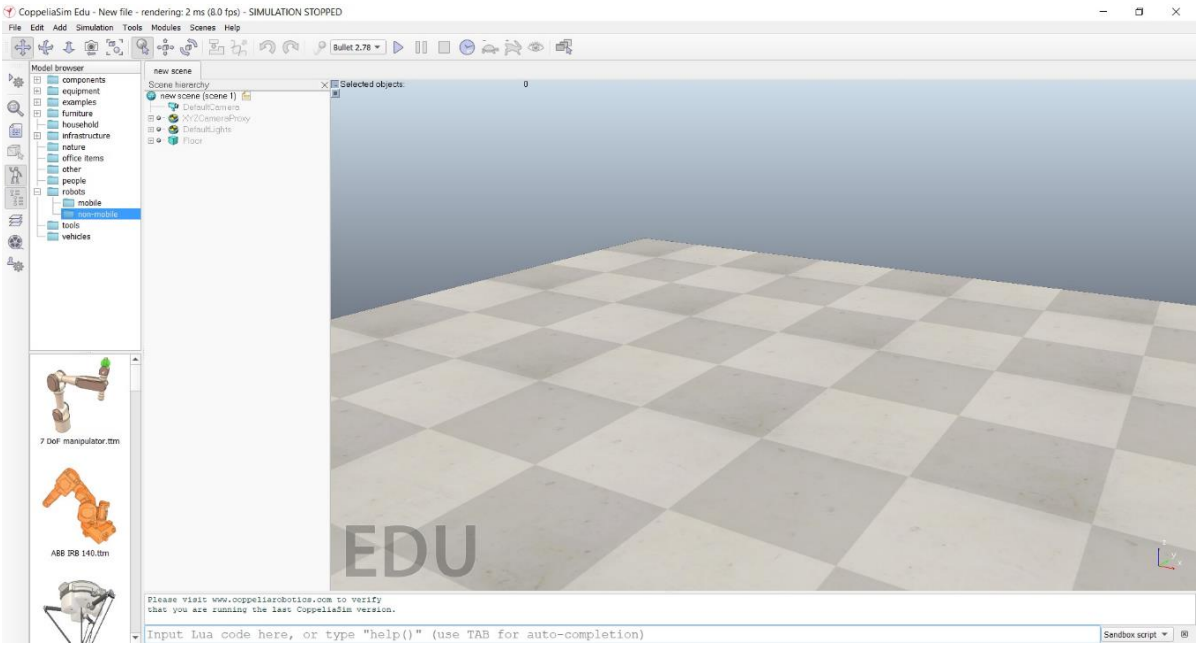
Co bardzo istotne, podmioty edukacyjne (w tym studenci czy nawet hobbyści) mogą korzystać z CoppeliaSim Edu za darmo.

Wszystkie wartości w API są podawane podstawowych jednostkach układu SI, a więc w metrach, kilogramach, sekundach i radianach lub ich kombinacjach (chyba, że zostało to inaczej określone).

W celu uruchomienia środowiska należy ze strony https://www.coppeliarobotics.com/ pobrać poprzez zakładkę Download wersję edu oprogramowania. Następnie należy zainstalować to oprogramowanie na komputerze, który będziemy wykorzystywać do pracy ze środowiskiem.

Po uruchomieniu pobranego pliku należy zainstalować oprogramowanie zgodnie z ustawieniami domyślnymi. Zainstalowane środowisko może zostać uruchomione.

W oknie uruchomionego środowiska widoczna jest scena wraz z drzewem przedstawiającym jej elementy. W domyślnej scenie znajdują się: kamera, światła i podłoga. Przyciski, pola i obiekty pomocnicze pozawalają na kreowanie odpowiedniego wyglądu sceny oraz importowanie gotowych lub tworzenie własnych modeli różnego rodzaju obiektów począwszy od prostych brył geometrycznych, a na złożonych zespołach robotów skończywszy.

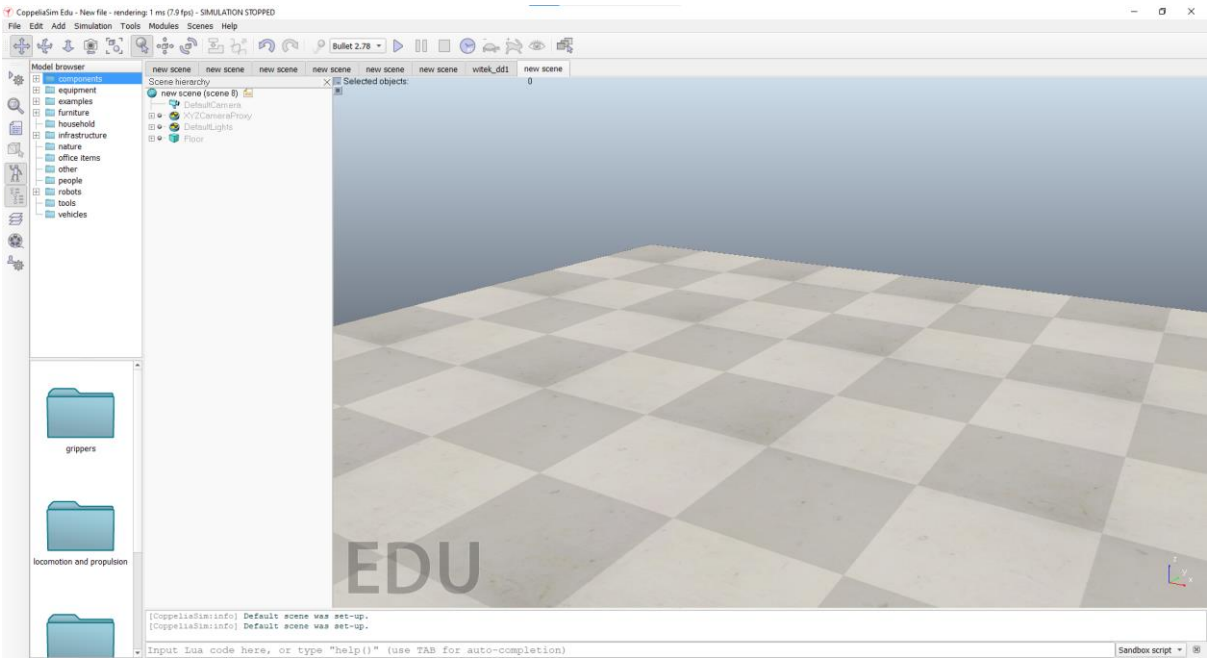


2 Budowa i sterowanie prostego robota o napędzie różnicowym

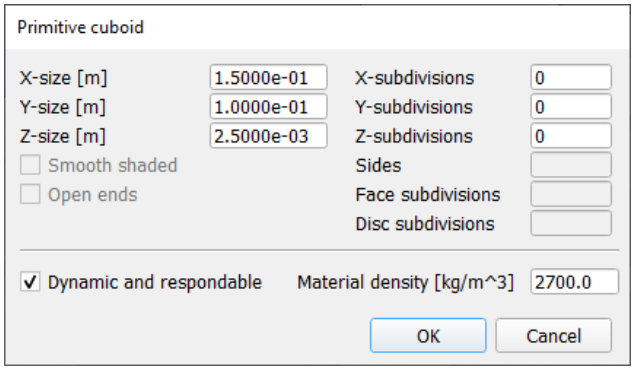
Pierwsze ćwiczenie polega na zapoznaniu się ze środowiskiem CoppeliaSim, a następnie na zbudowaniu własnego robota z podstawowych komponentów dostępnych w symulatorze, oprogramowaniu go i wizualizacji wybranych parametrów ruchu.

2.1 Bodowa robota

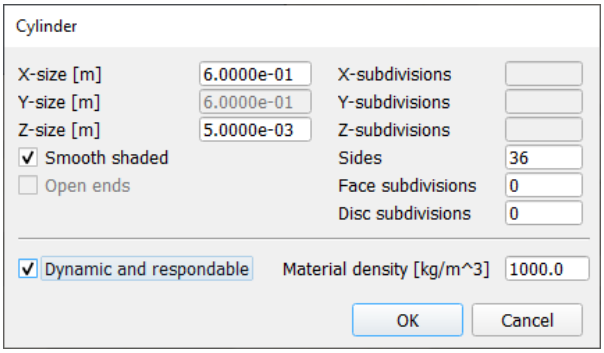
Po uruchomieniu programu zobaczymy domyślną scenę zawierającą podstawowe obiekty takie jak podłoga, światła i kamera. W takim środowisku możemy umieścić i symulować naszego robota.



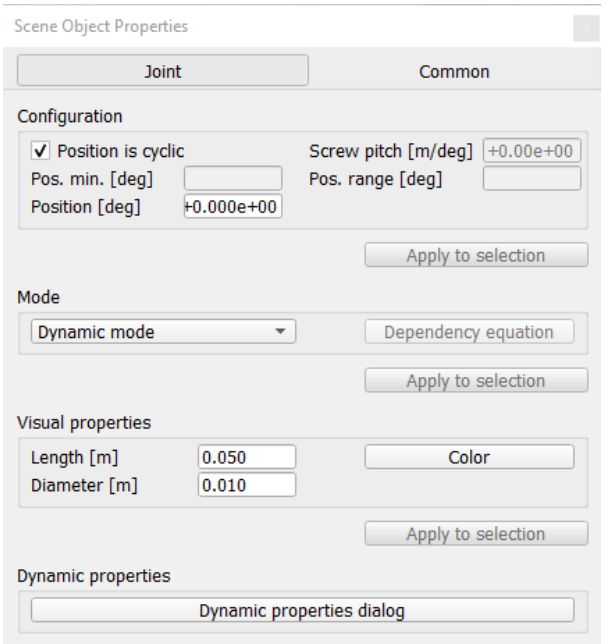
Prostego robota możemy stworzyć z podstawowych brył geometrycznych dostępnych z menu Add/Primitive Shape. Do zbudowania robota o napędzie różnicowym wystarczy podstawa robota w postaci płaskiego prostopadłościanu (Cuboid), dwa silniki (Joint/Revolute) i dwa koła (Cylinder). Zaczynamy od podstawy robota, dodając ją do sceny i ustawiając wymiary np. na 15x10 cm i grubość na 2,5 mm (rys.2).



W związku z tym, że CoppeliaSim symuluje dynamiczne zachowanie obiektów, istotne może być podanie parametrów materiałów zbliżonych do rzeczywistości. Podczas tworzenia nowego obiektu, poza jego wymiarami, podajemy także ich ciężar właściwy. Domyślnie wynosi on 1000 kg/m3, jednak jeśli podstawa naszego robota miałaby być wykonana z blachy aluminiowej, to należy uwzględnić ciężar właściwy tego metalu wynoszący ok. 2700 kg/m3. Koła niewielkich robotów zwykle są wykonywane z plastiku z gumową oponą, jednak dla uproszczenia zamodelujemy je jako w całości wykonane z jednego materiału o domyślnej gęstości. Przyjmiemy parametry jak na rysunku (średnica 6 cm, grubość 5 mm):

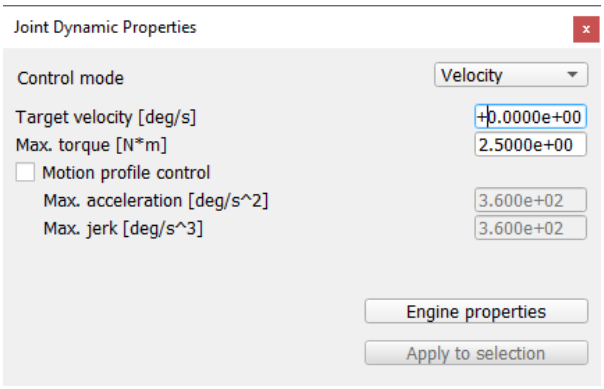


Pozostaje dodanie silników. Faktycznie nie będziemy modelować silnika jako takiego, ale dodamy złącza obrotowe, których długość i średnicę ustalimy odpowiednio na 5 cm i 1 cm. W przeciwieństwie do innych obiektów geometrycznych, złącza obrotowe pojawią się z domyślnymi parametrami, które możemy zmienić po dwukrotnym kliknięciu na symbol złącza w drzewie obiektów symulacji:



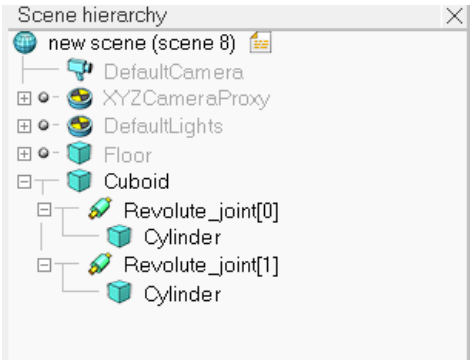
Istotne jest, by zmienić właściwości dynamiczne tych silników, gdyż domyślnie złącza są swobodne, tymczasem my będziemy realizować sterowanie prędkościowe. Należy więc

zmienić ich tryb (Control mode) na prędkościowy (Velocity), a docelową prędkość (Target velocity) ustawić na 0 (w przeciwnym razie silnik będzie się od razu obracał).

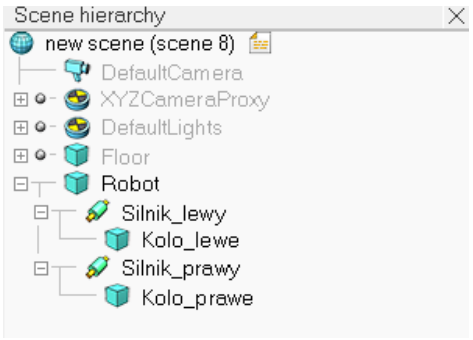


Powyższe okno dialogowe umożliwia ustawienie zaawansowanych parametrów silników, niemniej do podstawowej symulacji robota o napędzie różnicowym nie będzie to potrzebne.

Po utworzeniu wszystkich niezbędnych komponentów należy połączyć je ze sobą przeciągając odpowiednie obiekty w hierarchii na siebie. Nadrzędnym obiektem będzie podstawa robota, do niej będę podłączone silniki, a do każdego silnika – koło. Hierarchia obiektów powinna wyglądać następująco:

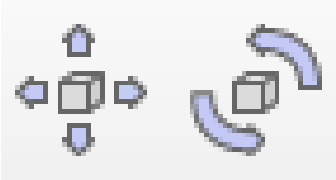


Warto w tym momencie zmienić domyślne nazwy komponentów robota na lepiej oddające ich charakter:

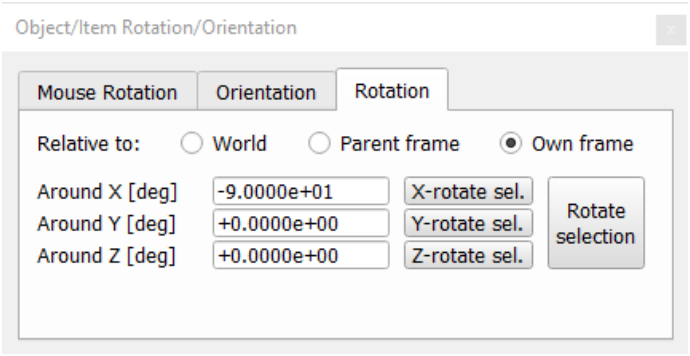


Pozostaje teraz odpowiednie ułożenie komponentów względem siebie, tak by zrobić z nich dwukołową platformę jeżdżącą. Na razie wszystkie elementy składowe robota znajdują „jeden

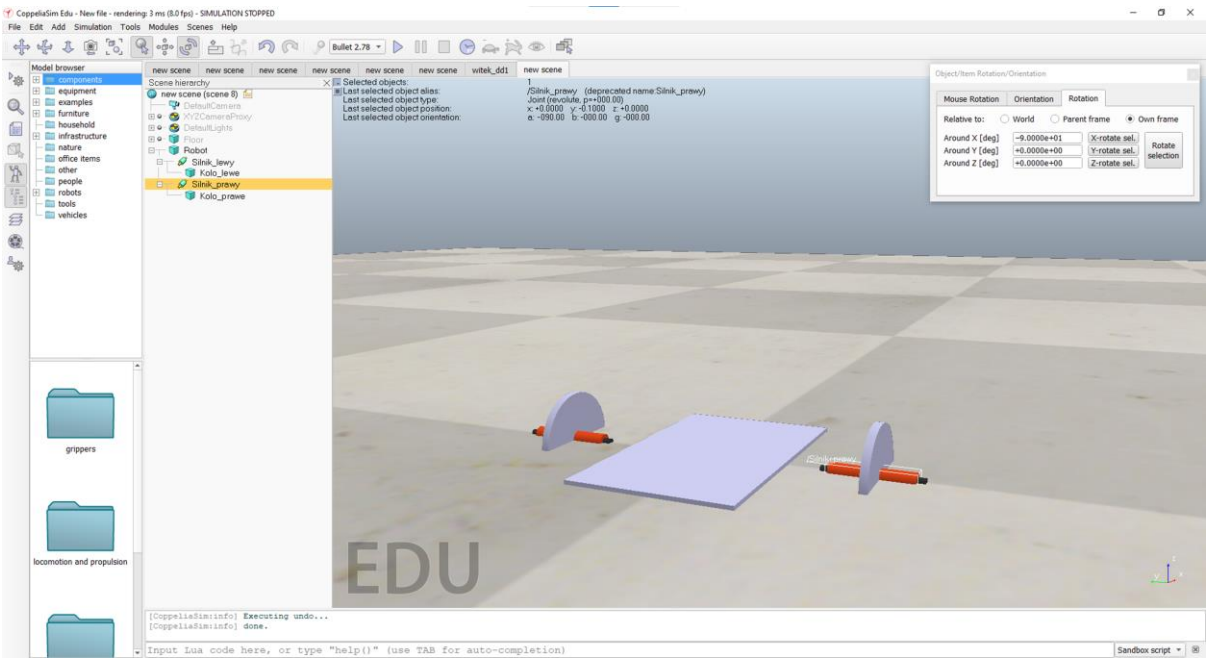
na drugim” w początku układu współrzędnych. Aby je przesunąć bądź obrócić, należy kliknąć na ikonę danego obiektu w hierarchii lub na sam obiekt w oknie sceny (co może być kłopotliwe, gdy obiekty zachodzą na siebie), a następnie wybrać jedną z dwóch ikon na pasku narzędziowym:



