CMP438/738

There and Back Again[1]

Event-based programming

Objectives:

- Design a program to deal with events asynchronously
- Use tasks
- Use global variables for communication between tasks
- Identify potential conflicts between tasks

Issues

Robots have to deal with events whose timing can't be predicted exactly. These are called *asynchronous events*. They occur according to their own schedule, not that of the programmer.

This means that as programmers, we can't organize our programs to do one thing first, and then another thing, and so forth, according to a given schedule. We have to react to events. Thus the robots are *reactive systems*: their environment presents them with events, according to its own timing, and the robots react.

Video games are also reactive systems: they have to react to the events that the player causes (and the player has to react also!)

Contrast this to programming a compiler: the compiler reads the program, parses it into individual pieces, translates the pieces into code, maybe optimizes – always in the same order.

GUI interfaces tend to be reactive systems: they present a user with a lot of options, which one the user chooses determines what the system does next.

A challenge for a reactive system comes when multiple events are happening at once, or at least, while other events are being processed.

Preliminary program: Going from here to there

This is a simple version of the Advanced Checkerboard Challenge, involving just one starting point and destination.

Watchers

Watchers are a useful tool to simplify a program. They let you check for something to happen while your robot is doing something else. For example, check that it crosses a line while it is moving and perhaps pushing an object or following a light.

Watchers

- Tasks that watch for something and change variables when it happens
- Examples: hitting an object, crossing a line, receiving a message

[1] See *The Hobbit*, by J. R. R. Tolkein

Tasks

In NQC, a task is a unit of a program that can run independently and concurrently with the rest of the program. The main routine is a task; subroutines are not tasks, since they must be called and executed to completion before the calling routine can continuer.
It is natural to implement a watcher as a task.

An algorithm

This section introduces as algorithm to go from here to there, where here and there are expressed as <x,y> coordinates.

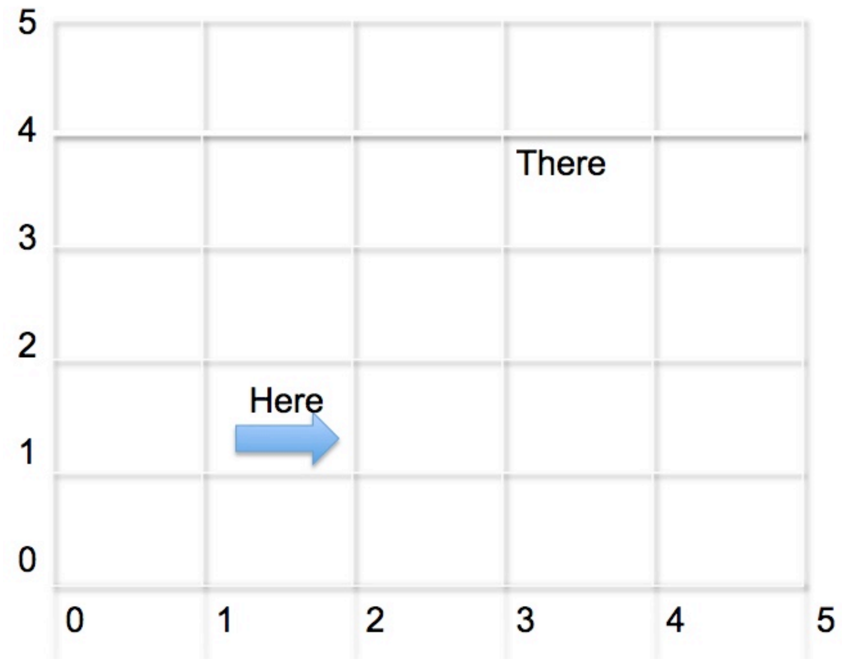
This is just one way to do it!!

```
Variables:
    here – an array with two entries (0 is x, 1 is y)
    there – an array with two entries (0 is x, 1 is y)
Line-watcher:
    Changes a here[i] when it sees a line for direction i has been crossed
Idea:
    let i be either 0 or 1 (x or y)
    change orientation to face from here[i] to there[i]
    start a line-crossing-watcher to change the value of here[i]
        parameters are i and +1 for positive direction or -1 for negative direction
    start motor
    while (here[i] != there[i]) {
        move forward
    }
    stop motor
    stop the watcher
    change i to 1-i (y or x)
    change orientation by 90 degrees
    change orientation to face from here[i] to there[i]
    start a line-crossing-watcher to change the value of here[j]
    while (here[j] != there[j] {
        move forward
    }
    stop the watcher
```

Since tasks don't have parameters, you will have to pass information in and out with global variables.
If you're not using rotation sensors, you will want extra watchers to correct the robot's direction after it hits a line
Some things we have to work out:

- How to decide what the orientation is
- The global variables to share with the tasks

Changing orientation



Cases:

Facing	Move in X direction from here to there	Move inY direction from here to there	Action
North		North	Go north until here[1]==there[1]
		South	Turn 180 degrees Go south until here[1]==there[1]
		None	Deal with x direction
South		North	Turn 180 degrees Go north until here[1]==there[1]
		South	Go south until here[1]==there[1]
		None	Deal with x direction (if necessary)
East	East		Go east until here[0]==there[0]
	West		Turn 180 degrees Go east until here[0]==there[0]
	None		Deal with y direction (if necessary)
West	East		Turn 180 degrees Go east until here[0]==there[0]
	West		Go east until here[0]==there[0]
	None		Deal with y direction (if necessary)

Algorithm:

Trying to keep the code as simple as possible!

```
if (here != there) {
    if (here[1]!=there[1] && robot direction != necessary move in y direction) {
        turn robot 180 degrees
    }
    set variables for line-crossing-watcher
        set i=1 // index to use when updating “here”
        set what the increment will be
    start line-crossing-watcher
    move until here[1] == there[1]
    stop line-crossing-watcher

    // similarly for the x direction
}
```

Exercises:

How do we decide what the needed direction is?

- North: here[1] < there[1]
- South: here[1] > there[1]
- East: here[0] < there[0]
- West: here[0] > there[0]

How do we decide what the increment is?

- North and East are +1
- South and West are -1

Could you use forward and reverse instead of turning 180 degrees? What would that require?