Simulation of Robotic Sensors in BYOB

Javier Arlegui[1], Michele Moro[2], Alfredo Pina[3]
[1]Psychology and Pedagogy Department, Public University of Navarra
Campus Arrosadía, Pamplona, Spain
arleguip@unavarra.es
[2]Department of Information Engineering, University of Padova
Via Gradenigo 6/A-B, Padova, Italy
mike@dei.unipd.it
[3]Mathematics and Computer Engineering Department, Public University of Navarra
Campus Arrosadía, Pamplona, Spain
pina@unavarra.es

***Abstract*—The paper presents a proposal to simulate several robotic sensors through an implementation in the BYOB authoring environment. The possibility to define custom blocks as specialized reporters is exploited to represent the information usually returned by relevant physical sensors in real robots. Some motivations to use simulated sensors and robots for educational purposes in a well know and not so complex environment like BYOB are also given.**

***Index Terms*—Scratch, BYOB, Robotic sensors, Simulation, Educational robotics**

I. Introduction

A reliable and rich sensorial endowment is crucial for an autonomous robot to realize effective behaviours. Thus the comprehension of the role, potentialities and limits of real robotic sensors is important under the educational point of view [1]. This knowledge can be initially promoted through easy-to-use simulation environments before working with real robots and facing all the uncertainties of a real scenario. Nonetheless younger students could find too hard to use sophisticated simulators able to reproduce 3D objects with all their physical parameters [2] [3] [4].

In [5] we showed that a widely spread authoring system like Scratch [6] provides 2D simulation features sufficient to permit a student to make significant robotic experiences. BYOB [7] goes further thanks to its powerful extensions, e.g. the possibility to define custom blocks, to use (recursive) procedures as data, to program in an object-oriented style. Both Scratch and BYOB have been used to make robotic simulations, for example Karel [8] and Valentino [9], and include commands to interact with external robotic components (Pico/Scratch boards, and LEGO WeDo natively but also other robots like LEGO NXT and Arduino-based architectures).

This paper aims at presenting a broad spectrum of possibilities to simulate robotic sensors without using external hardware with respect to the standard PC resources. Thus it presents a sequence of proposals, based on the realization of BYOB custom blocks, in order to simulate, among the others, the most significant sensors that in real robots support their autonomy. The focus of the paper is how to exploit the more advance features of the language to make simulated sensors able to report reasonable environmental data. This can help teachers in designing interesting 2D demonstrative robotic examples and in motivating their students to deepen some relevant scientific issues before working with real robots.

II. Some motivations

Sensors in robotics play a fundamental role, particularly when autonomy is concerned. Their variety, precision and complexity influence the control of the behaviour of a robot and make it more or less completely and effectively fulfil its tasks. Sensors are used both in finely controlling the robot's actuators and to permit it to take strategic decisions.

The simulation of a sensorized robot presents a fundamental difficulty: in a complete 3D environment objects have their physicality and the interaction between a sensor and the simulated reality must rely on that, i.e. an object occupies a certain volume within its surface and 'responds' following specific rules when subjected to some physical phenomenon like an ultrasonic emission. On a 2D simulation the situation is only a bit simpler: an object occupies an area of a plane within a close curve (its boundary) and any physical phenomenon takes essentially place planarly.

Scratch is widely adopted as an authoring system to give young students the possibility to experiment, in a pleasant and constructive way, several important aspects related to story telling, maths, geometry, computer graphics, computer programming. It promotes the knowledge of important but not so easy concepts in computer science like multitasking and message passing synchronization, hiding their most difficult details behind simple interfaces and procedures and exploiting the full potential of its hybrid (graphical/textual) programming style. BYOB reintroduces in Scratch first class (possibly recursive) procedures, lists and objects which were for decades fundamental elements of the previous LOGO available environments. These improvements make it possible to use BYOB as a powerful programming language to teach basic programming concepts like complex data structures, recursion, object-oriented programming and information hiding, etc.

BYOB, like Scratch, provides a set of sensing features associated with sprites which, when encapsulated into recognizable and simply interfaced functions, may be assimilated to several types of robotic sensors as discussed in the following section. Though most of these sensing functions are inherited from Scratch, for the sake of conciseness in this paper we refer only to the BYOB environment.

III. SENSORS IN BYOB

In the following we provide a uniform interface for sensing functions so that the robotic-oriented programming of sprites' scripts results enough simple and self explaining. This interface is based on the concept of 'port', an input connection between the robot and the sensor device, like what you can find, for example, in the LEGO NXT brick. In a configuration phase the robot is 'connected' through its ports, indexed with integer numbers, to one or more sensors which can be read calling a custom block, specific for each type of sensor. Therefore these blocks have always the connection port as one of their input parameters.

