Lesson

REPEAT A PATH WITH AN ARRAY

CREATED BY

Richard Born
Associate Professor Emeritus
Northern Illinois University
rborn2@niu.edu

TOPICS

Robotics, Programming

GRADES

7-12

METHOD

OzoBlockly

DURATION

45 minutes

Ozobot Bit/Evo Lesson: Repeat a Saved Path Using an Array

By Richard Born
Associate Professor Emeritus
Northern Illinois University
rborn2@niu.edu

Introduction

In this lesson students will learn how to make Ozobot Bit/Evo follow a random path on a Cartesian grid and then repeat the path after being placed back at the starting position. This will be accomplished by the use an array constructed from the OzoBlockly Mode 5 (Master) array-processing blocks. This lesson assumes no prior knowledge of arrays, so it can be used as a practical and independent learning experience for students.

What is an array?

Before working with arrays in OzoBlockly, it might be a good idea to think of how the word *array* is used in everyday conversation. Here are some example sentences:

- There is a beautiful *array* of flowers in that vase.
- The *array* of solar panels on my neighbor's roof is soaking up the sun.
- Mark has an *array* of baseball bats in the corner of his bedroom.
- You can choose from an *array* of colors to paint your house.
- Sarah has an *array* of dresses in her closet.

These sentences all share a common thread. Each sentence references a *group of identical or similar things with the same name*: flowers, solar panels, bats, colors, or dresses. The concept of an array in computer science also shows this common feature. An array in computer science is *a group of sequential memory locations for storing similar information*. Examples include:

- An array storing names of customers
- An array storing restaurants within fifteen miles of your house
- An array storing prices for items that you purchase while visiting the grocery store
- An array storing colors that Ozobot encounters while following a line
- An array storing the direction that Ozobot takes for each step in a random walk

The name of the array for the "storing colors" example above might be *color*. Individual elements in an array are identified by an *index*. The first element in an array has an index 0. Languages such as JavaScript place the index in square brackets. So we could identify the first three elements of the array *color* by *color[0]*, *color[1]*, and *color[2]*. Array elements in OzoBlockly can hold any integer from -128 through 127. If we let 1 represent the color red, 2 green, and 3 blue, then the contents of *color[0]*, *color[1]*, and *color[2]* might be 2, 3, and 1 if Ozobot encountered the colors green, blue, and red in that order.

A *declare array* block that can be placed anywhere in the OzoBlockly workspace is used to name and specify the length of an array. The maximum length of an array is 127, the minimum 1. You can have more than one array, if needed, in your program logic. It would be a good idea to study the array block references in the online OzoBlockly *Master Mode Reference*. OzoBlockly only supports one-dimensional arrays.

Repeat Saved Path Video

It is suggested that the video https://goo.gl/GRNBic be viewed now to clarify what the OzoBlockly program does. Ozobot Bit/Evo is placed on the origin (gray circle) of the Cartesian coordinate system facing the right. The start button is pressed twice to run the stored program. It begins by displaying a white LED, with its initial step straight ahead to the right. After that successive steps are randomly right, left, or straight ahead. The red LED signifies that Bit/Evo is on its initial random walk. After completing a total of five random steps, Bit/Evo displays a blue LED for ten seconds, giving the user time to move Bit/Evo back to the origin facing to the right again. Bit/Evo then repeats the same path as the original random path, showing a green light.

For simplicity of program logic, the program was written under the assumption that the Cartesian coordinate system is infinite in all directions. Clearly the coordinate system on the OzoMap is finite. Therefore, you may find that when Bit/Evo reaches the outer edge of the finite coordinate system, it stops and flashes the LED red and blue to indicate that it cannot perform a turn requested by the program. In most cases this will not happen if you keep the number of steps in the initial random walk to 10 or fewer. In other words, the greater the number of steps in the random walk, the more likely it is that Bit/Evo will reach the edge of the paper's coordinate system and fail.

The Main OzoBlockly Program

The program was written in a modularized version with a number of smaller functions (subroutines) that are called as needed in the program logic. Figure 1 shows the OzoBlockly blocks for the main program.

