Scratch Simulations of the Thymio Robot

Moti Ben-Ari

Version 1.1 for

Aseba 1.4

Scratch 2.0

© 2015 by Moti Ben-Ari.

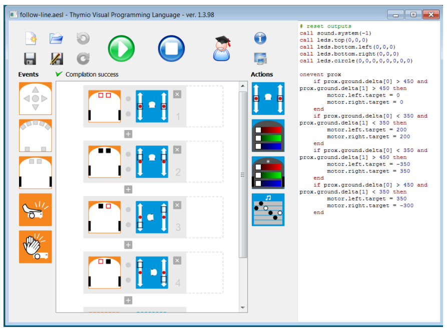
This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Contents

Preface

VPL and Scratch

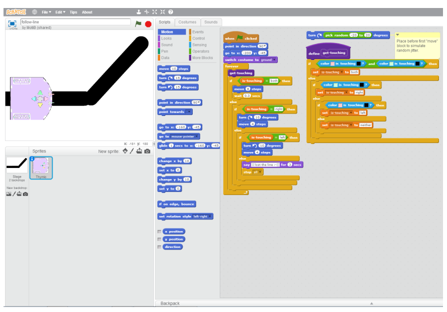
VPL is a visual programming environment for the Thymio robot (Figure 1(a)). Scratch (Figure 1(b)) is a web-based visual programming environment that animates *sprites* on the computer screen. The two environments have similar features: a program is created by dragging-and-dropping blocks onto the screen, and the fundamental programming construct is the *event handler*. Knowledge of one environment can be helpful in learning a second environment: we take a program written in the first environment and show how it can be written in the second. This is based on the educational theory called *mediated transfer*; see the Wikipedia article on Transfer of learning and the references by Salomon and Perkins cited there.


follow-line.aesl - Thymio Visual Programming Language - ver. 1.3.98
Events
Compilation success
Actions

```
# reset outputs
call sound.system(-1)
call leds.top(0,0,0)
call leds.bottom.left(0,0,0)
call leds.bottom.right(0,0,0)
call leds.circle(0,0,0,0,0,0,0,0)

onevent prox
    if prox.ground.delta[0] > 450 and
prox.ground.delta[1] > 450 then
        motor.left.target = 0
        motor.right.target = 0
    end
    if prox.ground.delta[0] < 350 and
prox.ground.delta[1] < 350 then
        motor.left.target = 200
        motor.right.target = 200
    end
    if prox.ground.delta[0] < 350 and
prox.ground.delta[1] > 450 then
        motor.left.target = -350
        motor.right.target = 350
    end
    if prox.ground.delta[0] > 450 and
prox.ground.delta[1] < 350 then
        motor.left.target = 350
        motor.right.target = -300
    end
```

(a) VPL



(b) Scratch

Figure 1: VPL and Scratch

Chapters 1–5 are about transfer from VPL to Scratch. We take programs from the VPL tutorial *First Steps in Robotics with the Thymio-II Robot and the Aseba/VPL Environment* (https://aseba.wikidot.com/en:thymioprogram) and show how to implement them as Scratch programs controlling a sprite that is an image of the Thymio robot.

Chapter 6 is intended for students with a good knowledge of Scratch who are learning VPL. The projects from Chapter 12 of the VPL tutorial on Braitenberg creatures have been implemented in Scratch.

References

The document is not intended as a tutorial on Scratch, but rather as a collection of projects that can be used when learning Scratch. For an introduction to Scratch, I recommend *Computer Science Concepts in Scratch* by Michal Armoni and Moti Ben-Ari, which can be downloaded for free at http://stwww.weizmann.ac.il/g-cs/scratch/scratch_en.html.

The projects are arranged in increasing complexity of the Scratch implementation, not in the order they appear in the VPL tutorial.

The Scratch projects described here can be found in my Thymio studio on the Scratch website (http://scratch.mit.edu/studios/1023692), except for the projects on the Braintenberg creatures which are in a separate studtio (https://scratch.mit.edu/studios/1452106).

For other robotics-related Scratch projects, see my website or my Robotics studio (http://scratch.mit.edu/studios/520857).

On the implementation

Aside from the obvious difference between the concrete Thymio robot and its image on the Scratch stage, the main difference between the robotic projects and the Scratch projects is how the sensors are implemented in Scratch. Full details of the implementation are given in Appendix A, but it is not necessary to understand them, since a set of abstractions has been introduced and they will be described as they are introduced in the projects.

- A sprite called `Pointer` is used to sense where the mouse is clicked. It broadcasts the messages `center`, `front`, `back`, `left`, `right` to simulate touching the buttons.
- The new block **get-pointer-direction** returns in the variable **direction-to-pointer** the direction from the `Thymio` sprite to the point where the mouse was clicked.
- The new block **get-touching** returns in the variable **is-touching** an indication if the `left` or `right` ground sensors are touching a black tape, or if `both` sensors or `neither` sensor are touching the tape.

The following costumes are used:

- The `Thymio` sprite is colored violet to make it stand out on the stage. You can change this color if you like. The buttons are colored and these colors should not be changed.
- Five costumes (`blank`, `red`, `green`, `blue`, `yellow`) simulate the top lights.
- The costume `ground` simulates the ground sensors.

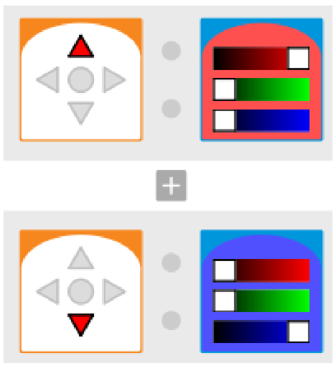
Chapter 1

Programming with Events

Events

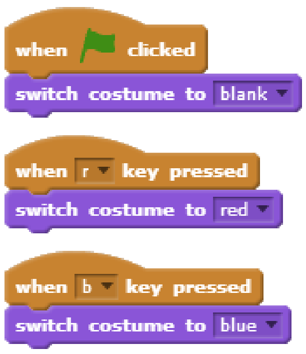
Scratch project: **tap-on-off**

In VPL an instruction consists of an event block followed by one or more action blocks:



Each time the event occurs, the actions associated with the event are performed. Here, when the front button of the robot is touched, the top light is turned on with color red, and when the back button is touched, the light becomes blue.