We assume that the reader has already a basic knowledge of BYOB in order to understand the proposed examples. The code is provided with a syntax which is a personal extension of the Scratch's Block Plugin Syntax [10]: the details of this extension will be presented gradually along with the examples.

A. Embedded sensors

Some state parameters are more or less directly returned by basic BYOB reporters, for example a sprite's position and direction. In these cases the realization of the equivalent of fundamental robotic sensors are straightforward and is presented as the first, simplest case.

The first example we propose is a **Compass** sensor, used to return the robot's orientation. In BYOB orientation is reported by the *direction* command and it is measured in degrees. The following correspondences with the cardinal points hold: S=±180°, W= -90°, N=0°, E=+90°. Here is the first code:

```
   def (Sensing reporter, for all, report[direction]) compass (port=1 Number)
     if < (item (port) of [ports v]) = [compass] > //check port
       report (direction)
     else
       report [Error!]
```

def(*desc*) represents the header of a custom block definition; *desc* describes the category, the type of the block and the reported value type when applicable. In this example the new *Sensing* block named *compass* takes one parameter, the port, a Number with 1 as its default value, to which the sensor should be connected, and reports the direction in degrees. Its implementation checks if the port is actually connected to the correct sensor: for this purpose let's imagine that, in a configuration phase, for each sensor you have orderly assigned its descriptive keyword to the *ports* list. In all our examples we assume a *config* custom block, local to each robot-sprite, initializing the *ports* list and other possible configuration parameters, like the following:

```
   def (Variables command, for this) config
     delete (all v) of [ports v]
     add [sensor1] to [ports v]
     add [sensor2] to [ports v]
   ...
     add [compass] to [ports v]
```

One of the interesting feature of BYOB, due to the fact that sprites are first class objects, is the way one sprite can ask another sprite to execute a script or a block: in this case even a custom block defined local to the called sprite (not global) can be executed and the execution can refer to the called sprite's local variables. One sprite can also ask another sprite to execute a command or a script concurrently. This permits the realization of a remote version of the *compass* reporter that could be imagined as one robot sending a request to another robot through a wireless connection to receive the latter's orientation. This modified version of the custom block, called *rcompass*, takes one further parameter, the name of the sprite whose orientation is requested (*myself* corresponds to the calling sprite). Its realization follows.

```
   def (Sensing reporter, for all, report[direction]) rcompass [sprite=myself Text] (port=1 Number)
     if < < (sprite) = [myself] > or < (sprite) = (attribute [name v]) > > // the calling sprite
       report (direction)
     else
       report (ask (object (sprite)) for {(rcompass (sprite) (port))} )
     end
```

Curly brackets indicate the special procedure of inserting a Reporter-type input parameter that BYOB provides to delay the evaluation of the reporter to the moment the parameter will be used by the called function, *ask* in our case. This procedure is illustrated as a 'grey border' in the BYOB documentation. *ask* is a library block that shortens the remote call of another sprite and it is defined as follows:

```
   def (Control reporter, for all, report[something]) ask (object) for (message) (args...)
     report call ((message) of (object)) [with input list] (args)
```

With these definitions, the two calls executed by *Sprite1*:

rcompass [myself] (1)

rcompass [Sprite2] (3)

report respectively the orientation of *Sprite1* and of *Sprite2*, provided the compass sensor is respectively 'connected' to port 1 for *Sprite1* and 3 for *Sprite2*.

The second example is the simulation of a **GPS** sensor: such a device usually returns the current absolute position of the robot. In BYOB a sprite's position is represented by a couple of Cartesian coordinates: a sprite can ask its own position calling separately the two reporters *x position* and *y position*. We define a new *gps* custom reporter which returns the two coordinates of the calling sprite in one single list of two elements, orderly *x* and *y*.

```
   def (Sensing reporter, for all, report[position list]) gps (port=1 Number)
     if < (item (port) of [ports v]) = [gps] > //check port
       report (list (x position) (y position))
     else
       report (list [Error!])
```

Also in this case you can easily define a remote version *rgps* with the additional sprite parameter. With this variant a *Sprite1* can smoothly reach the position of *Sprite2*, with a gps sensor on port 2, for example with the following piece of script code:

```
    set [gpsval v] to (rgps[Sprite2] (2))
    glide (1) secs to x: (item (1 v) of (gpsval)) y:
(item (2 v) of (gpsval))
```

If you define this sequence as a private custom block *reach* of *Sprite2*, *Sprite1* can force *Sprite2* to concurrently reach it with the command:

```
launch ([reach v] of [Sprite2 v])
```

Take notice that, in order to save space, in the following examples we will omit to include again the port check.