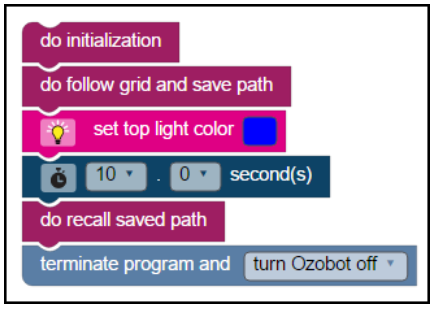

Figure 1

We'll take a detailed look at the three functions shown in purple blocks in upcoming sections of this document. Basically, the main program follows what you observed in the video. Some housekeeping initialization comes first, Bit/Evo then follows the Cartesian grid while saving its path, the LED is blue while the observer moves Bit/Evo back to the origin, Bit/Evo recalls and follows the saved path, and finally the program terminates and Ozobot turns off.

Array Declaration

Our program has a single array of length 6 whose name is *stepDirection*. The purpose of this array is for Bit/Evo to save the direction he turned for each step of his random walk. That way he can repeat the same walk later by looking up the direction for each successive step in the array. See Figure 2 for the array declaration block.

Figure 2

This array declaration can be placed anywhere in the OzoBlockly workspace. It causes OzoBlockly to set aside space in Bit/Evo's memory, in this case 6 locations for storing any integers from -128 to 127. Figure 3 provides a pictorial view of this array.

stepDirection[0]	stepDirection[1]	stepDirection[2]	stepDirection[3]	stepDirection[4]	stepDirection[5]

Figure 3

The indexes are shown in square brackets [ ]. Note that arrays are indexed starting at 0, so the last index in the array is always 1 less than the declared length of the array. In our array for this program, the length is 6 and the last index is therefore 5. For simplicity of logic we will not make use of stepDirection[0]. Rather than storing the direction of the first step in stepDirection[0], we will store it in stepDirection[1]. That way the step number matches the index number and causes less confusion. The price paid is that one memory location that is not used still reserves space in Bit/Evo's memory. It should be noted that in the event the index is lower than 0 or higher than *the array length minus 1*, the program will terminate and an *index out of bound* error is indicated by a blue-red flashing LED.

"Do Initialization" Subroutine

Figure 4 shows the blocks for the "do initialization" subroutine. The program begins by showing a white LED, and waits three seconds. This delay allows the student to start Ozobot Bit/Evo while holding it and then setting it down on the origin before it starts moving. The variable holding the number of steps planned for the walk is set to the length of the array minus 1, i.e., 6 – 1 = 5 steps. Note the use of the get length of array block, shown in light blue. The number of steps is always one less than the length of the array, as defined in the array declaration block. So if we want to change the number of steps in the walk, the only adjustment needed is to change the length of the array in its declaration.

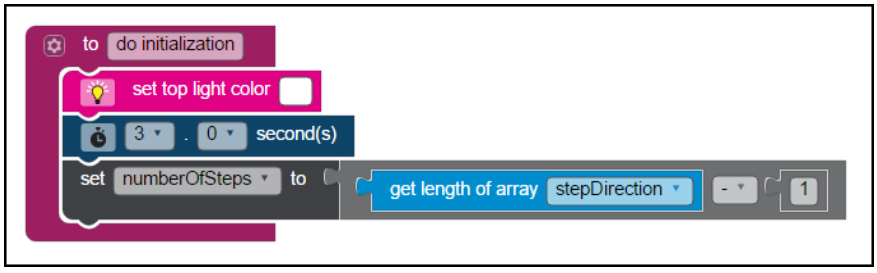

Figure 4

"Do Follow Grid and Save Path" Subroutine

Figure 5 shows all of the blocks for the "do follow grid and save path" subroutine. The first thing we do is set the LED red, since that is the color that Bit/Evo is supposed to have while taking the initial random walk. Then there is a loop that is executed once for each of the steps in the walk. Within the loop, the variable *direction* is set to a random number from 1 through 3, arbitrarily representing the direction right, left, and straight ahead, respectively. We'll see that mapping direction to integers is arbitrary when the "do take next step in path" subroutine is discussed. Saving the direction for this step is accomplished by the "in array --- set element --- to value" block, shown in light blue. The count variable *i* in the loop is, in fact, also the index of the element in the array where we want to save the direction. Finally, Bit/Evo takes the next step by calling the "do take next step in path" subroutine, which we will now study.

