

## Turtle Graphics

Scratch is one of the modern programming languages that include turtle graphics options, introduced for the first time in the *Logo* language some fifty years ago.

*Logo* enabled the control of a turtle device, a type of robot, which was connected to the computer. The turtle could move on a horizontal surface back and forth and change its direction and orientation. If a piece of paper was placed under the turtle, it could leave a mark, thus forming graphics called the *turtle graphics*.

This way of setting an image as a collection of figures, together with their data (parameters), which define how the figure will be drawn and where it will be placed is called **vector graphics**.


Unlike the vector graphics, in the **raster graphics** the image is stored as a rectangular grid of pixels - bitmaps. Bitmaps are technically characterized by the width and height of the image in pixels, and the number of bits needed to store the color of the pixels. For example, if there are only 16 colors, you need 4 bits per pixel to store the color. Raster graphics are resolution- dependent. They cannot be enlarged without the loss of quality of the image.

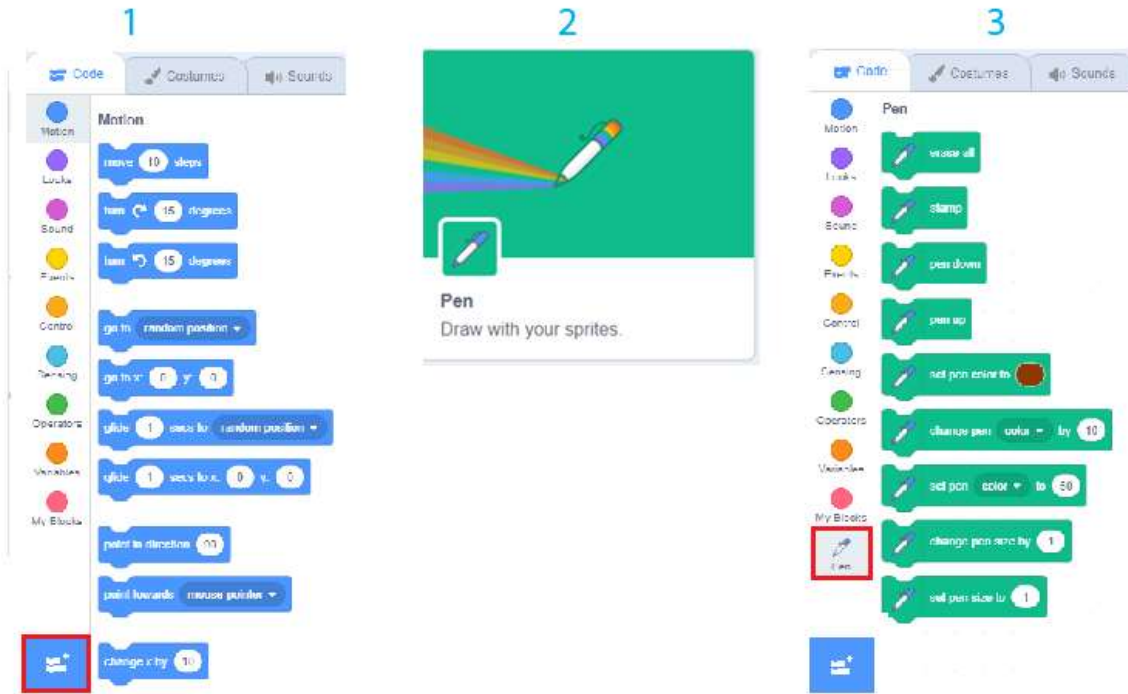
In Scratch, every sprite has the option to behave like the Logo turtle: you can determine their position, direction and orientation of movement, and they can leave a mark when they move. The size, the shape and the color of the sprite do not affect the mark they leave, because the sprites are drawing with a pen. The sprite can also be invisible or composed of only one dot, and this does not affect its drawing.

### Functions of the *Pen* blocks








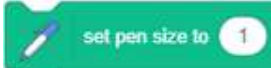

The *Pen* group of blocks, which enable the sprites to leave a mark when they move, i.e. to support the turtle graphics are located among the extensions.



In order to use blocks from this group, you need to:

1. Click on the  icon in the bottom left corner of the blocks palette.
2. Select the *Pen* extension from the opened gallery.
3. The icon and the blocks will appear in the blocks palette.