In Scratch, the construction equivalent to an event-actions pair is the *script*:



```
when 🏁 clicked
switch costume to blank ▾

when r ▾ key pressed
switch costume to red ▾

when b ▾ key pressed
switch costume to blue ▾
```

The image shows three scripts, each of which changes the costume of the sprite:

- Scratch programs are started by clicking on the *green flag* above the stage. When the green flag is clicked, the costume of the sprite is set to the `blank` costume, which displays the Thymio with the top lights off.

- The second script responds to the event of pressing the **r** key. The costume is changed to the `red` costume which displays the top lights in red.
- Similarly, the third script turns the lights blue when the **b** key is pressed.

The **Events** palette contains the event blocks which are colored brown. Most of these blocks have a "hat" shape to indicate that they can only be the first block in a script. After an initial event block, other blocks can be added to the script.

Costumes

A sprite can have one or more *costumes* which specify how the sprite is displayed on the stage. The sprite's costumes are displayed when you click the **Costumes** tab. You can create costumes by importing images, or by drawing or editing them using a paint program that is included in Scratch.

The costumes for this project
The project **thymio-costumes** contains all the costumes used in these projects. Open this project and copy (drag and drop) the costumes you need to your **Backpack**. Now open a new project and copy the costumes to the sprites.

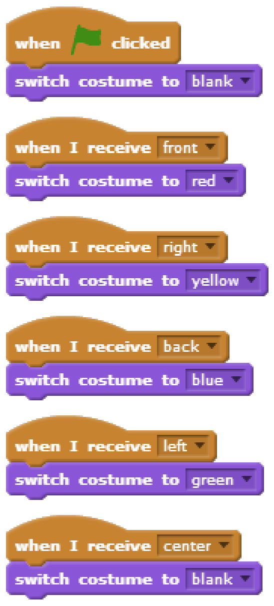
Setting the size of the costumes
You may have to adjust the size of the image of the sprites in order for a project to work. You can do this by shrinking a costume in the paint program: click on the icon and then on the image. A better solution is to use the block set size to 40 % from the **Looks** palette in the initialization of a program; experiment with different values until you have the size the need.

Sending and receiving messages

Scratch project: **colors**

Let us look now at the program **colors** which changes the color of the top lights according to which button is touched (Figure 1.1). In Scratch, touching a button is simulated by clicking on the image of a button. The click is interpreted by a second sprite called `Pointer`, which sends a message to the `Thymio` sprite, depending on which image is clicked: `center`, `front`, `back`, `left`, `right`.

Click on the `Pointer` sprite in the sprite area in the lower left of the Scratch window. Without going into detail, just notice that the second script uses the block broadcast front and wait, which causes a message (here, `front`) to be sent. Now, click back on the `Thymio` sprite and you will see the scripts in Figure 1.1. Scripts starting with



```
when [flag] clicked
switch costume to blank

when I receive front
switch costume to red

when I receive right
switch costume to yellow

when I receive back
switch costume to blue

when I receive left
switch costume to green

when I receive center
switch costume to blank
```

Figure 1.1: Changing colors using messages

the event block `when I receive front` are performed when the corresponding message is received. We see that different messages cause the costume of the `Thymio` to be changed to ones displaying different colors.

Sounds

Scratch project: **bells**

The VPL project **bells** caused notes to be played when the buttons are touched. In Scratch, the **Sound** palette contains a rich set of blocks for playing notes. Experiment with these blocks and then replace the blocks that change costumes in the above project with blocks that play different sounds when different buttons are clicked. You can also add sound blocks to the script in Figure 1.1, so that the sprite both changes colors and plays sounds.

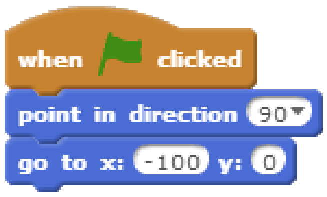
Chapter 2

Moving Sprites

Scratch project: **moving**

In the project **moving**, the robot moves forwards and backwards when the front and back buttons are touched, and it stops when the center button is touched. In Scratch, a sprite will move on the screen in response to running blocks from the **Motion** palette.

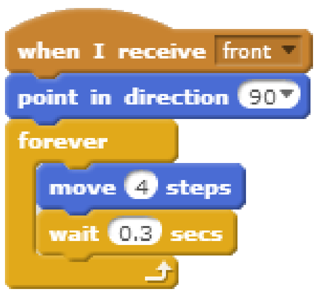
The following script initializes the sprite to start at the left side of the stage, pointing right (90°):



```
when [flag] clicked
point in direction 90
go to x: -100 y: 0
```

The size of the stage is from -240 to 240 pixels (picture elements) in the x (horizontal) direction and -180 to 180 pixels in the y (vertical) direction. You can explore these values by moving your mouse around the stage; its position is displayed below the lower right corner of the stage. The initial position of the `Thymio` sprite is set to $(-100, 0)$, so that it is at the left edge of the stage and centered vertically.

The sprite runs the following script when it receives the message `front` that is broadcast when the front button is clicked:

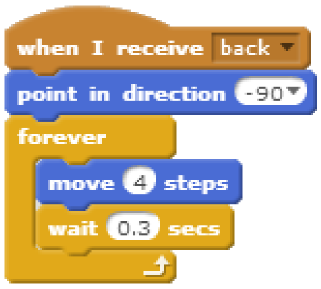


```
when I receive front
point in direction 90
forever
    move 4 steps
    wait 0.3 secs
```

The block `move 4 steps` is contained within a `forever` block that causes the instruction to be run repeatedly. This will cause the sprite to move steadily in the direction it is pointing.

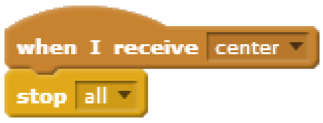
The `wait 0.3 secs` block causes the program to wait 0.3 seconds before continuing with the next block. This slows down the movement and makes it easier to click on a button.

