Laboratorium Inteligentnych Maszyn i Systemów

Politechnika Warszawska

Wydział Elektryczny

Zaawansowane sterowanie robotem Lego NXT

Jakub Matysiak (e-mail: 482372@poczta.fm)

Semestr V, informatyka/inżynieria oprogramowania

2009

Spis treści

1 Wprowadzenie

1.1 Dlaczego wybrałem ten projekt

W czasie mojego dzieciństwa bardzo lubiłem bawić się klockami LEGO. W szczególności interesowała mnie seria LEGO Technic za pomocą której konstruowałem przeróżne pojazdy mechaniczne. Miałem nawet możliwość tworzyć proste programy dzięki „klockowi" programowanemu na pomocą kodów kreskowych.

Podczas wyboru projektu indywidualnego moją uwagę od razu zwrócił temat o programowaniu robota z LEGO. Poszukałem informacji o zestawie LEGO NXT, i przekonawszy się, że jego możliwości są duże, zdecydowałem się na jego realizację.

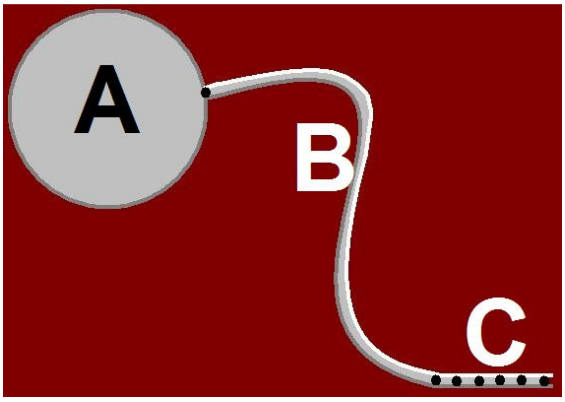
1.2 Tematyka projektu

Przedmiotem projektu jest sterowanie robotem LEGO NXT, nie miałem jednak narzuconych z góry, wymagań co do zakresu projektu czy języka programowania. Do tematyki projektu można zaliczyć także zapoznanie się ze środowiskami pozwalającymi na programowanie robota.

1.3 Cele

1.3.1 Pierwotne

Pierwotnym założeniem było dążenie ku temu, by robot jechał do obszaru [A] po linii [B], szukał tam obiektu, chwytał go, a następnie przewoził w inne, wolne miejsce obok linii w obszarze [C] po planszy jaka jest pokazana na rysunku 1.



Rysunek 1. Plansza po której robot miał jeździć w pierwotnym założeniu

1.3.2 Ostateczne

Po zweryfikowaniu możliwości sprzętu i swoich, ostatecznym celem projektu stało się zaprogramowanie robota, aby jechał po linii widocznej na rysunku 2 oraz usuwał napotkane przeszkody ze swojej drogi. Podczas dążenia do osiągnięcia tego celu musiałem postawić sobie wiele celów cząstkowych, jak na przykład zbudowanie funkcjonalnego robota czy znalezienie efektywnego sposobu na przesyłanie programów do kostki NXT. Ponadto, do celów tego projektu można zaliczyć zapoznanie się ze sprzętem i jego możliwościami.

Rysunek 2. Plansza z linią

2 Możliwe rozwiązania

2.1 Dostępne języki programowania

2.1.1 NXT-G

NXT-G to graficzne środowisko programowania dostarczane standardowo z zestawem LEGO Mindstorms. Napisanie zaawansowanego programu w NXT-G jest niemożliwe z uwagi na niewygodną nawigację po graficznym kodzie oraz na ograniczenia samego języka. Jednak największą wadą środowiska jest mała szybkość programów w nim tworzonych.

2.1.2 ROBOTC

Dzięki środowisku ROBOTC można zaprogramować kostkę NXT używając języka C. Mimo, że korzysta ze standardowego firmware kostki NXT, daje ono większe możliwości niż standardowe. Jest także znacznie szybsze od NXT-G – ponad 100 razy. ROBOTC nie jest jednak darmowe do użytku.

2.1.3 NXC

NXC czyli Not Exactly C to język programowania podobny do C dla NXT. NXC nie wymaga instalowania niestandardowego firmware do kostki, dzięki czemu możliwe jest jednoczesne posiadanie na kostce programów w NXC jak i NXT-G. Wadą są ograniczenia języka takie same jak w przypadku NXT-G. Programy napisane w NXC są kilkukrotnie szybciej wykonywane niż te w NXT-G.

2.1.4 LeJOS NXJ

LeJOS NXJ to pakiet narzędzi za pomocą których można napisane w Javie programy skompilować przesłać do kostki NXT. Aby korzystać z tego środowiska konieczne jest zainstalowanie firmware – wirtualną maszynę Java do kostki NXT. Deweloperzy LeJOS NXJ zapewnili wszystkie narzędzia potrzebne do efektywnego korzystanie z tego środowiska.

2.2 Metody sterowania robotem

2.2.1 Prosta sekwencja instrukcji

Prawdopodobnie można by było napisać algorytm sterowania robotem w konwencji programowania strukturalnego, jako serii warunków „if then” w pętli.

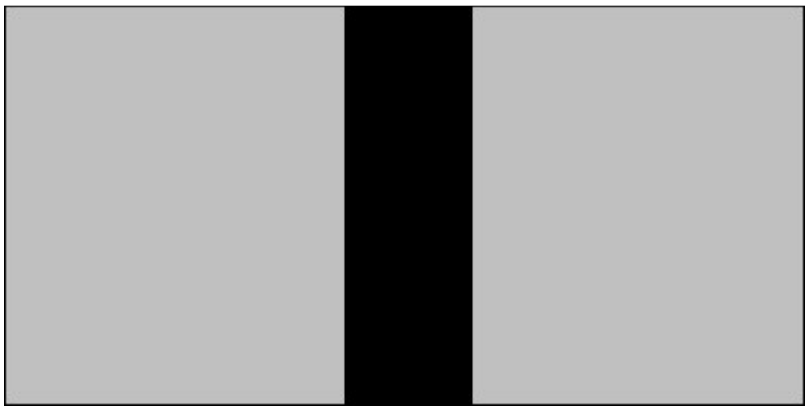
2.2.2 Model kontroli zachowawczej

Model kontroli zachowawczej polega na zdefiniowaniu serii zachowań oraz odpowiadającym im priorytetów i warunków jakie muszą zajść, aby dane zachowanie zostało wykonane. Zachowaniami zarządza klasa Arbitrator która uruchamia poszczególne zachowania w zależności od zaszłych warunków i priorytetów zachowań.

2.3 Metody śledzenia linii

2.3.1 Linia jednokolorowa

Robot śledzi linię jednokolorową, która znajduje się na tle innego koloru, jak na rysunku 3. Robot tak naprawdę śledzi krawędź linii czarnej, przez co musi cały czas jej szukać. W rezultacie, nie jest zdolny do szybkiej jazdy na wprost, nawet jeśli linia jest prosta, chyba, że założy się, że zakręty mogą być tylko w jedną stronę (jak np. jazda po elipsie)



Rysunek 3. Linia jednokolorowa

2.3.2 Linia trzykolorowa

Robot śledzi linię, która znajduje się wewnątrz linii zewnętrznych (czarnej i białej). Dopóki sensor światła znajduje się nad środkową linią - robot jedzie prosto. Tor jazdy koryguje dopiero jak wjedzie na jedną z linii zewnętrznych. Przykład takiej linii znajduje się na rysunku 4. Kolor tła nie ma znaczenia.

