Snake

Programmierprojekt des SS18
unter

Thorsten Wagener

Inhaltsverzeichnis

Gruppenmitglieder:

-Moritz Joachim
-Daniel Friesen
-Hannah Lenz
-Adrian Baron

Präsentation der Idee:

Unser Ziel war ein Snake-Spiel auf einer selbst erstellten LED-Matrix zu spielen, die aus einem individuell ansteuerbaren LED-Streifen besteht.

Probleme und Lösungen:

Die Ansteuerung des Streifens erwies sich als schwieriger als geplant. Da wir einen bestimmten Abstand zwischen den Pixeln der Matrix haben wollten, beschlossen wir den Strip komplett zu zerschneiden und jeden Pixel einzeln zu verlöten bzw. zu befestigen.
Als die Matrix soweit fertig gestellt war führten wir einige Tests aus und mussten feststellen, dass aus für uns unerklärlichen Gründen einige Pixel manchmal nicht leuchteten. Dies passierte ohne ein erkennbares Muster und wir konnten das Problem nicht genau feststellen. Vermutlich reichen die 3.3V des GPIO Pins nicht für die große Anzahl an LEDs aus und das Signal wurde deshalb verfälscht.

Wir beschlossen das Spiel nun auf einer vorgefertigten 8x8-dot-Matrix zu programmieren.

Allerdings erfolgt die Ansteuerung dort nicht mehr via Wert pro Pixel, sondern mit jeweils einer x- und einer y-Koordinate pro Pixel, was zur Folge hatte, dass ein neues Programm geschrieben werden musste.

Dies verlief ohne größere Komplikationen und funktioniert wie gewünscht!

Code:

Kommentare sind teilweise falsch eingerückt, im GitHub ist der Code als „Snake8x8. py“ zu finden

```
import
time
        import random
        import datetime
        from datetime import timedelta
        # enum is used for clarifying the button inputs and directions to right, left
        etc instead of plain numbers
        from enum import Enum
        import RPi.GPIO as GPIO

        # importing the necessary libraries for the 8x8 dot matrix
        from luma.led_matrix.device import max7219
        from luma.core.interface.serial import spi, noop
        from luma.core.render import canvas
        from luma.core.virtual import viewport
        from luma.core.legacy import text, show_message
        from luma.core.legacy.font import proportional, CP437_FONT, TINY_FONT,
        SINCLAIR_FONT, LCD_FONT

        GPIO.setmode(GPIO.BCM)

        # setting up the pins for the buttons
        GPIO.setup(17,GPIO.IN, pull_up_down=GPIO.PUD_UP)
        GPIO.setup(27,GPIO.IN, pull_up_down=GPIO.PUD_UP)

        # enumerated values for the direction - counting up by one if changed to the
        right, instead of 4 it goes back to 0.
        class Direction(Enum):
            RIGHT = 0
            DOWN = 1
            LEFT = 2
            UP = 3

            def succ(self):
                if self is Direction.UP:
                    return Direction.RIGHT
                else:
                    return Direction(self.value + 1)

            def pred(self):
                if self is Direction.RIGHT:
                    return Direction.UP
                else:
                    return Direction(self.value -1)
```

```
# defining the left and right button presses the same way
class ButtonPress(Enum):
    LEFT = 0
    RIGHT = 1

# this is the main snake class, everything gets initialized first
class Snake:
    def __init__(self):
        # the snake positions have to be split into x and y coordinates for
this matrix' library.
        self.current_xpos = [1,2]
        self.current_ypos = [4,4]
        # same goes for the fruit positions
        self.fruit_xpos = random.randint(0,7)
        self.fruit_ypos = random.randint(0,7)
        # variable for the fruit location is both of the coordinates combined,
this is important because we have to compare the fruit positions with the head
position of the snake.
        self.fruit_location = (self.fruit_xpos, self.fruit_ypos)
        # this is for testing:                print(self.fruit_location)
        # starting direction is from left to right
        self.direction = Direction.RIGHT
        # the snake is, of course, alive upon starting the game.
        self.alive = True

    def is_alive(self):
        return self.alive

    # this part was originally planned to be in a function, but right now, its
directly inside of the  move() function.
    # saving all of this for later, in case we run into errors

    #def button_input(self, button_pressed):
     #   if button_pressed is ButtonPress.LEFT:
      #      print("left")
       #     self.direction = self.direction.pred()
        #elif button_pressed is ButtonPress.RIGHT:
         #   print("right")
          #  self.direction = self.direction.succ()

    # function for spawning a new fruit that is not on a pixel where the snake
is:
    # this was difficult because we don't have a single value for pixels for
this matrix because of the way it works with coordinates

    def set_random_fruit_drop(self):
```

```
        xy = []
        # i iterates as often as there are entries in current_x/ypos, adding
the i'st entry to an array everytime
        for i in range (len(self.current_xpos)):
            xy_new = (self.current_xpos[i],self.current_ypos[i])
            xy.append(xy_new)
        # generating a random coordinate
        f = (random.randint(0,7),random.randint(0,7))
        # generating a new coordinate every time the generated one is inside
the array from above, this way we get an unoccupied pixel in the matrix for the
fruit location.
        while f in xy:
            f = (random.randint(0,7),random.randint(0,7))
        # splitting the value into its coordinates again so the fruit collision
check can work
        self.fruit_xpos = f[0]
        self.fruit_ypos = f[1]

        self.fruit_location = (self.fruit_xpos, self.fruit_ypos)

    # defining the function to check if the snake is colliding with itself:
    def check_collision_self(self, move_location):
        # same procedure as in the fruit drop, this time we compare it to the
location we are about to move to.
        xy = []
        for i in range (len(self.current_xpos)):
            xy_new = (self.current_xpos[i],self.current_ypos[i])
            xy.append(xy_new)
        # returning either True or False, the programm will stop if this
returns False.
        if move_location in xy:
            return True
        else:
            return False
    # checking for fruit collision is a bit simpler, we just compare the x
value of the fruit with the x value of the head and the same with the y value.
    # only if BOTH are identical it's considered a match and the snake eats the
fruit.
    def check_collision_fruit(self, move_location):
        if move_location[0] is self.fruit_location[0] and move_location[1] is
self.fruit_location[1]:
            print("it's a match")
            self.set_random_fruit_drop()
        else:
            # if no fruit is eaten, we delete the tail end of the snake since
it moves one pixel further.
```

```
            # if the snake ate a piece this turn, the tail will stay since it
got one pixel larger.
            self.current_xpos.pop(0)
            self.current_ypos.pop(0)

    # this is the main function for moving the snake:
    def move(self):
        # the head positions of the snake are the last entry in the array of
the stored x/y values since it will be added last with the .append() function.
        head_xpos = self.current_xpos[-1]
        head_ypos = self.current_ypos[-1]
        # the following ~25 lines look at the direction the snake is moving and
give out a location the head will move to, based on the current head positions
and direction.
        if self.direction is Direction.RIGHT:
            # if the snake reaches the border of our matrix it will reenter on
the other side, otherwise it's really hard to play.
            # the following lines are just calculations of the new coordinates
based on different possible movement options.
            if head_xpos is 7:
                move_location = (0, head_ypos)
            else:
                move_location = (head_xpos + 1, head_ypos)

        elif self.direction is Direction.DOWN:
            if head_ypos is 7:
                move_location = (head_xpos, 0)
            else:
                move_location = (head_xpos, head_ypos + 1)

        elif self.direction is Direction.LEFT:
            if head_xpos is 0:
                move_location = (7, head_ypos)
            else:
                move_location = (head_xpos - 1, head_ypos)

        elif self.direction is Direction.UP:
            if head_ypos is 0:
                move_location = (head_xpos, 7)
            else:
                move_location = (head_xpos, head_ypos - 1)

        # now that we have looked at the new head position it's time to check
if we crash into ourself and result in a Game Over:
        if self.check_collision_self(move_location) is True:
            # if so, we just kill the snake and the main loop further down
stops looping.
```

```
            self.alive = False
            return

        # splitting the location our snakehead will move to during this turn
into x and y coordinates again so we can append the current_x/ypos arrays.
        movex = move_location[0]
        movey = move_location[1]
        self.current_xpos.append(movex)
        self.current_ypos.append(movey)
        # checking if we will eat a fruit this turn, here it becomes clear why
we used the .pop() function when defining the fruit collision function,
        # when we eat a fruit we don't need to get rid of the last pixel, but
since we always append we need to drop our last pixel when we don't eat a
fruit.
        # this method is great because we only have to draw the actual snake
one time per round and not before and after consuming a fruit.
        self.check_collision_fruit(move_location)
        # drawing the i'st position of both x and y coordinate each time it
iterates, resulting in the whole snake.
        with canvas(device) as draw:
            for i in range(len(self.current_xpos)):
                draw.point((self.current_xpos[i],self.current_ypos[i]),
fill="white")
                # also drawing the fruit.
                draw.point((self.fruit_xpos,self.fruit_ypos), fill="white")

snake = Snake()

# initializing the 8x8 LED Matrix
serial = spi(port=0, device=0, gpio=noop())
device = max7219(serial, cascaded=1)
# little starting message:
msg = "Snake"
show_message(device, msg, fill="white", font=proportional(CP437_FONT))
time.sleep(1)


# this is the main loop while the snake is alive:
while snake.is_alive() is True:

    delta = 0
    # we look at the current time (an exact value)
    time_start = datetime.datetime.now()

    # loop as long as delta is less than 250000 microseconds (.25 seconds)
    while delta < 250000:
```

```
        # comparing the time we looked at earlier with the time right now, we
do this as long as delta is smaller than 250000 microseconds
        time_end = datetime.datetime.now()
        delta = (time_end - time_start).microseconds
    # upon pressing a button, the variable for the direction changes.
    if GPIO.input(17) == False:

        button_pressed = ButtonPress.RIGHT
        snake.direction = snake.direction.succ()
    elif GPIO.input(27) == False:

        button_pressed = ButtonPress.LEFT
        snake.direction = snake.direction.pred()
    # now that we adjusted the direction, the snake.move() function can finally
be called.
    snake.move()
    type(delta)

# little Game Over message.
while snake.is_alive() is False:
    msg = "Game Over"
    show_message(device, msg, fill="white", font=proportional(CP437_FONT))
    time.sleep(2)
    # this has to be called so the message doesn't loop
    snake.alive = True
```