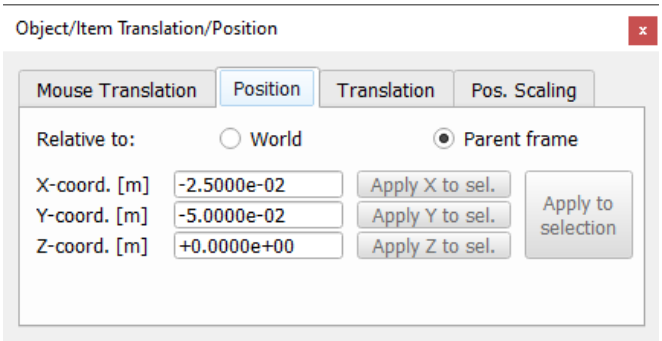
W pierwszej kolejności należy rozsunąć silniki na boki (koła leżące niżej w hierarchii przesuną się razem z nimi), a następnie obrócić je o -90 stopni wokół osi X w lokalnym układzie współrzędnych (Own frame) jak na rysunku poniżej:



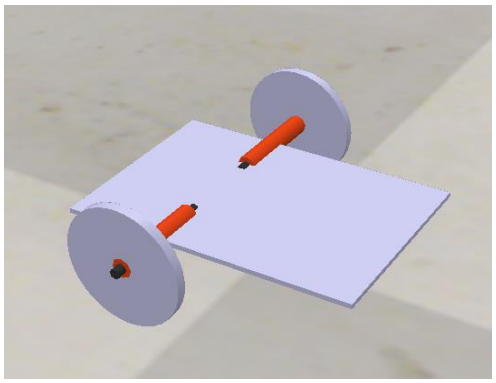
Ważne, by oba silniki obrócić w tę samą stronę – wówczas kierunek obrotów obu kół będzie jednakowy przy tym samym znaku zadanej prędkości obrotowej. Na bieżącym etapie komponenty robota powinny wyglądać jak na rysunku:



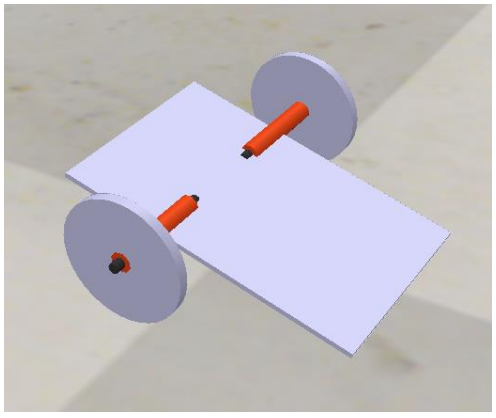
Wystarczy teraz przesunąć koła na końce silników, same silniki nasunąć na platformę i wszystko unieść nieznacznie nad powierzchnią podłogi. Przy przesuwaniu komponentów wygodnie jest poruszać nimi względem obiektu nadrzędnego, co znacznie ułatwia określenie współrzędnych dla obu silników – będą się one różnić jedynie znakiem przy współrzędnej Y.



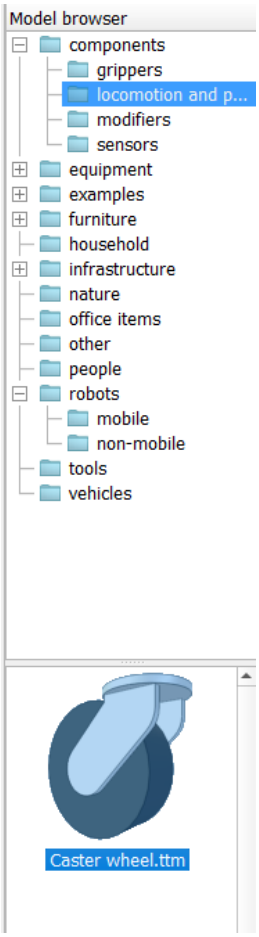
Po odpowiednim ułożeniu wszystkich elementów powinniśmy otrzymać platformę zbliżoną do tej na poniższym rysunku:



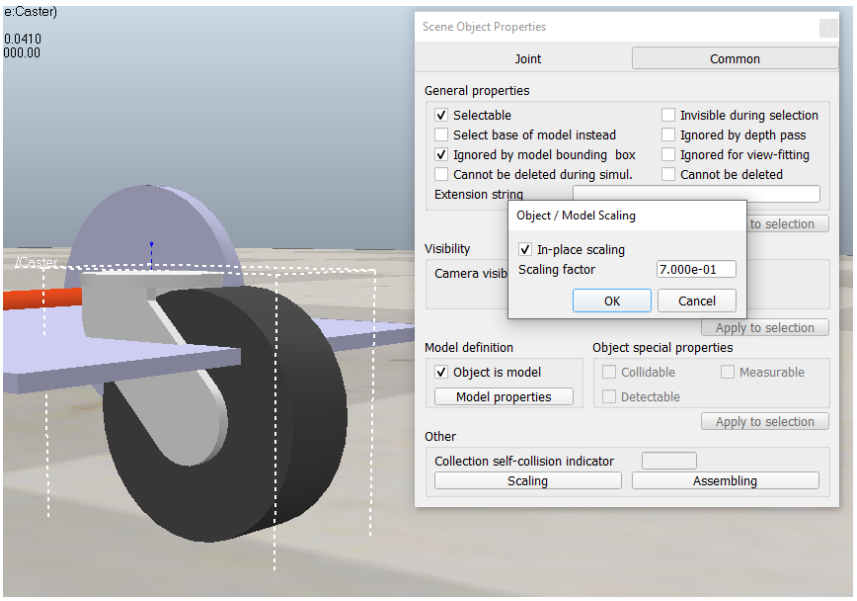
Po uruchomieniu symulacji platforma opadnie, a jeśli nadamy jej silnikom dodatnią prędkość, zacznie poruszać się do przodu (w prawo, na poniższym rysunku). Jeśli platforma jedzie w przeciwną stronę, oznacza to, że albo silniki zostały obrócone w drugą stronę, albo zostały przesunięte ku przodowi, a nie tyłowi platformy. Należy wówczas poprawić orientację lub położenie silników i upewnić się, że prawy silnik znajduje się z prawej strony robota, a dodatnia prędkość powoduje jazdę robota do przodu – pozwoli to uniknąć późniejszych pomyłek w kodzie.



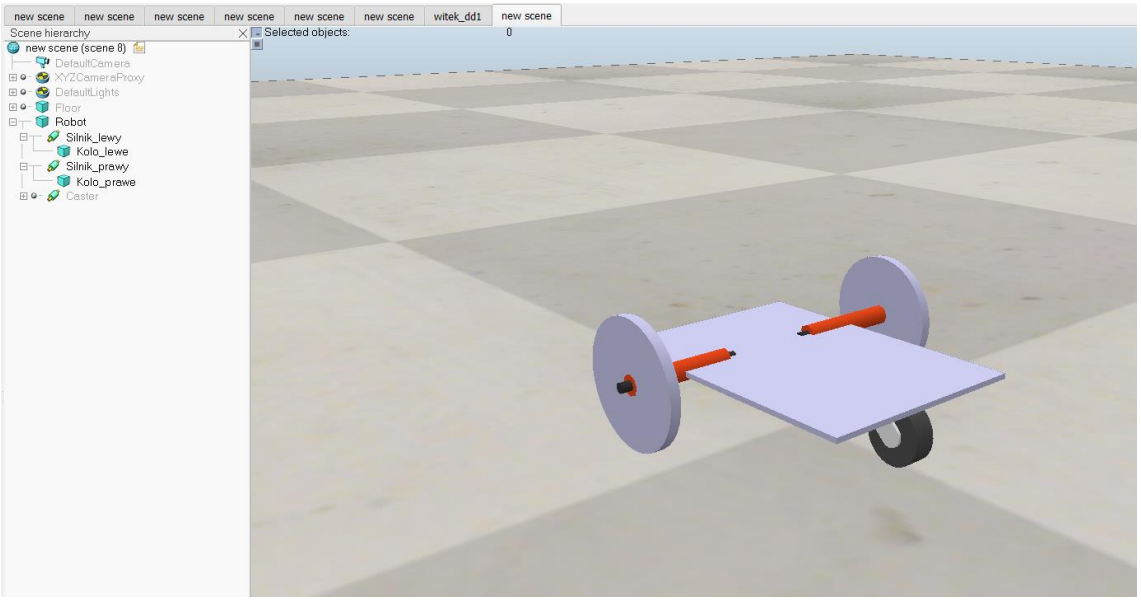
Tak skonstruowany robot będzie wprawdzie jeździł, ale warto dodać do niego przednie kółko podpierające, zwłaszcza jeśli w przyszłości będziemy chcieli umieścić jakieś czujniki z przodu robota. Kółko samonastawne najłatwiej wybrać z biblioteki gotowych elementów (Model browser/components/locomotion and propulsion):



Jest ono nieco za duże do naszego robota. Należy je odpowiednio przeskalować, klikając dwukrotnie w ikonę kółka (Caster) i wybierając przycisk Scaling z zakładki Common.



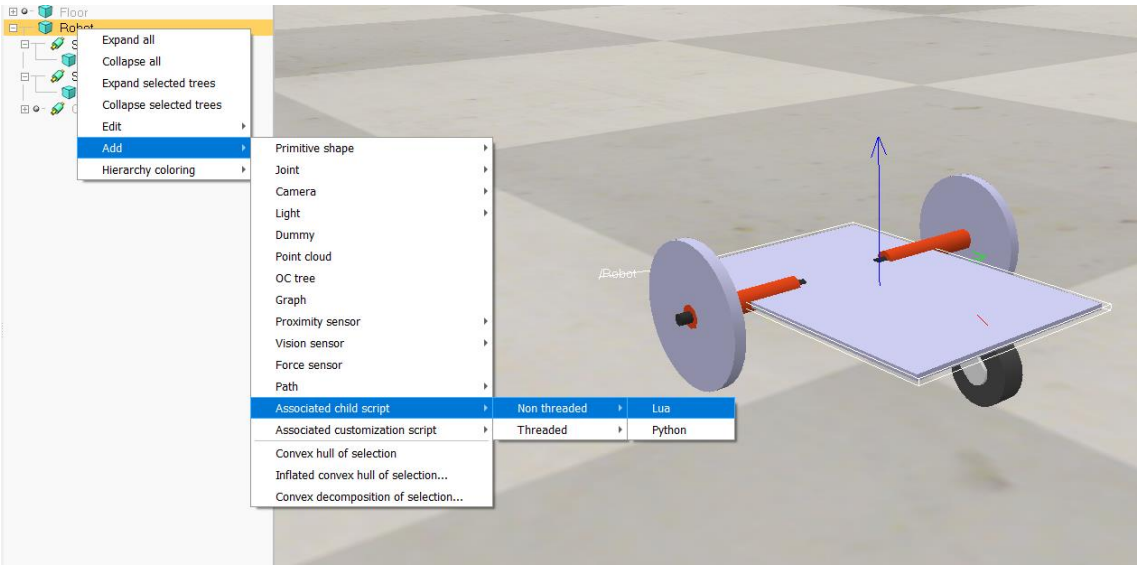
Zaznaczenie opcji In-place scaling oraz zastosowanie współczynnika skalowania 0,7 powinno dać odpowiedni rezultat. Pozostaje jedynie przesunięcie kółka nieco poniżej platformy i dołączenie go do hierarchii robota. Ostateczny efekt powinien wyglądać tak:



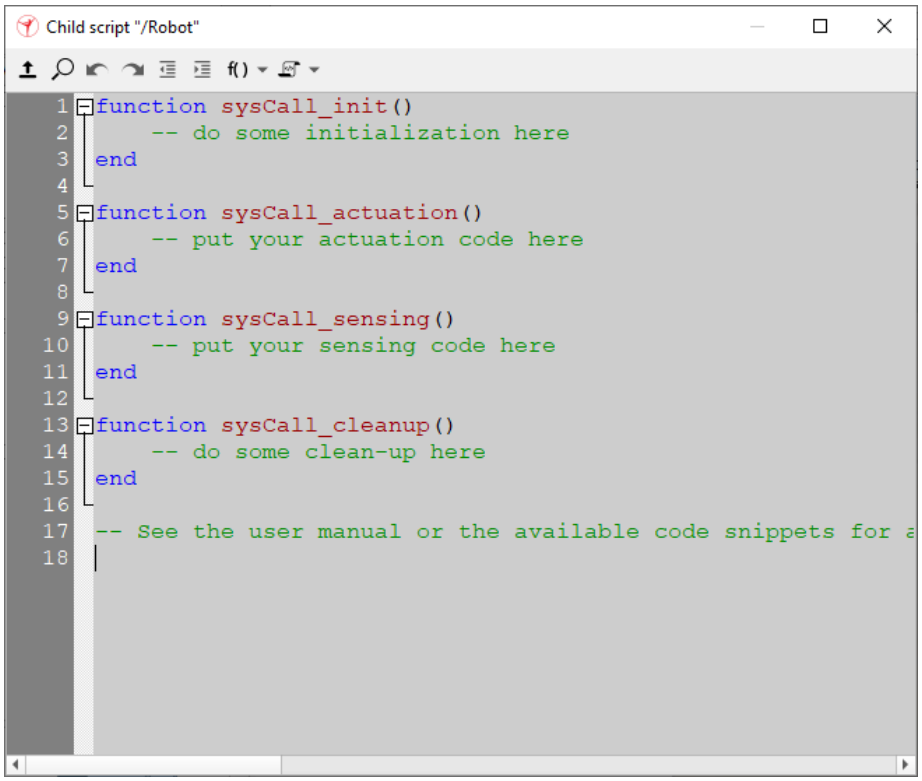
2.2 Podstawowe sterowanie robotem

W symulatorze CoppeliaSim istnieje wiele metod sterowania obiektami i wpływania na przebieg symulacji, a najprostszą z nich jest skorzystanie ze skryptów języka Lua. Obsługa tych skryptów jest wbudowana w środowisko i nie wymaga żadnych zewnętrznych narzędzi, zaś sam język Lua jest prostym językiem obiektowym zbliżonym składniowo do języka Pascal. Opis interfejsu użytkownika w CoppeliaSim dla języka Lua można znaleźć pod adresem: https://coppeliarobotics.com/helpFiles/en/apiFunctions.htm

Do naszego robota dodajemy skrypt, klikając na ikonie robota prawym przyciskiem myszy:



Skrypt składa się z czterech funkcji, która są początkowo puste.



Funkcja sysCall_init() zostanie wywołana raz, zaraz po uruchomieniu symulacji. Funkcje sysCall_actuation() oraz sysCall_sensing() będą wywoływane cyklicznie i umożliwią odpowiednio sterowanie oraz odczyt czujników robota podczas symulacji. Funkcja sysCall_cleanup() jest uruchamiana raz, na końcu symulacji. Aby wysterować robota, wystarczy nadać jego silnikom prędkość obrotową. W tym celu należy utworzyć uchwyty do obiektów w funkcji sysCall_init(), a następnie wykorzystać je do nadania prędkości silnikom:

```
function sysCall_init()
     silnik_L= sim.getObject("/Silnik_lewy")
     silnik_P= sim.getObject("/Silnik_prawy")
     sim.setJointTargetVelocity(silnik_L,3);
     sim.setJointTargetVelocity(silnik_P,3);
end
```

Po uruchomieniu symulacji, robot będzie poruszał się powoli po trajektorii prostoliniowej. Na zachowanie robota możemy wpływać zmieniając jego prędkość w cyklicznie wywoływanej funkcji sysCall_actuation(). Poniższy skrypt zmniejszy prędkość jednego z kół po 5 sekundach od rozpoczęcia symulacji, przez co robot zacznie jeździć w kółko. Po kolejnych 5 sekundach, koła robota będą się kręcić z tą samą prędkością, ale w przeciwnych kierunkach, w wyniku czego robot będzie obracał się w miejscu.

```
function sysCall_actuation()
    t1=sim.getSimulationTime()
    if (t1 > 5) then
        sim.setJointTargetVelocity(silnik_L,1)
    end
     if (t1 > 10) then
        sim.setJointTargetVelocity(silnik_L,5)
        sim.setJointTargetVelocity(silnik_P,-5)
    end
end
```

Wykorzystując liczniki czasu można łatwo napisać program, który będzie cykliczne zmieniał zachowanie robota. Przy odpowiednio dobranej długości cyklu i prędkości kół, można uzyskać na przykład trajektorię w kształcie ósemki.

```
function sysCall_init()
     silnik_L= sim.getObject("/Silnik_lewy")
     silnik_P= sim.getObject("/Silnik_prawy")
     t1=sim.getSimulationTime() -- inicjacja pomiaru czasu
(wyzerowanie stopera)
end


function sysCall_actuation()
    t2=sim.getSimulationTime()
    dt=t2-t1 -- delta t, czas od chwili wyzerowania stopera
    t=7.8 -- dlugosc  jednego kolka osemki
    if (dt < t) then -- pierwszy cykl w jedna strone
        sim.setJointTargetVelocity(silnik_L,5)
        sim.setJointTargetVelocity(silnik_P,1)
    end
    if (dt >= t and dt <2*t) then -- drugi cykl w druga strone
        sim.setJointTargetVelocity(silnik_L,1)
        sim.setJointTargetVelocity(silnik_P,5)
    end
```

```
    if (dt>=2*t) then
        t1=sim.getSimulationTime() -- resetowanie stopera
    end
end
```

2.3 Rysowanie trajektorii robota z wykorzystaniem cyklicznego bufora położenia robota

Bardzo przydatną możliwością, łatwą do zrealizowania w symulacji, a znacznie trudniejszą w przypadku rzeczywistego robota, jest opcja rysowania jego trajektorii ruchu. Pozwala to na znacznie lepszą ocenę przebiegu drogi robota niż zwykła obserwacja robota w ruchu. Do rysowania trajektorii wykorzystujemy informację o rzeczywistym położeniu robota w symulowanym świecie. Ta informacja w przypadku robota rzeczywistego jest niedostępna, nie należy więc nadużywać możliwości odczytania faktycznej pozycji robota do jego sterowania, niemniej rysowanie śladu robota takim nadużyciem nie jest. Aby to zrobić w symulatorze, trzeba najpierw dodać uchwyt do robota w funkcji sysCall_init():

```
robot= sim.getObject("/Robot")
```

a następnie dokonać właściwego odczytu pozycji w funkcji sysCall_actuation():

```
pozycja_robota=sim.getObjectPosition(robot,sim.handle_world)
```

Najprostsza metoda rysowania trajektorii zakłada wykorzystanie bufora, w którym przechowywane są kolejne współrzędne robota. Będą one następnie łączone linią. Ślad robota powinien zostać zainicjowany w funkcji sysCall_init() jak pokazano poniżej:

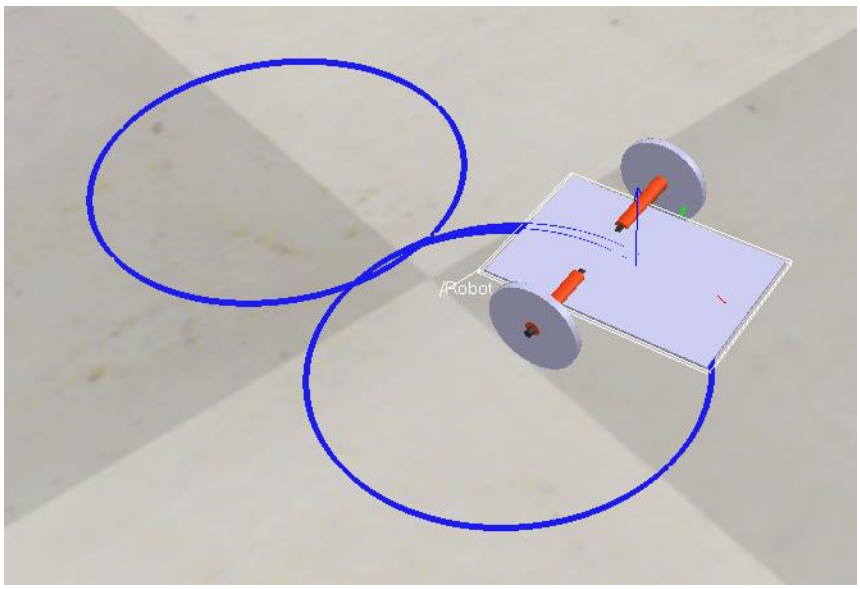
```
function sysCall_init()
robot= sim.getObject("/Robot")
sim.setJointTargetVelocity(silnik_L,3);
sim.setJointTargetVelocity(silnik_P,3);
slad_robota=sim.addDrawingObject(sim.drawing_linestrip,5,0,
-1,1000,{0,0,1})
end
```

Będzie on rysowany za pomocą linii (drawing_linestrip) o rozmiarze 5, bez pomijania żadnych punktów (0), w globalnym układzie współrzędnych (-1). Bufor ma rozmiar tysiąca punktów, a linia będzie miała kolor niebieski ({0,0,1}). Powyższe polecenie niczego jeszcze nie rysuje, jedynie inicjuje obiekt do narysowania.

