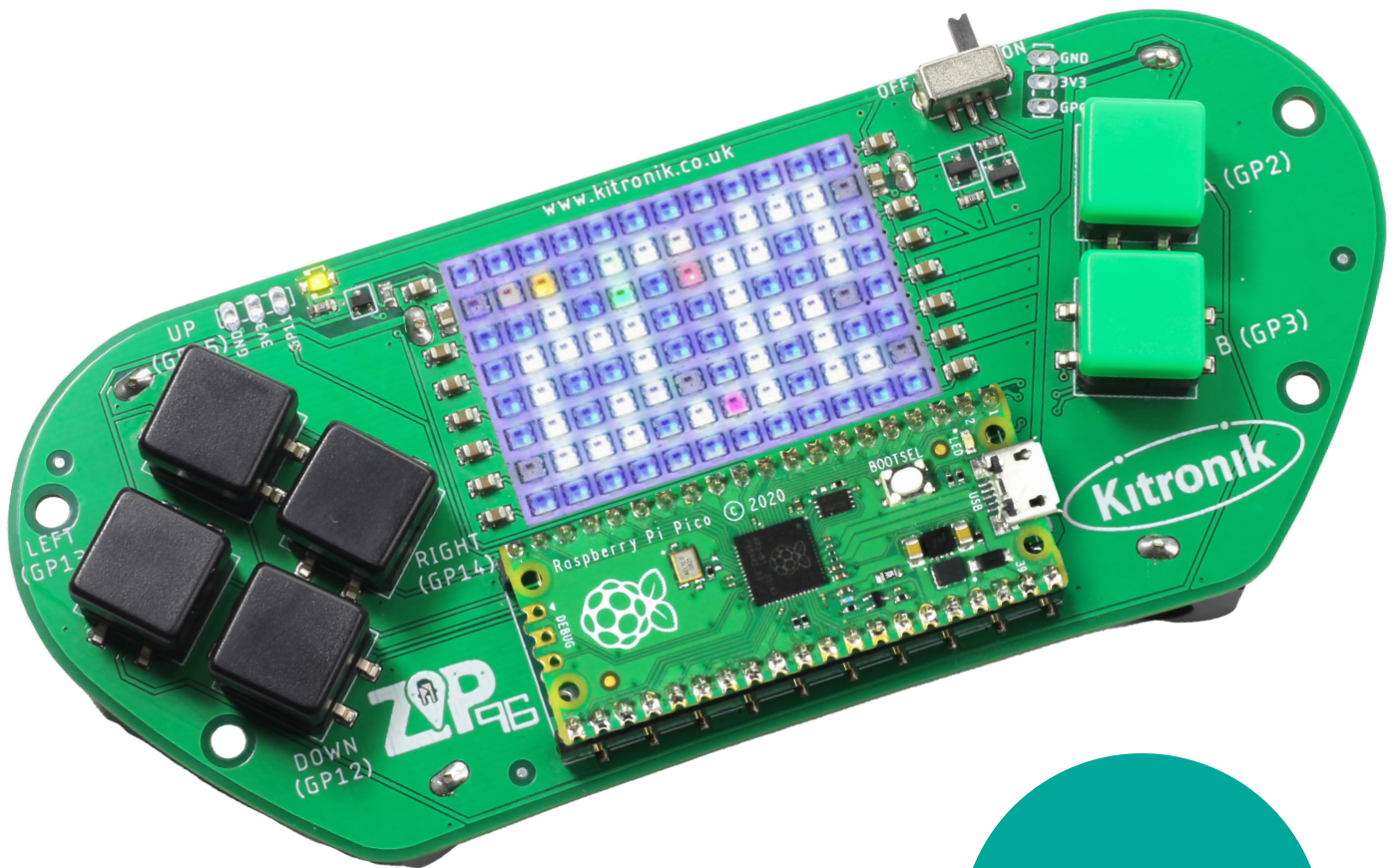


LESSON GUIDE TO THE PICO ZIP96



LESSON 10: THE A-MAZING GAME



This lesson includes curriculum mapping, practical exercises and a linked PowerPoint presentation.