The *Pen* group includes the following blocks:

-  - erases all marks made by the pen
-  - stamps the image of the sprite on the stage
-  - lifts the pen up
-  - puts the pen down
-  - sets the color of the pen
-  - sets the drawing parameters
-  - changes the drawing parameters
-  - sets the size (thickness) of the pen mark
-  - changes the size (thickness) of the pen mark

If you add the  block to your sprite, from then on, the sprite will leave a mark on the stage whenever it moves. When you add the  block, it will stop leaving

the mark until you add the  block again. The look of the mark is determined by the drawing parameters. The drawing parameters include the size (thickness), color, saturation, brightness and transparency of the mark left by the pen.

Note.   $\neq$  . The first block refers to drawings and the second to the sprites.



### Using messages for synchronization

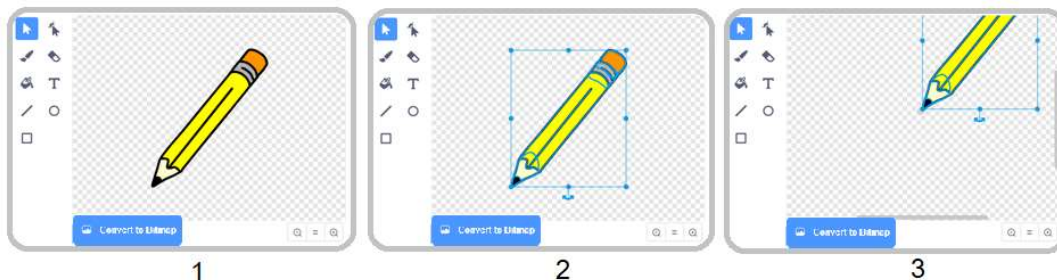
The behavior is always triggered by an event, which can be an action or receipt of a message. We will now show how the actions of the sprites (and the stage) can be coordinated, depending on whether a broadcast message event has occurred. You noticed that we didn't say that we **send** a message, but that we **broadcast** it. This is because, in Scratch, the message is directed to all objects, and not just one. Broadcasting and receiving


messages in Scratch is achieved with the following blocks ,  and  from the *Events* group.

### Some examples

#### Example 1 - "Drawing a Line"

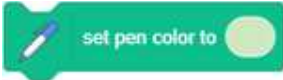
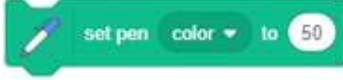
We will now draw a line that is 10 pixels wide and 300 pixels long. We will choose the Pencil sprite from the sprites library, and we will delete the cat sprite. Since we want the mark to be positioned along the tip of the Pencil sprite, we will move the center of the costume to the top. We can do this by opening the *Costume* tab. The available tools will appear on the left side of the drawing display; in our case, these will be vector tools because we chose a vector sprite (1). Use the  to select the whole sprite (2), and then drag the sprite so that the tip of the pencil is above the sign  which indicates the center of the screen (3).



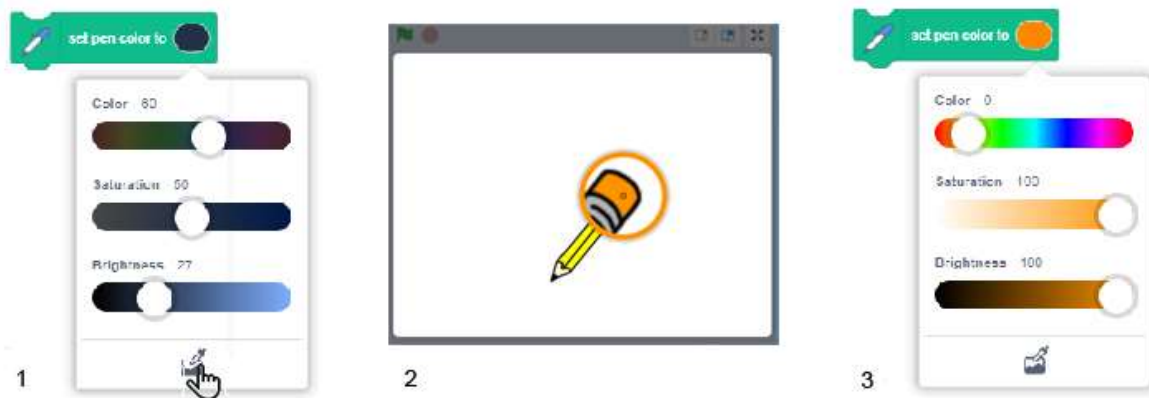
First, you need to erase everything that was previously drawn by using the  block, place the pencil in the starting position from which it will start drawing, set the color, and the size of the pen, and then use the **pen down** block to allow the pencil sprite to leave a mark when it moves. You need to add the **pen up** block at the end to stop the sprite from

leaving a mark in the process of returning to the starting position, each time the program runs.