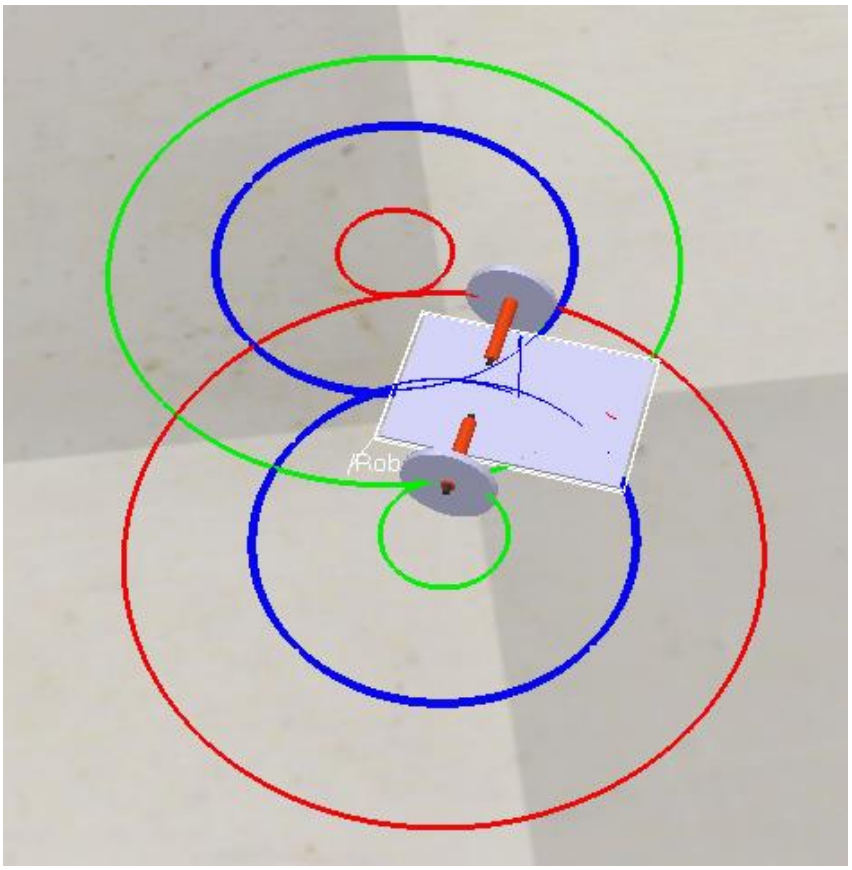
Kolejne punkty pozycji robota będą dodawane do jego trajektorii w funkcji sysCall_actuation() i automatycznie rysowane:

```
pozycja_robota=sim.getObjectPosition(robot,sim.handle_world)
sim.addDrawingObjectItem(slad_robota,pozycja_robota)
```

W efekcie, otrzymamy rysunek trajektorii robota:



Po dodaniu, w analogiczny sposób, linii dla kół otrzymamy:



Tak narysowany ślad robota będzie miał 1000 punktów i przestanie się rysować po około 50 sekundach (przy domyślnych ustawieniach symulatora). Nowe pozycje robota nie będą już rejestrowane i wizualizowane. Najprostszą radą jest zwiększenie liczby punktów, jednak może to spowodować, że rysunek śladu robota stanie się nieczytelny, gdyż nowe ślady będą się nakładać na stare. Poza tym, w przypadku długich symulacji, bufor pozycji robota musiałby być ogromny, co negatywnie odbiłoby się na wydajności symulatora. Jego twórcy przewidzieli jednak taką sytuację i umożliwiają wykorzystanie bufora cyklicznego, który po zapełnieniu będzie nadpisywał się od początku. W efekcie tego podejścia, ślad robota będzie się cały czas rysował, ale jego najstarsze punkty będą po jakimś czasie znikać. W ten sposób zawsze będziemy widzieć ostatnie n punktów trajektorii naszego robota, a wcześniejsze nie będą zaciemniać obrazu. Aby włączyć cykliczną wersję bufora punktów należy nieznacznie zmodyfikować polecenie tworzące ślad robota poprzez dodanie właściwości sim.drawing_cyclic:

```
slad_robota=sim.addDrawingObject(sim.drawing_linestrip + 
sim.drawing_cyclic,5,0,-1,1000,{0,0,1})
```

Wyżej pokazany ślad robota rysuje się w miejscu, w którym znajduje się obiekt tworzący ślad. Jest to zazwyczaj środek geometryczny takiego obiektu, a więc ślad zostawiony przez koła będzie „unosił się” nad podłożem na wysokości osi koła. W przypadku dużych kół, ślad będzie dość daleko od podłoża, a nie w miejscu styku kół z podłożem, co jest intuicyjnie oczekiwane przez użytkownika. Podczas obserwacji pionowo z góry, nie będzie to miało większego znaczenia, ale jeśli kamera będzie ustawiona pod kątem, powstanie błąd paralaksy, który może być mylący. Aby ślad wyglądał naturalniej, należy zmienić jego współrzędną Z na niewiele powyżej zera (powyżej powierzchni podłoża). Można to zrobić modyfikując trzecią współrzędną odczytanej pozycji obiektu np. tak:

```
pozycja_robota[3]=0.001
```

2.4 Scenariusz ćwiczenia

W ramach ćwiczenia należy zrealizować kolejno poniższe zagadnienia:

1. Zapoznać się z symulatorem, poznać jego interfejs, umieścić w przestrzeni roboczej kilka gotowych robotów i uruchomić symulację oraz obserwować roboty podczas jazdy.
2. Zbudować własnego robota o napędzie różnicowym.
3. Na podstawie przeprowadzonych eksperymentów wyznaczyć czas, w którym robot pokonuje zadaną odległość oraz obraca się w miejscu o zadany kąt i napisać dwie funkcje: jedna przemieszczająca robota o zadaną odległość, druga obracająca go o zadany kąt.
4. Wykorzystać funkcje z poprzedniego punktu do realizacji trajektorii o kształcie kwadratu.
5. Zwizualizować faktycznie zrealizowane trajektorie i określić błędy odometrii robota.

2.5 Pytania sprawdzające

1. Jakie są zalety napędu różnicowego?

Napęd różnicowy jest prosty w sterowaniu, a robot może zmienić swą orientację w miejscu, co pozwala na realizację dowolnych trajektorii ruchu. Ponadto, roboty o napędzie różnicowym zbudowane na podstawie w kształcie koła są w stanie odwrócić się w miejscu także w wąskich przestrzeniach, dzięki czemu mogą z nich wyjechać bez problemu przodem, a nie tyłem, co może być istotne w przypadku, gdy jakiś czujnik (np. kamera) umieszczony jest wyłącznie na przodzie robota.

2. Jaki wpływ na trajektorię robota ma podpierające go trzecie koło samonastawne?

Koło samonastawne powoduje nieznaczną zmianę orientacji robota, zwłaszcza w przypadku wykonywania obrotu w miejscu o duży kąt lub zmiany kierunku jazdy o 180 stopni bez obracania się. Jest to szczególnie widoczne w tym ostatnim przypadku. Koło podpierające zmienia wówczas swą orientację o 180 stopni i w czasie przekręcania się wpływa na trajektorię robota.

3. Czy istnieją inne rozwiązania niemające wad koła samonastawnego?

Tak, istnieją. W przypadku małych i lekkich robotów koło samonastawne można zastąpić ślizgaczami albo kulką. Oba rozwiązania wymagają także, aby powierzchnia po której porusza się robot była równa i czysta, w przeciwnym razie ślizgacze będą zawadzały o nierówności, a kulka zatrze się w swoim uchwycie.

4. Co zrobić w symulatorze CoppeliaSim, aby rysowana trajektoria zawsze pokazywała ostatnie n punktów, a najstarsze punkty były kasowane?

Do funkcji tworzącej trajektorię należy dodać opcję sim.drawing_cyclic.

5. Czy w przypadku robota o dużych kołach, ich trajektoria utworzona w standardowy sposób będzie poprawna?

Taka trajektoria znajdzie się na wysokości osi kół, a więc ślad koła nie znajdzie się na podłożu, co jest intuicyjnie oczekiwane przez użytkownika. Podczas obserwacji z góry nie będzie to miało większego znaczenia, ale przy obserwacji pod kątem pojawi się błąd paralaksy. Będzie on tym większy im większe są koła. Aby uniknąć tego błędu, należy przypisać Z-owej współrzędnej śladu wartość niewiele większą od zera.

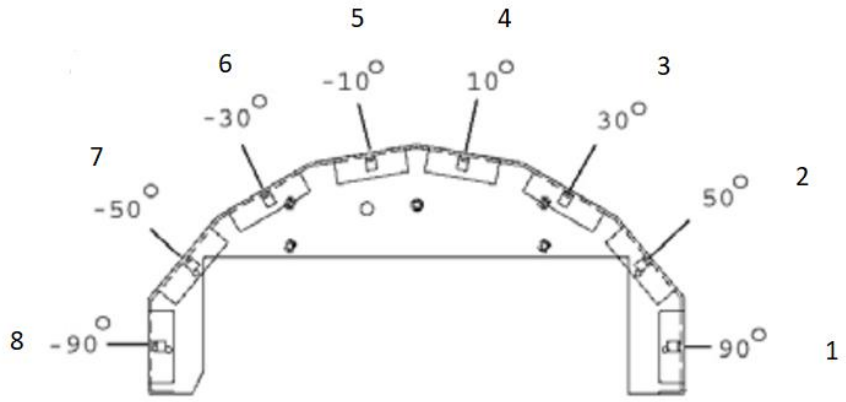
3 Behawioralne sterowanie robotem w zadaniu omijania przeszkód

Drugie ćwiczenie dotyczy zagadnienia sterowania behawioralnego. Wykorzystana w nim zostanie gotowa scena oraz robot z czujnikami ultradźwiękowymi z biblioteki CoppeliaSim. Zarówno scena, jak i robot będą wymagały nieznacznych modyfikacji w trakcie ćwiczenia. Trajektoria robota zostanie zwizualizowana w odmienny sposób niż w ćwiczeniu pierwszym, a ostatecznym celem tego ćwiczenia będzie takie wysterowanie robota, aby odwiedził on wszystkie dostępne miejsca w swoim otoczeniu.

3.1 Przygotowanie symulacji

W bieżącym ćwiczeniu zostanie wykorzystana gotowa scena mobileRobotVisualTraces.ttt oraz robot Pioneer3DX z biblioteki symulatora, wyposażony w czujniki ultradźwiękowe. Po załadowaniu wspomnianej sceny, należy usunąć z niej występujące tam roboty oraz dodać robota Pioneer. Ten robot jest wyraźnie wyższy od domyślnych robotów w tej scenie i jego czujniki ultradźwiękowe znajdują się ponad przeszkodami. Konieczne jest więc podwyższenie wszystkich przeszkód znajdujących się wokół robota.

Robot Pioneer3DX jest wyposażony w 16 ultradźwiękowych czujników odległości, umieszczonych dookoła robota, jednak na potrzeby tego ćwiczenia skorzystamy z połowy tych czujników usytuowanych na przodzie maszyny, co pokazano na rysunku poniżej:



3.2 Algorytm Braitenberga

Algorytm Braitenberga to koncepcja zaproponowana w eksperymencie myślowym przez włosko-austriackiego cybernetyka Valentino Braitenberga, zakładająca bezpośrednie połączenie pomiędzy czujnikami a elementami wykonawczymi. Pojazd Braitenberga to agent, który może poruszać się autonomicznie na podstawie informacji sensorycznych. Posiada prymitywne sensory, które mierzą pewien bodziec w danym miejscu (np. odległość) oraz

niezależnie napędzane koła pełniące rolę elementów wykonawczych, czyli efektorów. W najprostszej konfiguracji czujnik jest bezpośrednio połączony z efektorem, dzięki czemu wykrywany sygnał natychmiast powoduje ruch koła.

W zależności od sposobu połączenia czujników i kół, pojazd wykazuje różne, często złożone i pozornie inteligentne zachowania – wydaje się on dążyć do osiągnięcia określonych celów, jednocześnie unikając innych sytuacji, zmieniając kierunek ruchu w reakcji na stan otoczenia.

Idea algorytmu Braitenberga jest bardzo prosta. Każdy z czujników wpływa w pewien określony sposób na prędkość obu kół robota. Wpływ poszczególnych czujników na prędkość kół jest uzależniony od położenia czujnika oraz sygnału z tego czujnika, który jest odwrotnie proporcjonalny do zmierzonej przez czujnik odległości. W domyślnej implementacji czujniki zwracają wartość w zakresie od 0 do 1, gdzie wartość 0 oznacza brak przeszkody w promieniu 50 cm (bezpiecznie, nie trzeba podejmować żadnej akcji), zaś wartość 1 oznacza występowanie przeszkody w odległości 20 cm lub mniejszej (niebezpiecznie, trzeba skręcić). Każdy z czujników indywidualnie wpływa na zachowanie obu kół robota ustawiając ich prędkość na wartość będącą iloczynem sygnału z czujnika oraz predefiniowanej prędkości dla tego czujnika. Ostateczna prędkość danego koła jest wypadkową (sumą) wszystkich zachowań (iloczynów dla wszystkich czujników), w tym także zachowania domyślnego, jakim jest przypisanie prędkości V0=2 rad/s obu kołom. Wpływ wszystkich czujników na koła zdefiniowany jest następująco:

Nr czujnika	1	2	3	4	5	6	7	8
dV_l	– 0,2	– 0,4	– 0,6	– 0,8	– 1,0	– 1,2	– 1,4	– 1,6
dV_p	– 1,6	– 1,4	– 1,2	– 1,0	– 0,8	– 0,6	– 0,4	– 0,2

gdzie dV_l oznacza zmianę prędkości lewego koła, a dV_p zmianę prędkości koła prawego.

Ostateczna prędkość dla kół wyraża się wzorem:

$$V_{l,p} = V0 + \sum_{i=1}^{8} s_i dV_{l,p}$$

Ten prosty algorytm, będący de facto fuzją 9 elementarnych zachowań, jest zadziwiająco skuteczny. Nie wymaga on złożonej logiki polegającej na sprawdzaniu, które czujniki zwracają mniejsze, a które większe odległości i dobierania odpowiedniej prędkości kół. Tu siła reakcji robota wynika z sumy wpływu poszczególnych zachowań. Inną zaletą tego podejścia jest też fakt, że sterowanie odnosi się bezpośrednio do kół, a nie do robota. Algorytm ten zadziała więc na najprostszym robocie, którego sterownik potrafi wysterować silniki z zadaną prędkością, a niekoniecznie jest w stanie obrócić całego robota o zadany kąt.

Przykładowo, jeśli robot wykryje bardzo bliską przeszkodę tylko ze swojej lewej strony, jedynie czujnik nr 1 zwróci wartość 1, zaś pozostałe zwrócą zera. Zatem prędkości obu kół wyniosą:

V_l=2-0,2=1,8

V_p=2-1,6=0,4

Prędkość lewego koła nieznacznie spadnie, zaś prawe koło wyraźnie zwolni – robot skręci w prawo, oddalając się od przeszkody.

Jeśli przeszkody wystąpią z lewej strony oraz przed robotem, czyli aktywowane zostaną czujniki 1 – 4, wówczas:

V_l=2-0,2-0,4-0,6-0,8=0

V_p=2-1,6-1,4-1,2-1,0=-3,2

Robot obróci się w prawo wokół nieruchomego lewego koła. Chwilę po rozpoczęciu ruchu odczyty z czujników się zmienią i czujnik nr 4 przestanie widzieć przeszkody, a wówczas wysterowanie kół będzie następujące:

V_l=2-0,2-0,4-0,6=0,8

V_p=2-1,6-1,4-1,2=-2,2

Robot nadal będzie się obracał w prawo, ale już nie wokół nieruchomego lewego koła, które zacznie powolny ruch do przodu. Po pewnym czasie, gdy robot dostatecznie odwróci się od przeszkody, oba koła zaczną obracać się do przodu i robot będzie kontynuował jazdę po ominięciu przeszkody.