The script for responding to the `back` message is the same as that for the front message, except that the sprite is made to point left ($-90°$):



```
when I receive back
point in direction -90
forever
    move 4 steps
    wait 0.3 secs
```

When the `center` message is received, all the scripts in the program are stopped:



```
when I receive center
stop all
```

Warning!

Turning the sprite to move in the direction $-90°$ will also cause the *image* of the sprite to turn 180°. To prevent this, select the `Thymio` sprite icon in the area below the stage and click on the small **i** in the icon. Then click on the dot which is one of the options in **rotation style**.

Directions

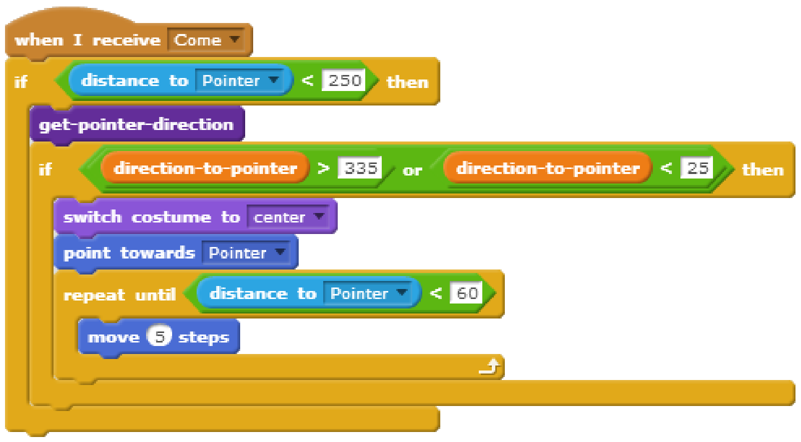
Scratch project: **obeys**

The `Thymio` can behave like a pet, following you around. In project **obeys**, click the mouse closer and closer to the center sensor on the image of the `Thymio` sprite. When it is close enough, the red light on the sensor turns on and the robot moves towards the pointer, stopping when it is close.

Initialize the project as in **moving** adding the block `switch costume to blank`. The main script for this project is shown in Figure 2.1. The script starts running when it receives the message `Come`, which is sent by the `Pointer` sprite when the mouse is clicked.

Distance to a sprite

The robot moves only when it detects that the pointer is close. The sprite `Pointer` tracks the mouse pointer. The block `distance to Pointer` returns the distance in pixels from the sprite that runs it—here the `Thymio` sprite—to the `Pointer` sprite. The block can be found in the **Sensing** palette.



```
when I receive Come
if distance to Pointer < 250 then
    get-pointer-direction
    if direction-to-pointer > 335 or direction-to-pointer < 25 then
        switch costume to center
        point towards Pointer
        repeat until distance to Pointer < 60
            move 5 steps
```

Figure 2.1: Move towards an object near the center sensor

Next, we compare this distance with the distance 250 that we decide is "close" enough to be detected. In the palette **Operators** you can find mathematical operators for performing comparisons: `distance to Pointer < 250`.

The block with comparison operator is a *condition block*. The `if` block `if then` has an angular area for a condition block and a "mouth" in which you can place other blocks. The meaning of the `if`-block is:

```
if the condition is true then
    the blocks contained in the "mouth" are run.
```

It follows that the rest of the blocks in the script in Figure 2.1 will be run only when the `Pointer` sprite is close to the `Thymio` sprite.

Direction to a sprite

The specification requires that the Thymio respond only if the pointer is detected in front of the center sensor. This is implemented by calling the block `get-pointer-direction` which returns in `direction-to-pointer` the direction of the `Pointer` from the front of the `Thymio` sprite. Its value is in degrees in the range 0 to 360, clockwise; that is, directions just to the right of the `Thymio` will have low values, while directions just to the left of the `Thymio` will have values close to 360. We decide that the `Pointer` is detected by the center sensor if the direction is greater than 325 or less than 25.

☆ **User-defined blocks**

`get-pointer-direction` and `direction-to-pointer` are not predefined in Scratch. The implementation of these blocks is described in Appendix A, but you can use the blocks without understanding the implementation.

A second if-block is used to run blocks only when the direction to the Pointer is greater than 325 or less than 25. The **Operators** palette contains the operator or which enables us to combine the two conditions into one. The result is a *compound condition* that is true if either of its parts is true.

If the compound condition is true, we turn the Thymio to face the Pointer sprite using the block point towards Pointer from the **Motions** palette, and we change the costume so that the red light next to the center sensor is turned on.

Approaching a sprite

If the Thymio sprite is close to the Pointer sprite and is pointed in the right direction, the robot must approach the sprite. We use repeat until which is similar to the forever block in that it causes the blocks contained in its "mouth" to be run again and again, but it is different in that it has a condition like an if-block. The movement of the Thymio sprite must stop if the distance to the Pointer sprite is too small and this is checked by the condition block distance to Pointer < 60 .

Chapter 3

Defining New Blocks

Scratch project: **likes**

The project **likes** is similar to **obeys** except that the `Thymio` sprite turns towards the object when one is detected by the left or right sensors and not just by the center sensor:

```
if center sensor faces Pointer
    switch costume to center
    go to Pointer
else
  if right sensor faces Pointer
      switch costume to right
      go to Pointer
  else
    if left sensor faces Pointer
      switch costume to left
      go to Pointer
```