Rysunek 4. Linia trzykolorowa

2.4 Wybór rozwiązania

Do oprogramowania mojego robota wybrałem LeJOS NXJ. Głównymi argumentami przemawiającymi na jego korzyść była znajomość języka Java oraz duża, w porównaniu z innymi sensownymi rozwiązaniami, ilość materiałów o nim w Internecie. Nie bez znaczenia jest także fakt, iż istnieje plugin do Eclipse pozwalający używać tej platformy do rozwoju oprogramowania. Dzięki niemu proces powstawania programu i przesyłania go do klocka może być znacznie przyspieszony.

Jako sposób sterowania robotem wybrałem model kontroli zachowawczej. Jest on o wiele ciekawszy i bardziej zaawansowany, a tematyka projektu wymaga zaawansowanych rozwiązań. Oprócz tego program napisany z użyciem tego modelu jest o wiele bardziej czytelny i podatny na przyszłe modyfikacje.

Jako drogę do naśladowania dla robota wybrałem linię 3 kolorową. Kierowałem się przy tym głównie większą możliwą szybkością robota. Oczywiście algorytm do śledzenia takiej linii jest bardziej skomplikowany, ale z uwagi na tematykę projektu nie dążyłem do implementowania rozwiązań jak najprostszych.

3 Założenia projektu

Jako środowisko programistyczne posłużył mi Eclipse wraz z wtyczką leJOS NXJ Eclipse Plugin 0.7. Sprzęt z którego skonstruowałem robota to zestaw Lego Mindstorms NXT, numer zestawu 8457. Na kostkę NXT wgrałem firmware LeJOS NXJ w wersji 0.7.

Robot sterowany jest przez program zapisany w jego wewnętrznej pamięci i wykonywany na nim samym. Podczas działania, robot nie ma żadnego połączenia z komputerem. Sterowanie poszczególnymi funkcjami robota oparte jest na programowaniu zachowawczym (behavior programming).

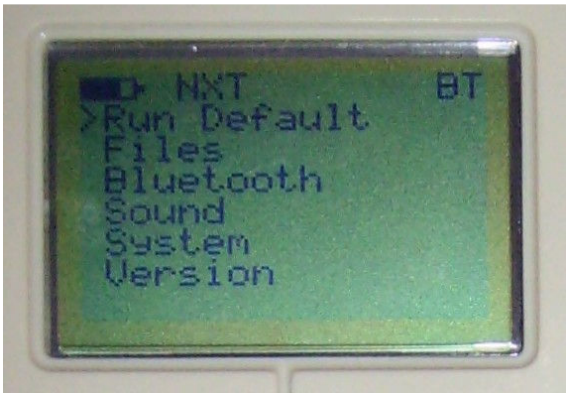
4 Realizacja projektu

4.1 Szczegółowy opis realizacji

4.1.1 Przygotowania

Pierwszym etapem w projekcie było zapoznanie się ze sprzętem na którym miałem pracować. Zbudowałem jednego ze standardowych robotów z instrukcji LEGO – tribota, oraz oprogramowałem go za pomocą środowiska programistycznego LEGO NXT-G. Następnie przystąpiłem do przystosowywania zestawu do pracy w środowisku LeJOS NXJ.

Zapoznałem się z dokumentacją tego środowiska oraz przeczytałem instrukcję instalacji. Gdy udało mi się już wgrać nowy firmware (rysunek 5) oraz połączyć robota z komputerem przystąpiłem do pisania pierwszych programów.



Rysunek 5. Menu główne w firmware LeJOS NXJ

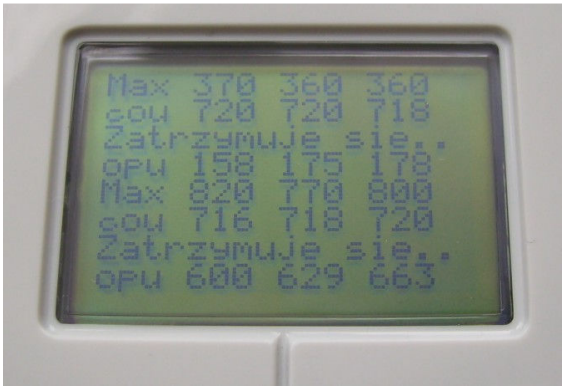
4.1.2 Eksperymentowanie i rozwiązania

Eksperymenty zacząłem od napisania programu Hello World, który przesłałem i uruchomiłem na kostce NXT. Następnie, korzystając z pomocy tutoriala LeJOS, poznawałem sposoby korzystania i możliwości kostki oraz dołączonych do zestawu sensorów i silników. Oto niektóre wnioski z moich obserwacji.

4.1.2.1 Silniki

Na rysunku 6 widoczne są odczyty maksymalnych prędkości, dokładności czujników rotacji oraz kąty potrzebne do zatrzymania się silników. Jeśli wierzyć czujnikom rotacji, każdy silnik ma nieco inne parametry.

Z powodu ograniczonej liczby silników musiałem dokładnie przemyśleć jakie rozwiązania zastosować aby funkcjonalność robota była wystarczająca przy wykorzystaniu zaledwie trzech silników. Dwa silniki zostały wykorzystane do napędu, więc do obsługi chwytaka musiał wystarczyć jeden. Dzięki sprytnemu mechanizmowi podnoszenia przedmiotów udało się zrealizować założenia. Szerszy opis tego mechanizmu znajduje się w punkcie 4.1.2.6.



Rysunek 6. Odczyty z silników

4.1.2.2 Sensor natężenia światła

Sensor światła może działać w dwóch trybach: bez podświetlenia (rysunek 7) oraz z podświetleniem (rysunek 8). Pierwszy tryb przydaje się, gdy chcemy zmierzyć natężenie światła w np. pokoju. Natomiast drugi tryb przeznaczony jest do pomiaru natężenia światła z bliskiej odległości. Dzięki własnemu doświetleniu odczyty prawie nie są wrażliwe na zewnętrzne zmiany natężenia światła. Trzeba zaznaczyć, że ten czujnik mierzy jedynie natężenie światła, a nie rozróżnia kolory (widzi na czarno – biało). Możliwe jest kalibracja, aby dostosować go do własnych warunków oświetleniowych. W moim przypadku, po kalibracji, sensor zwraca wartość 100 gdy znajduje się nad białą linią, i wartość 0 gry znajduje się nad czarną linią. Sensor ten wykorzystałem do śledzenia linii trójkolorowej w trybie z podświetleniem. Poniższy fragment kodu odpowiada za korygowanie toru jazdy gdy robot wyjeżdża ze środkowej linii.

```
if(light.readValue() < 30)          // Jak wjeżdża na czarną linie
      skrec('r');                   // Skręć w prawo

if(light.readValue() > 83)          // Jak wjeżdża na białą linie
      skrec('l');                   // Skręć w lewo
```