W tym miejscu można się zastanowić czy domyślne parametry prędkości robota są optymalne. Widać, że w przypadku, gdy robot wjedzie w zamknięty zaułek, gdzie wszystkie czujniki zwrócą wartość 1, prędkości kół wyniosą wówczas -5,2. Czyli w takiej sytuacji robot będzie się wycofywał znacznie szybciej niż normalnie jedzie do przodu. To nie wydaje się dobrym pomysłem. Raczej chcielibyśmy, aby wycofywał się wolniej niż jechał do przodu. Należałoby więc zmienić nieco ustawienia prędkości podstawowej lub prędkości reakcji dla czujników albo dodać proste przycinanie prędkości do wybranej wartości maksymalnej. Należy jednak zwrócić uwagę, że manipulowanie parametrami zachowań robota może prowadzić do nieoczekiwanych rezultatów (np. zablokowanie się robota w rogu), a nawet nieznaczne zmiany tych parametrów mogą drastycznie zmienić kształt jego trajektorii.

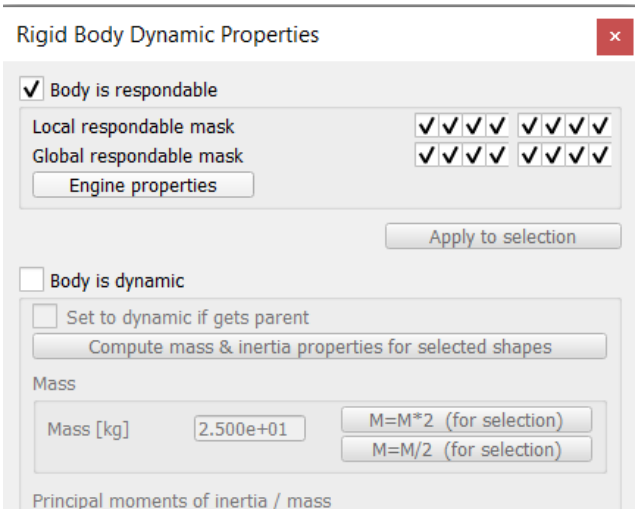
3.3 Metoda przełączanych zachowań

Alternatywnym podejściem do sterowania robota wobec algorytmu Braitenberga (być może bardziej intuicyjnym) jest przełączanie (zamiast sumowania) jego zachowań w obliczu różnych bodźców. W tym przypadku definiujemy (często wykluczające się wzajemnie, ale niekoniecznie) zachowania typu: przeszkoda z przodu – cofaj, przeszkoda bliżej lewej – skręcaj

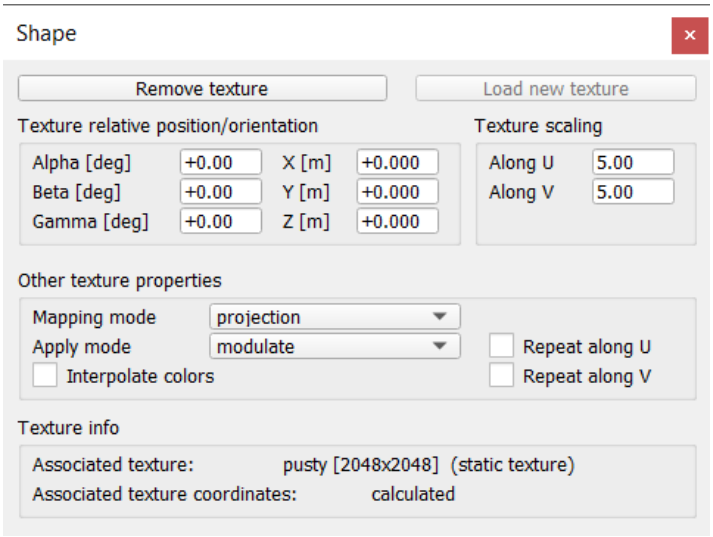
w prawo, przeszkoda bliżej prawej – skręcaj w lewo itp. Do tego należy dodać zachowanie podstawowe jakim jest jazda do przodu oraz, co może być przydatne, wycofanie w przypadku kolizji z przeszkodą. Takie podejście jest oderwane od prostego schematu bezpośredniej reakcji efektorów na bodźce, ale pozwala na zaprojektowanie bardziej złożonego algorytmu sterowania, co może dać lepsze efekty, ale i wymaga większych mocy obliczeniowych. Nadmierne skomplikowanie algorytmu przez zastosowanie złożonych obliczeń oraz wielopoziomowych instrukcji warunkowych może doprowadzić do tego, że pętla przetwarzania będzie wykonywała się zbyt długo, przez co robot (o niskiej wydajności obliczeniowej) nie zdąży na czas zrealizować obliczonego sterowania. W efekcie, jego reakcje będą spóźnione i zbyt silne lub zbyt słabe, a trajektoria niezgodna z oczekiwaniami.

3.4 Rysowanie trajektorii robota z wykorzystaniem tekstur

W niektórych przypadkach przydatna bywa rejestracja całego przebiegu trajektorii robota. Co więcej, istotna może być także informacja o tym, gdzie robot bywa częściej, a gdzie rzadziej czy też którędy chętniej się porusza. Poprzednia metoda nie udostępnia takiej funkcjonalności, ale symulator CoppeliaSim pozwala na wizualizację przebytej przez robota ścieżki metodą bezpośredniego malowania na powierzchni, po której się on porusza. W przeciwieństwie do poprzedniego sposobu, nie tworzymy tu nowych punktów, które na bieżąco łączymy linią, ale modyfikujemy graficznie teksturę obiektu stanowiącego podłogę dla robota. Procedura malowania po teksturze jest nieco bardziej złożona. Należy zacząć od stworzenia pustego obrazu w formacie PNG o wymiarach przynajmniej 2048 na 2048 pikseli (większych tekstur CoppeliaSim nie obsługuje). Taki obraz przypiszemy teksturze podłogi i będziemy po nim malować. Jako, że domyślna podłoga w symulatorze (Floor) jest obiektem złożonym, najprościej będzie ją skasować i stworzyć nową wykorzystując płaszczyznę (Plane). Należy pamiętać, że nowa podłoga musi podpierać naszego robota, a więc reagować z nim, ale nie w sposób dynamiczny – ma być statyczną, nieruchomą, nieprzenikalną podstawą. Musimy zatem zmienić jej właściwości w panelu właściwości dynamicznych (dostępnym w głównym oknie właściwości obiektu pojawiającym się po jego dwukrotnym kliknięciu):



Następnie z głównego panelu właściwości należy wybrać opcję Texture i załadować utworzony wcześniej plik przeskalowując go do maksymalnych wymiarów (2048x2048). Domyślnie tekstura pokrywa obszar o wymiarach 1x1 metr. Nasza podłoga będzie zapewne znacznie większa. Musimy więc przeskalować teksturę tak, by ją w całości pokryła, a nie powtarzała się jak płytki na domyślnej podłodze. W tym celu modyfikujemy parametry tekstury jak na rysunku poniżej, gdzie tekstura została przeskalowana 5 razy w każdym kierunku, gdyż takie były wymiary podłogi:



Pozostaje teraz dodać skrypty obsługujące teksturę. Jeden z nich będzie zawsze czyścił teksturę po każdorazowym uruchomieniu programu (nie chcemy zazwyczaj mieć na niej śladów poprzednich przejazdów). W tym celu dodajemy skrypt języka Lua do obiektu podłogi, analogicznie jak to pokazano na rys. xx i modyfikujemy funkcję sysCall_init() dodając do niej polecenia wypełniające teksturę kolorem białym:

```
teksturaPodlogi=sim.getShapeTextureId(sim.getObject('.')) --kropka
oznacza biezacy obiekt

local kolor=sim.packUInt8Table({255,255,255}) --kolor, ktorym
zostanie wypelniona tekstura

sim.writeTexture(teksturaPodlogi,0,kolor)
```

Tak przygotowana tekstura jest gotowa do tego, by po niej malować. Robimy to w skrypcie obsługującym robota. W funkcji sysCall_init() tworzymy uchwyt do tekstury oraz definiujemy kolor śladu:

```
teksturaPodlogi=sim.getShapeTextureId(sim.getObject('/Podloga'))

kolor=sim.packUInt8Table({255,0,0}) -- kolor czerowny
```

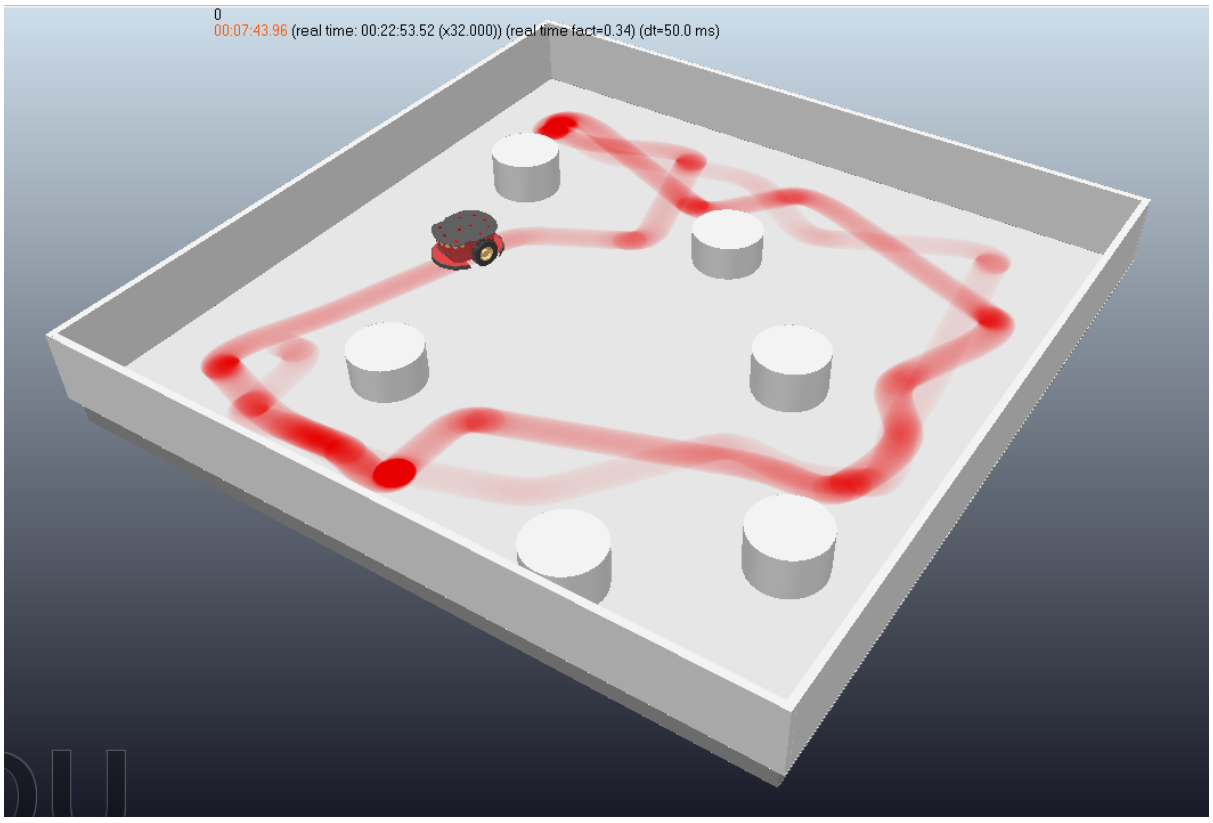
Następnie w funkcji sysCall_actuation() lub sysCall_sensing() rysujemy kółko w miejscu, w którym jest robot względem podłogi:

```
local p=sim.getObjectPosition(robot,sim.getObject('/Podloga'))

sim.writeTexture(teksturaPodlogi,4,kolor,math.floor(2048*(0.5+p[1]/5
))-r,math.floor(2048*(0.5+p[2]/5))-r,r*2,r*2,254/255)
```

Liczba 2048 jest wymiarem naszej tekstury, p[1] i p[2] to współrzędne x i y robota względem podłogi, zaś liczba 5 to jej wymiar w metrach. Układ współrzędnych podłogi zaczepiony jest domyślnie w jej środku, stąd pojawia się odpowiedni wzór na przeliczenie współrzędnych robota na piksele tekstury. Drugi parametr funkcji writeTexture (liczba 4) oznacza w tym przypadku, że nie cała tekstura zostanie pokryta wybranym kolorem (jak wcześniej z parametrem 0), ale jedynie elipsa (w naszym przypadku koło) wpisana w prostokąt (w naszym przypadku kwadrat) definiowany przez współrzędne lewego górnego rogu (dlatego od współrzędnych robota odejmujemy r) i długości boków (2*r). Ostatni parametr to intensywność malowanego śladu. Jej mała wartość spowoduje, że kolejne przejazdy robota w tym samym miejscu zostawią coraz mocniejszy ślad, a więc będzie widać gdzie robot bywał częściej lub spędził więcej czasu, co może być istotną informacją na temat zachowania robota.

W efekcie zastosowania powyższego skryptu otrzymamy taki ślad robota:



Widać wyraźnie, w których miejscach robot spędzał więcej czasu, którą trasę chętniej wybierał, a którą rzadziej.

3.5 Scenariusz ćwiczenia

W ramach ćwiczenia należy zrealizować kolejno poniższe zagadnienia:

1. Załadować scenę mobileRobotVisualTraces.ttt, usunąć występujące w niej roboty i dodać robota Pioneer3DX z biblioteki symulatora. Zmodyfikować przeszkody (zwiększyć ich wysokość) tak, aby robot wykrywał je swoimi czujnikami ultradźwiękowymi.
2. Skopiowac algorytm rysowania śladu z jednego z małych robotów do Pioneera, po czym skasować małe roboty. Zmodyfikować ten algorytm tak, aby robot zostawiał ślad nie co określony dystans, ale co określony czas. Ustalić ten czas na 0,1 sekundy.
3. Uruchomić robota z domyślnym algorytmem Braitenberga i zapisać ślad.
4. Zmodyfikować algorytm sterowania robotem tak, aby robot nie zapętlał się na powtarzalnej ścieżce, dojeżdżał do przeszkód i odwiedzał wszystkie dostępne miejsca. Można w tym celu wprowadzić dodatkowe losowe zachowania takie jak zmiana kierunku ruchu od czasu do czasu, jazda po łuku o różnym promieniu, zmieniającym się w czasie itp.
5. Dobrać parametry zachowań robota tak, aby cała powierzchnia podłogi była pokryta przez robota możliwie równomiernie (podłoga powinna być możliwie równomiernie czerwona).

3.6 Pytania sprawdzające

1. Co to jest sterowanie behawioralne i czym się charakteryzuje?
   Sterowanie behawioralne jest to fuzja elementarnych zachowań w reakcji na bodźce zewnętrzne, np. skręt w lewo, gdy wykryto przeszkodę z prawej strony, zmniejszenie prędkości w obecności przeszkód, jazda po spirali na wolnej przestrzeni. Zachowanie robota, które jest wypadkową wszystkich zachowań składowych może być skomplikowane i sprawiać pozory inteligentnego, jednak robot nie planuje w żaden sposób swojego działania, a jedynie reaguje na bodźce zewnętrzne – najczęściej na informacje o wykrytych przeszkodach.
2. Czy w sterowaniu behawioralnym potrzebna jest mapa otoczenia?
   Nie, nie jest potrzebna, a co więcej, wykorzystanie mapy (a więc planowanie) nie jest elementem sterowania behawioralnego.
3. Czy sterowanie behawioralne jest stosowane w przypadku robota mającego mapę otoczenia i planującego trajektorię według tej mapy?
   Tak, roboty wykorzystujące mapę i planujące swoje działanie najczęściej korzystają także z podejścia behawioralnego, gdyż zazwyczaj mapa nie odzwierciedla rzeczywistości w stu procentach – w otoczeniu robota mogą pojawić się ludzie czy inne obiekty, których nie ma na mapie.
4. Na czym polega algorytm Braitenberga?
   Algorytm Braitenberga zakłada bezpośrednie połączenie pomiędzy czujnikami a elementami wykonawczymi. Pojazd Braitenberga to agent, który posiada prymitywne

sensory mierzące pewien bodziec w danym miejscu (np. odległość) oraz niezależnie napędzane koła pełniące rolę elementów wykonawczych, czyli efektorów. W najprostszej konfiguracji czujnik jest bezpośrednio połączony z efektorem, dzięki czemu wykrywany sygnał natychmiast powoduje ruch koła. W zależności od sposobu połączenia czujników i kół, pojazd wykazuje różne, często złożone i pozornie inteligentne zachowania.

5. Czy do równomiernego pokrycia powierzchni pomieszczenia wystarczy odpowiednia modyfikacja algorytmu Braitenberga czy też niezbędne jest wprowadzenie dodatkowych zachowań robota?

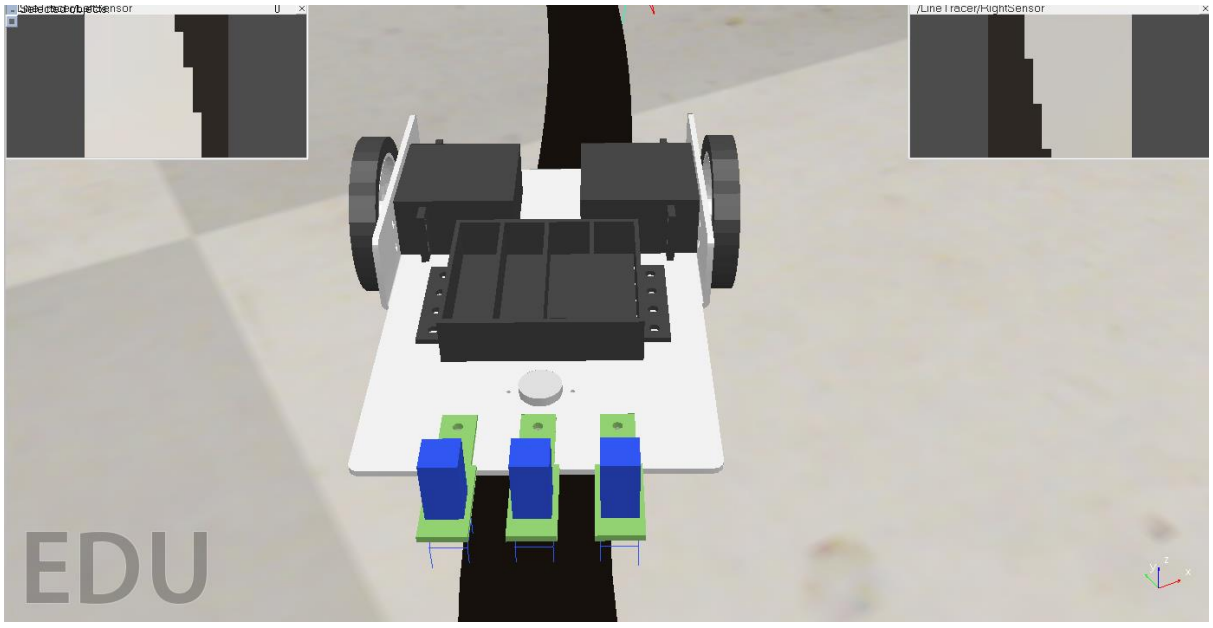
   Wprowadzenie różnych złożonych algorytmów behawioralnych może przynieść bardzo dobre efekty, niemniej odpowiednio dobrane parametry podstawowego algorytmu Braitenberga mogą pozwolić na uzyskanie zbliżonego efektu przy znacząco niższej złożoności algorytmu sterowania. Sam proces doboru tych parametrów może być jednak trudny, jako że sterowanie z przełączanymi zachowaniami wydaje się być nieco bardziej intuicyjne.

