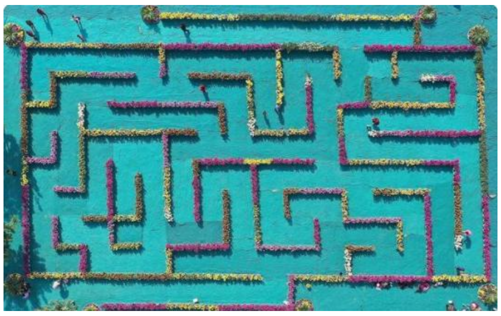
Lesson 5 2D Maze

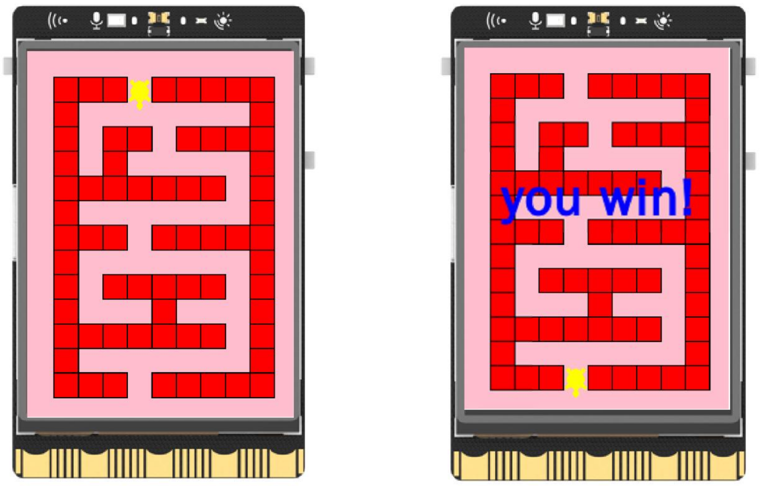
The maze game is a popular game that many of us played during our childhood. It can help to improve our spatial reasoning, logical thinking, patience, and perseverance, while also providing a lot of fun.

Let's design a maze game and experience it on the screen!



Task Objectives

We can obtain acceleration values through a three-axis accelerometer sensor, which can then be used to control the movement of the character on the map, allowing them to navigate and escape the maze.



Knowledge points

1. Understanding the Three-Axis Accelerometer Sensor

2. Learning How to Use the Pinpong Library to Read Acceleration Values

3. Learning How to Use the Turtle Library to Draw a Maze

Material List

Hardware List:

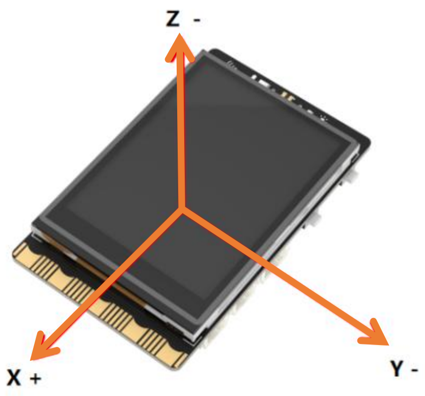
UNIHIKER x1	Type-C & Micro Dual-Use USB Cable x1

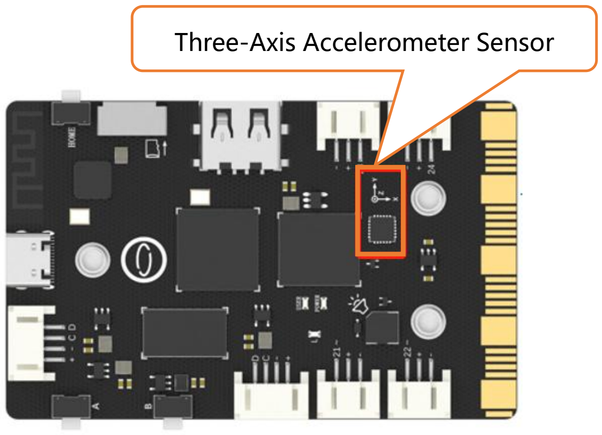
Software Preparation: Mind+ Programming Softwarex1

Knowledge background

1. What is Gravity Acceleration and an Accelerometer Sensor?

Gravity acceleration is the acceleration experienced by an object near the surface of the Earth due to the force of gravity, also known as freefall acceleration, which is represented by the symbol "g." An accelerometer sensor is a device that can measure acceleration and convert it into an electrical signal. The accelerometer sensor on the board can measure acceleration in three directions: X, Y, and Z. The X-axis is oriented in the direction of the side with the gold finger, the Y-axis is oriented in the direction of the side with the Home button, and the Z-axis is perpendicular to the board, with the positive direction facing the back of the screen.