There are two ways to set the color of the pen:

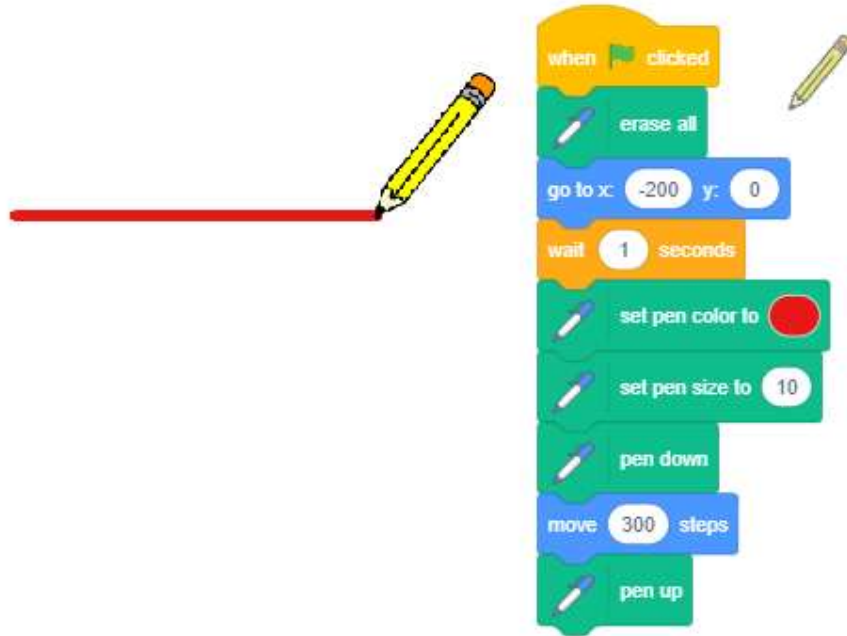
- by using the  block, where you can select a color by clicking on the input field
- by using the  block, where you can select a color by inserting a number into the second input field.

In this example, we will use the first option.



Clicking on the input field of the block for setting the pen color, we open the drop-down menu where we will find sliders with color components: number attached to the color, saturation and brightness, and at the bottom we will see a tool - a pipette for collecting color samples (1). The desired color can be set by moving the sliders or by clicking on the pipette. If you click on the pipette, a stage with a magnifier on it that has a circle in the center will light up (2). To choose a color, we need to place the center of the circle above the part of the stage where that color is located and then click on it. We will get the same result, as shown in Figure.

The result of the running of the script and the script itself are presented in the Figure below.




---

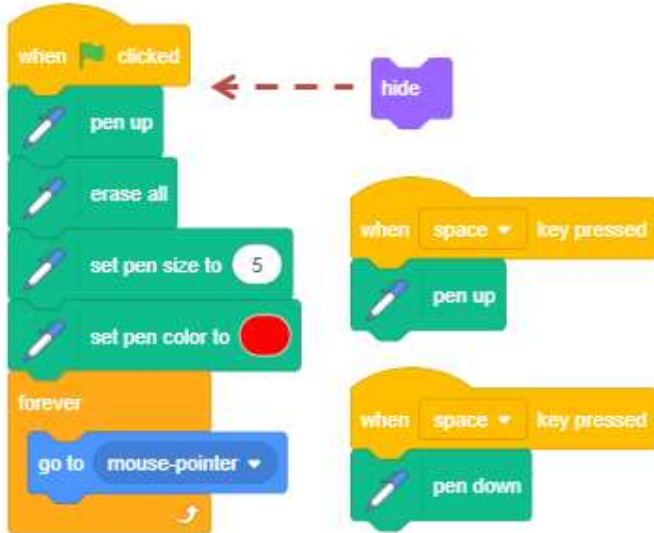
### Example 2 - “Free-hand Drawing”

This example should illustrate how we use drawing commands and show that the mark does not depend on the size of the sprite, nor on whether the sprite is visible or hidden. The pen is doing the drawing, so it does not matter which sprite is holding it. This time we will choose the *Ladybug 1* sprite from the sprites library.