Rysunek 7. Czujnik natężenia światła z wyłączonym podświetleniem

Rysunek 8. Czujnik natężenia światła z włączonym podświetleniem

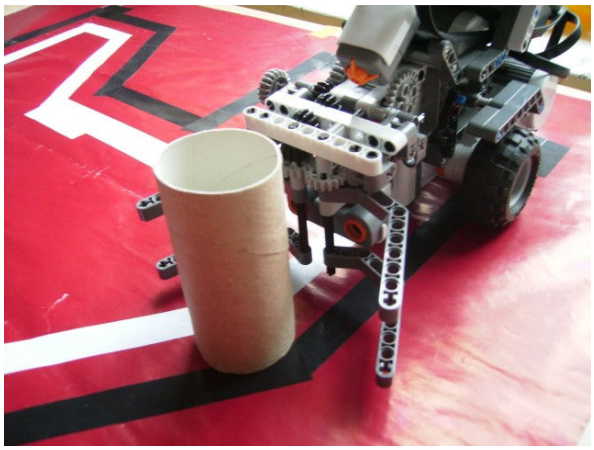
4.1.2.3 Sensor dotyku

Czujnik dotyku wykrywa nacisk na umieszczony z przodu obudowy pomarańczowy przycisk i zwraca go kostce NXT. Można wyszczególnić różne akcje na naciśnięcie oraz na puszczenie przycisku. W moim robocie używany jest do wykrywania, czy chwytak faktycznie chwycił przedmiot. Funkcja ta realizowana jest poprzez porównanie kąta o jaki musiał obrócić się silnik aby chwytak dotknął czujnika z pewną wartością graniczną. Jeśli chwytak nie ma nic wewnątrz kąt ten będzie większy. Dzięki tej funkcji robot nie będzie próbował przestawić przedmiotu, którego nie udało mu się chwycić.

4.1.2.4 Sensor ultradźwiękowy

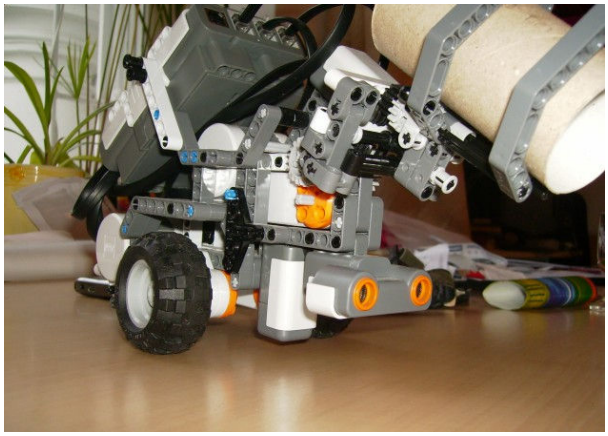
Sensor ultradźwiękowy mierzy w centymetrach odległość do przeszkody. W praktyce w miarę dokładne odczyty (+/- 5 cm) można uzyskać w zakresie od 4 do ok 150 cm. Jeśli

przeszkoda znajduje się dalej niż 150 cm, czujnik zazwyczaj zwraca wartość 255 (nie wykrywa nic). Czujnik miewa czasem problemy z lokalizowaniem przedmiotów okrągłych bądź kulistych. Ja wykorzystałem ten sensor do wykrywania przeszkód na drodze, jak na rysunku 9.



Rysunek 9. Robot wykrywa przeszkodę

Chwytak robota zaprojektowany jest w taki sposób, by po podniesieniu przeszkody nie zasłaniała ona „pola widzenia” sensorowi ultradźwiękowemu, jak widać to na rysunek 10. W pierwszym założeniu projektu było to wymagane, aby robot mógł znaleźć wolne miejsce do odstawienia przedmiotu, jednak obecnie robot nie sprawdza czy miejsce na którym chce postawić przedmiot jest wolne.



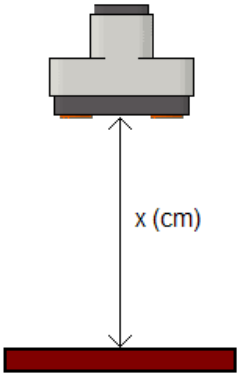
Rysunek 10. Podniesiony chwytak nie zasłania „pola widzenia” sensorowi ultradźwiękowemu

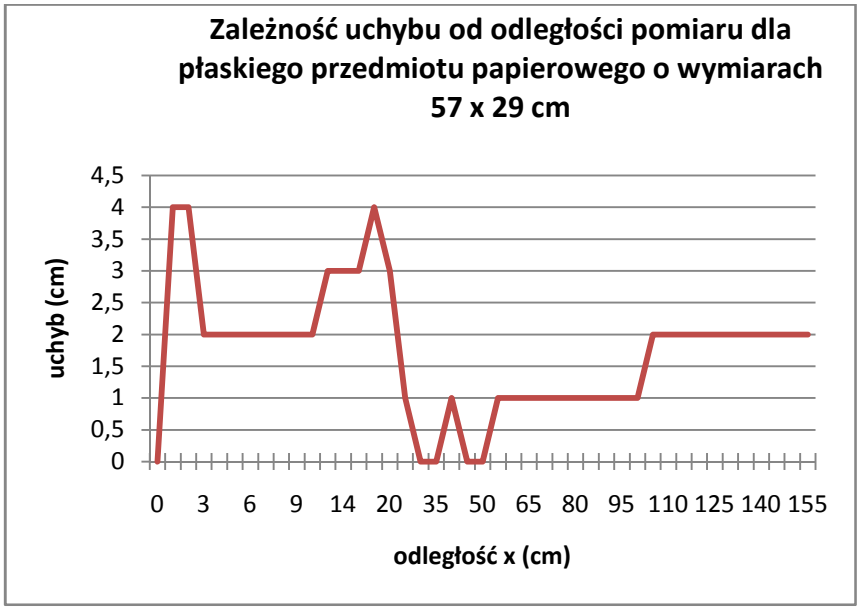
Sensor ultradźwiękowy jest głównym powodem, dlaczego nie udało mi się zrealizować pierwotnych celów projektu. Chciałem go wykorzystać do lokalizowania przedmiotów w obszarze [A] zaznaczonym na rysunku 1. Wykonałem serię eksperymentów jak efektywnie szukać przedmiotów znajdujących się najbliżej robota wewnątrz obszaru. Jednak mała dokładność, a przede wszystkim niezdolność do wykrywania niewielkich przedmiotów z większej odległości uniemożliwiła mi realizację początkowych założeń.

Na wykresach znajdują się wyniki pomiarów ukazujące uchyby między pomiarem zwróconym przez sensor ultradźwiękowy a prawdziwą odległością przy różnych odległościach i ustawieniach sensora względem obiektów.

1) Płaski papierowy przedmiot o wymiarach 57 cm x 29 cm umieszczony prostopadle przed sensorem.

Pierwszy odczyt przy ustawieniu widocznym na rysunku 11 udało się uzyskać przy odległości 1 cm, jednak był na bardzo niedokładny. Ostatni pomiar został wykonany, gdy obiekt znajdował się w odległości 155 cm od sensora. Przy większych odległościach sensor nie wykrywał nic. W miarę dokładne pomiary (+/- 2cm) otrzymuje się przy odległościach większych od 3 cm. Gdy przedmiot znajdował się w odległości 15 – 20 cm błędy były większe, dochodziły do 4 cm. Można także zauważyć, że odczytywane z kostki NXT pomiary są prawidłowe lub zawyżane, nigdy zaniżane. Można to wykorzystać do samodzielnego skalibrowania sensora w celu uzyskania dokładniejszych wyników.