2. Reading Acceleration Values with the Pinpong Library

The "get_x()", "get_y()", and "get_z()" methods of the "GD32Sensor_acc" class in the "pinpong.extension.unihiker" module of the Pinpong library can respectively obtain the acceleration values in the x, y, and z directions. Since an accelerometer object has already been instantiated in the file, we can directly use the command "accelerometer.get_x()", "accelerometer.get_y()", and "accelerometer.get_z()" to obtain the acceleration values in the respective directions. Of course, before doing this, we need to first import the relevant files of the Pinpong library and initialize the board.

```
from pinpong.board import Board  # Import the Board module from the pinpong.board package
from pinpong.extension.unihiker import *  # Import all modules from the pinpong.extension.unihiker package


Board().begin()  # Initialize the board and select the port number (auto-detected if not specified)
x = accelerometer.get_x()  # Get the X-axis acceleration value
y = accelerometer.get_y()  # Get the Y-axis acceleration value
```

In this sentence, "x" and "y" are variables used to store the detected acceleration values in the x and y-axis directions, respectively.

3. Common functions for screen control in the turtle library

There are many functions in the turtle library for screen control, but we only use a portion of them. When programming, after importing the library with "import turtle", we can use the format "turtle.function_name()" to achieve the desired functionality.

(1) The function "bgpic()" is used to set the background color of the drawing window (canvas)

With the "bgpic()" function, we can set the background color for the current window screen.

```
turtle.bgcolor("pink") # Set up the window background color
```

Here, "pink" refers to the color pink. This color can also be represented in three different ways: RGB values, hexadecimal values, and fixed values.

(2) The "clear()" function clears the window

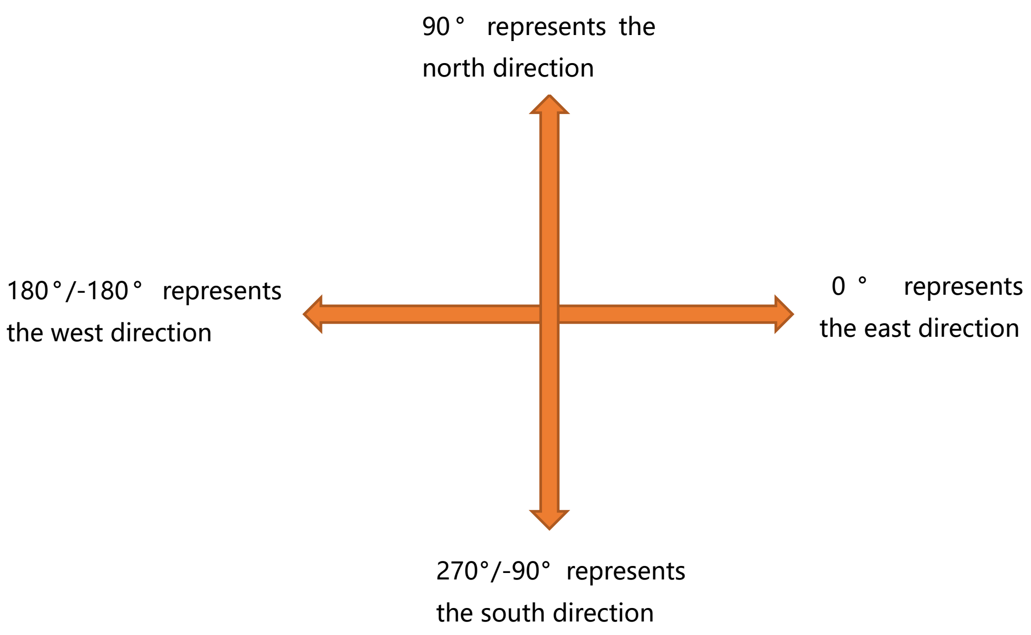
Using the "clear()" function, we can clear the contents on the window.

```
turtle.clear() # Clear the window
```

4. Common functions for pen movement in the turtle library

(1) The function setheading() sets the current orientation of the pen

We can use the setheading() function to make the turtle point in any direction. 0° represents the east direction, 90° represents the north direction, 180°/-180° represents the west direction, and 270°/-90° represents the south direction.


90° represents the north direction
180°/-180° represents the west direction
0° represents the east direction
270°/-90° represents the south direction

```
turtle.setheading(90) # Sets the orientation of the pen to the direction of 90 degrees
```

The value "90" refers to the specific angle of orientation to which the pen should point. By changing the value of the angle of orientation, the direction of the pen can be altered.

(2) The write() function is used to write text

We can use the write() function to make the pen write text on the window screen.

```
turtle.write('you win!', align='center', font=('Microsoft YaHei', 30)) # Writes 'you win!' in the cen
ter of the screen with a font size of 30 using the Microsoft YaHei font
```

In this case, "you win!" refers to the specific text content that will be written, align refers to the alignment method, with "center" indicating center alignment, and font refers to the font type, which is the Microsoft YaHei font with a size of 30.