4 Zaawansowane sterowanie robotem na przykładzie zadania śledzenia linii

Celem trzeciego ćwiczenia jest zbadanie kilku prostych algorytmów sterowania robota o napędzie różnicowym w zagadnieniu śledzenia linii. W tym ćwiczeniu zostanie wykorzystany gotowy robot dedykowany do tego typu zadań, czyli line follower, dostępny w bibliotece CoppeliaSim. Jest to prosty robot o napędzie różnicowym wyposażony w 3 czujniki wizyjne. W ramach ćwiczenia należy otworzyć wstępnie przygotowaną scenę sledzenie_linii.ttt i rozbudować skrypt sterujący robotem aby zweryfikować kilka różnych algorytmów sterowania, począwszy od najprostszego, typu włącz-wyłącz, a skończywszy na regulatorze PID.

W tym ćwiczeniu przyjmujemy, że maksymalna prędkość obrotowa silników robota to nieco ponad 3 obroty na sekundę, czyli w przybliżeniu 20 radianów na sekundę. Robimy to po to, by uzyskane wyniki były porównywalne pomiędzy zespołami. W ostatniej części ćwiczenia wszelkie ograniczenia na prędkość kół zostaną zdjęte.

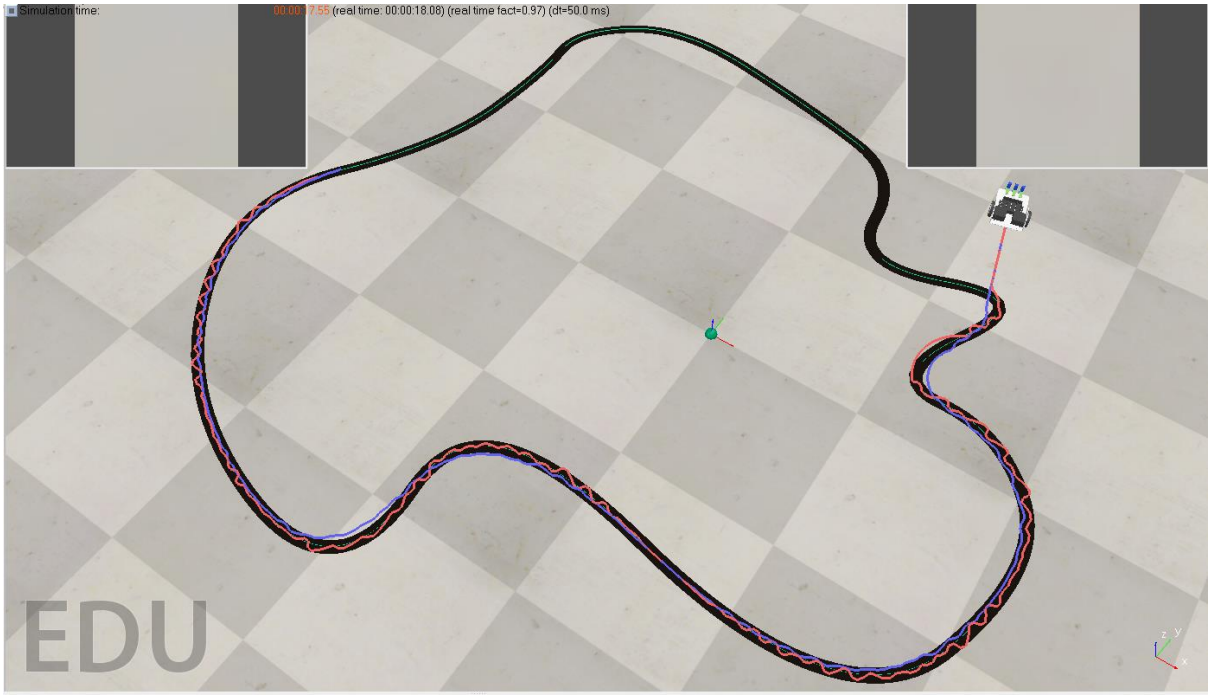
Pozycja początkowa robota zakłada, że jest on ustawiony centralnie na linii, a dwa skrajne czujniki widzą jej fragment:



4.1 Sterowanie typu włącz-wyłącz

W najprostszej wersji algorytmu sterowania będziemy zmniejszać prędkość jednego koła w momencie, gdy któryś z czujników znajdzie się całkowicie nad czarną linią. Będzie to oznaczać, że robot nie jedzie wzdłuż linii, ale że na nią najechał. W takiej sytuacji robot skręci w odpowiednim kierunku, aż linia znajdzie się z powrotem między czujnikami – wówczas oba koła będą obracały się z tą samą prędkością, a robot będzie jechał prosto. Prosto, ale

niekoniecznie wzdłuż linii. Po chwili robot przekroczy linię z drugiej strony i skoryguje swoją trajektorię w przeciwnym kierunku. Ruch robota będzie miał charakter silnie oscylacyjny, a jego reakcja na zjechanie z linii będzie zawsze tak samo silna, niezależnie od tego czy robot zjedzie z linii dużo czy mało (gdyż zawsze, gdy jeden z czujników znajdzie się nad czarną linią tak samo przyhamowujemy jedno z kół). Siła reakcji robota będzie zależała od tego jak bardzo zwolnimy jedno z kół. Niewielka siła reakcji (niewielkie spowolnienie) da w efekcie gładszą trajektorię, ale nie pozwoli na pokonanie łuków o mniejszym promieniu. Te będą wymagały mocniejszej reakcji, która na odcinkach prostych będzie zbyt duża i wyraźnie spowolni robota wprowadzając go w zbędne oscylacje. Bardzo prawdopodobne jest, że robot nie będzie w stanie pokonać ostrych zakrętów i wypadnie z trasy, a w najlepszym przypadku zetnie zakręt.



4.2 Sterowanie proporcjonalne z przełączaniem

Jednakowa reakcja robota na każdorazowe zjechani z linii nie jest najkorzystniejszym rozwiązaniem. Znacznie lepiej by było, gdyby robot reagował proporcjonalnie do tego jak bardzo oddali się od linii. Wówczas niewielka odchyłka będzie skutkować niewielką korektą, a nie gwałtownym szarpnięciem. W pierwszym przypadku wykrywaliśmy zjechanie robota z linii, gdy jeden z czujników znalazł się całkowicie nad nią. Tym razem chcemy wykryć nawet niewielkie odchylenie robota. Za błąd pozycji najprościej przyjąć różnicę wskazań lewego i prawego czujnika światła (które zwracają poziom jasności obserwowanego fragmentu podłoża). Różnica zerowa oznacza, że robot jest umieszczony symetrycznie nad linią, zaś każda zmiana tej wartości jest proporcjonalna do oddalenia się robota w lewo lub prawo od linii. Wystarczy teraz odpowiednio, proporcjonalnie do różnicy, zmniejszyć prędkość jednego koła lub jednocześnie spowolnić jedno koło, a przyspieszyć drugie (oba podejścia są stosowane). Przykładowy kod spowalniający jedno koło podany jest niżej. Jeżeli błąd jest dodatni, oznacza

to, że prawy czujnik widzi mniej czarnej linii niż lewy, a więc robot zboczył w prawo – trzeba zatem spowolnić jego lewe koło, by robot skręcił w lewo.

```
k=2
e=jasnosc_P-jasnosc_L
if(e>0) then v_L=v0-k*e end
if(e<0) then v_P=v0+k*e end
```

Kluczowy w powyższym kodzie jest dobór stałej k. Zbyt mała wartość spowoduje, że robot będzie wypadał z ostrych zakrętów, a zbyt duża, że będzie wpadał w oscylacje na odcinkach prostych. Są to typowe wady podstawowego regulatora proporcjonalnego, który nie będzie w stanie sprowadzić uchybu układu do zera, a wzmocnienie jego działania jest możliwe jedynie przez zwiększenie stałej k, co objawia się przesterowaniami i oscylacjami. Pewnym wyjściem z sytuacji jest zmiana wartości wzmocnienia k dla różnych poziomów błędu. Innymi słowy, dla niskich wartości błędów przyjmujemy niski współczynnik k, który doprowadza uchyb do akceptowalnej wartości, ale nie wprowadza oscylacji. Natomiast w przypadku, gdy uchyb gwałtownie się zwiększy (robot wjedzie w zakręt) podnosimy wartość stałej k, by reakcja robota była wyraźnie silniejsza. Taki przełączany regulator będzie działał znacznie lepiej niż sterowanie włączy–wyłącz i lepiej niż pojedynczy regulator proporcjonalny, niemniej dobór wartości oraz progów przełączania wzmocnienia k będzie arbitralny, a ostateczne efekty dalekie od optymalnych.

4.3 Regulator PID

Regulator proporcjonalny zmniejsza błąd regulacji, ale go nie zeruje. Dodanie członu całkującego, którego działanie jest proporcjonalne do całki, a w zasadzie sumy, poprzednich błędów wyeliminuje ten problem. Jeśli błąd z części proporcjonalnej cały czas występuje, to jego całka (suma) będzie szybko rosnąć, zatem szybko pojawi się dodatkowe sterowanie ze strony członu całkującego. Owo sterowanie wyzeruje błąd, a więc doprowadzi robota na środek linii. Najczęściej robot przekroczy linię i wówczas błąd zmieni swój znak. Całka (suma) błędów zacznie więc maleć, a więc i osłabi się dodatkowe działanie członu całkującego, a robot powróci nad linię. Kluczowy jest tu dobór wzmocnienia członu całkującego. Zbyt duża wartość spowoduje przesterowania i oscylacyjną jazdę, a zbyt mała zapewni wprawdzie gładką trajektorię, ale może być niewystarczająca na ostrych zakrętach.

Ostatnim członem regulatora PID jest człon różniczkujący. W przypadku dyskretnego układu sterowania będziemy wyznaczać zwykłą różnicę pomiędzy bieżącym a poprzednim błędem. Jeśli błąd spada, wynik różnicy będzie ujemny, a więc człon różniczkujący zmniejszy wysterowanie. Ten człon może się przydać szczególnie w sytuacji, kiedy robot nagle wróci na środek linii (bo ta gwałtownie skręci), a scałkowany błąd jest jeszcze duży i człon całkujący ma cały czas tendencję do przesterowania. Wówczas człon różniczkujący powinien ją zahamować.

4.4 Szczególne uwarunkowania zadania

W przypadku robota śledzącego linię charakter uchybu jest nieco inny niż np. w zadaniu wysterowania silnika. W przypadku silnika, który obraca się zbyt wolno lub zbyt szybko, uchyb jest cały czas ujemny lub dodatni, a zmienia się jedynie jego wartość – jest tym większa im prędkość obrotowa silnika bardziej odbiega od zadanej. Człon całkujący cały czas zwiększa więc swoją wartość, a co za tym idzie, wysterowanie. W przypadku robota śledzącego linię dzieje się podobnie, ale tylko do momentu, kiedy oba czujniki znajdą się po jednej stronie linii (robot zjedzie z linii) – wówczas błąd będący różnicą wskazań czujników jest bliski zeru (ze względu na niejednorodność podłoża), a więc algorytm przestaje zmieniać swoje nastawy, a robot odjeżdża od linii. Istotne jest więc aby szybko zareagować na narastający błąd i za wszelką cenę nie dopuścić robota do zjechania z linii.

Drugim ważnym aspektem jest krok próbkowania w symulacji. Domyślnie ustawiany jest on na 50 ms. Oznacza to, że robot będzie reagował na otaczający go świat 20 razy na sekundę. Tak często będą odpytywane czujniki i wysterowywane koła. Problem pojawia się w zadaniach wymagających szybkiej reakcji robota, a takim jest zadanie śledzenia linii. Przez 50 ms robot może przesunąć czujniki znad linii poza linię, a algorytm sterowania tego nie wychwyci. Będzie to szczególnie dobrze widoczne przy dużych prędkościach, kiedy reakcja układu sterowania powinna być natychmiastowa. Jedyną radą w takiej sytuacji jest zmniejszenie kroku próbkowania (Simulation/Settings/Simulation dt). Nie można jednak tego kroku zmniejszać w nieskończoność, jako że rzeczywiste układy sterowania będą miały jakąś skończoną wydajność i nie pozwolą na dokonanie pomiarów i wykonanie obliczeń oraz wystawienie wysterowań w zerowym czasie.

Krok próbkowania w scenie do tego ćwiczenia został ustawiony na 10 ms.

4.5 Wizualizacja parametrów robota

Nie mniej ważnym od pokazania trajektorii robota jest wizualizacja innych jego parametrów, a w szczególności prędkości robota czy poszczególnych kół, tego co widzą czujniki wizyjne czy też parametrów sterowania. Obserwacja tych wielkości i ich zmian w czasie jest bardzo pomocna przy projektowaniu algorytmów sterowania.

Aby dokonać wizualizacji tego co widzi czujnik, należy prawym przyciskiem myszy kliknąć na dowolnym punkcie sceny i z wyskakującego menu wybrać Add/Dloating view. Pojawi się wówczas okienko podglądu czujnika. Pozostaje teraz wybrać w hierarchii obiektów interesujący nas czujnik i przypisać go do wybranego okienka (klikając na nim prawym klawiszem myszki i wybierając opcję View\Associate view with selected vision sensor. Efekt wizualizacji dwóch czujników robota śledzącego linię pokazano na początku tego rozdziału.

Wizualizacja danych numerycznych w postaci wykresów zmienności tych danych w czasie możliwa jest po dodaniu analogicznego okienka wykresów (prawy guzik myszy, Add/Graph).

Tym razem przypisanie danych do wykresu nie jest już tak proste jak poprzednio. W funkcji sysCall_init() należy zainicjować strumień danych i powiązać go z okienkiem wykresu:

```
v_robot=sim.addGraphStream(sim.getObject('/Graph[2]'),'predkosc
robota', 'm/s', 0,{1,0,0})
```

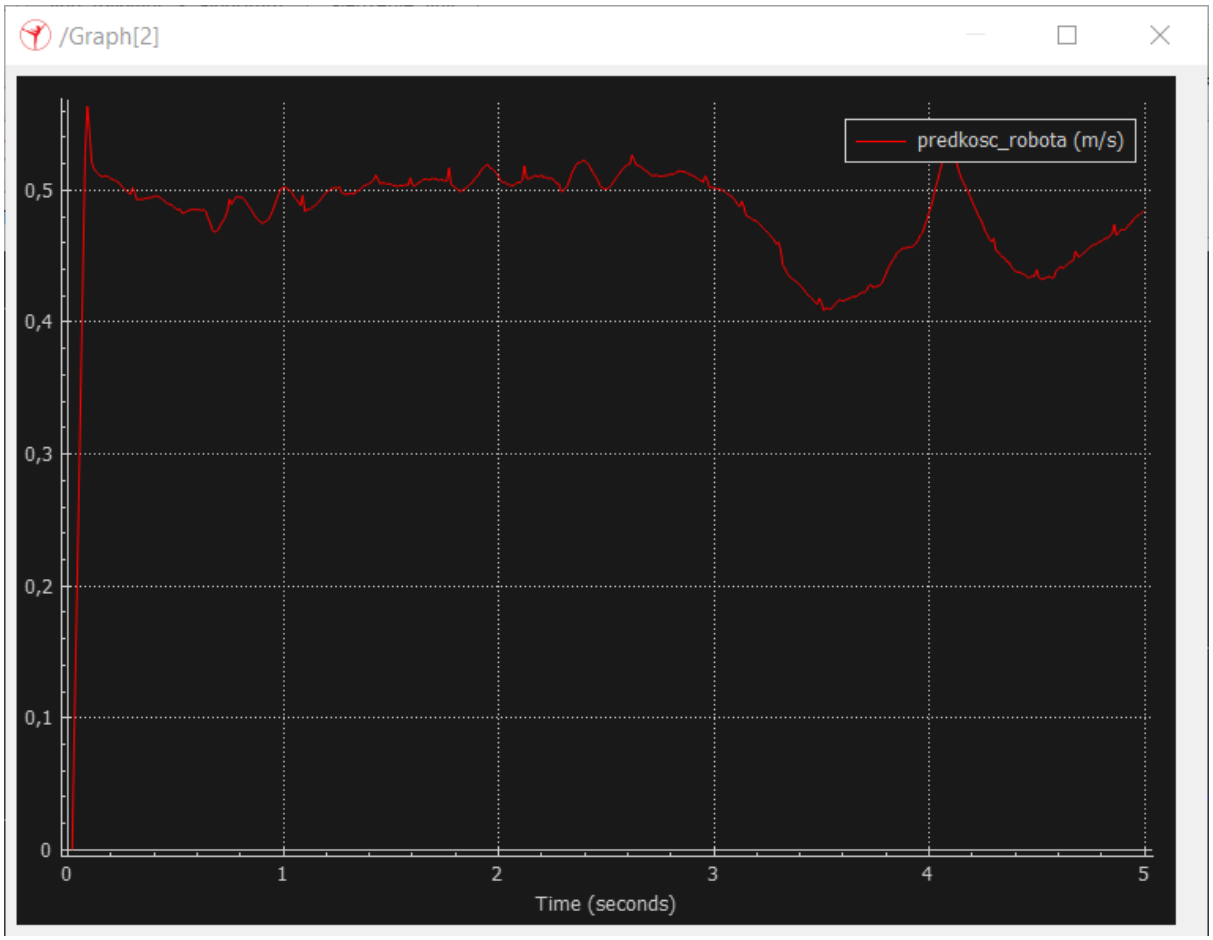
Graph[2] to wybrane okienko graficzne, potem następuje opis wykresu i jednostka, parametr 0 (nie zmieniać) i na końcu kolor wykresu. Wykres domyślnie przechowuje 1000 punktów, więc przy czasie próbkowania 10 ms, będzie to zaledwie 10 sekund. Aby zwiększyć pojemność bufora należy kliknąć dwukrotnie na obiekt wykresu w drzewie obiektów i zwiększyć rozmiar bufora. Dodanie kolejnych punktów wykresu odbywa się w funkcji sysCall_actuation() lub sysCall_sensing() w następujący sposób:

```
lv,av=sim.getObjectVelocity(robot) -- odczytanie predkosci robota

v_r=math.sqrt(lv[1]*lv[1]+lv[2]*lv[2]) -- wyznaczenie predkosci
wypadkowej ze skladowych

sim.setGraphStreamValue(sim.getObject('/Graph[2]'),v_robot,v_r) --
dodanie punktu do wykresu
```

W efekcie tych zabiegów zobaczymy wykres interesującej nas wielkości:



4.6 Scenariusz ćwiczenia

W ramach ćwiczenia należy zrealizować kolejno poniższe zagadnienia:

1. Załadować scenę sledzenie_linii.ttt i uruchomić zaimplementowany tam algorytm śledzenia włącz–wyłącz. Dodać rysowanie śladu robota oraz środkowego czujnika. Współrzędną Z obu śladów ustalić na bliską 0, tak żeby ślady robota były na poziomie podłoża. Zapisać obraz trajektorii robota.
2. Zaimplementować sterowanie proporcjonalne z przełączeniami. Czy da się tak dobrać wzmocniania i przedziały przełączeń, by robot przejechał całą trasę? Zapisać obraz trajektorii robota.
3. Zaimplementować regulator PI. Dobrać parametry tak, by robot jak najpłynniej pokonywał całą trasę. Zapisać obraz trajektorii robota.
4. Dodać człon różniczkujący. Zoptymalizować parametry. Czy człon różniczkujący poprawił działanie algorytmu? Zapisać obraz trajektorii robota.
5. Porównać kształty trajektorii i czasy przejazdu dla wszystkich algorytmów.
6. Zdjąć ograniczenia na prędkość silników i dobrać regulator PID, który przeprowadzi robota wzdłuż linii w najkrótszym możliwym czasie.