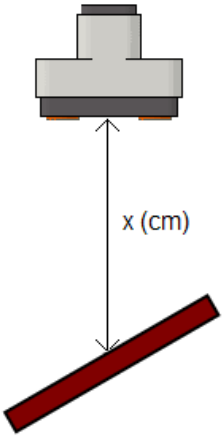


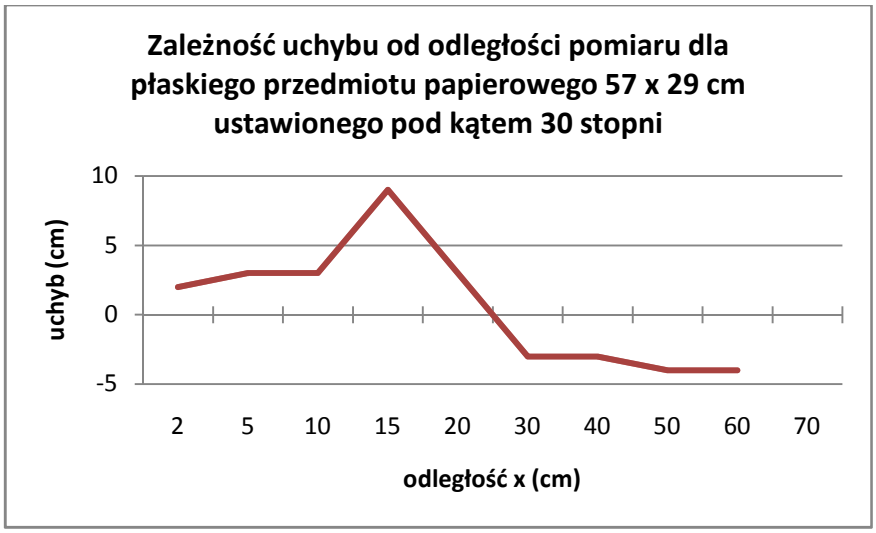


Rysunek 11.

2) Płaski przedmiot papierowy 57 x 29 cm umieszczony pod kątem 30 stopni

Pomiary obiektu znajdującego się pod kątem 30 stopni względem sensora (rysunek 12) okraszone są większymi błędami. Pierwszy odczyt udało się uzyskać przy odległości 2 cm. Ostatni pomiar został wykonany, gdy obiekt znajdował się w odległości 60 cm od sensora. Przy większych odległościach sensor nie wykrywał nic.

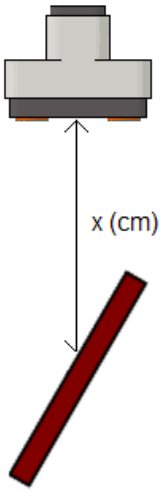


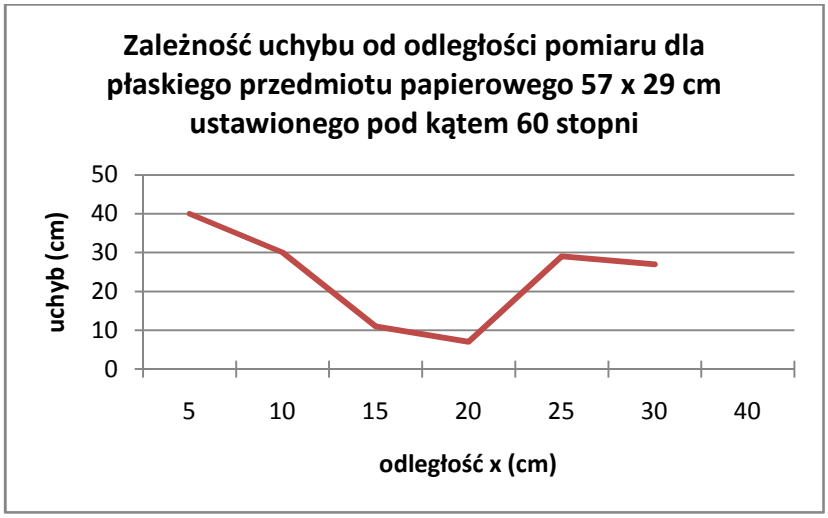


Rysunek 12.

3) Płaski przedmiot papierowy 57 x 29 cm umieszczony pod kątem 60 stopni

Pomiary obiektu znajdującego się pod kątem 60 stopni względem sensora (rysunek 13) obarczone są znacznie większymi błędami. Pierwszy odczyt udało się uzyskać przy odległości 5 cm. Ostatni pomiar został wykonany, gdy obiekt znajdował się w odległości zaledwie 30 cm od sensora. Przy większych odległościach sensor nie wykrywał nic.

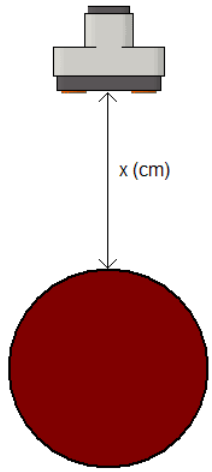


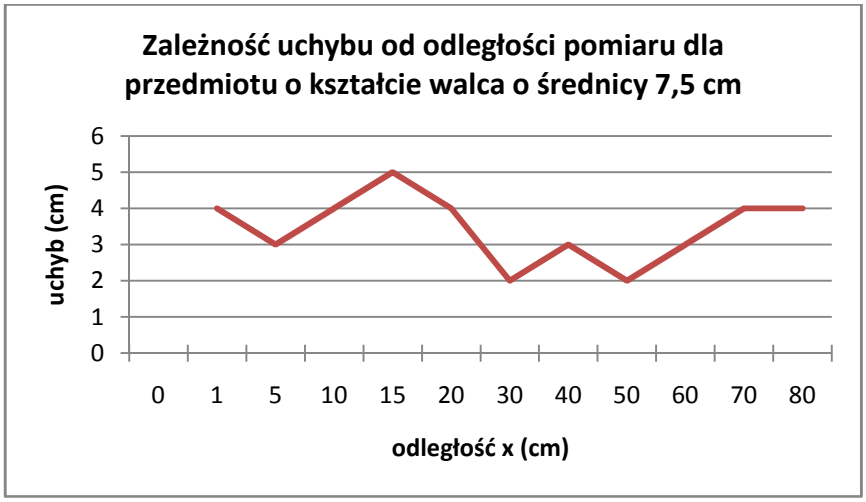


Rysunek 13.

4) Walec o powierzchni gładkiej (szkło) o średnicy 7,5 cm

Podczas pomiaru odległości do obiektu o kształcie walca o średnicy 7,5 cm (rysunek 14) błędy oscylują w granicach 2 – 5 cm. Wyniki, podobnie jak w przypadku przedmiotu umieszczonego prostopadle, zawsze są zawyżone. Pierwszy odczyt udało się uzyskać przy odległości 1 cm. Ostatni pomiar został wykonany, gdy obiekt znajdował się w odległości 80 cm od sensora. Przy większych odległościach sensor nie wykrywał nic.

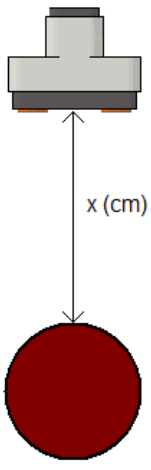


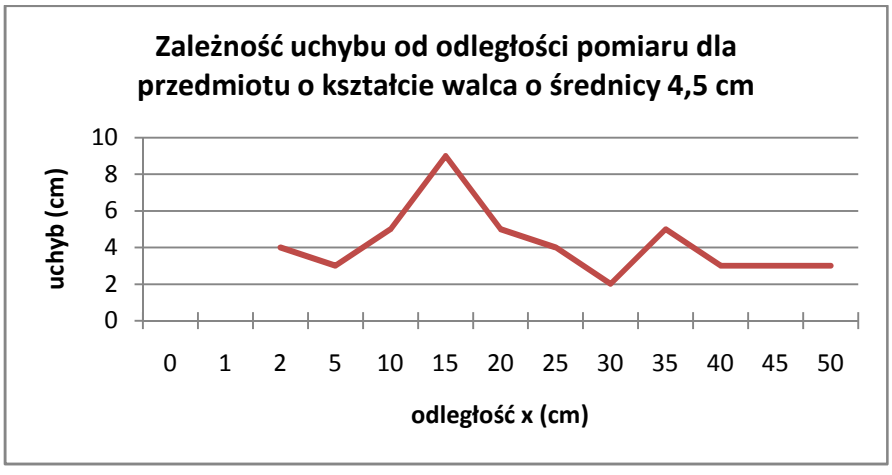


Rysunek 14.