14+

www.kitronik.co.uk

TEACH YOUR STUDENTS HOW TO CREATE GAMES WITH CODE!

LESSON 10

INTRODUCTION & SETUP



This is the tenth lesson in the 'A-mazing Game' series for Pico ZIP96. The final enemy to add to the game requires a more involved searching algorithm – this lesson goes through the theory and design stage, ready for coding in the last lesson.



CLASSROOM SETUP



Students will be working in pairs. They will need:

- Pen & Paper

The teacher will be writing on the board as well as demonstrating code on a projected board (if available).



Curriculum mapping

- Decompose problems and solve them using algorithms. Explore different searching and sorting algorithms.
- Decompose problems into smaller components, and make use of subroutines to build up well-structured programs.
- Learn how to handle strings, and simplify solutions by making use of lists and arrays.
- Apply iteration in program designs using loops.
- Make decisions in programs, making use of arithmetic, logic and Boolean expressions and operators. Create nested selection statements.

KEYWORDS:

ITERATION, WHILE LOOPS, FOR LOOPS, NESTED STATEMENTS, DESIGN PROGRAMS, SELECTION, CONTROL STRUCTURES, LOGIC, BOOLEAN, NESTED STATEMENTS, SUBROUTINES, PROCEDURES, FUNCTIONS, MODULES, LIBRARIES, VARIABLE SCOPE, WELL-DESIGNED PROGRAMS, STRINGS, LISTS, STRING HANDLING, ARRAYS (1D, 2D), MANIPULATION, ITERATION, ALGORITHMS, DECOMPOSITION, ABSTRACTION, DESIGN METHODS, TRACE TABLES, SEARCHING AND SORTING ALGORITHMS

WHAT IS OUR A-MAZING GAME?



Our A-Mazing Game is a maze-based game where the player runs through a maze collecting gems. Once you have collected all the gems in the maze then you have won! But it won't be that easy, as there are enemies in the maze trying to catch you before you collect all of the gems. The three enemies each have their own level of difficulty. The first enemy moves randomly. The second enemy tries to move towards you, without trying to avoid the maze walls. The third and final enemy is smart and moves around the maze walls finding the best path to you. When an enemy catches you, you lose a life and everyone in the game is reset back to their starting positions. You have three lives to collect all the gems, and if you don't, then you lose.



Curriculum mapping

Decompose problems and solve them using algorithms. Explore different searching and sorting algorithms

MOVESMART() SEARCH ALGORITHM

To create an Enemy that is smart in the way it moves around our game we'll need to use a basic Artificial Intelligence algorithm to select how the Enemy moves. To do this we can use a search algorithm to find a path from the Enemy to the Player.