If in the definition of the *compass* block you substitute *direction* with the basic *loudness* reporter, you obtain a **Sound** sensor: the so defined *sound* custom block reports a sound level between 0 and 100 as measured on the PC sound input (the microphone or whatever selected as input source).

Instead of using the PC sound system, a sound function can also be simulated imagining the robot-sprite provided with a device able to measure a sound level and the scenario includes just one sound source represented by a sprite of known name, namely 'bell'. Imagining a punctual source, we are interesting in a theoretical 2D circular sound diffusion. The sound pressure level at distance *d* from the source is given in decibels by:

$Lp = Lw - 20 \cdot \log_{10} d - 11$

$Lw = 10 \cdot \log_{10} (W/W_0)$

where *Lw* is the acoustic (constant) power level of the source that is emitting the sound with power *W*, and W_0 is the minimum audible power, conventionally set at 10^{-12} W(att). Now assuming a certain value for *Lw* and evaluating the current distance from the robot-sprite and the source, you can realize a *mic* custom block reporting the virtually measured dBs.

```
  def (Sensing reporter, for all, report[sound level
dB]) mic (port=1 Number)
    report (((LSource)  - ((20)*([log v] of ( (distance
to [bell v]) / (Scale) ))) ) - (11))
```

LSource represents the source power and *Scale* is an accessory scale factor.

The next example is slightly more elaborated but it refers again to a basic sensing function, the touching predicate of one color with respect to another color. For this example imagine a rectangular robot provided with two colored small rectangular probes on its front, one red and one blue. Imagine also that on the stage some circular orange 'objects' of different radiuses are drawn (Fig. 1). Unfortunately neither Scratch nor BYOB allow to use a color code for this touching feature. Therefore we define a **Bumper** sensor for every type of probe, one for blue and one for red: in the block definition you must use the GUI to set the pertinent colors. Nonetheless we use color codes in our scripting language to describe the block implementation, as taken from the BYOB color palette.

```
  def (Sensing reporter, for all, report[pressed or
bumped]) bumperblue (port=1 Number) (type=pressed Text)
    if < color [#0042FF] is touching [#FF9500] ? >  //
blue touching orange
      if < (type) = [pressed] >
        set [bub_state v] to <true>
        report <true>
      end
      if <(bub_state)> // bumped not complete
        report <false>
      end
      set [bub_state v] to <true> // bump started
      report <false>
    end
    else
      if < (type) = [pressed] >
        set [bub_state v] to <false> // not pressed
        report <false>
      end
      if <<not (bub_state)>>  // no bump
        report <false>
      end
      set [bub_state v] to <false>  // bump complete
      report <true>
    end
```

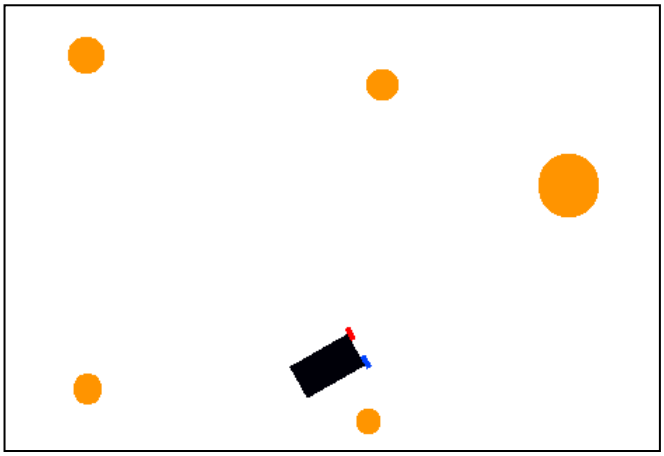

Fig. 1 Bumpers as colored probes

You can choose between two types of sensing: *pressed* and *bumped*. The second has memory and returns true when a transition between pressed and not pressed is sensed. A similar *bumperred* custom block can be defined substituting the touching color code with [#FF0000].

You can also define a similar *key* custom block substituting the first *if* instruction with:

if < key [k v] pressed? >

This block may be used to signal the robot with a user's action represented by typing the *k* key on the keyboard.