5) Walec papierowy o średnicy 4,5 cm

Podczas pomiaru odległości do obiektu o kształcie walca o średnicy 4,5 cm (rysunek 15) błędy większe niż w przypadku walca o większej średnicy. Wyniki, podobnie jak w przypadku przedmiotu umieszczonego prostopadle, zawsze są zawyżone. Pierwszy odczyt udało się uzyskać przy odległości 2 cm. Ostatni pomiar został wykonany, gdy obiekt znajdował się w odległości 50 cm od sensora. Przy większych odległościach sensor nie wykrywał nic.

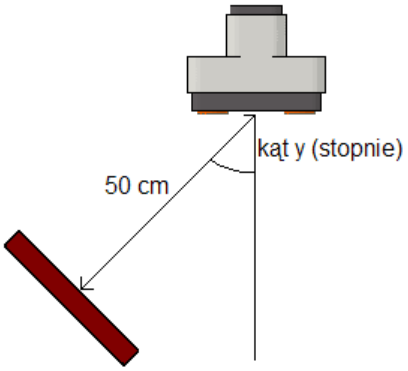


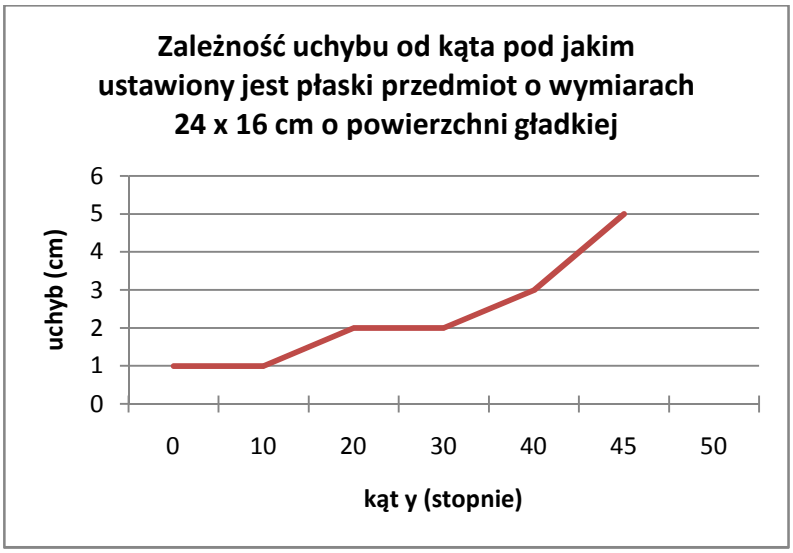


Rysunek 15.

6) Płaski przedmiot o powierzchni gładkiej (pokrytej folią) 24 cm x 16 cm umieszczony pod różnymi kątami względem sensora przy stałej odległości 50 cm

Ten eksperyment miał ukazać, jak zachowuje się sensor ultradźwiękowy w przypadku, gdy przedmiot nie znajduje się wprost przed nim (rysunek 16). Jak można się było spodziewać, błędy w odczytach są tym większe im większy jest kąt odchylenia obiektu. Pierwszy odczyt udało się oczywiście uzyskać gdy przedmiot znajdywał się wprost przed sensorem (kąt 0 stopni). Ostatni pomiar został wykonany, gdy obiekt znajdował się pod kątem 45 stopni. Przy większych kątach sensor nie wykrywał nic.





Rysunek 16.

Podsumowując powyższe wyniki można stwierdzić, że sensor najlepiej radzi sobie z dużymi płaskimi przedmiotami umieszczonymi prostopadle przed nim. Im obiekt jest mniejszy i bardziej „krzywy" lub chropowaty tym pomiary są gorsze.
W przypadku przedmiotu pokrytego folią udało mi się go zlokalizować nawet z odległości 210 cm od sensora. Jest to znacznie więcej niż w przypadku przedmiotu papierowego, który jest bardziej chropowaty.

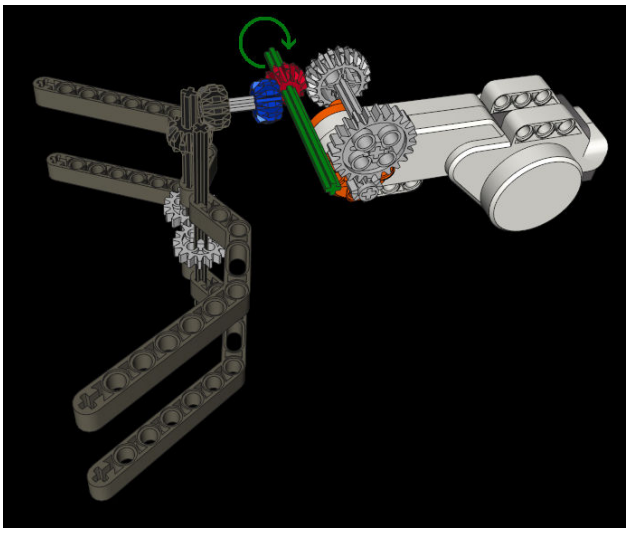
4.1.2.5 Sensor natężenia dźwięku

Sensor natężenia dźwięku w jest najmniej użytecznym sensorem w zestawie. Zwraca on wartość natężenia w zakresie od 0 do 100%, przy czym cisza panująca w pustym pokoju to natężenie rzędu kilku procent a przy głośnym klaśnięciu sensor zwraca wartość dochodzącą do 100%. Nie używam go w moim projekcie.

4.1.2.6 Mechanizm podnoszenia przedmiotów

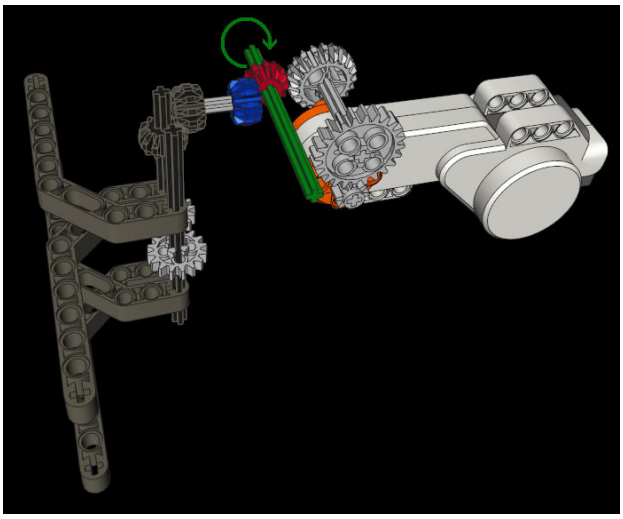
Mimo, że chwytak wykonuje pozornie dwie różne czynności (chwytanie i podnoszenie), działa za pomocą zaledwie jednego silnika. Udało się to dzięki zastosowaniu mechanizmu, którego działanie przedstawię na trzech kolejnych rysunkach:

1) Na rysunku 17 chwytak jest w pozycji gotowej do chwycenia przedmiotu. Gdy robot wykryje obiekt silnik zaczyna działać, zielona oska obraca się w kierunku pokazanym przez zieloną strzałkę, czerwona zębatka napędza niebieską. W wyniku tego ramiona chwytaka składają się do wewnątrz.