4.7 Pytania sprawdzające

1. Dlaczego regulator proporcjonalny nie sprawdzi się dla linii o silnie zróżnicowanych promieniach łuków?
   Regulator proporcjonalny nie umożliwia wyzerowania uchybu, a jedynie jego zmniejszenie. Dla ostrych łuków uchyb będzie duży, a więc aby go zminimalizować należy użyć dużego wzmocnienia, co wprawdzie pozwoli pokonać ostre zakręty, ale spowoduje silnie oscylacyjny charakter ruchu robota.
2. Dlaczego symulacja tego samego robota w czasie rzeczywistym oraz w zwolnionym tempie może przebiegać inaczej?
   Wynika to z wielkości kroku próbkowania. W zwolnionym tempie krok próbkowania jest mniejszy, a więc dane z czujników i nastawy regulatora są częściej wyznaczane. Inaczej mówiąc, robot szybciej reaguje na zmianę swojego stanu. W przypadku szybkich robotów, duży krok próbkowania może oznaczać pominięcie istotnych informacji (przekroczenie linii bez „zauważenia” tego przez czujnik), co może odbić się negatywnie na zachowaniu robota.
3. Jaki jest efekt zastosowania członu całkującego?
   Człon całkujący zeruje błąd algorytmu sterowania dzięki akumulowaniu poszczególnych błędów, których człon proporcjonalny nie jest w stanie wyzerować. Całkowany (sumowany) błąd wzrasta, jednocześnie zwiększając wartość wysterowania, co przekłada się na zmniejszanie się błędu jednostkowego i ostatecznie jego wyzerowanie.
4. Dlaczego algorytm PID pozwala wypaść robotowi z trasy, mimo że człon całkujący powinien w końcu doprowadzić uchyb do zera?

W przypadku robota śledzącego linię charakter uchybu jest nieco inny niż np. w zadaniu wysterowania silnika. Tam, gdy silnik obraca się zbyt wolno lub za szybko, uchyb jest cały czas ujemny lub dodatni, a zmienia się jedynie jego wartość. Człon całkujący cały czas zwiększa swoją wartość i wysterowanie. W przypadku robota śledzącego linię dzieje się podobnie, ale tylko do momentu, kiedy oba czujniki znajdą się po jednej stronie linii (robot zjedzie z linii) – wówczas błąd będący różnicą wskazań czujników jest bliski zeru (ze względu na niejednorodność podłoża), a więc algorytm przestaje zmieniać swoje nastawy, a robot odjeżdża od linii.

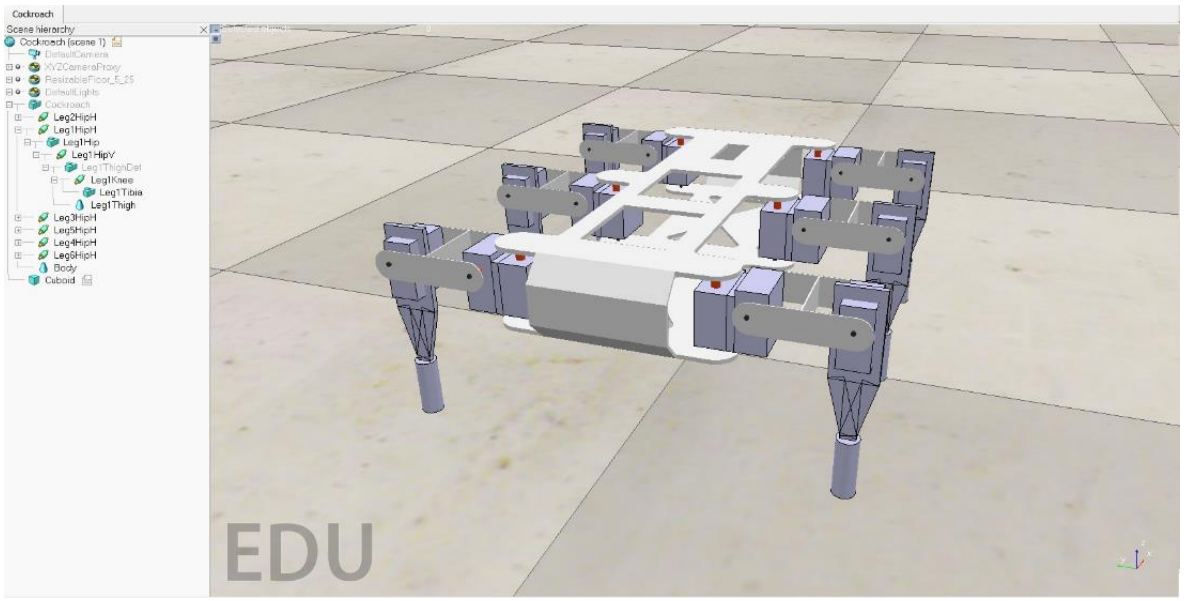
5. Czy człon różniczkujący jest równie ważny co człon całkujący?
   Zdecydowanie nie. Robot śledzący linie będzie działał w wielu przypadkach niemal identycznie z samym regulatorem PI. Człon różniczkujący może jednak pomóc stłumić niektóre oscylacje i zmniejszyć przeregulowania.

5 Podstawowe sterowanie maszyną kroczącą z poziomu języka C++

Tematem tego ćwiczenia jest zapoznanie się z budową oraz podstawowymi sposobami poruszania sześcionożnej maszyny kroczącej o 18 stopniach swobody. Do tego ćwiczenia został sporządzony plik Karaluch1.zip zawierający pliki niezbędne do realizacji zadań polegających na poruszaniu robotem i implementacji chodu dla maszyny kroczącej. Do realizacji tego ćwiczenia niezbędna jest wiedza o podstawowych chodach maszyn kroczących.

5.1 Model robota

W ćwiczeniu wykorzystywany jest plik Cockroach.ttt. Plik ten stanowi symulację sześcionożnego robota mobilnego w pustym środowisku jak przedstawiono na poniższym rysunku



Na tym rysunku widzimy tył Karalucha co powoduje, iż jego prawa strona to również nasza prawa strona oraz lewa to lewa. Nogi Karalucha ponumerowane są od 1 do 6 gdzie noga pierwsza to prawy przód, druga – prawy środek, trzecia – prawy tył, czwarta – lewy przód, piąta – lewy środek i szósta – lewy tył. Elementy ruchome nóg nazywają się kolejno: LegXHipH, LegXHipV oraz LegXKnee, gdzie X to numer nogi. Trzy przeguby czyli trzy stopnie swobody w każdej nodze pozwalają na osiągnięcie przez końcówkę nogi dowolnego punktu w zakresie jej ruchu.

5.2 Programowanie z poziomu języka wysokiego poziomu C++

Programowanie na poziomie języka wysokiego poziomu będzie w ćwiczeniach z Karaluchem realizowane za pomocą języka C++ jako projekt w środowisku MS Visual Studio. Aplikacja MS Visual Studio to projekt typu Console Application. Należy tu zwrócić uwagę na pewne ustawienia projektu (menu: Project− > ...Properties):

Advanced

Character Set = Use Multi-Bate Character Set

Whole Program Optimization = Use Link Time Code Generation

C/C++ -> General

Do **Additional Include Directory** należy dodać:

C:\Program Files\CoppeliaRobotics\CoppeliaSimEdu\programming\legacyRemoteApi\remoteApi

C:\Program Files\CoppeliaRobotics\CoppeliaSimEdu\programming\include

C/C++ -> Preprocessor

Do **Preprocessor Definitions** należy dodać:

NON_MATLAB_PARSING

MAX_EXT_API_CONNECTIONS=255

_CRT_SECURE_NO_WARNINGS

DO_NOT_USE_SHARED_MEMORY

Należy ustawić **Undefine preprocessor definitions**

C/C++ -> Precompiled Headers

Dla **Precompiled Header** ustawić:

Not Using Precompiled Headers

Do parametru **Additional Include Directory** zostały dodane katalogi w których znajdują się pliki biblioteczne niezbędne do utworzenia aplikacji zarządzającej robotami zasymulowanymi w środowisku CoppeliaSim. W związku z tym należy dołączyć pliki biblioteczne znajdujące się w dołączonych katalogach lub (by uniknąć ewentualnych problemów ze zmianą lokalizacji aplikacji) skopiować je do folderu aplikacji a następnie dołączyć do projektuj jako **Source Files** lub odpowiednio **Header Files**:

- **extApi.h,**
- **extApi.c,**
- **extApiPlatform.h,**
- **extApiPlatform.c.**

Pliki znajdują się w katalogu:

CoppeliaInstallFolder\programming\legacyRemoteApi\remoteApi.

5.3 Biblioteka i klasa opisująca model robota w języku C++

Programowanie robota odbywa się w programie C++ z wykorzystaniem MSVisualStudio. W pliku Karaluch1.zip znajduje się spakowany projekt Karalucha w środowisku CoppeliaSim oraz projekt aplikacji w VisualStudio, który stanowi prototyp programu pozwalającego na zarządzanie robotem. W projekcie tym znajduje się plik w którym zaimplementowane zostały klasy pozwalające na tworzenie instancji reprezentującej Karalucha oraz jego nogi.

Klasa Karaluch:

Pola:

dostęp prywatny:

x:int, y:int, z:int {Współrzędne robota w środowisku}

klientID:int {Identyfikator klienta symulacji}

KaraluchID:int = 0 {Identyfikator ciała robota}

dostęp publiczny:

PP:Noga* = new Noga(150, -90, 5) {Zmienna i kostrukcja nogi prawy przód}

PS:Noga* = new Noga(0, -90, 5) {Zmienna i kostrukcja nogi prawy środek}

PT:Noga* = new Noga(-150, -90, 5) {Zmienna i kostrukcja nogi prawy tył}

LP:Noga* = new Noga(150, 90, 5) {Zmienna i k ostrukcja nogi lewy przód}

LS:Noga* = new Noga(0, 90, 5) {Zmienna i kostrukcja nogi lewy środek}

LT:Noga* = new Noga(-150, 90, 5) {Zmienna i kostrukcja nogi lewy tył}

Metody:

dostęp publiczny:

Karaluch():void {konstruktor klasy Karaluch}

Karaluch(iPort:int):void {konstruktor klasy Karaluch z numerem portu symulacji}

Karaluch(cAdresIP:simxChar* {IP maszyny z symulacją}, iPort:int {numer portu}):void {konstruktor klasy Karaluch z definicją IP i numeru portu symulacji}

Uruchomione():bool {Sprawdzenie poprawności uruchomienia robota}

Start():void {Uruchomienie połączenia z symulacją na lokalnej maszynie IP lokalne "127.0.0.1" i port nr 19000}

Start(cAdresIP:int {IP komputera z symulacją}, iPortid:int {numer portu}):void //Uruchomienie połączenia z symulacją robota z definicją numeru portu}

GetKlientID():int {pobranie identyfikatora klienta, zwraca numer ID klienta}

Klasa Noga:

Pola:

dostęp prywatny:

Baza:struct{float x, y, z;} {Położenie nogi we współrzędnych robota [mm]}

biodroH:float = 0, biodroV:float = 0, kolano:float = 0 {Kąty przegubów nogi [rad]}

dostęp publiczny:

Staw:enum{BIODROH=0, BIODROV=1, KOLANO=2} {Stałe definiujące przeguby}

klientID:static int {identyfikator klienta symulacji robota (jedna wartość da wszystkich nóg)}

BiodroHID:int = 0 {Identyfikator pierwszego przegubu -` biodra horyzontalnie}

BiodroVID:int = 0 {Identyfikator drugiego przegubu - biodra wertykalnie}

KolanoID:int = 0 {Identyfikator trzeciego przegubu – kolana}

Metody:

dostęp publiczny:

Noga():void {Konstruktor klasy Noga}

Noga(fX:float, fY:float, fZ:float) {Konstruktor klasy Noga z możliwością zdefiniowania położenia Nogi we współrzędnych robota}

BiodroH():float {Zwraca wartość kąta przegubu pierwszego BIODROH [rad]}

GetBiodroH(bCzekaj:bool = false {Czekaj na wykonanie}):float {Odczyt wartości kąta przegubu pierwszego roota BIODROH z możliwości oczekiwania na wykonanie, zwraca kąt przegubu pierwszego nogi BIODROH [rad]}

SetBiodroH(fKat:float {Kąt przegubu pierwszego nogi}, bCzekaj:bool = false {Czekaj na wykonanie}):void {Zadanie wartości kąta przegubu pierwszego BIODROH z możliwości oczekiwania na wykonanie}

BiodroV():float {Przekazuje wartość kąta przegubu drugiego BIODROV, zwraca – kąt przegubu drugiego BIODROV [rad]}

Get BiodroV(bCzekaj:bool = false {Czekaj na wykonanie}):float {Odczyt wartości kąta przegubu drugiego w nodze – BIODROV z możliwości oczekiwania na wykonanie

SetBiodroV(fKat:float {kąt przegubu 2}, bCzekaj:bool = false):void {Zadanie wartości kąta przegubu drugiego BIODROV z możliwości oczekiwania na wykonanie}

Kolano():float {Przekazuje wartość kąta przegubu trzeciego KOLANO, zwraca [float] - kąt przegubu trzeciego KOLANO [rad]}

GetKolano(bool bCzekaj = false):float {Odczyt wartości kąta przegubu trzeciego nogi KOLANO z możliwości oczekiwania na wykonanie, zwraca [float] - Kąt przegubu trzeciego robota KOLANO [rad]

SetKolano(fKat:float {kąt przegubu trzeciego nogi [rad]}, bCzekaj:bool = false):float {Zadanie wartości kąta przegubu trzeciego KOLANO z możliwości oczekiwania na wykonanie}

GetMoment(stStaw:Staw, bCzekaj:bool = false):float {Odczyt momentu przegubu nogi robota z możliwością oczekiwania na wykoananie, zwraca wartość momentu [Nm]}

SetMoment(stStaw:Staw, fMoment:float, bCzekaj:bool = false):void {Ustawienie momentu przegubu nogi robota z możliwością oczekiwania na wykonanie}

GetMomenty(fMomenty:float[3], bCzekaj:bool = false):void {Odczyt momentów przegubów nogi robota z możliwością oczekiwania na wykonanie}

SetMomenty(fMoment:float, bCzekaj:bool = false):void {Ustawienie momentu przegubów nogi robota z możliwością oczekiwania na wykonanie}

GetMomentMAX(stStaw:Staw, bCzekaj:bool = false):float {Odczyt maksymalnego momentu przegubu nogi robota z możliwością oczekiwania na wykonanie}

SetMomentMAX(stStaw:Staw, fMomentMax:float, bCzekaj:bool = false):void {Ustawienie maksymalnego momentu przegubów nogi robota z możliwością oczekiwania na wykonanie}

GetMomentyMAX(fMomentyMax:float[3], bCzekaj:bool = false):void {Odczyt maksymalnych momentów przegubów nogi robota z możliwością oczekiwania na wykonanie}

SetMomentyMAX(fMomentMax:float, bCzekaj:bool = false):void {Ustawienie maksymalnego momentu przegubów nogi robota z możliwością oczekiwania na wykonanie}

SetPredkosc(stStaw:Staw, fPredkosc:float, bCzekaj:bool = false):float {Ustawienie prędkości przegubu nogi robota z możliwością oczekiwania na wykonanie}

SetPredkosci(fPredkosc:float, bCzekaj:bool = false):void {Ustawienie prędkości przegubów nogi robota z możliwością oczekiwania na wykonanie}

SetKatyNogi(fBiodroH:float, fBiodroV:float, fKolano:float):void {Zadanie wartości kątów przegubów robota}

GetKatyNogi(fBKS:float[3]):void {Odczyt wartości kątów przegubów robota}

SetPozycjaNogiAtRobot(fX:float, fY:float, fZ:float):void {Ustawienie pozycji końcówki nogi robota we współrzędnych robota}

