High-Level Sensors in LeJOS using Threads
Notes and exercises

March 5, 2018

Henning Christiansen
Roskilde University
http://www.ruc.dk/~henning/, henning@ruc.dk

1 Sensors and High-Level Sensors

Admittedly, the distinction between "sensors" and "high-level sensors" is blurry. We tend to think of EV3's touch sensor as a "simple" or "low-level" sensor, since it registers physical events that are simple, instant and easy to comprehend. The same thing for a thermometer (if we had had such one as a sensor) although it does register events, but a continuously changing property, which, however, still has an immediate physical relationship.

But the touch sensor does not really register "touching". The physical sensor consists of some mechanics and electrical wires and a few electronic components – so what it (most likely) registers are some changes in voltage inside this electro-mechanical aggregate. And an *interpretation* of these changes in voltage is necessary for claiming that a touching event has been observed. EV3's infrared sensor requires a quite advanced interpretation of various physical phenomena, which is only comprehensible for especially educated and skilled people, in order to report that some physical object is observed in some distance. The mentioned interpretation processes may be embedded inside the plastic thing (that we usually refer to as "the sensor") in terms of electronic circuits or a program running ona very small and dedicated computer, or by software running on the EV3 brick. Likely it is a combination.

We take "high-level sensor" to mean a software handle that makes it possible for a program to check properties that usually requires cognitive skills, including perhaps cultural norms etc. To implement a new high-level sensor requires some software for the interpretation as well as some data to be interpreted; these data may be obtained directly from less high-level sensors or internal data structures and processes. The exercises to be given below, considers a bumpy road detector, which gets its data from an EV3 touch sensor and keeps track of the density of touch events (understood here as single bumps) over time.

2 Sensors and high-level sensors in LeJOS – and some comments on the nightmare of Java style object orientation and how to bypass it

In the lecture last week, we saw how the different EV3 sensors could be accessed in LeJOS. For all sensors, a reading is returned as a float array – and this is independent of whether this makes intuitive sense or not for the particular sensor.

If, for example, the reading of a sensor essentially should return a `true` or `false` value, this is returned as a float array of length 1, with (e.g.) `true` represented as `1.0000000` and `false` represented as `0.0000000`.

Each sensor type has its own class, and to use a sensor attached to an EV3 brick, we need to create an object of that type, plus one (or more) objects of class `SampleProvider` for reading the sensor. For example, for using a touch sensor, the following declarations are needed.

```
EV3TouchSensor touch = new EV3TouchSensor(SensorPort.S1);
SampleProvider touched = touch.getTouchMode();
float[] sample = new float[touched.sampleSize()];
```

To read the sensor, we use the declared `SampleProvider` as follows.

```
touched.fetchSample(sample,0);
```

This method sends the result back to the calling code by side-effects in the first argument `sample`, i.e., it modifies its argument (not good programming style according to your teacher). ((To find out what the last argument with value `0` means, check the documentation.))

When we introduce a new high-level sensor it may be considered a virtue to make it appear in the same way as other sensors, so we have the possibility to replace the use of one (high-level or not) sensor by another without modifying the code in any essential way. This will also make it possible to have existing classes that use a sensor (as arguments, through generics or inheritance or other ways) to accept our new high-level one. More precisely, we should find the right place in the LeJOS class hierarchy to place the new sensor class. The simplest way to do that is not to try to understand the entire class hierarchy, but approach the problem in an analytic way.

We show here the position of some of those sensor classes we are familiar with, copy-pasted from the documentation.

Class EV3TouchSensor

java.lang.Object
- lejos.hardware.Device
  - lejos.hardware.sensor.BaseSensor
    - lejos.hardware.sensor.AnalogSensor
      - lejos.hardware.sensor.EV3TouchSensor

Class EV3IRSensor

java.lang.Object
- lejos.hardware.Device
  - lejos.hardware.sensor.BaseSensor
    - lejos.hardware.sensor.UARTSensor
      - lejos.hardware.sensor.EV3IRSensor

Class RCXThermometer

java.lang.Object
- lejos.hardware.Device
  - lejos.hardware.sensor.BaseSensor
    - lejos.hardware.sensor.AnalogSensor
      - lejos.hardware.sensor.RCXThermometer

Class EV3ColorSensor

java.lang.Object
- lejos.hardware.Device
  - lejos.hardware.sensor.BaseSensor
    - lejos.hardware.sensor.UARTSensor
      - lejos.hardware.sensor.EV3ColorSensor

As it appears, the paths upwards in the hierarchy meet in the `BaseSensor` class, so we may implement a new high-level sensor as an immediate subclass of `BaseSensor`. It may also be considered if our new sensor has more in common with one of the subclasses `AnalogSensor` or `UARTSensor` and make it an immediate subclass of that instead; in any case, this choice is easy to change later.

Good reasons for not adapting to the LeJOS class hierarchy

As we have seen, the uniform way of accessing a sensor from LeJOS makes the programming quite clumsy, so for most usages, it seems better to provide a specific and simple interface to a new high-level sensor (rather than using 20 unreadable code lines to get the result into a float array and another 20 such lines to get it out).

For the bumpy road sensor, it will give good sense to access it by a boolean function, or even simpler, having a global boolean variable that tells whether or not the road is currently considered bumpy.

3 The exercises

Extend your EV3 robot with a touch sensor that can be used for a high-level bumpy road sensor. Do not spend time on a sophisticated mechanical arrangement so that it actually can sense bumps on the road, but you can place the touch sensor in such a way that you can easily click it many times with your finger, thus giving an impression of a bumpy road.

Write a program for your EV3 robot with two threads that have access to a common boolean variable `theRoadIsBumpyRightNow`.

Thread 1 reads the touch sensor continuously and maintains data structures so that the boolean function `roadIsBumpy()` indicates the following: It is `true` if during the last two seconds, five or more "bumps" has been registered by the touch sensor; otherwise it is `false`.

Thread 2 makes the robot drive (straight ahead or in a closed curve, as you prefer). When the road is bumpy, i.e., `theRoadIsBumpyRightNow` is `true`, it drives with half of its normal speed. When the road is not bumpy, it goes back to normal speed.

A hint

A standard way of keeping track of the number of events of a certain kind is to use a queue of timestamps.[1] When an event occurs, put the current time[2] into one end of the queue and take timestamps that are too old in the other end of the queue. The length of the queue determines the result of `roadIsBumpy()`.

[1] `Queue<Long> queue = new LinkedList<Long>()`

[2] `System.currentTimeMillis()`

Another hint

If you take it as a single-bump event whenever the sensor indicates “I’m pressed” when it is tested by the program look, you will likely get far too many registrations; the event is when the sensor changes status from “I’m not pressed” to "I’m pressed”.