Exercise for course day 2, 11-feb-2009

Implementing a simple behavioural robot using Lego's graphical programming language

1 Introduction and specification of the task

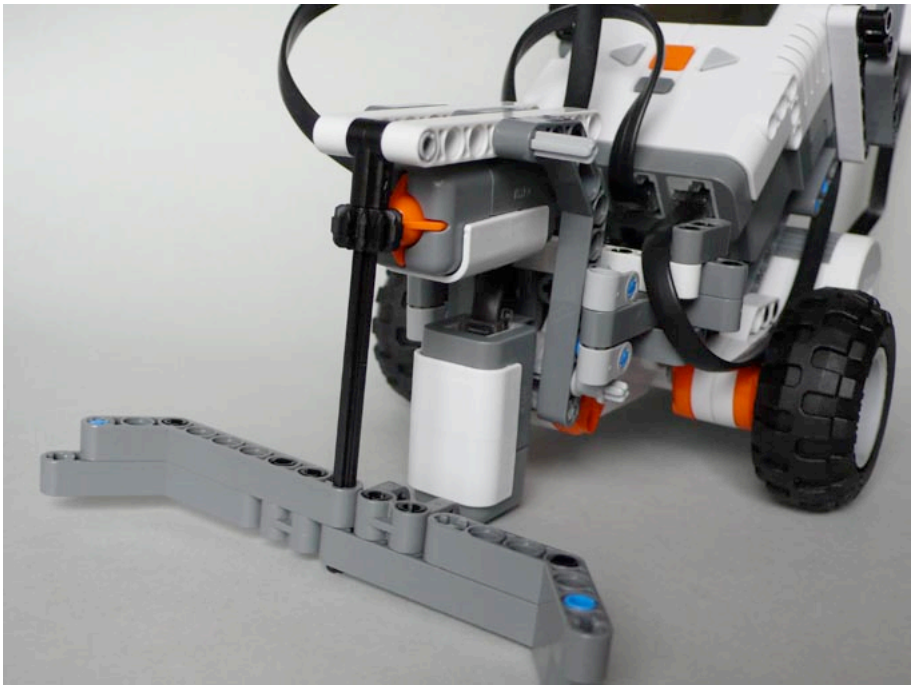
You are going to implement a robot based on the principle of behavioural robotics as it is outlined in the first two chapters of [Jones], although we take it still at an intuitive way, and postpone the strict definition of behaviours until later.The overall task is to make a robot that performs the following task:

> *Maintain an area, delimited by a thick black line, free of any objects (except the robot itself). Objects should be pushed out of the area, but the robot must not move outside the area.*

This is inspired by the central example introduced in the textbook, page 1 and further on, but adapted to the sensors that we have available for the Lego robot. We will suggests extensions of the tasks later for those of you who get the basic task implemented quickly. You may use the "test pad" included with the Lego sets, but if you have the opportunity, you may try with a larger area, e.g., using the floor and use black tape or paper (but be sure you don't get into a conflict with the cleaners).

The robot can be based on a modification of the TriBot described in Lego documentation, and it should be programmed in NXT-G graphical programming language. You can remove the grab and you should instead attach to the moving parts that affect the push sensor, some sort shovel or bar that is suited for pushing (not too heavy and not too light) objects and at the same time make it possible for the push sensor to register a resistance indicating that there is something to push.

The light sensor should be mounted in the same way as in for the TriBot. Your robot may look somewhat like the following.



Be aware that there will be given a written assignment very soon on describing how you solved this exercise; requirements and format will be explained very soon. This means that you are strongly advised to take notes of how you solve the exercise; exercises are expected to be made in groups of 2–3 students, but answers to written assigments must be written individually be each students.

2 Behaviours

The task can be solved by introducing three different robot behaviours (in the sense introduced in the textbook).

- Pushing an object (expecting that it sooner or later to be pushed out from the delimited area.
- Avoiding the black line.
- Wandering around (searching for objects to remove).

The robot has no way of checking that the area is free of objects that must be removed and new objects may be dropped in the area; this means that it may look as if the robot is patrolling, searching eagerly for unwanted objects to throw out.