GetPozycjaNogiAtNoga(fXYZ:float[3], bCzekaj:bool = false):void {Odczyt pozycji nogi robota we współrzędnych nogi z możliwością oczekiwania na wykonanie}

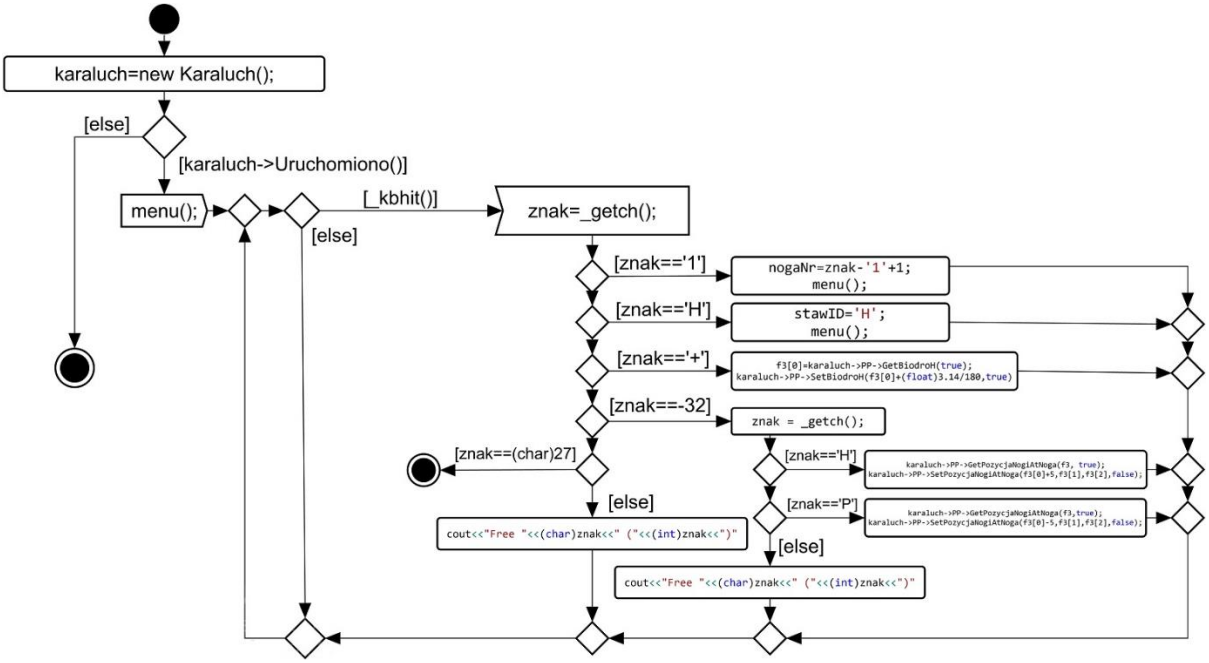
SetPozycjaNogiAtNoga(fX:float, fY:float, fZ:float, bCzekaj:bool = false):int {Ustawienie pozycji końcówki nogi robota we współrzędnych nogi z możliwością oczekiwania na wykonanie}

AddPozycjaNogiAtNoga(fdX:float, fdY:float, fdZ:float, bCzekaj:bool = false):int {Zmiana pozycji końcówki nogi robota we współrzędnych nogi robota z możliwością oczekiwania na wykonanie}

Ważnym jest również to, aby uruchamiać aplikację w trybie Release. W domyślnym ustawieniu **Debug** aplikacja nie zostanie poprawnie uruchomiona.

5.4 Szablon programu

Projekt aplikacji znajdującej się w pliku Karaluch.zip to szablon programu służącego do zrealizowania zadań przypisanych do ćwiczenia. Na poniższym rysunku znajduje się diagram prezentujący budowę programu zaimplementowanego w aplikacji Karaluch.



Program na początku inicjuje zmienne robocze i wyświetla menu. Po zainicjowaniu wszystkich elementów przechodzimy do głównej pętli programu, która jest pętlą działającą w nieskończoność. Podstawową funkcjonalnością realizowaną w tej pętli jest odczyt klawiatury i realizacja czynności zdefiniowanych w menu po ich wybraniu przez użytkownika.

Nie można zapominać o sprawdzeniu dodatkowych parametrów i ustawień projektu aplikacji. Niewłaściwa definicja ustawień projektu aplikacji spowoduje wystąpienie błędów krytycznych

i nie pozwoli na skompilowanie programu. Należy również pamiętać o tym, iż aplikacja musi być kompilowana i uruchamiana w trybie Release.

5.5 Scenariusz ćwiczenia

Do zadań, które kolejno powinny zostać wykonane przez studenta należą:

1. Uruchomienie symulacji robota.
2. Otwarcie i kompilacja aplikacji szkieletowej dla Karalucha.
3. Implementacja obsługi wszystkich pozycji menu, czyli:
    - wybór nogi,
    - wybór stawu,
    - poruszanie wybraną nogą i stawem.
4. Implementacja chodu trójpodporowego.
5. Implementacja chodu zdefiniowanego/opracowanego przez studenta/

Ocenie podlega realizacja zdefiniowanych zadań. Do wykonania są trzy główne zadania. Po zrealizowaniu każdego z poleceń student przedstawia prowadzącemu swoje wyniki.

5.6 Pytania sprawdzające

1. Czym charakteryzuje się chód trójpodporowy?

   Chód trójpodporowy charakteryzuje się tym, iż zawsze robot stoi na co najmniej trzech nogach. Jest to chód stabilny. Nogi maszyny tworzą dwie grupy, które poruszają się podobnie jak nogi maszyny dwunożnej, gdzie każda z grup zachowuje się analogicznie do jednej nogi.

2. Co to jest kinematyka prosta?`

   Kinematyka prosta dotyczy obliczania pozycji końcówki roboczej. Polega na obliczeniu pozycji na podstawie danych o położeniu przegubów w łańcuchu kinematycznym danego manipulatora.

3. Co to jest kinematyka odwrotna?

   Kinematyka odwrotna to przekształcenie odwrotne dla kinematyki prostej i polega na wyznaczeniu parametrów przegubów łańcucha kinematycznego danego manipulatora tak by końcówka robocza osiągnęła zadane położenie.

4. O czym decyduje liczba nóg maszyny kroczącej?

   Liczba nóg maszyny kroczącej decyduje o liczbie możliwych do zdefiniowania rodzajów chodu. Ilość możliwych chodów określa wzór:

   k=(2*n-1)!

   gdzie n – liczba nóg. k - liczba możliwych do zdefiniowania rodzajów chodu.

5. Kiedy maszyna krocząca jest stabilna?

Maszyna krocząca jest stabilna statycznie jeśli jej środek ciężkości znajduje się wewnątrz figury wyznaczonej przez jej punkty podparcia.

6 Wykorzystanie czujników odległości do omijania przeszkód przez maszynę kroczącą

Celem tego ćwiczenia jest zaproponowanie algorytmu omijania przeszkód wykrytych przez czujniki odległości dodane do robota kroczącego. Użycie algorytmu Braitenberga, omówionego w ramach ćwiczenia nr 2 będzie raczej trudne (ze względu na złożoność robota i aż 18 stopni swobody), niemniej meta przełączanych zachowań powinna się tu dobrze sprawdzić.

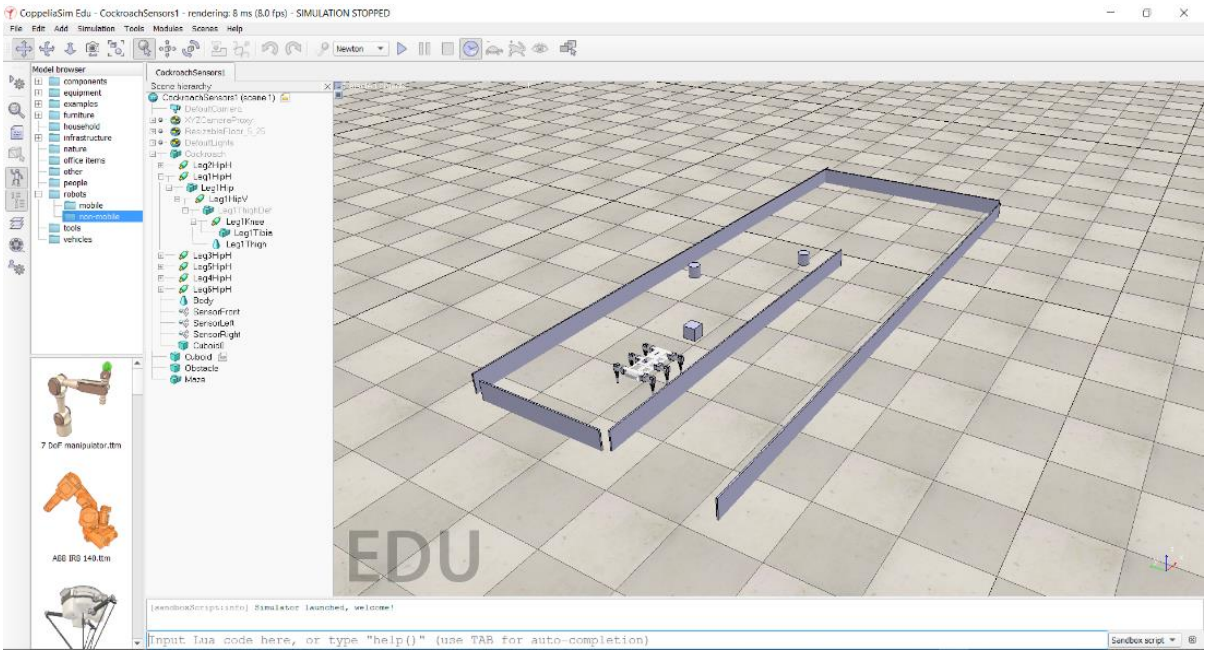
6.1 Model robota i środowiska

Do realizacji tego ćwiczenia zostały przygotowane pliki **CockroachSensors1.ttt**, **CockroachSensors2.ttt**, **CockroachSensors3.ttt** i **CockroachSensors4.ttt**. Różnica pomiędzy symulacją do pierwszego ćwiczenia z Karalucha polega na tym, iż sam robot dodatkowo posiada czujniki odległości:

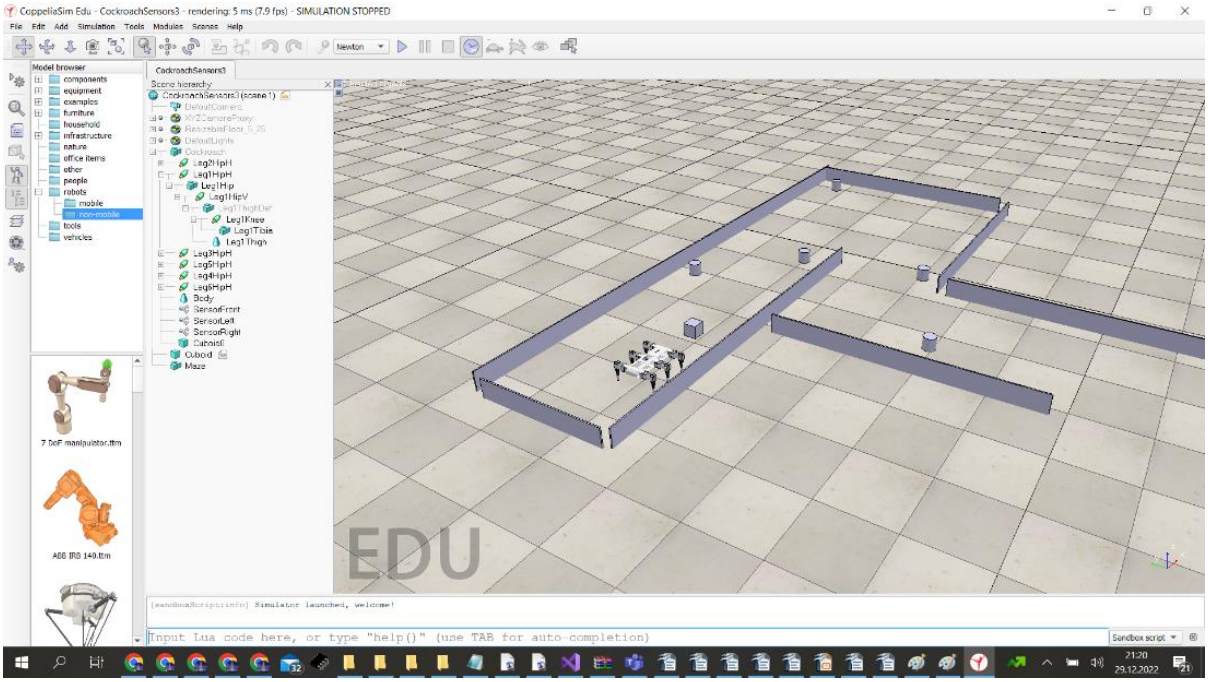
- SensorFront,
- SensorMiddle,
- SensorRight.

We wszystkich tych plikach zaimplementowany robot jest taki sam, a różnice dotyczą jedynie elementów środowiska. Tory, którymi ma poruszać się maszyna są zróżnicowane i zostały wyposażone w różne dodatkowe obiekty stanowiące przeszkody dla maszyny.

Symulacja z pliku CockroachSensors1.ttt:



Symulacja z pliku CockroachSensors3.ttt:



6.2 Biblioteka i klasa opisująca model robota w języku C++

Aplikacja języka C++ dla omawianego ćwiczenia charakteryzuje się dokładnie takimi samymi parametrami, jak w pierwszym ćwiczeniu z Karaluchem, czyli Console Application w MS VisualStudio. Klasy reprezentujące Karalucha i wykorzystywane w aplikacji zostały zaimplementowane również w pliku **KaraluchSim.h**. Jednakże plik ten został zmodyfikowany w stosunku do zadania pierwszego. Modyfikacja pliku KaraluchSim.h polega na zwiększeniu funkcjonalności dostępnych w klasie Karaluch. Klasa Noga nie została zmodyfikowana.

Klasa Karaluch została dodatkowo wyposażona w:

Pola:

dostęp prywatny:

…

USSensorID:int = 0 {Identyfikator centralnego czujnika odległości}

USSensorLewyID:int = 0 {Identyfikator lewego czujnika odległości}

USSensorPrawyID:int = 0 {Identyfikator prawego czujnika odległości}

distanceID:int = 0 {Identyfikator pomiaru odległości}

…

Metody:

dostęp publiczny:

...

GetDystans(cJaki:char = ‘c’ {definicja sensora ‘c’-centralny(domyślny, ‘l’’-lewy,

‘p’-prawy}:float {Odczyt odległości od obiektu widzianego przez sensor, zwraca odległość}

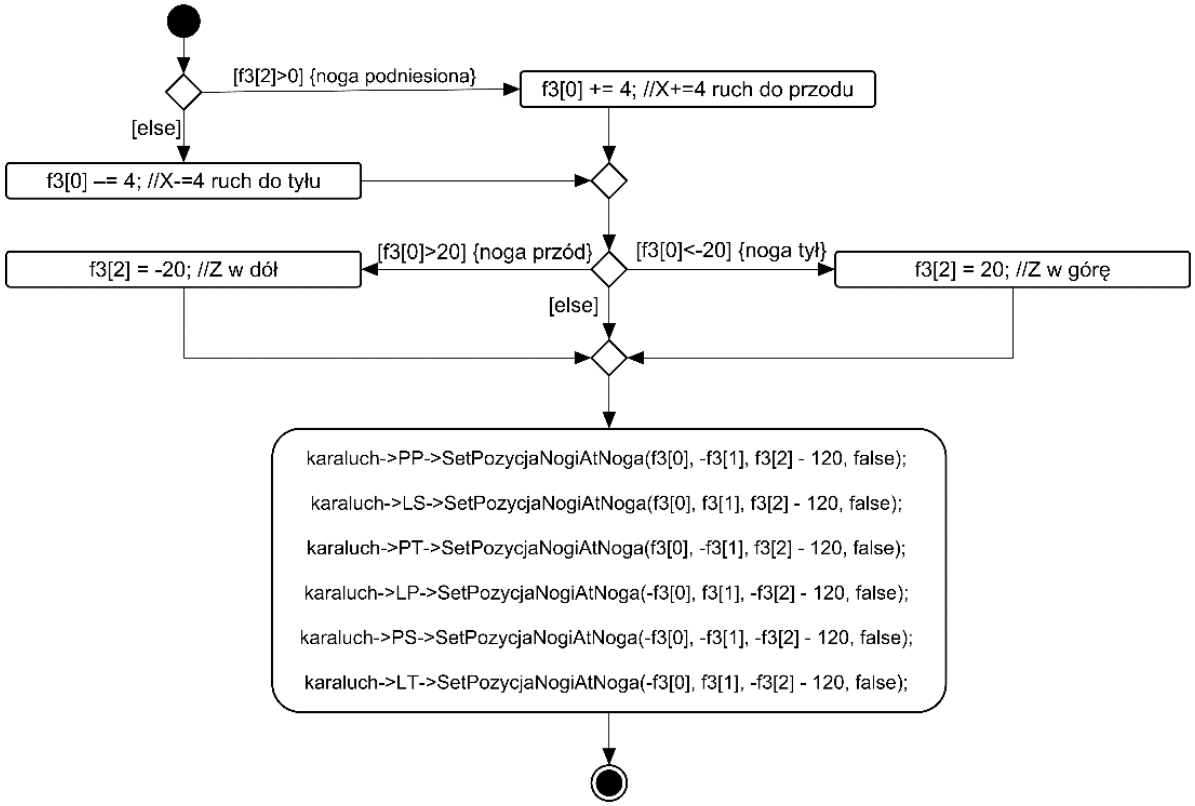
Wprowadzone nowe pola oraz metoda służą do obsługi czujników odległości, w które wyposażony został karaluch używany w zadaniach drugim i trzecim z karalucha.

6.3 Szablon programu

Szablon programu, podobnie jak plik biblioteczny KaraluchSim.h, nie uległ wielkim modyfikacjom. Najważniejsza zmiana dotyczy menu i zaimplementowanych komend. Pierwszą podstawową zmianą jest implementacja obsługi strzałek na klawiaturze komputera, które pozwalają na sterowanie ruchem robota. Realizacja ruchu robota do przodu została zaimplementowana zgodnie ze sposobem poruszania się chodem trójpodporowym.

Zaimplementowane zostały funkcje pozwalające na realizację ruchy do przodu *doPrzodu*, w prawo *wPrawo*, oraz w lewo *wLewo*.