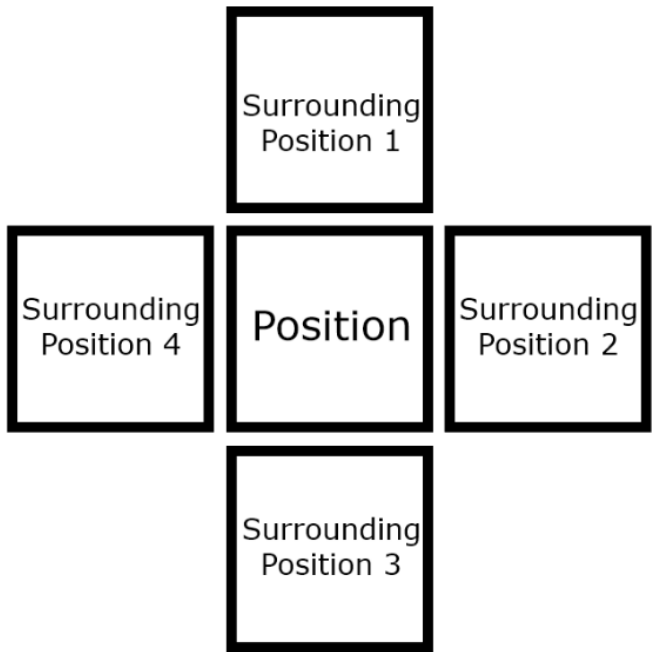
Students take note

The general idea of the search algorithm is to check all of the directions we can move from our starting position. We'll pretend to move in each of these directions and see if we reach the destination. If one of these is the destination, then we can just move there. If none of these are the destination, then we'll add all of these new position to a list and do the same checks we just did, on each of the positions in the list. When we add the new positions to the list we also store how many moves it took to get there. Then, when we find our destination we will have created a path from our starting point and can work backwards up the path to find out which move to make. This is done by moving from the destination in the direction which took the lowest number of moves to reach from our starting point, repeating this until we get back to our starting point.

MOVESMART() SEARCH ALGORITHM - KEY DEFINITIONS

Position is an x, y pair representing an LED on the ZIP96 screen.

Surrounding positions are the four positions directly around a position.



LESSON 10

Distance travelled is the number of moves taken to get from one position on the screen to another.

The toVisit array is an array of positions we need to visit, ordered from largest distance travelled at the start to smallest distance travelled at the end.

The visited array is a 2-dimensional (2D) array to represent all the positions on the screen, with a size of screenWidth by screenHeight.



Where should the algorithm start searching from – the Player's position or the Enemy's position? What are the reasons for your answer?

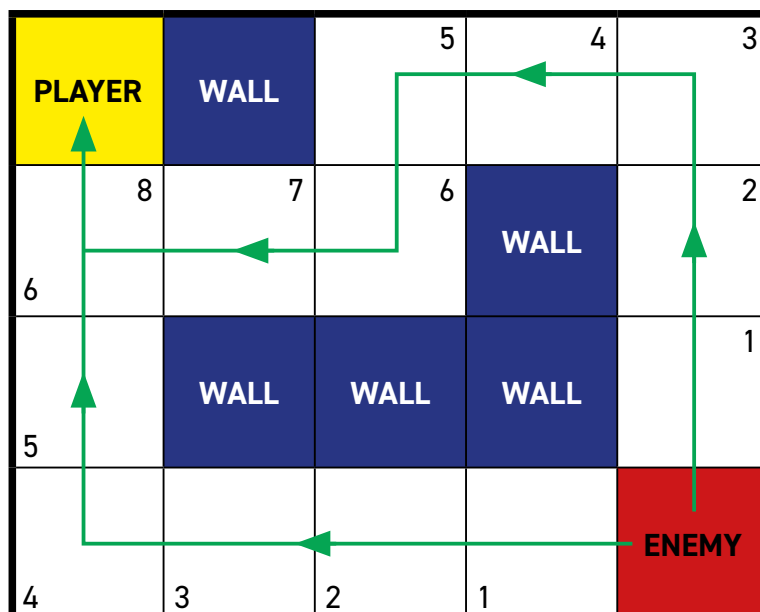


MOVESMART() SEARCH ALGORITHM - WHERE SHOULD WE START?