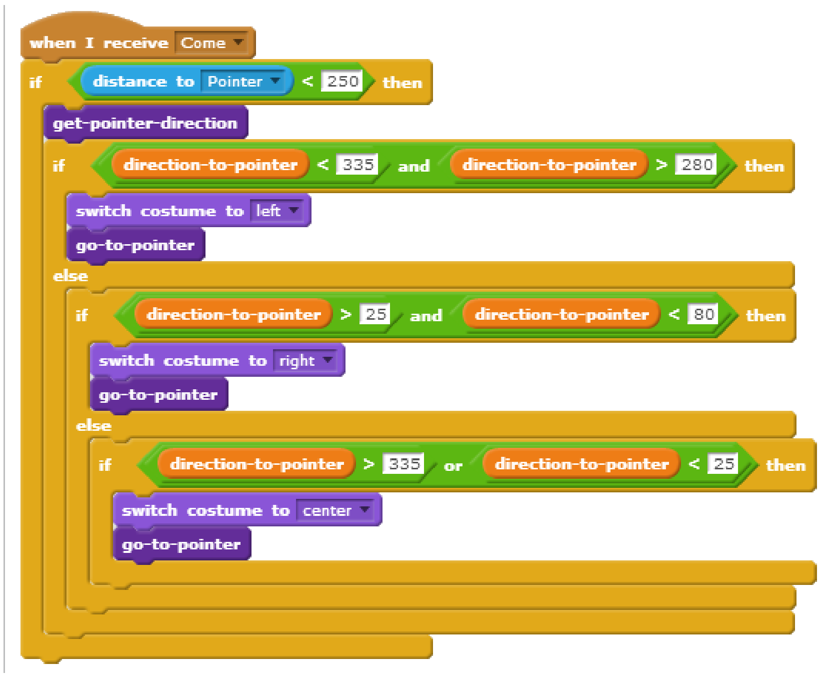
In Scratch, we can define new blocks such as go-to-pointer . Once the block is defined, we can use it to write the script for **likes** (Figure 3.1).

How do we define the new block? Go to the palette **More Blocks** and click on Make a Block . Enter the name of the new block, `go-to-pointer`, in the purple field in the `New block` window that is opened. This will create the block define go-to-pointer block in the script area and the go-to-pointer in the block area for this palette. Now you can drag-and-drop the blocks required to implement the new block (Figure 3.2).

Trick

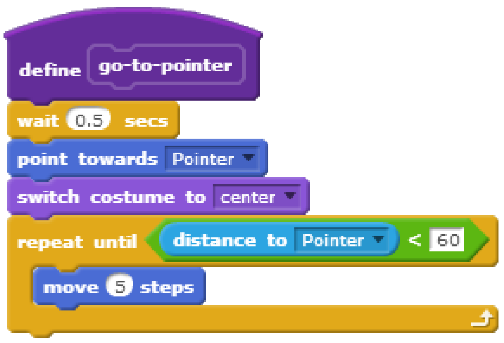
When defining a new block, click on **Options** and **Run without screen refresh**. This ensures that the new block is run as one action and the user does not see the result of running each block in the definition.

The new block go-to-pointer can be used in different scripts and even more than once in the same script as shown in Figure 3.1. Clearly, this script is much shorter than it would be if we had to copy the blocks again and again.



```
when I receive Come
if distance to Pointer < 250 then
    get-pointer-direction
    if direction-to-pointer < 335 and direction-to-pointer > 280 then
        switch costume to left
        go-to-pointer
    else
        if direction-to-pointer > 25 and direction-to-pointer < 80 then
            switch costume to right
            go-to-pointer
        else
            if direction-to-pointer > 335 or direction-to-pointer < 25 then
                switch costume to center
                go-to-pointer
```

Figure 3.1: Script for **likes**



```
define go-to-pointer
wait 0.5 secs
point towards Pointer
switch costume to center
repeat until distance to Pointer < 60
    move 5 steps
```

Figure 3.2: Definition of the block **go-to-pointer**

Chapter 4

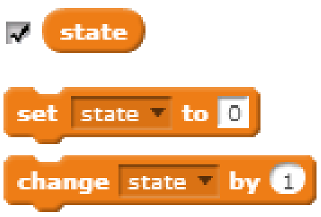
Variables

Using a variable to store a state

In advanced mode, VPL projects can use *states* to remember situations and values. There are four elements ("quarters") to the state block, so there are 16 different states. Scratch supports *variables* which can be used to store values. They offer many more capabilities than VPL states, because there can be as many variables as we want and each variable can store a large range of values.

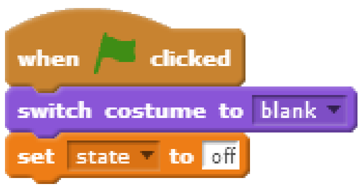
Scratch project: **tap-on-off-state**

Let us implement the project **tap-on-off** from the tutorial. Clicking on the `Thymio` sprite will turn the top light on if it is off and turn it off if it is on. A variable will remember the current state `on` or `off`. First, we *declare* the variable. Go to the **Data** palette and click Make a Variable ; give the variable a name that makes clear what its purpose is. An oval block with the variable's name appears, together with some new blocks:

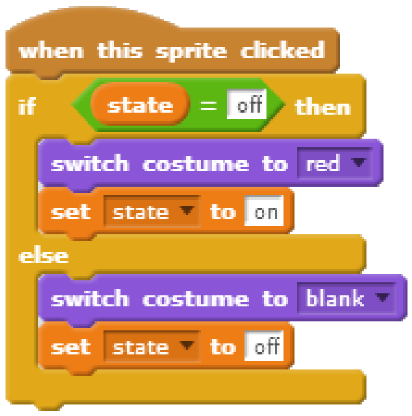


Displaying the value of the variable
The check next to the variable block is used to indicate that you want the variable and its current value to be displayed on the stage.

We need to *initialize* the variable in the script that is run when the green flag is clicked. This gives the variable a value before it is used for the first time. Here, the block set state to off sets the value to `off`:



The following script starts with the block when this sprite clicked, meaning that it begins to run when the sprite that contains the script is clicked:



```
when this sprite clicked
if state = off then
    switch costume to red
    set state to on
else
    switch costume to blank
    set state to off
```

It checks the value of the variable `state` and decides whether to switch the costume to the `red` costume or to the `blank` costume. It also changes the current value of the variable `state` to the opposite value.