Hands-on practice

Task Description 1: Drawing the Maze

Draw a maze with intricate and winding paths on the screen.

1. Hardware setup

STEP 1: Connect the UNIHIKER to the computer via a USB cable.

2. program coding

STEP 1: Create and Save Project File

Launch Mind+, save the project as "005 2D Maze".

STEP 2: Create and Save Python File

Create a Python program file named "protract.py" and double-click to open it.

STEP 3: Programming 1

In this task, we will draw a maze. Since the process of drawing a maze is quite complex, we will write it in a separate program file called "protract" so that we can easily call it later.

(1) Import necessary libraries

Here, we need to use the turtle library to draw the maze, so we need to import it first.

```
import turtle # Import the turtle library
```

(2) Create a graphics window

To display the maze on the screen, we need to create a graphics window that is the same size as the screen, and then create a canvas in the graphics window, set the appropriate size and background color, and set the screen delay to 0 to avoid lag.

```
# Set up the window size and background color

turtle.setup(240, 320)

turtle.bgcolor("pink")


# Set the screen delay to 0 to avoid lag

turtle.delay(0)
```

(3) Design the maze route plan

For drawing the maze, we will proceed in two steps. First, we set a small square with a width and height of 20 pixels as a wall unit. Then, we use this small square as a unit to set the entire maze. Since the screen is 240 pixels wide and 320 pixels high, we can draw 13 squares horizontally (11 whole squares displayed on the screen + 1 half square on each side), and also 13 squares vertically.

Here, we use "0" and "1" to represent the presence or absence of a small square, and represent the maze route through nested lists.

```
# Maze list

maze_list = [
    [0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0], # row 0
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0], # row 1
    [0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0], # row 2
    [0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0], # row 3
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0], # row 4
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0], # row 5
    [0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0], # row 6
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0], # row 7
    [0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0], # row 8
    [0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0], # row 9
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0], # row 10
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0], # row 11
    [0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0], # row 12
]
```

(4) Create a maze class and set the basic drawing properties in the class

When programming, we first create a maze class, and then create maze objects through class instantiation. In the maze class, we first need to define the size of a grid and set some basic drawing properties.

```
class Maze(turtle.Turtle):
    # Define the Maze class
    size = 20 # Set the size of each wall in the maze to 20 pixels


    def __init__(self, maze_list):
        # Initialize the Maze class
        turtle.Turtle.__init__(self) # Unbind the methods in the Maze class and inherit the methods in the Turtle class
        self.maze_list = maze_list # Set the maze list
        self.hideturtle() # Hide the turtle pen to speed up the drawing process
        self.speed(0) # Set the turtle movement (drawing) speed to the fastest
        self.draw_walls() # Draw the entire maze
```

(5) Create a function in the maze class to draw a single square

Furthermore, we need to define a function in the class to draw a single square, which can be called when drawing the entire maze.

```
# Draw a wall in the maze
def draw_wall(self):
    self.pendown() # Put the pen down
    self.begin_fill() # Begin filling the wall
    self.fillcolor('red') # Set the fill color to red
    # Draw a horizontal line with a distance of 20 pixels, then turn right 90 degrees and repea
t 4 times to form a square
    for i in range(4):
        self.forward(20) # Move forward 20 pixels
        self.right(90) # Turn right 90 degrees
    self.end_fill() # End the fill
    self.penup() # Pick up the pen
```

(6) Create a function in the maze class to draw the entire maze

Here, we can draw the entire maze by calling the function that draws a single square and combining it with the maze layout.

```
# Draw the walls of the entire maze
def draw_walls(self):
    self.penup() # Pick up the pen
    self.goto(-120, 120) # Go to the starting position
    # Draw the walls, with nested loops for rows and columns (the entire maze is composed o
f 13 walls in length and width)
    for row in range(13): # Row loop
        for col in range(13): # Column loop
            # If the value in the maze_list is 1, draw a wall
            if self.maze_list[row][col] == 1:
                self.draw_wall() # Draw a single wall
            self.goto(20 * (col + 1) - 130, 130 - 20 * row) # Move right one column
        self.goto(-130, 130 - 20 * (row + 1)) # Move down one row
```