Here an example that makes the robot avoid the orange obstacles and turn when the *k* key is bumped:

```
when green flag clicked
  config
  forever
    if < (bumperred (1) [pressed]) >
      rtglide (-30) steps in (0.5) s
      turn left (60) degrees
    end
    if < (bumperblue (2) [pressed]) >
```

```
      rtglide (-30) steps in (0.5) s
      turn right (60) degrees
    end
    if < (key (3) [pressed]) >
      turn right (90) degrees
    end
    move (5) steps
    if on edge, bounce
  end
```

The *config* block sets port 1 connected to the red bumper, port 2 to the blue one, port 3 to the *k* 'key sensor', and initializes all the sensors' state variables. *rtglide* is a auxiliary custom block, a relative-motion alternative to the basic *glide* command: *rtglide* accepts a relative steps parameter instead of an absolute value. We propose also a *rvglide* variant accepting a speed parameter in place of the duration parameter. Their implementations follow.

```
  def (Motion command, for all) rtglide (dist=1 Number)
steps in (time=1 Number) secs
    glide (time) secs to x: ((x position) + ((dist) *
([cos v] of (90 - (direction))) ))
      y: ((y position) + ((dist) * ([sin v] of (90 -
(direction))) ))

  def (Motion command, for all) rvglide (dist=1 Number)
steps in (speed=1 Number) steps/secs
    glide ((dist) / (speed)) secs to x: ((x position) +
((dist) * ([cos v] of (90 - (direction))) ))
      y: ((y position) + ((dist) * ([sin v] of (90 -
(direction))) ))
```

Another demonstrative application is a black/white maze: the robot has a red probe on the left side and a blue prove on the front edge and the example applies the so called 'left-hand' algorithm establishing that the robot must continuously follow the left wall until it reaches the exit point (Fig. 2).

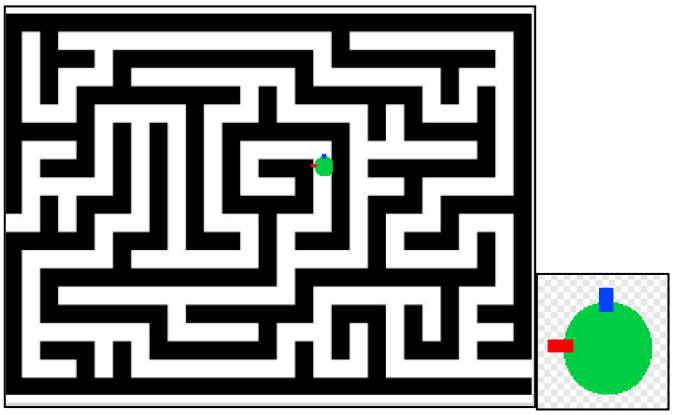

Fig. 2 a) The maze b) The 'robot'

```
forever
  if < (bumperblue (2) [pressed] >
    turn right (90) degrees
  else
    move (16.5) steps
    if < not (bumperred (1) [pressed]
      turn left (90) degrees
    end
  end
  if < (x position) > (224) >  // exit reached
    stop all
  end
end
```

B. *Light and color sensors*

When you want to simulate a sensor which cannot be directly associated with one of the PC devices or which is not directly simulated by the BYOB environment, a very simple solution is to represent the sensor value with a variable which is changed as effect of some user's actions (e.g. typing some keys on the PC keyboard), and to provide a general read custom block. We adopt this approach to simulate an **Ambient Light** sensor, returning a light level between 0 and a configurable maximum value (*maxlight*). When you type the up/down arrows keys, you modify the value from that moment returned by the *lightlev* custom reporter. This modification is performed by two specific scripts fired by the user's action, like an interrupt routine.

```
  def (Sensing reporter, for all, report[light level])
lightlev (port=1 Number)
    report (lightvar)

  when [up arrow v] key pressed
    if < (lightvar) < (maxlight) >
      change [lightvar v] by (1)
    end
  when [down arrow v] key pressed
    if < (lightvar) > [1] >
      change [lightvar v] by (-1)
    end
```

In the next example we provide the robot of the equivalent of a **Light** (grey level) sensor reporting a level in the range 0..100. This sensor can be used for example to recognize objects or markers on the ground or to realize a line follower. Imagine to equip your robot somewhere with a colored probe, for example a small red rectangle on the front edge, and to put some other sprites on the stage, each having a uniform color, that is with a specific, known grey level. Such a level must be initially set for every sprite, included the Stage, who has a uniform color, into a local *lightlev* variable. The *light* custom block reports the grey level of the sprite which is touched by the colored probe.

```
  def (Sensing reporter, for all, report[light level])
light (port=1 Number)
    script variables (i) (sp)
    set [i v] to [1]
    set [sp v] to (object [allSprites])  // list of
sprites
    repeat (length of (sp))
      if < < not <(item (i) of (sp)) = (object
[myself]) > > and < ask (item (i) of (sp)) for {(touching
color [#FF0000] ?)}  > >  // the calling sprite excluded
        report (ask (item (i) of (sp)) for {(lightlev)}
)
      end
      change [i v] by [1]
    end
    report (ask (object (Stage)) for {(lightlev)})  //
report the stage light level
```

An almost identical approach can be adopted to define a **Color** sensor: this time the *color* custom block reports the value of the touched sprite's *colorcode* variable that keeps the initially assigned color code string of the sprite.