Using a variable to store a number

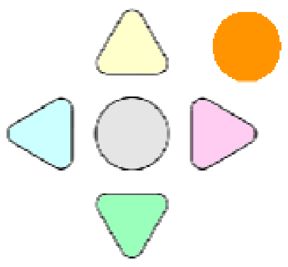
Scratch project: **count-to-two**

Scratch project: **count-to-four-binary**

Scratch project: **add**

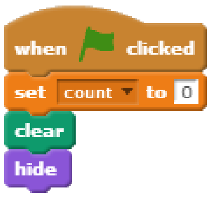
Variables are commonly used to store numbers. The VPL project **count-to-two** encoded the numbers 0 and 1 in states, and later projects counted to 4 and even to 16. We show how to use variables to count to two in binary, and leave the extension to other numbers as exercises.

The Thymio robot displays the current state by lighting the diagonal segments of the circular lights around the buttons. We simulate this by displaying an orange circle between the buttons on an image:



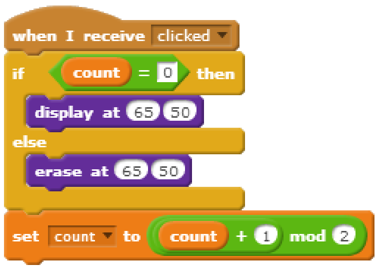
There are two aspects to the simulation in Scratch: (1) initializing, incrementing and checking the variable, and (2) displaying and erasing the orange circle.

We first discuss the operations on the variable. The variable is named count and is initialized to zero in the first script:



```
when [flag] clicked
set count to 0
clear
hide
```

In the second script, the choice whether to display or erase the circle is done in an if-statement that checks if the value of `count` is 0 or not, using the `if ⟨ ⟩ then … else …` block:



```
when I receive clicked
if <(count) = 0> then
    display at 65 50
else
    erase at 65 50
set count to ((count) + 1) mod 2
```

If the condition is true, the blocks in the first "mouth" are run; if not, the blocks in the second "mouth" are run.

The actual display of the circle is done within the two blocks `display at 1 1` and `erase at 1 1` whose definition is given below.

The final block of the script is `change count by 1` which changes the (numerical) value in the variable by the amount written in the small square. Here, `((count) + 1) mod 2` adds 1 to the value of `count` and then takes the remainder (called mod) by 2. The result will be either 0 or 1.

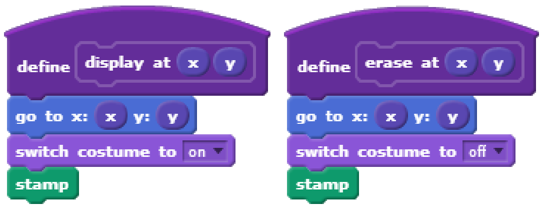
Warning!

Be sure not to confuse `set count to 1` with `change count by 1`. The first block ignores the current value in the variable and sets the variable to the value written in the small square. The second block takes the current value of the variable, adds the value written in the small square and then sets the value of the variable to the result of the computation. To subtract a value from the current value, simply change by a negative number.

Drawing on the stage

Scratch supports drawing on the stage using blocks available in the **Pen** palette. We will not explain all the blocks here, just the ones used to display and erase the orange circle.

In the initialization script, `clear` erases existing marks on the stage , if any. The blocks `display at 1 1` and `erase at 1 1` contain `stamp` which prints the image of the current costume of the sprite on the stage:



```
define display at x y
go to x: x y: y
switch costume to on
stamp
```

```
define erase at x y
go to x: x y: y
switch costume to off
stamp
```

Warning!

Be sure not to confuse sprites and stamps. A sprite is an actor in a Scratch animation; it has scripts and costumes. When it moves on the stage it does not leave a mark. The block `stamp` makes a mark on the stage; the mark is the current costume of the sprite that runs the block. The stamp is not an actor and does not move, and the mark remains on the stage until removed by running `clear` .

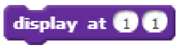
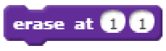
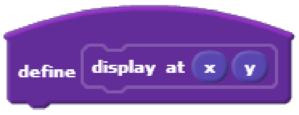
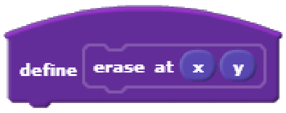
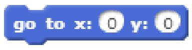
Don't use the `clear` when you only need to erase one mark such as an orange circle. Instead, change the costume of the sprite to one that is all white and stamp it in exactly the same place as the mark of the orange circle.

The block `hide` is used in the initial script because we don't want to display the sprite, only the marks that it makes on the stage.

Why two sprites?

There are two sprites in this project: a `Thymio` sprite (well, only the buttons), and a `circle` sprite. The `Thymio` sprite is the one that is clicked on; it broadcasts a message to the `circle` sprite. We don't want to sense a click on the `circle` sprite, because it may not be visible, and we don't want to stamp the `Thymio` sprite because we want marks of the circle, not of the buttons.

Parameters

We want to use the blocks  `display at 1 1` and  `erase at 1 1` in more than one place with different values for the positions of the marks. The definitions of the blocks  `define display at x y` and  `define erase at x y` declare two *parameters* called x and y. When the blocks are used, the x- and y-values of the position need to be provided in the small squares, just as is done in predefined blocks like  `go to x: 0 y: 0` .

Within the definition of a block with parameters, the current values of the parameters are available as oval blocks that can be used like any other variable.

Chapter 5

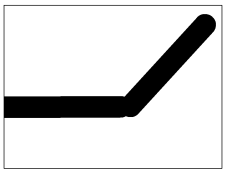
Guided Projects

These projects introduce no new concepts so we present them as a guided projects.

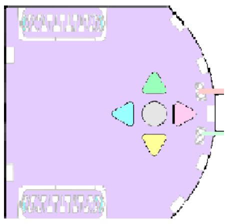
Line following

Scratch project: **follow-line**

Construct a backdrop with a wide line that represesnts a strip of black tape on the floor:



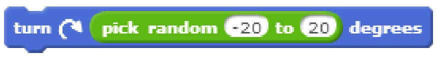
Next, import the costume called `ground`:



This costume has two small areas projecting from the front of the robot and is used to implement the block get-touching as described in Appendix A. The block returns in the variable is-touching one of `both`, `left`, `right`, `neither` depending on which projecting area is touching the black area on the stage. The `Thymio` sprite needs one script whose algorithm is shown in Figure 5.1.