Tips: The complete example program is as follows:

```
import turtle # Import the turtle library

# Set up the window size and background color
turtle.setup(240, 320)
turtle.bgcolor("pink")

# Set the screen delay to 0 to avoid lag
turtle.delay(0)

# Maze list
maze_list = [
  [0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0], # row 0
  [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0], # row 1
  [0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0], # row 2
  [0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0], # row 3
  [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0], # row 4
  [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0], # row 5
  [0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0], # row 6
  [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0], # row 7
  [0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0], # row 8
  [0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0], # row 9
  [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0], # row 10
  [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0], # row 11
  [0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0], # row 12
]

class Maze(turtle.Turtle):
    # Define the Maze class
    size = 20 # Set the size of each wall in the maze to 20 pixels

    def __init__(self, maze_list):
        # Initialize the Maze class
        turtle.Turtle.__init__(self) # Unbind the methods in the Maze class and inherit the method
s in the Turtle class
        self.maze_list = maze_list # Set the maze list
        self.hideturtle() # Hide the turtle pen to speed up the drawing process
        self.speed(0) # Set the turtle movement (drawing) speed to the fastest
        self.draw_walls() # Draw the entire maze

    # Draw a wall in the maze
    def draw_wall(self):
        self.pendown() # Put the pen down
        self.begin_fill() # Begin filling the wall
        self.fillcolor('red') # Set the fill color to red
        # Draw a horizontal line with a distance of 20 pixels, then turn right 90 degrees and repe
at 4 times to form a square
        for i in range(4):
            self.forward(20) # Move forward 20 pixels
            self.right(90) # Turn right 90 degrees
        self.end_fill() # End the fill
        self.penup() # Pick up the pen

    # Draw the walls of the entire maze
    def draw_walls(self):
        self.penup() # Pick up the pen
        self.goto(-120, 120) # Go to the starting position
        # Draw the walls, with nested loops for rows and columns (the entire maze is composed
of 13 walls in length and width)
        for row in range(13): # Row loop
            for col in range(13): # Column loop
                # If the value in the maze_list is 1, draw a wall
                if self.maze_list[row][col] == 1:
                    self.draw_wall() # Draw a single wall
                self.goto(20 * (col + 1) - 130, 130 - 20 * row) # Move right one column
            self.goto(-130, 130 - 20 * (row + 1)) # Move down one row
```

STEP 4: Programming 2

In the aforementioned "protract" program, we only created a maze class, but we cannot directly see the effect. Therefore, next, we will create a "main1.py" Python file, double-click to open and write the program, and observe the effect of the drawn maze.

(1) Import necessary libraries

For this task, in order to create a maze object and maintain the window interface, we need to import the previously written "protract" program file and the turtle library file.

```
import turtle # Import the turtle library
import protract # Import the protract module
```

(2) Generate the maze object and

Finally, we instantiated a maze object using a class and ended the drawing by keeping the window interface.

```
protract.Maze(protract.maze_list) # Instantiate the Maze class to create a maze object

turtle.done() # End the drawing and keep the window open
```

Tips: The complete example program is as follows:

```
import turtle # Import the turtle library
import protract # Import the protract module

protract.Maze(protract.maze_list) # Instantiate the Maze class to create a maze object
turtle.done() # End the drawing and keep the window open
```

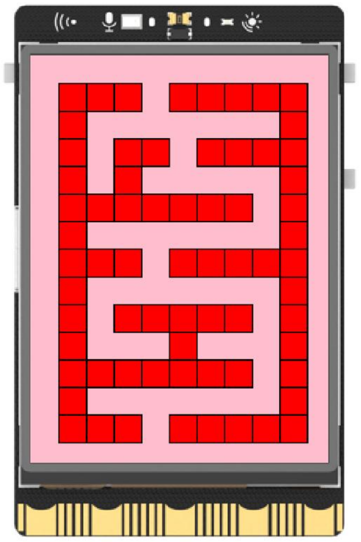
3. Running the Program

STEP 1: Remote connection to the UNIHIKER

STEP 2: Click the "Run" button in the upper right corner to execute the "main1.py" program.

STEP 3: Observing the Results

Observing the output, you can see that first a pink background appears, followed by small red squares appearing one by one from the upper left corner. Finally, they are pieced together to form a complex maze.



Task Description 2: Detecting Acceleration Values

In the previous task, we completed the drawing of the maze. In order to control the movement of the character on the maze using acceleration values, we need to first detect the changes in acceleration values in different directions.

1. program coding

STEP 1: Create and Save Project Files

Create a Python program file named "main2.py" and double-click to open it.

STEP 2: Programming

(1) Import necessary libraries

As we will be displaying the acceleration values on the screen, we need to first import the unihiker library and the time library. Additionally, to detect acceleration, we also need to import the Pinpong-related libraries and initialize the UNIHIKER board.

```
from unihiker import GUI  # Import the unihiker library
import time  # Import the time library

from pinpong.board import Board  # Import the Board module from the pinpong.board packa
ge
from pinpong.extension.unihiker import *  # Import all modules from the pinpong.extension.u
nihiker package

Board().begin()  # Initialize the board and select the port number (auto-detected if not specifie
d)
```

(2) Instantiate the GUI class and display the initial acceleration values on the x and y axes on the screen