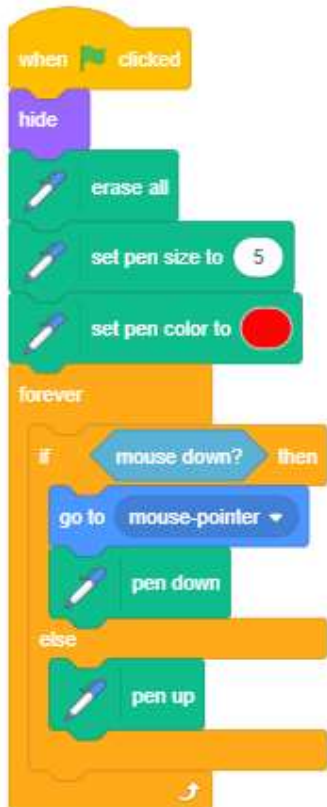
The script, which is activated by clicking on the green flag, allows the ladybug to follow the mouse-pointer for the duration of the running. At the beginning of the script, all drawings from previous running of the example will be erased from the stage, and the pen will be raised. The pen down command will be added to the *when down arrow is pressed* event block, and the pen up command will be added to the *when up arrow is pressed* event block. This way we will make sure that the sprite will not leave any marks when it moves until we press the *down arrow* key. The sprite will stop leaving marks when we press the *up arrow* key.

The scripts added to the ladybug are presented in the Figure below. The drawing would be


performed the same way even if the block  was inserted where the red arrow in the Figure is pointing, but then the sprite wouldn't be visible, and it would appear that the drawing is being done by the mouse-pointer.



Of course, it would be more natural if we didn't have to press the keys of the keyboard to lower and lift the pen, but just draw while holding the mouse button pressed, and then stop drawing by lifting the finger. This type of drawing is achieved by the following script.



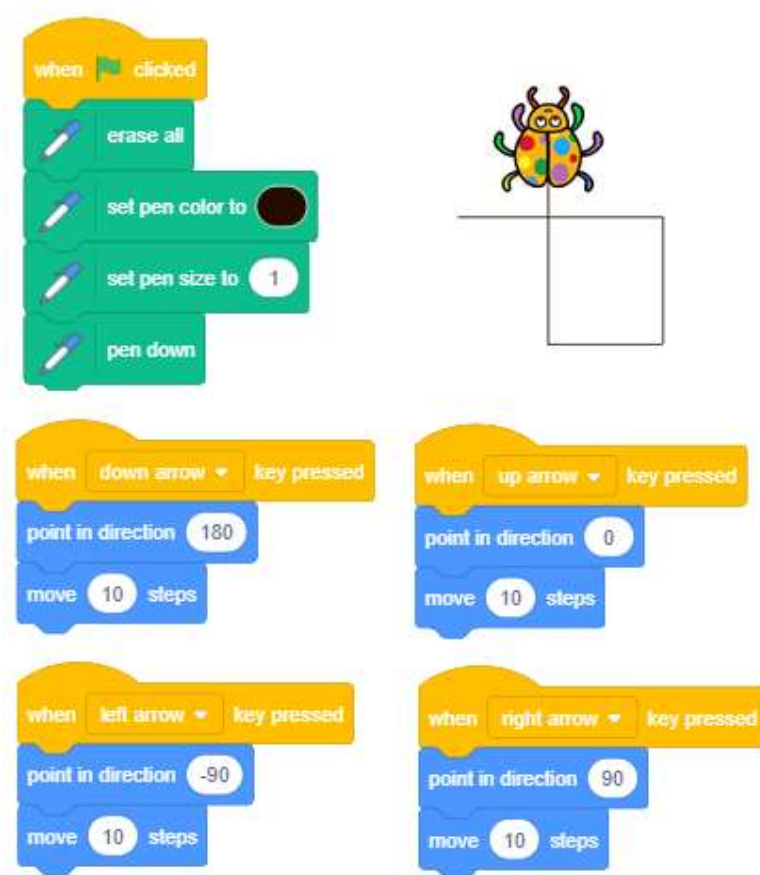
The effects of the **if then else** command can be understood. For now, let's just say that the script in this E-block will either execute the **go to random position** and pen down command

from the upper slot, or the pen up from the lower slot. Which command is being executed will depend on whether the mouse button is pressed or not, which is determined by the value set in the  block. This block belongs to the *Sensing* group, and it reports whether the value is true or false. Hexagonal function blocks that report only the values *true* or *false* are called **Boolean blocks**.

---

### Example 3 - “Follow my Trace”

In this example, the movement of the **ladybug** sprite is guided by the use of arrow keys. The mark left by the ladybug is a broken line made of horizontal and vertical lines. The following Figure includes the scripts and the look of the stage after one example execution.