Real robots don't move straight

From your experience with the Thymio robot, you know that it will not move straight even if both motors are set to the same power. The wheels and motors are not identical, the friction may vary, and so on. This noisy behavior is called *jitter*. Place the following block before first `move` block to simulate jitter:



```
turn ↻ pick random -20 to 20 degrees
```

```
initialize the position of the Thymio
forever
    call get-touching-block
    if is-touching = both
        move forward
    else
        if is-touching = right
            turn right
        else
            if is-touching = left
                turn left
            else
                stop the script
```

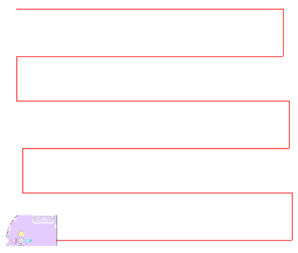
Figure 5.1: Algorithm for line following

The block causes the robot to turn by an angle that is randomly chosen in the range -20 to 20. You will see that the robot moves very erratically, but the algorithm succeeds in keeping the robot on the line.

Sweeping the floor

Scratch project: **sweep**

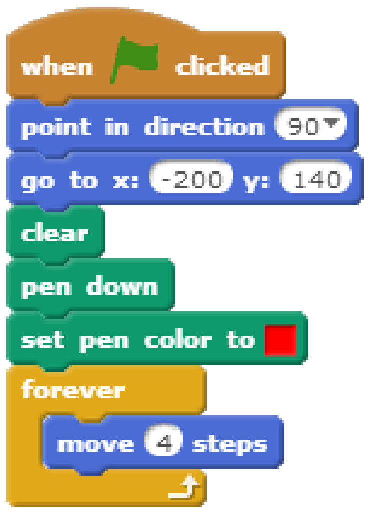
We want the `Thymio` sprite to traverse the stage so as to cover the whole surface:



In order the evaluate the quality of the sweep, the robot draws a thin red line as it moves. pen down causes an imaginary pen in the sprite to be lowered "down" so that it is in contact with the stage. The pen color is set to red.

The simulation could be programmed simply by commanding the sprite to move the correct distances—280 steps horizontally and 60 steps vertically—turning right and left by 90° as required. However, there is no way to command the Thymio *robot* to travel a certain distance. All we can do is to set the motors to a constant power for a fixed time, but the speed will not be precisely constant for the reasons mentioned above, so the distance will not be predictable.

The script shown in Figure 5.2 causes the sprite to move 4 steps at a time. A second script (not shown) consists of a repeat until block that contains blocks for left and



```
when [flag] clicked
point in direction (90 v)
go to x: (-200) y: (140)
clear
pen down
set pen color to [ ]
forever
    move (4) steps
```

Figure 5.2: Sweeping the stage

right turns of 90° with `wait (0.3) secs` blocks between them. Adjust the durations in the `wait` blocks to obtain a good sweep of the stage.

Finite automata

Scratch project: **finite-automaton**

The finite automata project in the VPL tutorial was inspired by the light-painting projects described on the Thymio website. The idea is to have the robot sweep over an area and respond to codes placed on the floor. The code consists of sync (synchronization) marks and marks of symbols on the stage. Figure 5.3 shows the stage with 4 rows of 4 sync marks (the blue circles) and 9 vertical black lines representing the symbol 1.

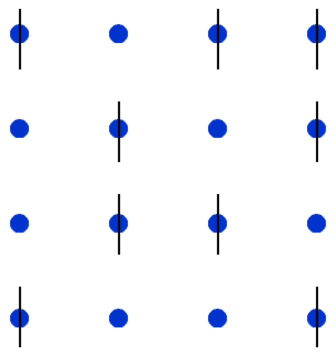


Figure 5.3: Synchronization and symbol marks

The Thymio sprite is required to sweep the stage and record the number of 1's. (For a finite automata we need to specify a finite range, such whether the number of 1 symbols is even or odd. This can be done using the mod operator.)

The sprite can be in three states: (1) it is touching a sync mark alone, (2) it is touching both a sync mark and the black line encoding 1, and (3) it is touching neither. Use `touching color 🟦 ?` to detect the blue ball representing the sync marks and `touching color ⬛ ?` to detect black line representing 1.

To check the behavior of the sprite, display different colored top lights in each state.

> **Warning!**
> The sync marks should be sufficiently far apart so that the Thymio sprite doesn't touch two marks at the same time.

Synchronizing the sprites

There will be two sprites: the Thymio sprite that sweeps the floor and decodes the marks, and a ball sprite that stamps marks on the stage before the Thymio sprite starts to move. Use messages to synchronize the sprites:

- The Thymio sprite performs initialization and then broadcasts the message draw.
- When the ball sprite receives the draw message, it stamps the marks and then broadcasts the message go.
- When the other scripts in the Thymio sprite receive the do message, they begin the tasks of sweeping the stage and decoding the marks.

Timed behavior

Scratch project: **shy**

Specification: If the Pointer is clicked opposite the right sensor, the Thymio sprite turns until the Pointer is opposite the center sensor and then 4 seconds later turns back. Similarly, if the Pointer is clicked opposite the left sensor. Be sure that the small red lights on the left, right and center sensors are turned on and off as appropriate.

Guidance: The behavior for detection by the left and right sensors is the same except for the costumes. Create a new block turn-and-turn-back with one parameter—the direction—that includes all the blocks needed to implement the behavior of the Thymio sprite once the Pointer sprite is detected.

Catch the mouse

Scratch project: **mouse**

Specification:

- The `Thymio` sprite detects a click of the `Pointer` sprite opposite its left sensor.
- It turns left (counterclockwise) until the right sensor points towards the `Pointer`.
- It turns right (clockwise) until the center sensor points towards the `Pointer`.
- It moves towards the point of the click and then stops.