When displaying the initial acceleration value, we can leave it blank by not inputting a numerical value.

```
gui = GUI()  # Instantiate the GUI class and create a gui object

value_1 = gui.draw_text(x=65, y=65, text='X-axis acceleration:', font_size=13)  # Display text
value_2 = gui.draw_text(x=65, y=178, text='Y-axis acceleration:', font_size=13)  # Display text
value_x = gui.draw_text(x=85, y=122, text='', font_size=13)  # Display initial X-axis acceleration
 value
value_y = gui.draw_text(x=85, y=240, text='', font_size=13)  # Display initial Y-axis acceleration
 value
```

(3) Loop to detect acceleration values and update the readings

Next, we will continuously detect the acceleration values in the x and y directions every 1 second and update them in real-time on the screen.

```
while True:
    x = accelerometer.get_x()  # Get the X-axis acceleration value
    y = accelerometer.get_y()  # Get the Y-axis acceleration value
    value_x.config(text=x)  # Update the display of the X-axis acceleration value
    value_y.config(text=y)  # Update the display of the Y-axis acceleration value
    time.sleep(1)
    print(x, y)
```

Tips: The complete example program is as follows:

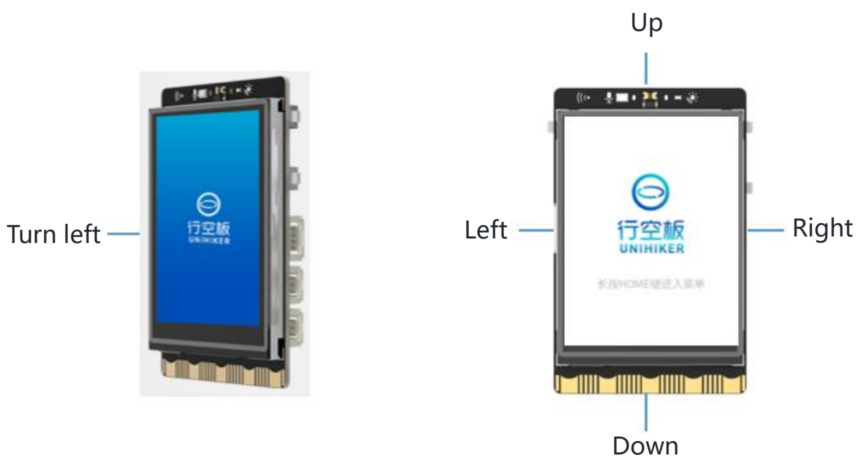
```
from unihiker import GUI  # Import the unihiker library
import time  # Import the time library

from pinpong.board import Board  # Import the Board module from the pinpong.board package
from pinpong.extension.unihiker import *  # Import all modules from the pinpong.extension.unihiker package

Board().begin()  # Initialize the board and select the port number (auto-detected if not specified)

gui = GUI()  # Instantiate the GUI class and create a gui object

value_1 = gui.draw_text(x=65, y=65, text='X-axis acceleration:', font_size=13)  # Display text
value_2 = gui.draw_text(x=65, y=178, text='Y-axis acceleration:', font_size=13)  # Display text
value_x = gui.draw_text(x=85, y=122, text='', font_size=13)  # Display initial X-axis acceleration value
value_y = gui.draw_text(x=85, y=240, text='', font_size=13)  # Display initial Y-axis acceleration value

while True:
    x = accelerometer.get_x()  # Get the X-axis acceleration value
    y = accelerometer.get_y()  # Get the Y-axis acceleration value
    value_x.config(text=x)  # Update the display of the X-axis acceleration value
    value_y.config(text=y)  # Update the display of the Y-axis acceleration value
    time.sleep(1)
    print(x, y)
```