Rysunek 17.

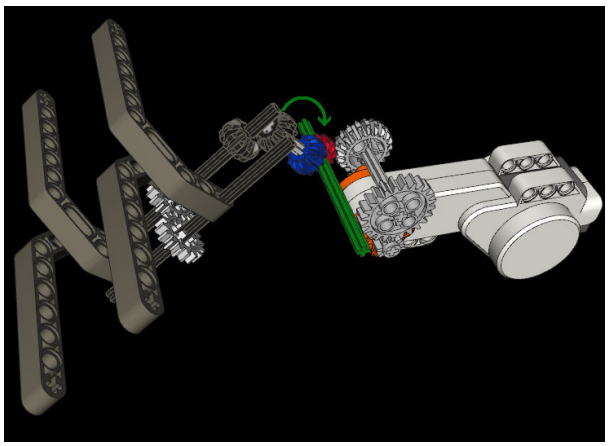
2) Gdy ramiona złożą się maksymalnie do środka, ich dalszy ruch nie jest możliwy w płaszczyźnie poziomej. Mimo to, silnik działa nadal a zielona oska nadal się obraca w tym samym kierunku. Pokazane jest to na rysunku 18.



Rysunek 18.

Dla lepszego uwidocznienia działania mechanizmu, na rysunku przedstawiony jest przypadek gdy ramiona chwytaka składają się bez przedmiotu wewnątrz.

3) Zielona oska pełni funkcję przeniesienia napędu na kolejne zębatki oraz jest osią obrotu całego chwytaka w płaszczyźnie pionowej. Gdy dalszy ruch ramion chwytaka nie jest możliwy czerwona zębatka „podnosi" niebieską zębatkę razem z całym chwytakiem. W tym momencie zaczyna się ruch chwytaka w płaszczyźnie pionowej. Gdy chwytak podniesie się do odpowiedniej wysokości, spowoduje on naciśnie sensora dotyku i silnik chwytaka zatrzyma się. Dzięki zastosowaniu sensora dotyku chwytak podniesie się zawsze na taką samą wysokość, niezależnie czy przedmiot został chwycony czy nie. Na rysunku 19 widać końcowe stadium podnoszenia chwytaka.



Rysunek 19.

4.2 Opis implementacyjny

4.2.1 Model kontroli zachowaniami

Do sterowania robotem wykorzystałem model kontroli zachowaniami (behavior control model). Składają się na niego klasy implementujące interfejs Behavior reprezentujące zachowania oraz klasa zarządzająca tymi zachowaniami. Każda klasa reprezentująca pewne zachowanie musi zawierać 3 metody:

- **public boolean takeControl()**

Zwraca wartość boolean pokazującą, czy to zachowanie powinno zostać aktywowane; w przypadku klasy Zakoncz, zwraca wartość true, gry naciśnięty zostaje przycisk Escape na kostce NXT, a wygląda następująco:

```
public boolean takeControl() {
      return Button.ESCAPE.isPressed();
}
```

- **public void action()**

Zwraca wartość boolean pokazującą, czy to zachowanie powinno zostać aktywowane; dla klasy Zakoncz wygląda następująco:

Kod znajdujący się w tej metodzie wykonuje się, gdy zachowanie aktywuje się. W przypadku klasy Zakoncz kod ten zakańcza cały program.

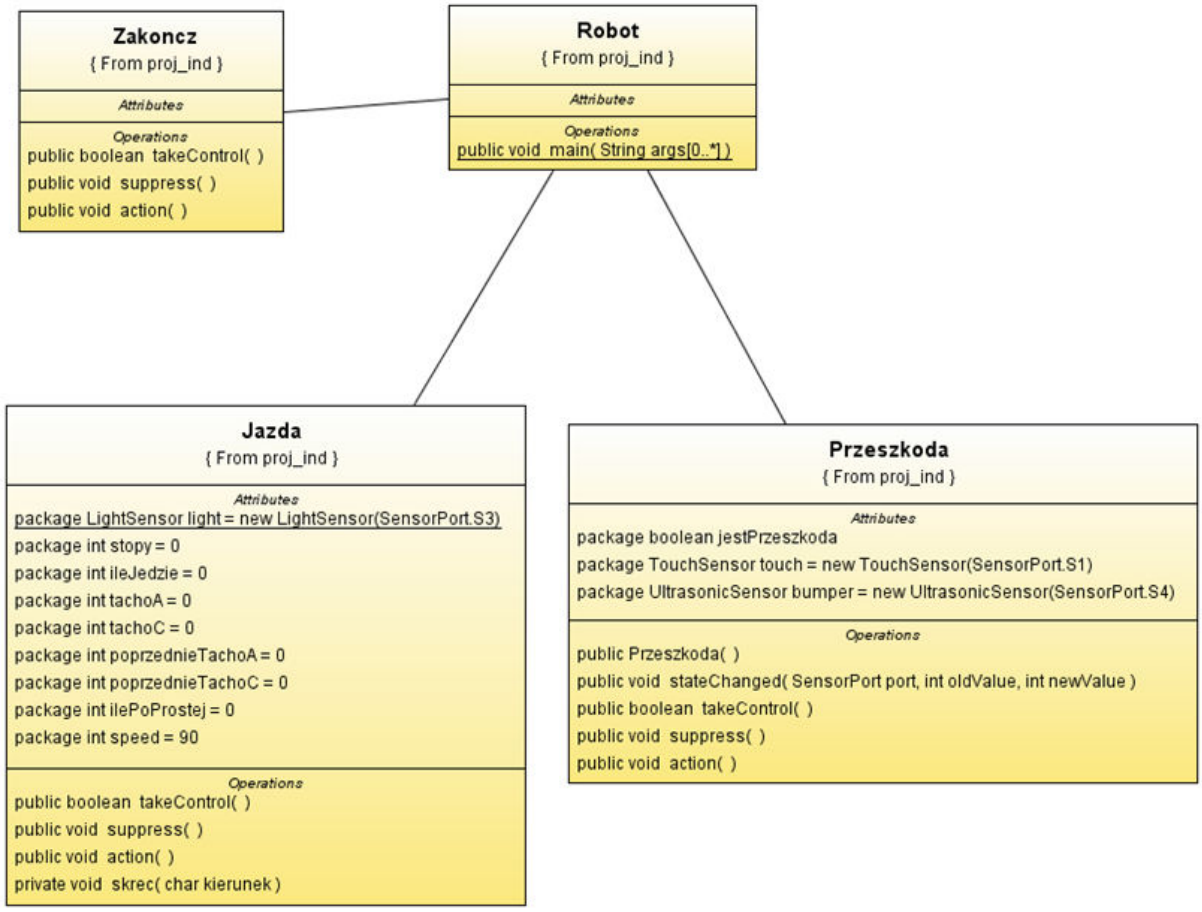
```
public void action() {
      System.exit(0);
}
```

- **public void suppress ()**

Kod w metodzie suppress() powinien natychmiast zakończyć wszystkie akcje rozpoczęte w metodzie action(), może też zostać wykorzystana do zaktualizowania zmiennych zanim dane zachowanie się zakończy. P przypadku klasy Zakoncz nie ma potrzeby umieszczania jakiegokolwiek kody w tej metodzie.

```
public void suppress() {
}
```

4.2.2 Diagram klas



4.2.3 Opis metod

- **Metoda public void action() w klasie Jazda**

Metoda ta odpowiedzialna jest za jazdę po linii. Jeśli robot znajduje się w środkowej linii – jedzie prosto. Jeśli wjedzie na jedną z zewnętrznych linii, wywołuję metodę skręć z parametrem, w którą stronę ma skręcić, w zależności od koloru linii na jaką wjechał.