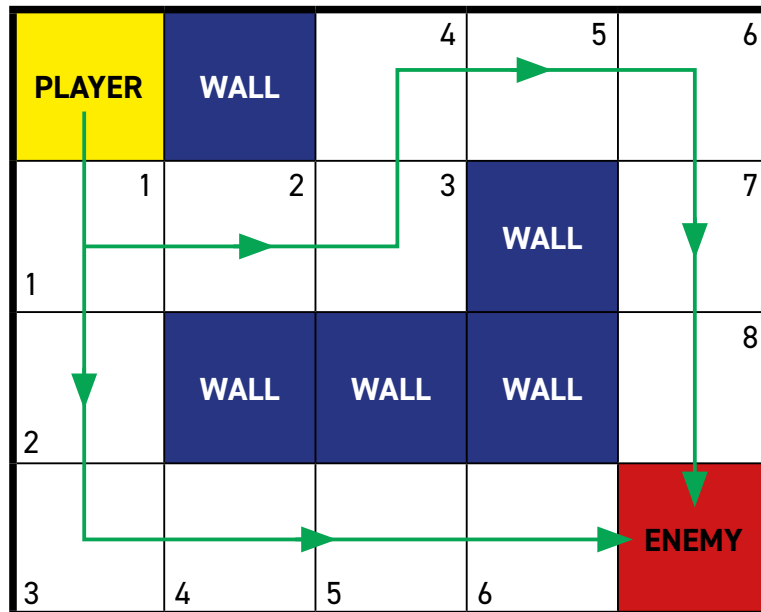
In our code we are going to set the starting point to the Player's position rather than the Enemy's position. This may seem odd, but there is a good reason. When starting from the Player, the destination of our search then becomes the Enemy. When we find the Enemy we'll only need to check its surrounding positions to find the move with the shortest path. This is because we started our search at the Player, and built our path back to the Enemy.

In the diagram below where we start searching from the Enemy, at the start of the path we have a distance travelled of one. So from this point all the moves available have the same distance travelled of one and we still don't know which provides the best path. This would mean if we started our search from the Enemy, then when we then found the Player we would need to work backwards following the shortest distance back to the Enemy.



LESSON 10

However, because we already know where the Player is, we can use this as the starting point in our search and create the path going in the opposite direction. Then when we get back to the Enemy it'll be easy to see from the Enemy's position which of the available moves is the quickest path to the Player. This means we don't need to follow the entire path to decide which move to make.



Curriculum mapping

Decompose problems into smaller components, and make use of subroutines to build up well-structured programs.

MOVESMART() SEARCH ALGORITHM - PSEUDOCODE



- create array toVisit of positions to visit
- create 2D array visited of positions visited
- add the start position to the toVisit array
- while there are still positions in toVisit:
 - set visiting as the position at end of the toVisit array and remove it from the array
 - add the visiting position to the visited array
 - if visiting is our destination position:
 - leave the loop
 - check the positions surrounding visiting as surrounding:
 - if we have not visited surrounding before:
 - add surrounding to the toVisit array with a distance travelled of visiting distance travelled + 1
- check the positions surrounding the start position for the shortest distance travelled and move to that position

MOVESMART() SEARCH ALGORITHM - PSUEDOCODE EXPLANATION

create array toVisit of positions to visit



Curriculum mapping

Learn how to handle strings, and simplify solutions by making use of lists and arrays.



First we setup the toVisit array which we'll use to store positions we haven't seen and checked yet. The number of elements inside toVisit will increase and decrease as we add new positions and later remove the positions to check them.

create 2D array visited of positions visited



The visited array is 2D, meaning we can store an array of arrays which we will use to store positions we have already seen and checked. We use this to allow us to check if we have already seen a position on the screen using the x and y coordinates of the position. We'll use this by having 8 arrays inside of the visited 2D array, one for each row on our screen. Inside these 8 arrays we'll have 12 positions, one for each column on our screen.

add the start position to the toVisit array



Then we add our starting position to the toVisit array, which for our code will be the Player's position.

while there are still positions in toVisit:



Curriculum mapping

Apply iteration in program designs using loops.

The majority of our logic sits inside of a loop. We will use a while loop and the condition will be to loop until there are no more positions left to check inside of our toVisit array.

set visiting as the position at end of the toVisit array and remove it from the array



The first thing we'll do inside our loop is get the position with the shortest distance travelled from our toVisit array and store it inside a variable visiting. We know that the position at the end of the array has the shortest distance travelled as we'll use the distance travelled to determine where in the array we add a position later on.

add the visiting position to the visited array



Then we'll add the visiting position into our visited array so that we can access it later if it's on the path from our starting position (the Player) to our destination position (the Enemy).

if visiting is our destination position:

leave the loop



LESSON 10



Curriculum mapping

Make decisions in programs, making use of arithmetic, logic and Boolean expressions and operators.
Created nested selection statements.