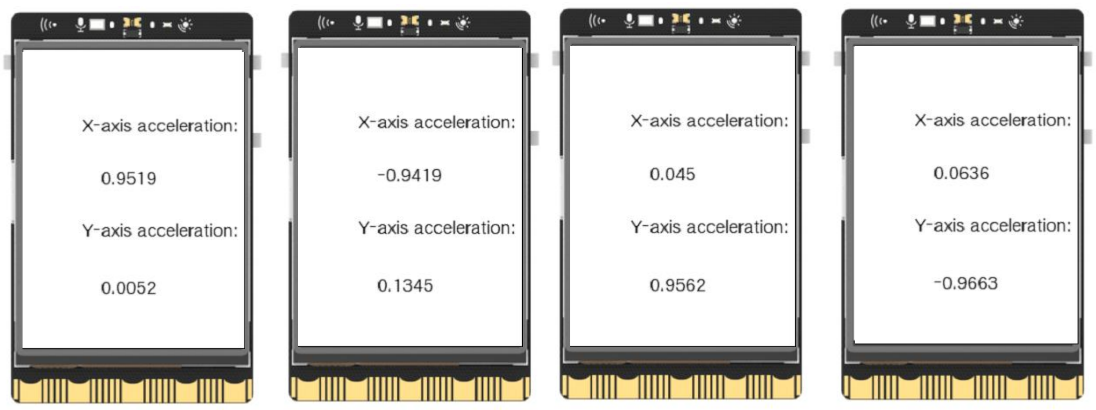
2. Running the Program

STEP 1: Remote connection to the UNIHIKER

STEP2: Running the Program and Observing the Results

After clicking run, tilt the board in four different directions from its horizontal position. Since the board only experiences one gravitational acceleration when it is stationary, and the gravitational acceleration always points towards the ground, you can see that when the board tilts downwards, the acceleration value in the x direction continuously increases, and when it is vertical, the value is around 1; when the board tilts upwards, the acceleration value in the x direction continuously decreases, and when it is vertical, the value is around -1; when the board tilts to the left, the acceleration value in the y direction continuously increases, and when it is vertical, the value is around 1; when the board tilts to the right, the acceleration value in the y direction continuously decreases, and when it is vertical, the value is around -1.




X-axis acceleration:
0.9519
Y-axis acceleration:
0.0052

X-axis acceleration:
-0.9419
Y-axis acceleration:
0.1345

X-axis acceleration:
0.045
Y-axis acceleration:
0.9562

X-axis acceleration:
0.0636
Y-axis acceleration:
-0.9663

Task Description 3: Maze Navigation

Next, we will combine the drawn maze with the acceleration detection to control the character's movement through the maze based on the different orientations of the acceleration.

1. program coding

STEP1: Create and Save Python File

Create a Python program file named "main3.py" and double-click to open it.

STEP2: Programming

(1) Import necessary libraries

For this task, we will create a turtle character on the drawn maze as the player and control its movement using the onboard three-axis accelerometer. Therefore, we need to first import the turtle library, the protract program file for drawing the maze, the time library, and the Pinpong library, and initialize the board.

```
import turtle # Import the turtle library
import time # Import the time library
import protract # Import the protract module
from pinpong.board import Board # Import the Board module from the pinpong.board package
from pinpong.extension.unihiker import * # Import all modules from the pinpong.extension.unihiker package

Board().begin() # Initialize the board and select the board type and port number. If no input is provided, the board is automatically recognized.
```

(2) Drawing the Maze

As our game takes place on a maze, we need to first instantiate the "Maze" class created in the protract program file from the previous task to generate a maze object.

```
protract.Maze(protract.maze_list) # Instantiate the Maze class and create a maze object
```

(3) Creating the Player Class and Setting its Attributes and Functions

For the player character on the maze, we generate a player object by first creating a player class and then instantiating it.

At the same time, we need to set some basic attributes and functions for the player in the created player class.

Here, we set the initial and final positions of the player, set the map where the player character is located as the maze map, set the movement speed to the fastest, display the player as a turtle on the screen after moving to the starting point of the maze, and set the color and initial direction of the turtle player.

```
class Player(turtle.Turtle): # Create the Player class
    def __init__(self, maze_list, start_m, start_n, end_m, end_n):
        turtle.Turtle.__init__(self) # Initialize the parent class
        self.m = start_m # Set the starting row number
        self.n = start_n # Set the starting column number

        self.end_m = end_m # Set the ending row number
        self.end_n = end_n # Set the ending column number

        self.maze_list = maze_list # Set the maze list
        self.hideturtle() # Hide the turtle pen
        self.speed(0) # Set the turtle movement (drawing) speed to the fastest
        self.penup() # Lift the turtle pen

        self.goto(self.n * 20 - 120, 120 - self.m * 20) # Move the player to the corresponding posit
ion
        self.shape('turtle') # Set the player shape to a turtle
        self.color('yellow') # Set the player color to yellow

        self.setheading(270) # Set the initial direction of the player
        self.showturtle() # Show the player
```

(4) Set the movable range in the player class

Due to the game mechanism, we need to set that the player can only move within the maze's passages.

```
def can_move(self, m, n): # Define the positions where the player can move, i.e. only allow mov
```

```
ement in the maze passages
        if 0 <= m and m <= 12: # If it's between the 0th and 12th row
            print(m,n) # Print
            return self.maze_list[m][n] == 0 # Return 0, indicating that there is no wall in row m an
d column n, and the player can move
```

(5) In the player class, we need to create a method to set the position change during movement

After the player is able to move, we need to set the position change that occurs during movement.

```
    def move(self, m, n): # Define the player's movement
        self.m = m # Define the row number
        self.n = n # Define the column number
        self.goto(self.n * 20 - 120, 120 - self.m * 20) # Move to that position
        self.reach_exit(m, n) # Check if the player has reached the end
```