C. Touch and Bump sensors

This kind of sensors, realized in a simplified version in section A, can be generalized exploiting the same enumeration of sprites done in the realization of the *light* custom block. Now imagine to put one green probe and one orange probe on different position of the border of the robot-sprite. We propose three custom blocks, *touch*, *bumpergreen*, *bumperorange* whose meaning is similar to the blocks of section A.

The **Touch** sensor, represented by the *touch* custom block, exploits another one of the touching basic sensing reporters: it returns true if the robot's border touches another sprite's border.

```
   def (Sensing reporter, for all, report[touching
condition]) touch (port=1 Number)
     script variables (i) (sp)
     set [i v] to [1]
     set [sp v] to (object [allSprites])
     repeat (length of (sp))
       if < < not <(item (i) of (sp)) = (object
[myself]) > > and < touching (item (i) of (sp)) ? > >
         report <true>
       end
       change [i v] by [1]
     end
     report <false>
```

bumpergreen and *bumperorange* can be realized with the structure of the preceding *bumperblue* and *bumperred* blocks but enumerating all the other sprites like in *touch* above, and storing in a state variable the touched sprite as an object instead of a simple Boolean value (the robot-sprite itself if not touching):

```
   def (Sensing reporter, for all, report[pressed or
bumped]) bumpergreen (port=1 Number) (type=pressed Text)
     script variables (i) (sp)
     set [i v] to [1]
     set [sp v] to (object [allSprites])
     repeat (length of (sp))
       if < < not <(item (i) of (sp)) = (object
[myself]) > > and < ask (item (i) of (sp)) for {(touching
color [#00FF52] ?)} > > // light green probe
   // change this color code to #FF9400 for bumperorange
         if < (type) = [pressed] >
           set [bug_state v] to (item (i) of (sp))
           report <true>
   . . .
       change [i v] by [1]
     end
     if < (type) = [pressed] >
       set [bug_state v] to (object [myself]) // not
pressed
       report <false>
   . . .
```

D. Proximity and sonar sensors

So far we have taken advantage of several basic sensing reporters referring to sprites' position, orientation, color and, in one case (*touching <sprite>*), partly taking into account the actual border of the sprites' costumes. Unfortunately the 'physicality' of a sprite cannot be completely sensed by another sprite without some pre-knowledge of its shape and dimensions. Thus for a sake of simplicity, we assume that we have some 'obstacles' on the stage represented by sprites for which we know the minimum radius of a circle completely covering their costume: for this you can refer to the maximum of the two dimensions of the picture representing the costume. This radius is stored during a configuration phase in the local variable *orad*. We assume also that, for our sensing purposes, the border of the actual covering circle is also the border virtually limiting the obstacle: we will measure the distance robot-sprite/obstacle with respect to this virtual border. With these assumptions we propose the following *prox* custom block:

```
   def (Sensing reporter, for all, report[minimum
distance]) prox (port=1 Number)
     script variables (i) (sp) (min) (dist)
     set [i v] to [1]
     set [sp v] to (object [allSprites]) // list of
sprites
     set [min v] to [1000]
     repeat (length of (sp))
       if < < not <(item (i) of (sp)) = (object
[myself]) >
         set [dist v] (((distance to (item (i) of (sp)))
- (((ask (item (i) of (sp)) for {(orad)}) * (ask (item
(i) of (sp)) for {(attribute [size v])})) / 100)) -
(sensoff))
         if < (dist) < (min) >
           set [min v] to (dist) // update minimum
         end
       end
       change [i v] by [1]
     end
     report (min) // return the minimum
```

The loop is repeated for every sprite and it evaluates the distance between the centres of the robot-sprite and the current sprite, minus the 'radius' of the sprite, for taking into account the area it occupies, and the relative distance of the simulated sensor on the robot-sprite with respect to its centre, kept in the *sensoroff* configuration variable. The minimum among these distances is regularly updated and finally returned by the reporter.

Consider now the following auxiliary custom block *vdir*:

```
   def (Operators reporter, for all, report[vector
direction]) vdir (x Number) (y Number)
     if < (y) > [0] > // upper half of the plane
       report ([asin v] of ((x) / ([sqrt v] of ( ((x) *
(x)) + ((y) * (y)) ) )) )
     else
       if < (x) < [0] > // left lower quadrant of the
plane
         report ( [0] - ([acos v] of ((y) / ([sqrt v] of
( ((x) * (x)) + ((y) * (y)) ) ))) )
       else // right lower quadrant of the plane
         report ([acos v] of ((y) / ([sqrt v] of ( ((x)
* (x)) + ((y) * (y)) ) )) )
       end
     end
```

This block returns the orientation of a (directed) vector whose components are the two parameters *x* and *y*.