- **Metoda private skrec(char kierunek) w klasie Jazda**

Metoda odpowiedzialna za skręcanie, wywoływana w przypadku wjechania na jedną z linii zewnętrznych. Posiada jeden parametr, informującą w którą stronę należy skręcić. Po odpowiednim przypisaniu silników do obiektów klasy Motor zaczyna się pętla while. Wykonuje się ona dopóki robot znów nie wjedzie na linię środkową. Wewnątrz pętli, zatrzymywany jest silnik po tej stronie w którą robot chce skręcić. Następnie liczony jest kąt obrotu silnika. Jeśli przekroczy od wartość graniczną przed tym jak robot wyjedzie z linii

zewnętrznej, skręt jest traktowany jako ostry. Aby przyspieszyć wykonywanie tego skrętu (oraz zapewnić, żeby robot nie wyjechał poza zewnętrzną linię) stojący do tej pory silnik zaczyna obracać się do tyłu. Gdy robot wjedzie znów na środkową linię, wychodzi z pętli. Następnie, jeśli zakręt był ostry, obraca jeszcze silniki w strony w które się obracały o pewien kąt, aby robot wjechał głębiej w środkową linię.

- **Metoda takeControl() w klasie Przeszkoda**

Zwraca wartość boolean pokazującą, czy chwytak powinien podjąć próbę chwycenia przedmiotu. Metoda ta zwróci wartość true jeśli zajdą dwa warunki: sensor ultradźwiękowy zwróci wartość mniejszą niż graniczna, oraz sensor dotyku zwróci wartość 0 (tzn. będzie wyciśnięty). Zapobiega to próbom chwytania przedmiotu, gdy chwytak jest już podniesiony i naciska na sensor dotyku.

- **Metoda public void action() w klasie Przeszkoda**

Metoda odpowiedzialna za obsługę chwytaka. Wprawia ona w ruch silnik chwytaka oraz liczy kąt o jaki silnik ten się obrócił, dopóki chwytak nie nacisnął sensora dotyku. Jeśli kąt ten jest większy niż wartość graniczna, znaczy to, że chwytak jest pusty. W takim przypadku opuszcza chwytak do pozycji początkowej i się kończy. Jeśli kąt ten jest mniejszy niż wartość graniczna, robot obraca się o 90 stopni, jedzie do przodu pewną odległość, opuszcza chwytak, jedzie do tyłu o taką samą odległość i obraca się o kąt -90 stopni. W rezultacie tej procedury robot znajduje się w takiej samej pozycji jak w chwili wykrycia przedmiotu.

- **Metoda public static void main(String [] args)**

Metoda uruchamiająca cały program. Tworzone są instancje wszystkich zdefiniowanych uprzednio zachowań (Behavior) oraz instancja obiektu Arbitrator z tablicą zachowań jako parametr. Kolejność zachowań w tej tablicy decyduje o ich priorytetach zachowań. Następnie wywoływana jest metoda start() zaczynająca arbitrowanie, czyli decydowanie które zachowanie powinno zostać aktywowane.

```
public static void main(String [] args) {
      Behavior jazda1 = new Jazda();
      Behavior przeszkoda = new Przeszkoda();
```

```
Behavior zakoncz = new Zakoncz();
      Behavior [] bArray = {jazda1, przeszkoda, zakoncz};
      Arbitrator arby = new Arbitrator(bArray);
      arby.start();
}
```

4.3 Opis uruchomieniowy

4.3.1 Czynności wstępne

- Zainstalować Eclipse SDK
- Zainstalować dystrybucję LeJOS NXJ 0.7
- Zmienna środowiskowa *NXJ_HOME* wskazująca katalog główny powyższej instalacji
- Do zmiennej PATH należy dodać: D:\nxt\lejos_nxj\bin;
- Zainstalować leJOS NXJ Eclipse Plugin 0.7:

  Należy użyć menagera aktualizacji Eclipse (Help -> Software Updates) i wybrać LeJOS NXJ Eclipse bądź wpisać adres URL: http://lejos.sourceforge.net/tools/eclipse/plugin/nxj/

4.3.2 Wgrywanie firmware LeJOS NXJ na kostkę NXT

Aby wgrać firmware należy ustawić kostkę NXT w „reset mode” poprzez naciśnięcie przycisku reset znajdującego się w jednej z dziur na pin z tyłu kostki. Kiedy kostka znajduje się w stanie umożliwiającym wgranie nowego firmware będzie wydawać cichy, pulsujący dźwięk.

Następnie trzeba wybrać w Eclipse menu LeJOS NXJ -> Upload Firmware. Gdy proces wgrywania ukończy się na wyświetlaczu LCD kostki pojawi się logo LeJOS NXJ. Teraz kostka NXT jest już gotowa do odbierania i uruchamiania programów.

4.3.3 Kompilowanie programu

Aby móc kompilować programy przy użyciu LeJOS NXT API i wysyłać je do kostki NXT trzeba stworzyć projekt odpowiedniego typu i ustawić zmienną Classpath aby nowe biblioteki wykorzystywane było do kompilacji i ustanawiania połączenia. Aby to zrobić należy przekonwertować projekt na projekt LeJOS NXJ (Projekt -> leJOS NXJ -> Convert to LeJOS NXJ Project)

4.3.4 Wysyłanie programu do kostki NXT

Aby ustanowić połączenie i wysłać program należy wskazać klasę zawierającą i otworzyć menu kontekstowe. Następnie wybrać menu LeJOS NXJ i Upload program to the NXT brick.

Aby uruchomić program należy użyć opcji Run w menu kostki NXT.

W wersji pluginu 0.7 procedura instalacji znacznie się uprościła. Na początku pracy nad projektem znaczną część z tych czynności zmuszony byłem robić ręcznie w linii komend.

5 Podsumowanie i wnioski

Niniejszy projekt dotyczy zagadnienia programowania robotów LEGO NXT. Ostatecznym celem było zaprogramowanie robota tak, aby śledził linie trójkolorową oraz usuwał przeszkody ze swojej drogi.

Program sterujący robotem napisany jest w języku Java za pomocą platformy Eclipse. Wykonywanie programu na kostce NXT możliwe jest dzięki firmware LeJOS NXJ. Do sterowania robotem wykorzystałem model kontroli zachowaniami.

Do sukcesów można zaliczyć pełne skonfigurowanie środowiska, zapoznanie się ze sprzętem i jego możliwościami, zrealizowanie celów ostatecznych.

Początkowych założeń nie udało się zrealizować. Głównym powodem była mała skuteczność w wykrywaniu niewielkich przedmiotów przez sensor ultradźwiękowy.