(6) Setting movement in different directions in the player class

Afterwards, we also need to set the player's movement in all four directions.

```
def go_up(self): # Move upward
        if self.can_move(self.m - 1, self.n): # If it's possible to move upward
            self.setheading(90) # Set the direction
            self.move(self.m - 1, self.n) # Move upward
    def go_down(self): # Move downward
        if self.can_move(self.m + 1, self.n): # If it's possible to move downward
            self.setheading(270) # Set the direction
            self.move(self.m + 1, self.n) # Move downward
    def go_left(self): # Move to the left
        if self.can_move(self.m, self.n - 1): # If it's possible to move to the left
            self.setheading(180) # Set the direction
            self.move(self.m, self.n - 1) # Move to the left
    def go_right(self): # Move to the right
        if self.can_move(self.m, self.n + 1): # If it's possible to move to the right
            self.setheading(0) # Set the direction
            self.move(self.m, self.n + 1) # Move to the right
```

(7) Setting endpoint detection in the player class

When the player's coordinates match the endpoint coordinates, we determine that the game is over, display a winning message on the screen, and restart the game.

```
    def reach_exit(self, m, n): # Check if the player has reached the end
        if m == self.end_m and n == self.end_n: # If it has reached the end
        text = turtle.Turtle() # Create a text object
        text.hideturtle() # Hide the pen
        text.penup() # Lift the pen
        text.goto(0,0) # Move to the origin
        text.color('blue') # Set the color to blue
        text.write('You Win!', align="center", font=('Helvetica', 30, 'normal')) # Write "You Win!"
        time.sleep(2) # Delay for two seconds
        text.clear() # Clear the text
        self.restart() # Restart the game
```

(8) Setting up the restart mechanism in the player class

After reaching the endpoint, we make the player return to the initial position.

```
    def restart(self): # Restart the game
        self.goto(-20, 120) # Move to the (-20,120) coordinate point
        self.m = 0 # Set the starting row number
        self.n = 5 # Set the starting column number
        self.end_m = 12 # Set the ending row number
        self.end_n = 5 # Set the ending column number
```

(9) Generating the player

Afterwards, we generate a player character by instantiating the class.

```
player = Player(protract.maze_list, 0, 5, 12, 5) # Instantiate the Player class and create a player object
```

(10) Setting Gyroscope Reference Value

In order to determine the orientation based on the acceleration values on the x and y axes in the future, we need to establish a reference value for comparison in advance. After the detection and analysis performed in task 2, we have selected "0.4" as the reference value.

```
thread_value = 0.4 # Define a threshold value
```

(11) Looping through the acceleration values and performing detection

Finally, we continuously obtain the values of the acceleration on the x and y axes of the gyroscope in an infinite loop, compare them with the reference value, and determine the direction of movement accordingly.

```
while True:
    x = accelerometer.get_x() # Get the x-axis acceleration value
    y = accelerometer.get_y() # Get the y-axis acceleration value
    time.sleep(0.15)
    if x > thread_value : # If the x-axis acceleration value is greater than the threshold value
```

```
        player.go_down() # Move downward
        time.sleep(0.1)
    if x < -thread_value: # If the x-axis acceleration value is less than the threshold value
        player.go_up() # Move upward
        time.sleep(0.1)
    if y > thread_value : # If the y-axis acceleration value is greater than the threshold value
        player.go_left() # Move to the left
        time.sleep(0.1)
    if y < -thread_value : # If the y-axis acceleration value is less than the threshold value
        player.go_right() # Move to the right
        time.sleep(0.1)
```

Tips: The complete example program is as follows:

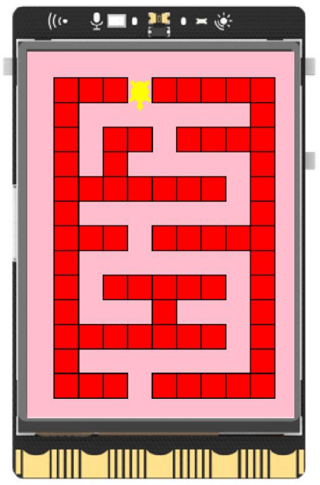
```
import turtle # Import the turtle library
import time # Import the time library
import protract # Import the protract module
from pinpong.board import Board # Import the Board module from the pinpong.board packa
ge
from pinpong.extension.unihiker import * # Import all modules from the pinpong.extension.u
nihiker package

Board().begin() # Initialize the board and select the board type and port number. If no input is
 provided, the board is automatically recognized.
thread_value = 0.4 # Define a threshold value
protract.Maze(protract.maze_list) # Instantiate the Maze class and create a maze object

class Player(turtle.Turtle): # Create the Player class
    def __init__(self, maze_list, start_m, start_n, end_m, end_n):
        turtle.Turtle.__init__(self) # Initialize the parent class
        self.m = start_m # Set the starting row number
        self.n = start_n # Set the starting column number

        self.end_m = end_m # Set the ending row number
        self.end_n = end_n # Set the ending column number

        self.maze_list = maze_list # Set the maze list
        self.hideturtle() # Hide the turtle pen
        self.speed(0) # Set the turtle movement (drawing) speed to the fastest
        self.penup() # Lift the turtle pen

        self.goto(self.n * 20 - 120, 120 - self.m * 20) # Move the player to the corresponding position
        self.shape('turtle') # Set the player shape to a turtle
        self.color('yellow') # Set the player color to yellow

        self.setheading(270) # Set the initial direction of the player
        self.showturtle() # Show the player

    def can_move(self, m, n): # Define the positions where the player can move, i.e. only allow movement in the maze passages
        if 0 <= m and m <= 12: # If it's between the 0th and 12th row
            print(m,n) # Print
            return self.maze_list[m][n] == 0 # Return 0, indicating that there is no wall in row m and column n, and the player can move
    def move(self, m, n): # Define the player's movement
        self.m = m # Define the row number
        self.n = n # Define the column number
        self.goto(self.n * 20 - 120, 120 - self.m * 20) # Move to that position
        self.reach_exit(m, n) # Check if the player has reached the end
    def go_up(self): # Move upward
        if self.can_move(self.m - 1, self.n): # If it's possible to move upward
            self.setheading(90) # Set the direction
            self.move(self.m - 1, self.n) # Move upward
    def go_down(self): # Move downward
        if self.can_move(self.m + 1, self.n): # If it's possible to move downward
            self.setheading(270) # Set the direction
            self.move(self.m + 1, self.n) # Move downward
    def go_left(self): # Move to the left
        if self.can_move(self.m, self.n - 1): # If it's possible to move to the left
            self.setheading(180) # Set the direction
            self.move(self.m, self.n - 1) # Move to the left
    def go_right(self): # Move to the right
        if self.can_move(self.m, self.n + 1): # If it's possible to move to the right
            self.setheading(0) # Set the direction
            self.move(self.m, self.n + 1) # Move to the right
    def reach_exit(self, m, n): # Check if the player has reached the end
        if m == self.end_m and n == self.end_n: # If it has reached the end
            text = turtle.Turtle() # Create a text object
            text.hideturtle() # Hide the pen
            text.penup() # Lift the pen
            text.goto(0,0) # Move to the origin
            text.color('blue') # Set the color to blue
            text.write('You Win!', align="center", font=('Helvetica', 30, 'normal')) # Write "You Win
!"
            time.sleep(2) # Delay for two seconds
            text.clear() # Clear the text
            self.restart() # Restart the game
    def restart(self): # Restart the game
        self.goto(-20, 120) # Move to the (-20,120) coordinate point
        self.m = 0 # Set the starting row number
        self.n = 5 # Set the starting column number
        self.end_m = 12 # Set the ending row number
        self.end_n = 5 # Set the ending column number

player = Player(protract.maze_list, 0, 5, 12, 5) # Instantiate the Player class and create a player
 object

while True:
    x = accelerometer.get_x() # Get the x-axis acceleration value
    y = accelerometer.get_y() # Get the y-axis acceleration value
    time.sleep(0.15)
    if x > thread_value : # If the x-axis acceleration value is greater than the threshold value
        player.go_down() # Move downward
        time.sleep(0.1)
    if x < -thread_value: # If the x-axis acceleration value is less than the threshold value
        player.go_up() # Move upward
        time.sleep(0.1)
    if y > thread_value : # If the y-axis acceleration value is greater than the threshold value
        player.go_left() # Move to the left
        time.sleep(0.1)
    if y < -thread_value : # If the y-axis acceleration value is less than the threshold value
        player.go_right() # Move to the right
        time.sleep(0.1)
```

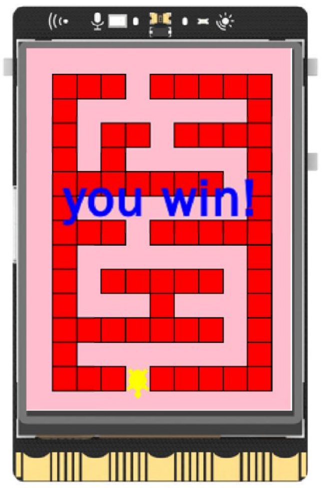
2. Running the Program

STEP 1: Remote connection to the UNIHIKER

STEP2: Running the Program and Observing the Results

After clicking "run", upon observing the screen, we can see that a maze is first drawn, followed by a small turtle appearing at the exit of the maze. Then, we can use the UNIHIKER to control the turtle to navigate through the maze by tilting it in different directions.





Challenge Yourself

Please try modifying the "0,1" in the maze route list to design your own maze and have some fun playing the game!