The behaviours should be implemented by a kind of servo-mechanism, which we (for now) can consider as a principle of performing a selected behaviour only in a very short time interval before the sensors are checked again.

You will need firstly to describe these behaviours in more detail and for each specify under which conditions of trigger readings they are relevant to perform. Specify also an order of importance in which the different behaviours should be activated, when more that one is possible.

The textbook describes a component called an “arbiter” that selects between the different actions proposed by the different behaviours. Due to the lack proper data

structures (for representing actions), it is proposed that you replace the arbiter with a choice which determines from the sensors which one of the possible behaviours that takes precedence.

Take notes about the different choices that you make when solving the exercise.

3 Some advice and warnings concerning the use of the Mindstorms NXT programming environment

Be aware that the graphical editor has several disadvantages, the most annoying one is, perhaps, that it is impossible to get an overview of your program. You see only a small fragment, so it is strongly recommended that you maintain a consistent sketch of you program's overall structure on a piece of paper. There is a sort of procedural abstraction called "My bricks" but which is a bit difficult to use due to the lack of proper parameter passing mechanisms (and lack of data types as well as arithmetic expressions).

You may need to use the so-called data wires that port information between bricks. Be aware that the editor has several problems with those.

- Sometimes the line drawing algorithm puts the lines behind bricks and their so-called data hubs, so the wires seem to be attached to other connecting points that the actually are.
- The editor has the bad habit of deleting your data wires in case you move a brick, e.g., to improve the layout.
- Sometimes the editor interprets your mouse movements and clicks as if you draw data wires from a connecting point to nowhere. You cannot explicitly delete those; the only way seems to remove the brick and start over again, placing a brick once again and setting up the data wires to and from it.

Your teacher did some experiments with different ways of structuring a program based on behaviours. The only thing that seemed to work is to use a nested structure of so-called switches that tests the behaviours one at a time in the order of importance, and then select one. In a traditional programming syntax it can be sketched as follows

```
if <behaviour-1-trigger condition is met> then  <do a small step of behaviour-1>
  else if <behaviour-2-trigger conditions is met> then  <do a small step of behaviour-1>
     else if
     .
     .
     .
       else if  <behaviour-n–1-trigger conditions is met> then  <do a small step of behaviour-n–1>
          else if  <behaviour-n-trigger conditions is met> then  <do a small step of behaviour-n>
```

If you can structure your trigger conditions in such a way that you need only check a single sensor at the entry of each switch (= if-statement), it fits best with the language; you may then rely on the fact that the previous conditions were false a few milliseconds ago, and most like are still.

Another computer scientist's way of thinking, such as checking all sensors at the same moment, and then having one switch to select behaviour from the reading,

becomes far to clumsy due to inherent limitations of the language. (NB: You may try to do so, if you want to learn why this is the case).

Using interrupt techniques does not seem possible.

4 Possible extensions

If you have got the robot to perform the task described above in a proper way, you may extend the problem and your solution to include one or more of the following.

- Use the ultrasound sensor together with the touch sensor to identify some objects as "tall objects" (tall from the robot's perspective), and which might be interpreted as a piece of furniture or a person's leg.
  Such tall objects should not be pushed, but turned away from.
- Use the rotation sensor to check if the robot is trying to push an object which is too heavy for the robot. It is reasonable that the robot gives up for such objects (or perhaps move backwards to obtain a run-up for another attempt, but be sure that the robot does not get stuck in a loop here)
- Use the rotation sensor to identify if the robot has got stuck in a corner or for some other reason, so that it has not moved nearly as much as expected within a time interval.

If you extend you robot to cope with some of these problems, make sure that you describe the necessary behaviours, their triggering conditions and level of importance. Notice that some of these tasks may involve fine-tuning of the mechanical parts of the robot and its interplay with the program. Some of them may also require that you extend your program structure with global variables (as being able to measure progress or lack of such).

Literature

[Jones] Joseph L. Jones: *Robot programming, A practical guide to behavior-based robotics*. McGraw-Hill, 2004.