Guidance:

- Declare a variable that takes four values: `search-left`, `search-right`, `found`, `stop`. This variable will take on successive values as it accomplishes each part of the mission.
- The behavior of the `Thymio` sprite will depend on which state it is in. It is convenient to define new blocks for each state: `go-left`, `go-right`, `catch-mouse`.

Chapter 6

Braitenberg Creatures

What are Braitenberg creatures?

Valentino Braitenberg was a neuroscientist who wrote a book describing the design of virtual vehicles which exhibited surprisingly complex behavior.[1] Braitenberg's vehicles have been widely used in educational robotics. Researchers at the MIT Media Lab created hardware implementations of the vehicles called *Braitenberg creatures*.[2] The vehicles were build from *programmable bricks* that were the forerunner of the LEGO Mindstorms robotics kits.

This document describes an implementation in Scratch of most of the Braitenberg creatures from the MIT report. The implementation is based upon a simulation in Scratch of the Thymio robot using the VPL programming environment. It is intended to be used as an introduction to Thymio / VPL for students with experience in Scratch.

The MIT hardware used light and touch sensors, while the Thymio robot relies primarily on infrared proximity sensors. To enable comparison with the MIT report, the names of the creatures used there have been retained, even though they may not be appropriate for the Thymio implementations. The order of presentation from the report has also been retained, although this does not correspond to the difficulty of implementation in either Scratch or VPL.

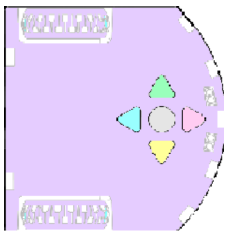
[1] V. Braitenberg. *Vehicles: Experiments in Synthetic Psychology* (MIT Press, 1984).

[2] David W. Hogg, Fred Martin, Mitchel Resnick. *Braitenberg Creatures*. MIT Media Laboratory, E&L Memo 13, 1991. http://cosmo.nyu.edu/hogg/lego/braitenberg_vehicles.pdf.

Overview

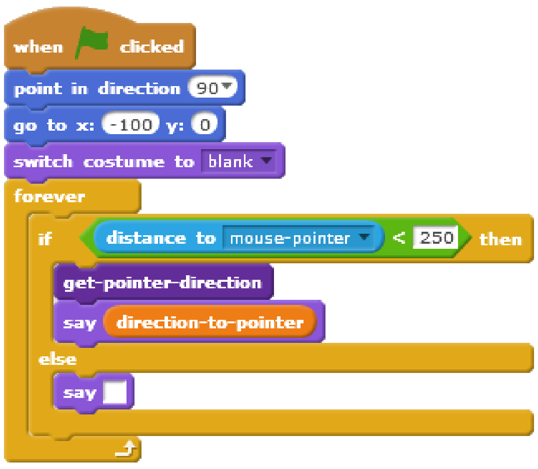
The project `template` contains the necessary sprites, the costumes and the outline of the scripts. The Braitenberg projects can be built starting with this project.

The Thymio robot is simulated by a sprite called `Thymio` that appears on the screen as:



The sprite has seven costumes: the `blank` costume shown above; four costumes (`center`, `right`, `left`, `rear`) with the corresponding sensors lit, and two (`red`, `green`) with the top lights turned on.

The `Thymio` sprite *detects an object* when the mouse pointer is close to the sprite. A new block called `get-pointer-direction` returns in the variable `direction-to-pointer` the angle from the sprite to the mouse pointer.[3] Zero degrees is the direction that the `Thymio` faces and the angles increase clockwise. The following test script in the template project demonstrates how the simulation works:



```
when [flag] clicked
point in direction 90
go to x: -100 y: 0
switch costume to blank
forever
    if distance to mouse-pointer < 250 then
        get-pointer-direction
        say direction-to-pointer
    else
        say
```

Start the program by clicking on the green flag. If the mouse pointer is near the `Thymio` sprite, the sprite says the direction to the pointer.

[3]The implementation is described in Appendix A, except that the direction is to the mouse pointer and not to a pointer sprite.

Specification of the creatures

Timid When the robot does not detect an object, it moves forwards. When it detects an object, it stops.

Indecisive When the robot does not detect an object, it moves forwards. When it detects an object, it moves backwards. Experiment with the definition of the distances for "detect" and "not detect" so that the robot *oscillates*: move forwards and backwards in succession.

Paranoid When the robot detects an object, it moves forwards. When it does not detect an object, it turns to the left.

Paranoid1 When an object is detected by the center sensor of the robot, it moves forwards. When an object is detected by the right sensor, it turns right. When an object is detected by the left sensor, it turns left.

Dogged When the robot detects an object in front, it moves backwards. When the robot detects an object in back, it moves forwards. When an object is not detected, it stops.

Insecure If an object is detected by the left sensor, the robot turns right. If an object is not detected by the left sensor, it turns left. Experiment with the angles of the turns until the robot can track the mouse pointer placed ahead and to its left.

Driven If an object is detected by the left sensor, the robot turns left. If an object is detected by the right sensor, it turns right. The robot will approach the mouse pointer in a zigzag.

Persistent The robot moves forwards until it detects an object. It then moves backwards for one second and reverses to move forwards again.

Attractive and repulsive When an object approaches the robot from behind, the robot moves forward until the object is no longer detected.

Consistent The robot cycles through four states when it is clicked on: moving forwards, turning left, turning right, moving backwards.

Frantic The top light flashes red.

Observant The robot turns the top light green when the right sensor detects an object. The robot turns the top light red when the left sensor detects an object. Once a light is turned on, the robot waits three seconds before turning off; during this period, the light does not change.

Appendix A

Implementation

This Appendix explains advanced programming techniques in Scratch that were used in the simulations of the Thymio robot. The details are encapsulated in sprites and new blocks so that the student need not be concerned with them, but the Appendix can serve are a tutorial to these programming techniques.

The ***`Pointer`*** *sprite*

The *touching* blocks from the **Sensing** palette are used to detect events. They are condition blocks that return true if the ***sprite running the script where the block appears*** touches something else.

- touching Thymio ▾ ? returns true if the sprite is touching another sprite or the mouse pointer.
- touching color 🟩 ? returns true if the sprite is touching an area with this color.
- color 🟪 is touching 🟨 ? returns true if an area ***on this sprite*** with the first color is touching an area with the second color.