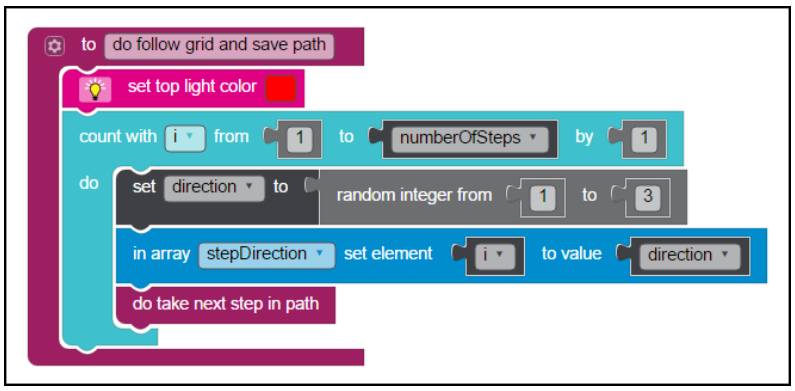

Figure 5

"Do Take Next Step in Path" Subroutine

Figure 6 shows the OzoBlockly blocks for the "do take next step in path" subroutine. First Bit/Evo is instructed to follow the line to the next intersection. Then there are a pair of nested "if…else" blocks to pick the direction. You can see that right has been arbitrarily assigned to a direction of 1, left to 2, and straight to 3.

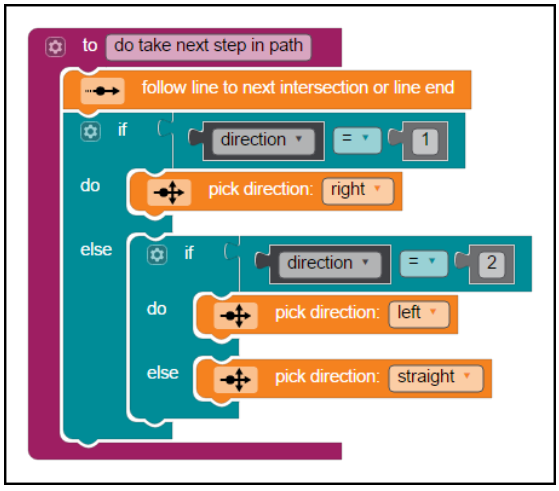

Figure 6

"Do Recall Saved Path" Subroutine

Figure 7 shows the blocks for the "do recall saved path subroutine. First, the LED is set to the color green, as this is what Bit/Evo is supposed to display when repeating the saved path. Next, there is a "count with" loop block identical to that discussed in the "do follow grid and save path" subroutine. However, the internals of the loops are different. Here, the array-processing block "get element – from array" is used to fetch the direction number of step *i* from the array *stepDirection*. Finally, the "do take next step in path" subroutine is called to actually force Bit/Evo to take the step.

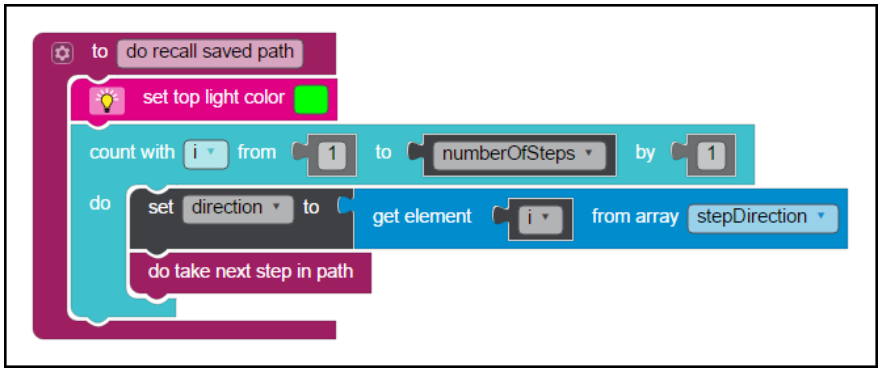

Figure 7

In conclusion, we have learned important concepts related to arrays in computer science. We have also had experience working with OzoBlockly's four array-processing blocks: declare array, get array length, set element in an array, and get element in an array.

Classroom Activity

After discussing the OzoBlockly program with the class, provide each student or student group with a copy of the *RepeatSavedPath.ozocode* file. Give them a program maintenance task as follows:

> *Users have requested that the program should also allow Bit to move backward during a step, in addition to the existing capability of moving right, left, or straight ahead. All four moves should be random and equally probable.*

The last page of this document contains an OzoMap that can be duplicated for student use with this lesson.