The figure shows the Scratch scripts and the resulting stage for the "Follow my Trace" example. On the left, there are four scripts:

- when green flag clicked**
  - erase all
  - set pen color to black
  - set pen size to 1
  - pen down
- when down arrow key pressed**
  - point in direction 180
  - move 10 steps
- when up arrow key pressed**
  - point in direction 0
  - move 10 steps
- when left arrow key pressed**
  - point in direction -90
  - move 10 steps
- when right arrow key pressed**
  - point in direction 90
  - move 10 steps

On the right, a ladybug sprite is shown on a stage. A broken line is drawn on the stage, consisting of a horizontal line segment followed by a vertical line segment, forming an L-shape.

### Example 4 - “Lines”

In the *Lines* example we will draw a purple line, 400 pixels long and 2 pixels wide, starting from the point (-200,0) in five different styles.

- 1 —————
- 2 .....
- 3 - - - - -
- 4 - - - - -
- 5 - - - - -

The sprite drawing the line is invisible. It will appear in the upper left corner of the stage, only when it finishes drawing the line, and it will say how the line was drawn. The drawing is



activated by the event block, the script for the first style is activated by pressing number 1 on the keyboard, the second style by pressing 2, and so on.

Clicking on the green flag erases everything that was previously on the stage, sets values for the color and size of the pen, sprite appears and gives instructions on how to start the example.

Each of the scripts associated with keys 1-5 on the keyboard first erases everything that was previously drawn on the stage, hides the sprite, draws the line in the given style, and then shows the sprite in the upper left corner of the stage, which tells us how the line was drawn.

The first style is a continuous straight line. This can be done immediately, by giving just one command "go 400 steps", but to make this last almost as long as other scripts we added the repeat command, so the sprite would move 10 steps 40 times.

The second style - the sprite repeats the same pattern 100 times: it moves 1 step with the pen down, and 3 with the pen up.

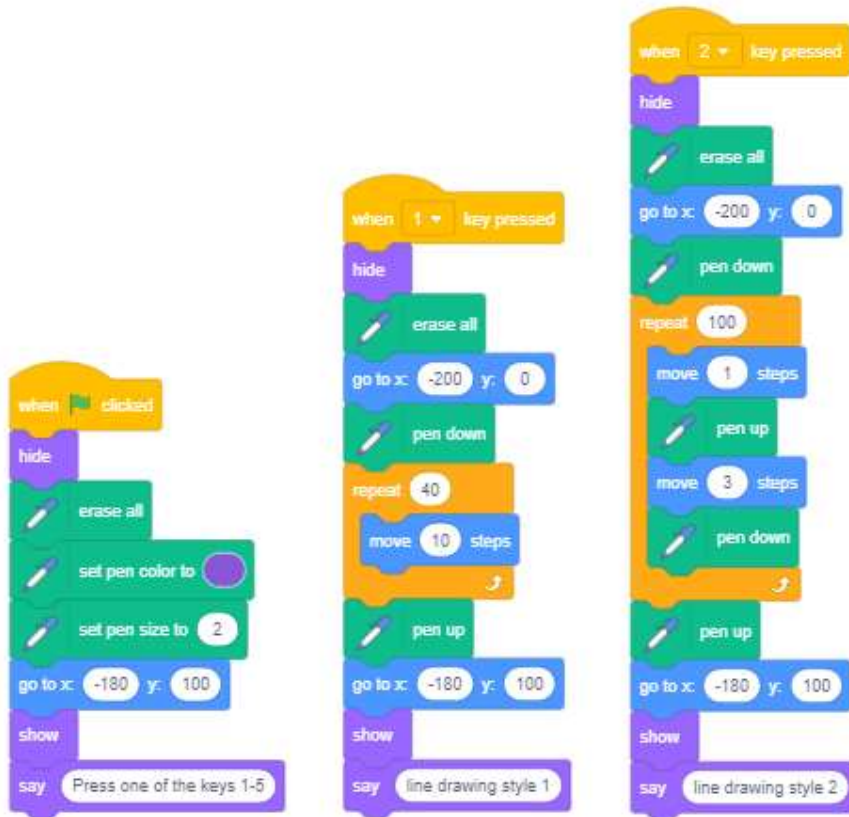
The third style - the sprite repeats the same pattern 50 times: it moves 3 steps with the pen down and, and 5 with the pen up.

The fourth style - the sprite repeats the same pattern 40 times: it moves 6 steps with the pen down and, and 4 with the pen up.

The fifth style - the sprite repeats the same pattern 25 times: it moves 6 steps with the pen down and, and 4 with the pen up, 2 steps with the pen down, and 4 with the pen up.

In the Figure below you will find the scripts for the events *when the green flag is clicked*, *when 1 key is pressed*, and *when 2 key is pressed*.





Note that repetition commands do not shorten program running time, but only allow the programmer to write programs more clearly and concisely.