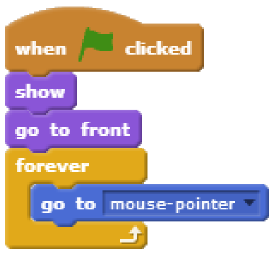
> **Trick**
>
> To set the color, click the little square, move the mouse pointer and click an area on a sprite or the backdrop that has the color you want.

Scratch does not support checking if the mouse pointer is touching a color; therefore, we define a sprite called `Pointer` that tracks the location of the mouse pointer (Figure A.1). The costume of this sprite is a very small gray dot that the user will not see.

To simulate the buttons on the Thymio robot, the image of the `Thymio` sprite has different colors for each button. When the `Pointer` sprite is clicked, the script in Figure A.2 checks if it is touching one of the colors and broadcasts the appropriate message.

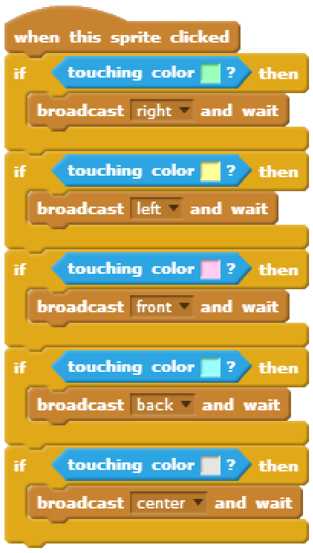
Computing the direction to the ***`Pointer`*** *sprite*

Many projects require that the direction from the `Thymio` sprite to the `Pointer` sprite be determined. The implementation of the block get-pointer-direction is shown in Figure A.3.



```
when 🏁 clicked
show
go to front
forever
    go to mouse-pointer ▾
```

Figure A.1: Tracking the mouse pointer



```
when this sprite clicked
if touching color 🟩 ? then
    broadcast right ▾ and wait
if touching color 🟨 ? then
    broadcast left ▾ and wait
if touching color 🟪 ? then
    broadcast front ▾ and wait
if touching color 🟦 ? then
    broadcast back ▾ and wait
if touching color ⬜ ? then
    broadcast center ▾ and wait
```

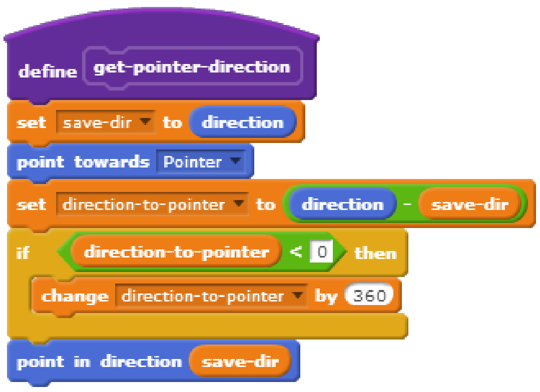
Figure A.2: Sensing a click on the buttons

The variable save-dir is used internally by this block, and the variable direction-to-pointer is used to return the direction to the calling sprite. Extensive use is made of the block direction which is predefined in the **Motion** palette and gives the current direction in which the sprite is pointing.

The *current direction* in which the sprite is pointing is saved in save-dir. Then, the `Thymio` sprite is turned so that it points to the `Pointer` sprite. The block direction now contains this new direction. By subtracting the value in save-dir we get the difference between the two directions. The `if`-block ensures that this direction is positive between 0 and 360. Finally, we turn the `Thymio` sprite to point in its original direction which was stored in save-dir.

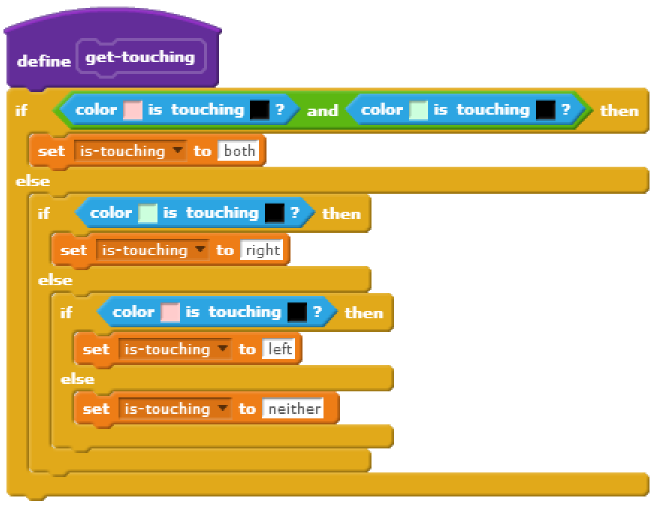
Simulating the botton sensors

To simuate the bottom sensors, create a new costume for the `Thymio` sprite with light-colored areas projecting out in front of the image:



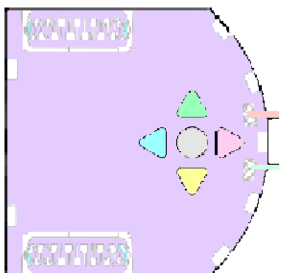
```
define get-pointer-direction
set save-dir to direction
point towards Pointer
set direction-to-pointer to (direction - save-dir)
if direction-to-pointer < 0 then
    change direction-to-pointer by 360
point in direction save-dir
```

Figure A.3: Get the direction to the `Pointer` sprite



```
define get-touching
if color 🟥 is touching ⬛ ? and color 🟩 is touching ⬛ ? then
    set is-touching to both
else
    if color 🟩 is touching ⬛ ? then
        set is-touching to right
    else
        if color 🟥 is touching ⬛ ? then
            set is-touching to left
        else
            set is-touching to neither
```

Figure A.4: Simulation of the ground sensors



Now declare a variable is-touching and define the block get-touching (Figure A.4), which returns in is-touching one of `both`, `left`, `right`, `neither` depending on which projecting area is touching the black area on the stage.