Poniższy rysunek przedstawia diagram czynności opisujący działanie funkcji doPrzodu.



Zmienna **f3** to trzyelementowa tablica wartości typu *float*:

```
float f3[3] = {0, 80, -20};
```

Odnosi się ona do położenia końcówki nogi robota w układzie nogi. Wartości odnoszą się kolejno do osi X, Y i Z. Założono, iż wartość w osi X może zmieniać się w zakresie od –20 do 20, w osi Y to konkretna liczba 80, a w osi Z to 20 lub –20. Powyższe wytyczne zostały wykorzystane do zdefiniowania warunków dotyczących realizacji zmian położenia końcówki nogi w X, Y i Z.

Poniżej znajduje się listing funkcji doPrzodu

```
void doPrzodu()

{

if (f3[2] > 0)  //Z>0 to noga w górze

f3[0] += 4;     //X+=4

else

f3[0] -= 4;     //X-=4

if (f3[0] > 20)

f3[2] = -20;

else if (f3[0] < -20)

f3[2] = 20;

karaluch->PP->SetPozycjaNogiAtNoga(f3[0], -f3[1], f3[2] - 120,
false);

karaluch->LS->SetPozycjaNogiAtNoga(f3[0], f3[1], f3[2] - 120,
false);

karaluch->PT->SetPozycjaNogiAtNoga(f3[0], -f3[1], f3[2] - 120,
false);

karaluch->LP->SetPozycjaNogiAtNoga(-f3[0], f3[1], -f3[2] - 120,
false);

karaluch->PS->SetPozycjaNogiAtNoga(-f3[0], -f3[1], -f3[2] - 120,
false);

karaluch->LT->SetPozycjaNogiAtNoga(-f3[0], f3[1], -f3[2] - 120,
false);

}
```

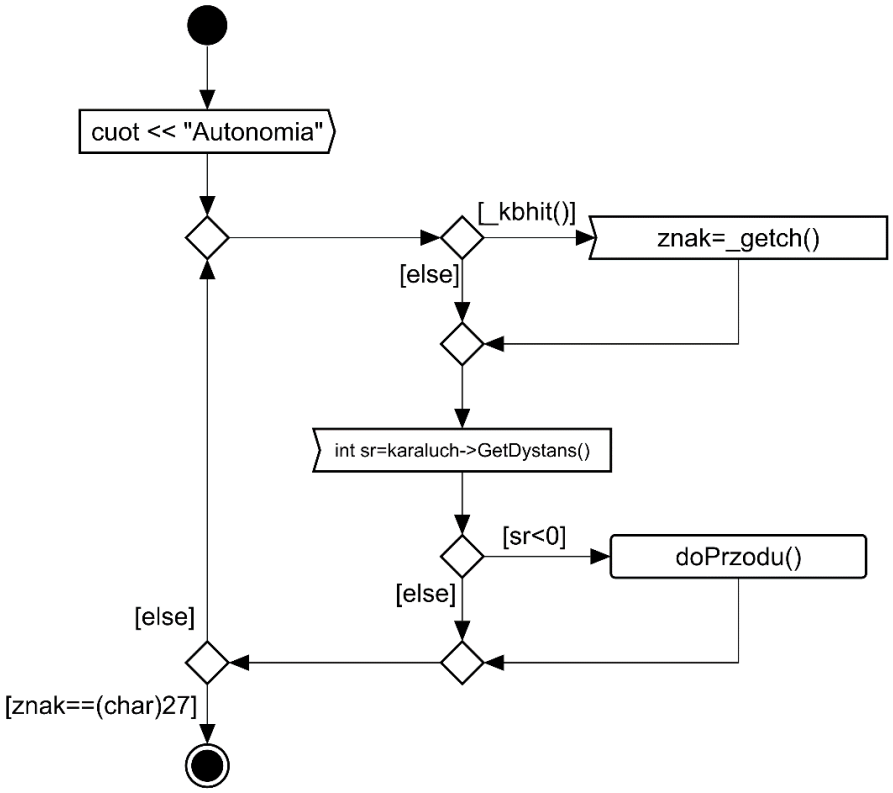
W analogiczny sposób zostały zaimplementowane funkcje *wLewo* oraz *wPrawo*. Funkcja *doTylu* nie została zaimplementowana, a jej realizację pozostawiono osobom wykonującym ćwiczenia.

6.4 Scenariusz ćwiczenia

W ramach realizacji ćwiczenia należy zrealizować następujące zadania:

1. Zaimplementować funkcję *doTylu* i zrealizować obsługę wszystkich strzałek na klawiaturze komputera (prawo, lewo, góra(przód) i dół(tył)).
2. Zaimplementować tryb pracy autonomicznej po wybraniu klawisza ‘A’.
3. Zaimplementować zachowanie robota tak, by wyszedł z obszaru ograniczonego ścianami.

Poniżej znajduje się diagram pomocniczy przedstawiający przykład kodu realizującego zachowanie autonomiczne robota:



Aplikacja w ćwiczeniu powinna tak sterować robotem by omijał on przeszkody i efektywnie podążał do wyjścia obszaru w którym się znajduje. Końcowym efektem realizacji ćwiczenia powinien być program sterujący karaluchem w symulacji CoppeliaSim w taki sposób by był on w stanie wyjść na zewnątrz obszaru ograniczonego ściankami.

6.5 Pytania sprawdzające

1. W jaki sposób realizuje się wykonanie skrętu sześcionożnym robotem kroczącym?

   Skręt sześcionożnym robotem kroczącym realizowany jest poprzez zróżnicowanie amplitudy ruchu nóg znajdujących się po przeciwnych stronach maszyny. Zmniejszenie amplitudy ruchu nóg robota po prawej stronie spowoduje, że skręci on w prawo a po lewej stronie w lewo.

2. W jaki sposób realizuje się obrót robota wokół własnej osi i czym on się różni od skrętu?

Obrót robota wokół własnej osi można wykonać ruszając nogami z jednej jego strony przeciwnie do tych po drugiej stronie. Jeśli nogi po prawej stronie idą do przodu to te po lewej powinny iść do tyłu i wtedy robot obróci się w lewo czyli przeciwnie do ruchu wskazówek zegara. Obrót wokół własnej osi różni się tym od skrętu, iż w skręcie mamy zróżnicowanie amplitudy nóg po obu stronach maszyny a w obrocie nogi poruszają się przeciwnie.

3. Jak wykrywamy i omijamy przeszkody w robocie w ćwiczeniu?

W robocie wykorzystanym w ćwiczeniu przeszkody wykrywane są za pomogą czujników odległości. Sposób omijania przeszkód może być uzależniony od algorytmu, który jest wykorzystywany do sterowania robotem. Standardowo w momencie, gdy obiekt znajduje się bliżej niż określona odległość to należy go ominąć. W momencie wykrycia obiektu czujnikiem prawy omijamy go ze strony lewej a gdy będzie obiekt wykryty czujnikiem lewym to po stronie prawej, W momencie wykrycia przeszkody za pomocą czujnika środkowego obiekt można ominąć zarówno po stronie prawej jak i lewej.

4. Jak znaleźć wyjście z obszaru otoczonego ścianami?

Wyjście można znaleźć poprzez zastosowanie odpowiedniego algorytmu. Może to być algorytm Wall follower, gdzie robot porusza się tak by przez cały czas widzieć ścianę po swojej prawej lub lewej stronie.

5. Co to jest układ sensoryczny maszyny?

Układ sensoryczny maszyny to zestaw jej czujników. Układy sensoryczne stosuje się w maszynach po to by mieć pełniejszą informacje o otaczającym środowisku. Przykładowo robot w ćwiczeniu został wyposażony w trzy czujniki odległości co powoduje, iż jest on w stanie nie tylko stwierdzić, że pojawiła się jakaś przeszkoda ale również może określić po której stronie ona się znajduje i odpowiednio zareagować.

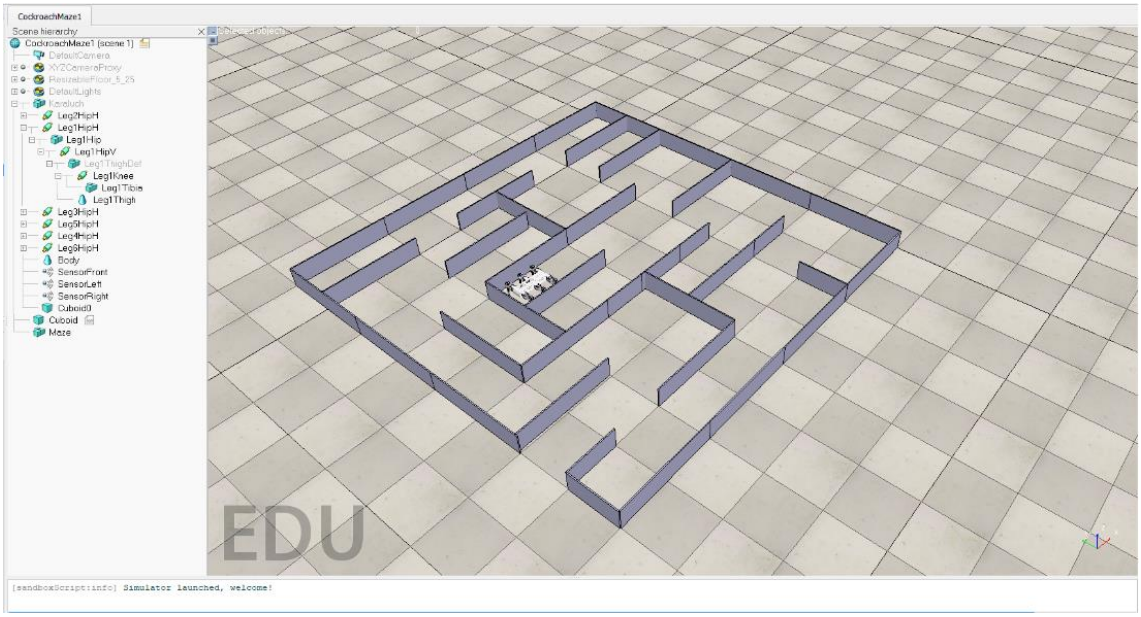
7 Planowanie drogi w labiryncie

W tym ćwiczeniu należy wyznaczyć najkrótszą drogę na znanej mapie (w labiryncie) i wyprowadzić robota z labiryntu. Metody poruszania robotem kroczącym zostały opracowane w ramach poprzednich ćwiczeń i należy je wykorzystać,

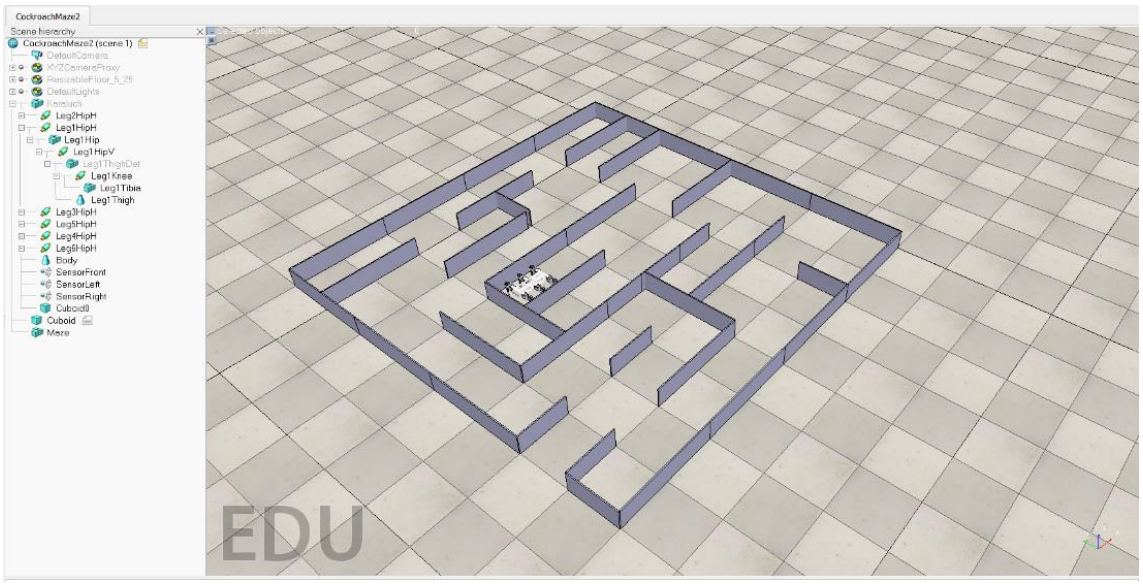
7.1 Model robota i środowiska

Do realizacji tego ćwiczenia zostały przygotowane pliki **CockroachMaze1.ttt** i **CockroachMaze2.ttt**. Budowa robota w tej symulacji jest dokładnie taka sama jak w ćwiczeniu poprzednim. Zupełnie inna jest natomiast scena, która nie ma obiektów stanowiących przeszkody dla Karalucha, a jednie odpowiednie ścianki, z których składa się labirynt.

Labirynt w wersji 1:



Labirynt w wersji 2:



7.2 Biblioteka i klasa opisująca model robota w języku C++

Zarówno biblioteka, jak i klasa w tym ćwiczeniu są dokładnie takie same, jak w poprzednim.

7.3 Szablon programu

Ze względu na brak zmian w bibliotece i samym robocie można w tym ćwiczeniu wykorzystać program, który został utworzony w ćwiczeniu poprzednim. Oczywiście zachowanie robota w ramach autonomii, która została zaimplementowana poprzednio, nie będzie tu efektywne choć, zgodnie z prawami Murphy'ego, może się zdarzyć, że zaimplementowany algorytm powoduje znalezienie optymalnej drogi prowadzącej do wyjścia.

7.4 Planowanie ścieżki

Problem znajdowania najkrótszej drogi z wykorzystaniem mapy można rozwiązać na wiele sposobów, których omówienie daleko wykracza poza ramy tego ćwiczenia. Dla zagadnień o niskim stopniu złożoności (np. takich jak niewielki labirynt) dobrze sprawdzają się metody siatkowe, to znaczy takie, w których przestrzeń konfiguracyjna robota została podzielona na (najczęściej kwadratowe) pola przylegające do siebie. Część z pól jest ze sobą połączona (robot może się między nimi swobodnie przemieszczać), a część nie (pola oddzielone przeszkodami). Wyznaczenie najkrótszej ścieżki na takiej mapie nie jest zadaniem trudnym. Jedną z prostszych metod jest metoda propagacji fali, ale można też potraktować utworzoną siatkę jako graf, którego węzły oznaczają kolejne pola, a krawędzie – połączenia pomiędzy nimi. Jeśli znamy strukturę labiryntu, bez trudu dokonamy jego stosownej reprezentacji grafowej, a następnie wykorzystamy jeden z licznych algorytmów znajdowanie najkrótszej drogi w grafie, takich jak algorytm Djikstry, A*, BFS czy DFS. Implementacje tych algorytmów bez trudu można znaleźć w sieci.

7.5 Scenariusz ćwiczenia

W ramach poruszania się po labiryncie należy zrealizować poniższe zadania, które pozwolą na swobodne poruszanie się maszyny po labiryncie:

1. Podzielić labirynt na sektory zgodnie z kwadratami na planszy symulacyjnej i zgodnie z tym podziałem zaimplementować funkcję realizującą przejście do sąsiedniego kwadratu, oczywiście tylko jeśli jest to możliwe.
2. Zgodnie z tym podziałem należy odpowiednie zdefiniować i oznaczyć/ponumerować poszczególne pola labiryntu, a następnie wybranym przez siebie algorytmem znaleźć wyjście z labiryntu, drogę do tego wyjścia i wyjść robotem z labiryntu.

Finalnym efektem realizacji ćwiczenia powinno być takie zachowanie autonomiczne robota by sam wyszedł z labiryntu.

7.6 Pytania sprawdzające

1. Jak skręcać robotem sześcionożnym w miejscu?

Skręcanie robotem w miejscu realizowane jest poprzez realizację ruchu nóg w przeciwnych kierunkach. Jeśli nogi po stronie prawej idą do przodu a po lewej do tyłu to robot obróci się w lewo. Jeśli nogi po stronie prawej idą do tyłu a po stronie lewej do przodu to robot obróci się w prawo.

2. W jaki sposób można znaleźć wyjście z labiryntu?

Do znajdowania wyjścia z labiryntu można wykorzystać odpowiedni algorytm. Jeśli znamy mapę labiryntu to możemy skorzystać z metody propagacji fali, która wyznacza optymalną drogę do wyjścia poprzez wyznaczenie odpowiednich wartości pól labiryntu. Jeśli nie znamy mapy labiryntu musimy zastosować algorytm, który bazuje na przeszukiwaniu labiryntu i znajdowaniu odpowiedniej drogi jak np.. Brute force - metoda w której przeszukujemy labirynt pamiętając, gdzie wcześniej byliśmy i wykluczamy realizację ruchu, który nie prowadzi do wyjścia a do miejsca wcześniej już przez nas odwiedzonego.

3. Co wymaga metoda Wall follower od robota?

Metoda ta wymaga by robot miał możliwość kontrolowania obecności ściany po swojej prawej lub lewej stronie. Tym samym musi on być wyposażony w odpowiednie czujniki odległości na podstawie, których będzie można stwierdzić, iż ściana za bardzo się oddala lub zbliża.

4. Kiedy podążając wzdłuż jednej ściany nie dotrzemy do wyjścia i co zrobić by tego uniknąć?

Podążając wzdłuż jednej ściany nie dotrzemy do wyjścia jeżeli ściana ta nie jest połączona ze ścianami zewnętrznymi labiryntu. W celu uniknięcia takiej sytuacji należy w momencie, gdy podążając wzdłuż ściany docieramy do miejsca w którym już byliśmy zmieniamy ścianę.

5. Na czym polega metoda propagacji fali?

Metoda propagacji fali wymaga znajomości mapy labiryntu. W labiryncie wyznaczamy pola, którym można będzie nadawać wagi. Wartości wag definiujemy od wyjścia, gdzie pole wyjściowe ma wagę 1. Przechodząc do każdego kolejnego pola, w którym nie została zdefiniowana waga, przypisujemy mu wagę o jeden większą. Jeśli cały labirynt ma pola o zdefiniowanych wagach to w łatwy sposób możemy z każdego miejsca labiryntu dotrzeć do wyjścia najkrótszą drogą. Realizuje się to poprzez wybieranie robotem kolejnego, sąsiadującego z obecnym pola o najmniejszej wadze.