Now, starting from the implementation of the *prox* block, if you select only the objects that are positioned within ±*semiview* degrees with respect to the robot-sprite axis (the one defining its direction), you obtain the value that could be returned by a **Sonar** sensor, oriented in the same direction of the sprite: in fact this type of sensor presents a limited angle of view like the abovementioned one. To implement this selection, it suffices to verify that the absolute value of the difference between the direction of the vector connecting the robot-sprite's and the obstacle's centres, and the direction of the robot-sprite itself is less than (*semiview*+1).

```
   def (Sensing reporter, for all, report[minimum
distance]) sonar (port=1 Number)
     script variables (i) (sp) (min) (dist) (dirr)
   . . .
       if < < not <(item (i) of (sp)) = (object
[myself]) >
         set [dirr v] to (vdir ( ([x position v] of
(item (i) of (sp))) - (x position)) ( ([y position v] of
(item (i) of (sp))) - (y position)) ) - (direction))
         if < ([abs v] of (dirr)) < ( (semiview) + [1])
>
           set [dist v] . . .
   . . .
```

To show the use of this simulation of the so important contactless distance sensor, we briefly present an emulation of the ultrasonic system that a bat uses to identify the position of a possible prey. This emulation has been physically realized with an NXT robot and illustrated in [11]: in this realization we assumed to know the distance *a* between the two ultrasonic sensors that represent the bat's ears, alternatively used to measure the distance from each one of them and the 'prey'. It is rather simple to calculate the (signed) distance *x* of the prey with respect to the axis orthogonal to the segment joining the two sensors (considered as punctual sources). Say *d1* and *d2* the respective distances measured by the two sensors (Fig. 3) it holds:

$x = (d1^2 - d2^2) / (2 \cdot a)$

from which the distance of the prey with respect to the line joining the two sensors may be calculated as:

$y = \text{sqrt}\ (d2^2 - (x-a/2)^2)$

We used these relations to move the bat towards its prey, re-evaluating in a loop the two relative coordinates above.

```
   forever
     set [d1 v] to (ask (object [ear1]) for {(sonar
(1))} )
     set [d2 v] to (ask (object [ear2]) for {(sonar
(1))} )
     set [x v] to ((((d1)*(d1)) - ((d2)*(d2))) /
((2)*(a)))
     set [y v] to ([sqrt v] of ( ((d2)*(d2)) - (((x) -
((a) / (2))) * ((x) - ((a) / (2))))) )
     if < (((x)*(x))+((y)*(y))) < (threshold) >
       stop all
     end
     glide (0.5) secs to x: ((xposition) + ((x) / (3)))
y: ((yposition) + ((y) / (3)))
   end
```

ear1 and *ear2* are the names of two little sprites, representing the bat's ears, 'anchored' to the bat at a distance *a* one another. *threshold* is a suitable value equal to the square of the minimum distance bat-prey which has to be reached to stop the hunting.

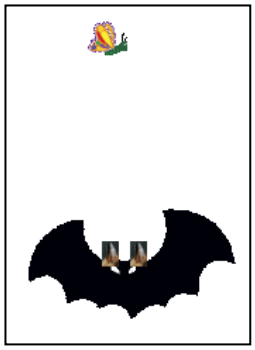

Fig. 3 The bat

E. A range scanner

This type of sensor is very powerful: it can report direction and distance of all the objects not 'hidden' by other objects in the whole arc of 360°. We can simulate this **Scanner** sensor combining the techniques presented so far. The *scan* custom block reports a list of two elements which are in turn two lists: the first one is the sequence of directions of the other sprites, the second one the sequence of their distances measured similarly as in the *prox* block. Directions and distances are evaluated with respect to the current position and direction of the robot-sprite. For simplicity, our simulation does not check whether one sprite could hide another sprite: *scan* reports measures for all the sprites different from the robot.

```
   def (Sensing reporter, for all, report[the list of
the lists of directions and distances]) scan (port=1
Number)
     script variables (i) (sp) (min) (dist) (dirr)
(objDir) (objDist)
     set [i v] to [1]
     set [sp v] to (object [allSprites])  // list of
sprites
     set [objDir v] to (list ()) // init with empty list
     set [objDist v] to (list ())
     repeat (length of (sp))
       if < < not <(item (i) of (sp)) = (object
[myself]) >
         set [dirr v] to (vdir ( ([x position v] of
(item (i) of (sp))) - (x position)) ( ([y position v] of
(item (i) of (sp))) - (y position)) ) - (direction))
         set [dist v] (((distance to (item (i) of (sp)))
- (((ask (item (i) of (sp)) for {(orad)}) * (ask (item
(i) of (sp)) for {(attribute [size v])})) / 100)) -
(sensoff))
         add (dirr) to (objDir)
         add (dist) to (objDist)
       end
       change [i v] by [1]
     end
     report (list (objDir) (objDist))
```