Next we want to check if the visiting position is our destination (the Enemy). If it is then we'll leave the while loop to reach the end of our code, where we'll decide where to move now that we know a path from our starting position (the Player).

check the positions surrounding visiting as surrounding:



Now we know the visiting position isn't our destination, we can check each of its surrounding position to see if we have already visited each of them or not.

if we have not visited surrounding before:

add surrounding to the toVisit array with a distance travelled of visiting distance travelled + 1



For each of the surrounding positions we only care if we haven't visited them. If we haven't visited any of the surrounding positions we want add it to the toVisit array, as a new position to visit. When adding it to the array we'll set its distance travelled to the distance travelled by visiting plus one, as we need to take one extra move to reach the new position.

check the positions surrounding the destination for the shortest distance travelled and move to that position



Here we are outside of our while loop meaning we have successfully found a path to our destination (the Enemy). We will now use the path to decide which direction to move in based on the distance travelled of the surrounding positions. For each of the surrounding positions we only care if we haven't visited them. If we haven't visited any of the surrounding positions we want add it to the toVisit array, as a new position to visit. When adding it to the array we'll set its distance travelled to the distance travelled by visiting plus one, as we need to take one extra move to reach the new position.

CONCLUSION

LESSON 10 CONCLUSION



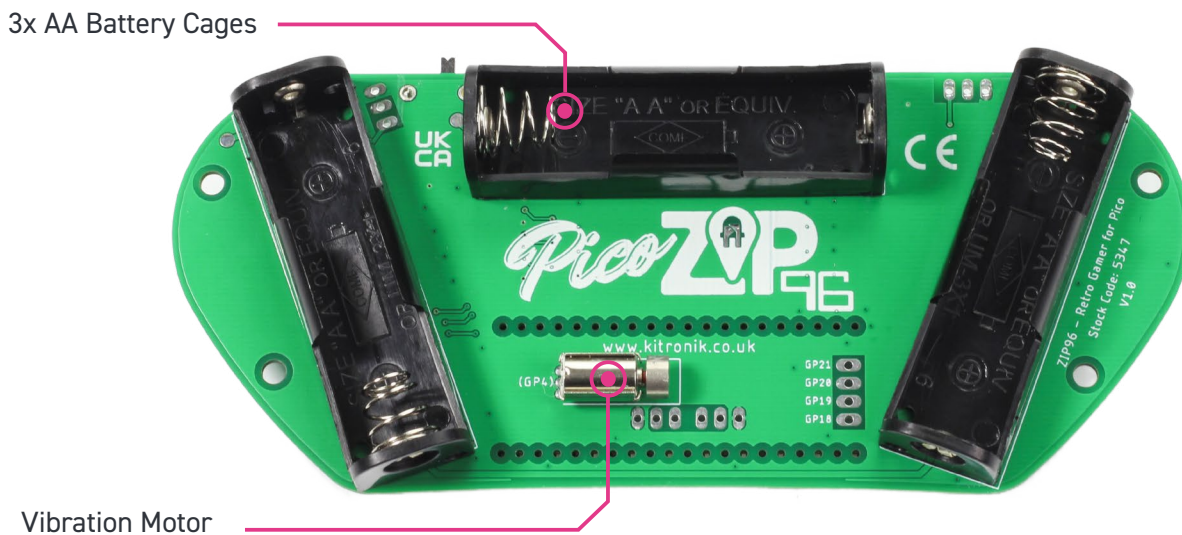
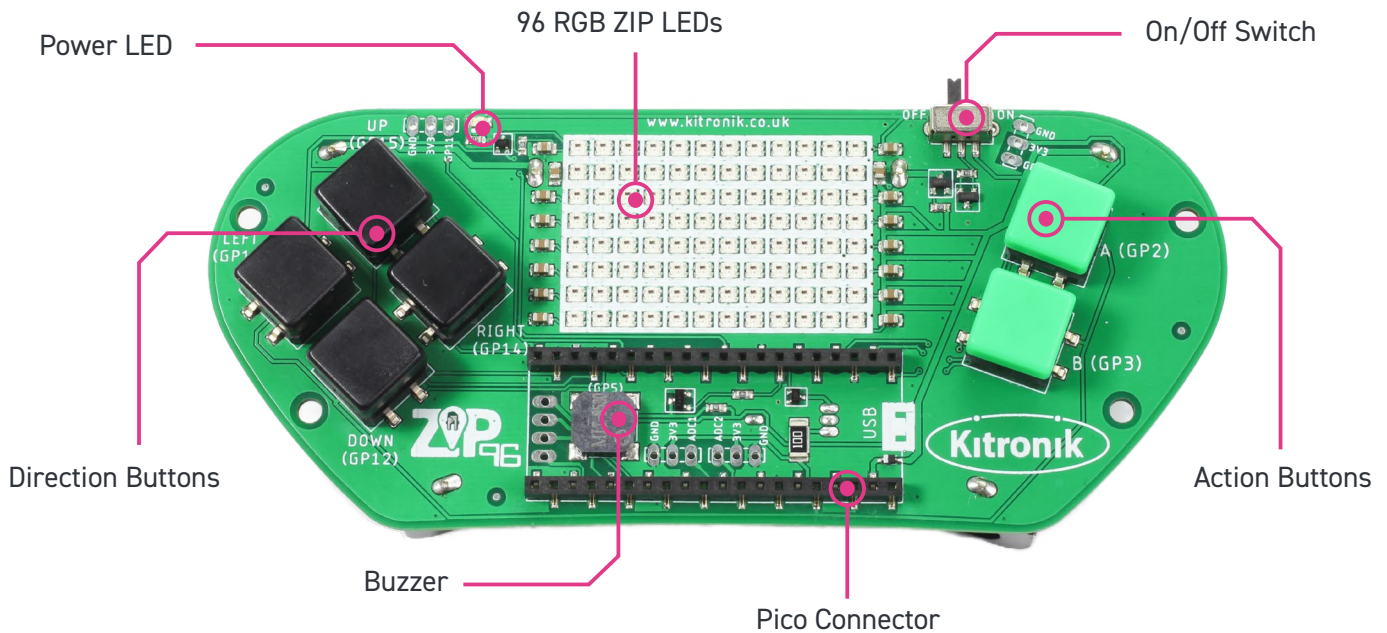
In this lesson, you have:

- Discussed how arrays and 2D arrays can be used
- Created pseudocode for basic 2D search algorithm
- Explained insertion sort
- Explained basic 2D search algorithm



THE ZIP96 IS A PROGRAMMABLE RETRO GAMEPAD FOR THE RASPBERRY PI PICO.

It features 96 colour addressable LEDs arranged in a 12 x 8 display, a buzzer for audio feedback, a vibration motor for haptic feedback, and 6 input buttons. It also breaks out GP1, GP11, ADC1 and ADC2, along with a set of 3.3V and GND for each, to standard 0.1" footprints. GP18 to 21 are also broken out on a 0.1" footprint underneath the Pico. The Pico is connected via low profile 20-way pin sockets.



T: 0115 970 4243

W: www.kitronik.co.uk

E: support@kitronik.co.uk



[kitronik.co.uk/twitter](https://twitter.com/kitronik)



[kitronik.co.uk/facebook](https://www.facebook.com/kitronik)



[kitronik.co.uk/youtube](https://www.youtube.com/kitronik)



[kitronik.co.uk/instagram](https://www.instagram.com/kitronik)



Designed & manufactured
in the UK by Kitronik



For more information on RoHs and CE please visit kitronik.co.uk/rohs-ce. Children assembling this product should be supervised by a competent adult. The product contains small parts so should be kept out of reach of children under 3 years old.