Mimo, że napotkałem na trudności podczas użytkowania narzędzi LeJOS, wywarły one na mnie pozytywne wrażenie. Cieszący jest też fakt, że są one cały czas rozwijane przez społeczność entuzjastów. Zestaw LEGO NXT także okazał się sprzętem z którym pracuje się przyjemnie. Zastrzeżenia budzą jedynie dokładność sensora ultradźwiękowego oraz mała użyteczność sensora natężenia dźwięku.

6 Możliwości rozbudowy

W robocie powstałym w tym projekcie można poprawić następujące elementy:

- Rozwiązanie problemu zatrzymywania się silników podczas manewrów przy słabych bateriach
- Sprawdzanie, czy miejsce gdzie robot chce postawić przedmiot jest puste

Bibliografia

1. Programming Solutions for the LEGO Mindstorms NXT
   http://www.botmag.com/articles/10-31-07_NXT.shtml, ostatni dostęp 26.01.2009
2. Porównanie języków programowania dostępnych dla NXT
   http://mi007.wikispaces.com/file/view/nxt.pdf, ostatni dostęp 26.01.2009
3. LeJOS – strona projetku
   http://lejos.sourceforge.net/nxj.php, ostatni dostęp 26.01.2009

Dodatek

1. **Klasa Jazda.java**

```
package proj_ind;


import lejos.subsumption.*;
import lejos.nxt.*;

public class Jazda implements Behavior {
	static LightSensor light = new LightSensor(SensorPort.S3);

	int stopy = 0;
	int ileJedzie = 0;
	int tachoA = 0, tachoC = 0,
		poprzednieTachoA = 0, poprzednieTachoC = 0;
	int ilePoProstej = 0;
	int speed = 90;

	public boolean takeControl() {
		return true;
	}

	public void suppress() {}

	public void action() {
		LCD.drawString("PROST", 0, 0);
		light.setHigh(560);		// Kalibrowanie sensora światła
		light.setLow(420);
		int speed = 150;		// Predkosc silnikow napedowych
		Motor.A.setSpeed(speed);
		Motor.C.setSpeed(speed);
		Motor.A.forward();		// Uruchomienie silnkow
		Motor.C.forward();

		if(light.readValue() < 30)	// Jak wjeżdża na czarną linie
			skrec('r');		// Skręć w prawo

		if(light.readValue() > 83)	// Jak wjeżdża na białą linie
			skrec('l');		// Skręć w lewo

	}



	private void skrec(char kierunek){  // Metoda wywolywana,
		//gdy robot wjedzie na jedna z linii zewnetrznych

		boolean ostrySkret = false;   // Zmienna zawierajaca
			informacje, czy robot pokonuje teraz ostry zakret

		Motor motor1 = null;
		Motor motor2 = null;

		if(kierunek == 'l'){
			motor1 = Motor.A;
			motor2 = Motor.C;
```

```
            }
            if(kierunek == 'r'){
                  motor1 = Motor.C;
                  motor2 = Motor.A;
            }

            int tacho2 = motor2.getTachoCount();
            LCD.drawString("SKREC", 0, 0);
            LCD.drawString("tac1", 0, 1);
            LCD.drawInt(tacho2, 3, 5, 1);

            motor1.forward();
            motor2.forward();

            while(light.readValue() > 80 || light.readValue() < 30){
                  //jesli robot jest poza srodkowym obszarem     80 30

                  LCD.drawString("gett", 0, 2);
                  LCD.drawInt(motor2.getTachoCount(), 3, 5, 2);
                  LCD.drawString("rozn", 0, 3);
                  LCD.drawInt(motor2.getTachoCount() - tacho2, 3, 5, 3);

                  if (!ostrySkret){
                        //jesli nie ostry skret zatrzymaj motor1,

                        motor1.stop();
                              //jesli ostry skret nie zatrzymuj motor1 zeby
                              // mogl obracac sie do tylu
                  }

                  if(motor2.getTachoCount() > tacho2 + 40){//jesli po
                              //obrocie motor2 o 40 stopni nie wyjedzie z
                              //bocznego obszaru

                        ostrySkret = true;      //jest ostry skret
                        motor1.backward();      //obracaj motor1 do tylu i
                                                // idz dalej

                        LCD.drawString("OSTRY", 0, 0);
                        LCD.drawString("tac1", 0, 4);
                        LCD.drawInt(motor1.getTachoCount(), 3, 5, 4);
                        LCD.drawString("tac2", 0, 5);
                        LCD.drawInt(motor2.getTachoCount(), 3, 5, 5);
                  }
            }

            if (ostrySkret){
                    //jak byl ostry skret a juz wyjechal

                  motor1.rotate(-20);
                        //obrocaj jeszcze motory doboki motor1 nie obroci
                         //sie o 20 stopni

            }
      }
}
```

2. Klasa Przeszkoda.java

```
package proj_ind;


import lejos.subsumption.*;
import lejos.nxt.*;

public class Przeszkoda implements Behavior, SensorPortListener {
	boolean jestPrzeszkoda;
	TouchSensor touch = new TouchSensor(SensorPort.S1);
	UltrasonicSensor bumper = new UltrasonicSensor(SensorPort.S4);

	// Constructor:
	public Przeszkoda() {
		jestPrzeszkoda = false;
		SensorPort.S1.addSensorPortListener(this);
	}

	public void stateChanged(SensorPort port, int oldValue, int
		newValue) {	//Port listener
		if(bumper.getDistance()<10)
			jestPrzeszkoda = true;
	}

	public boolean takeControl() {
		if(jestPrzeszkoda && !touch.isPressed()) {
			jestPrzeszkoda = false; // reset value
			return true;
		} else
			return false;
	}

	public void suppress() {
		System.out.println("Przeszkoda stop");
	}

	public void action() {
		System.out.println("Przeszkoda");
		Motor.A.stop();
		Motor.C.stop();

		Motor.B.setSpeed(100);				//chwyta
		System.out.println("Chwytam");
		while(!touch.isPressed()){
			Motor.B.backward();
		}
		if(Motor.B.getTachoCount() < -450){		//nie chwycil
			System.out.println("Nie chwycil");
			Motor.B.stop();
			Motor.B.rotateTo(0);
		}
		else{						//chwycil
			System.out.println("Chwycil");
			Motor.B.rotate(-20);
			Motor.B.stop();

			Motor.A.rotate(-150,true);		//skreca w lewo
```

```
                    Motor.C.rotate(150);

                    Motor.A.rotate(150,true);           //jedzie do przodu
                    Motor.C.rotate(150);
                    Motor.B.rotateTo(0);
                    Motor.A.rotate(-150,true);                //cofa sie
                    Motor.C.rotate(-150);
                    Motor.A.rotate(150,true);
                    Motor.C.rotate(-150);
              }
        }
}
```

Klasa Zakoncz.java

```
package proj_ind;


import lejos.nxt.Button;
import lejos.subsumption.Behavior;

public class Zakoncz implements Behavior {

      public boolean takeControl() {
            return Button.ESCAPE.isPressed();
      }

      public void suppress() {
            System.out.println("Zakoncz stop");
      }

      public void action() {
            System.out.println("Zakoncz");
            System.exit(0);
      }
}
```

3. Klasa Robot.java

```
package proj_ind;

import lejos.subsumption.*;

public class Robot {

	public static void main(String [] args) {

		Behavior jazda1 = new Jazda();
		Behavior przeszkoda = new Przeszkoda();
		Behavior zakoncz = new Zakoncz();

		Behavior [] bArray = {jazda1, przeszkoda, zakoncz};
		Arbitrator arby = new Arbitrator(bArray);
		arby.start();
	}
}
```