A demo program, which moves the robot subsequently towards the border of the various sprites around it, is the following (Fig. 4):

```
when green flag clicked
broadcast [config v] and wait
config
script variables (i) (ris)
point in direction [10 v]
set [ris v] to (scan (1))
set [i v] to (1)
repeat (length of (item (1 v) of (ris)))
  turn right (item (i) of (item (1 v) of (ris))) degrees
  rtglide (item (i) of (item (2 v) of (ris))) steps in (1) secs
  wait (1) secs
  rtglide ((0) - (item (i) of (item (2 v) of (ris)))) steps in (1) secs
  turn left (item (i) of (item (1 v) of (ris))) degrees
  change [i v] by [1]
end
```

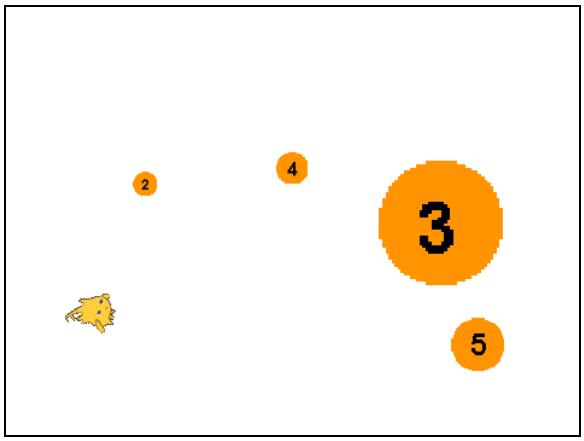

Fig. 4 The scan sensor

The *config* script is locally defined in every sprite to set the *orad* variable (see *prox* block above), apart from the robot-sprite which initialize this variable in its *config* block.

F. Acceleration and gyroscopic sensors

In this last example we simulate an **Accelerometer**, assuming for simplicity that the motion is on a horizontal straight line. Consider the following code:

```
when green flag clicked
config
go to x: (0) y: (0)
reset timer
broadcast [gosensor v]
forever
  go to x: ((200) * ([sin v] of ((angspeed) * (timer)))) y: (0)
  if < ([abs v] of (acc (1))) > [18] >
    change [color v] effect by (2)
  end
end
```

This code moves the sprite along a horizontal segment with a sinusoidal offset with respect to its medium (0, 0) point. The *angspeed* (angular speed, also called angular frequency) parameter is set in the *config* custom block to a reasonable value (for example 360/20 = 18 degrees/s, 20 s being the period). In the fragments when the evaluated acceleration is greater than 17, the sprite's costume changes color. These values are compatible with the following theoretical formulas:

$$x(t) = A \cdot \sin_r(\omega_r \cdot t) = A \cdot \sin_g(\omega_g \cdot t)$$
$$v(t) = dx/dt = A \cdot \omega_r \cdot \cos_r(\omega_r \cdot t) =$$
$$= A \cdot \omega_g \cdot (\pi/180) \cdot \cos_g(\omega_g \cdot t)$$
$$a(t) = dv/dt = -A \cdot \omega_r^2 \cdot \sin_r(\omega_r \cdot t) =$$
$$= -A \cdot (\omega_g \cdot \pi/180)^2 \cdot \sin_g(\omega_g \cdot t)$$

being ω_r and ω_g the angular frequency respectively in radians and degrees and sin/cos$_r$ and sin/cos$_g$ the sinusoidal functions with the parameter again in radians and degrees. Thus the maximum acceleration in absolute value is reached at the two extremes of the motion and it is given by:

$$a_{max} = A \cdot (\omega_g \cdot \pi/180)^2 = 200 \cdot (18 \cdot \pi/180)^2 = 2 \cdot \pi^2 \cong 19.7.$$

Now we imagine that the sensor is mounted over the robot-sprite so that it can measure the component of the acceleration parallel to the sprite's motion orientation. This example shows how to simulate a sensor through a separate concurrent thread that updates a common variable (*acc*) which can be reported to the main thread by the *acc* custom block. To preserve a sufficient precision the updating thread executes periodically on the basis of the internal timer, first calculating the speed *v* as the ratio $\Delta offset/\Delta t$ and then the ratio $\Delta v/\Delta t$.

```
when I receive [gosensor v]
script variables (pos) (newpos) (t) (newt) (vel) (newvel) (next)
set [pos v] to (x position) // starting position
set [vel v] to (0)  // starting speed
set [t v] to (timer)  // starting time
set [next v] to ((timer) + (0.5)) // next period
forever
  wait ((next)-(timer)) secs // wait next period
  set [newt v] to (timer)  // ending time
  set [newpos v] to (x position)  // ending position
  set [newvel v] to ( ((newpos) - (pos)) / ((newt) - (t)) )
  set [acc v] to ( ((newvel) - (vel)) / ((newt) - (t)) )
  set [t v] to (newt)  // update time
  set [pos v] to (newpos) // update position
  set [vel v] to (newvel) // update speed
 change [next v] by (0.5) // update next period
end
```

With a substantially similar approach we can also simulate a **Gyroscope**, a sensor that measures angular speed with respect to one of its axis. In our simulation we imagine that the sensor is mounted over the robot-sprite centre and therefore it must measure the turning speed of the sprite: that means that in the simulation you must read the sprite's direction through the homonymous reporter.

Obviously, due to scheduling of threads, angle resolution and other inaccuracies, the simulations in this sections have more a qualitative value than a quantitative one.

IV. Conclusions

The paper aspires to prove that a known and powerful environment like BYOB, through the realization of suited custom blocks and, in some cases, of supporting service scripts, can include several fundamental robotic sensors provided of a homogeneous interface, becoming a rather complete 2D robotic simulator. With little modifications to the adopted approach and code you could also easily add some uncertainty to the sensors' model in order to better reproduce a real environment.

Though testing in class will be conducted in future, we would emphasize that a student coming from previous experiences with Scratch, and possibly BYOB, can be smoothly leaded to challenging robotic experiences which anticipate the following work with real robots in real environments. We are convinced that this learning progression constitutes a valuable tool for promoting a deep consciousness of important facts regarding perception, algorithms, control theory, programming and technical aspects of robotics with a pleasant and rewarding approach. Moreover, this approach does not prevent from successively making students to work with real robots: Scratch already provides a general interface to connect its environment to external devices. For example this interface has been adapted in *Scratch for Arduino* (S4A) [12] to control an Arduino board. A little effort would be required to adapt the API here proposed to the S4A commands in order to support external real sensors.

V. References

[1] R. Siegwart, and R. Nourbakhsh, *Introduction to Autonomous Mobile Robots*, Cambridge, MA: MIT Press, 2004.

[2] J. Åkesson, U. Nordström, and H. Elmqvist, "Dymola and Modelica_EmbeddedSystems in Teaching – Experiences from a Project Course", in *Proc. 7th Modelica Conference*, Como, Italy, Sep. 20-22, 2009, pp. 603-611.

[3] D. M. Lofaro, T. T. Giang Le, P. Oh, "Mechatronics Education: From Paper Design to Product Prototype Using LEGO NXT Parts", in *Progress in Robotics, Communications in Computer and Information Science*, 2009, Volume 44, Part 3, 232-239, DOI: 10.1007/978-3-642-03986-7_27.

[4] (2012) The NXT-ROS wiki. [Online] Available: http://www.ros.org/wiki/nxt

[5] J. Arlegui, M. Moro, and A. Pina, "How to enhance the robotic experience with Scratch", accepted for: *Contructionism 2012: Theory, Practice and Impact Int. Conf.*, Athens, Greece, 2012.

[6] (2012) Scratch website. [Online] Available: http://scratch.mit.edu

[7] (2012) BYOB website. [Online] Available: http://byob.berkeley.edu

[8] R. E. Pattis, J. Roberts, and M. Stehlik, *Karel the Robot: A Gentle Introduction to the Art of Programming*, 2nd ed., John Wiley and Sons, Inc, New York, NY, 1995.

[9] M. Ben-Ari (2012) Software and Learning Materials for Computer Science Education. [Online] Available: http://www.weizmann.ac.il/sci-tea/benari/home/software.html

[10] (2012) Scratch's Block Plugin Syntax. [Online] Available: http://scratch.mit.edu/wiki/Block_Plugin/Syntax

[11] M. Moro, E. Menegatti, F. Sella, M. Perona, *Imparare con la robotica: Applicazioni di problem solving*, Ed. Centro Studi Erickson, Trento, Italy, 2012 (in Italian).

[12] (2012) Scratch for Arduino (S4A) website. [Online] http://seaside.citilab.eu/scratch/arduino