An Interactive Arcade Platform: Integrating Python Game Logic with Physical Control Interfaces

Audrey

March 2026

1 Introduction

This project outlines the development of a physical arcade machine. The primary objective was to move beyond conventional software-based gaming, which can feel generic and dull, and integrate programming with tangible electronic components, thereby creating a more immersive and authentic gaming experience.

Due to the lack of funding and equipment, it was paramount that I was resourceful in sourcing and assembling the hardware components. I acquired one joystick, four push buttons, and four 16-by-16 NeoPixel boards (totalling 1024 individually addressable LEDs). The joysticks were designed for in-game character or piece control, replacing the conventional keyboard, while the buttons were assigned to functions such as action commands and launch.

2 Development

The software architecture was developed concurrently with the hardware assembly. I developed a `convert()` function to translate the two-dimensional grid coordinates into the precise linear LED index. This function underpins all visual output for the three classic games selected for emulation.

```python
def convert(y, x):
    block = y // 16
    y = y % 16
    if x % 2 == 0:
        led = 16*x+y
    else:
        led = 16*x+15-y
    led = led + block*512
    return led
```

```python
def coord(x,y):
    value = 0
    XSaver = x
    if x%2 == 1: x -= 1
    X2Total = int(x/2)*32
    if XSaver%2 == 0: value = X2Total + y%16
    else: value = X2Total+31-y%16
    if y >= 16: value += 512
    return value
```

Figure 1: convert() function in the Snake and Pac-Man code (left), and Tetris (right)

2.1 Snake

The snake's body is represented as a list of coordinate pairs. Game state updates involve appending a new head coordinate and simultaneously removing the tail segment based on food consumption. Movement is dictated by reading the joystick's directional pins, with implemented safeguards to prevent the snake from reversing into its own body. Collision detection terminates the game upon the head intersecting the boundary walls or any segment of its own body. Food generation incorporates logic to ensure spawn coordinates do not conflict with

```
if up.value():
    if direction != "down":
        direction = "up"
if direction == "up":
    new_head = (head[0] - 1, head[1])
    snake.append(new_head)
    for n in range(len(snake)):
        head = snake[-1]
        itemy = snake[n][0]
        itemx = snake[n][1]
        lights = convert(int(itemy), int(itemx))
        pixels[lights] = bodycolour
        heady = head[0]
        headx = head[1]
        head_coor = convert(int(heady), int(headx))
        pixels[head_coor] = headcolour

    #eating fruit
    if new_head == food_coor:
        score += 1
        foodx = random.randint(2,29)
        foody = random.randint(2,29)
        food_coor = (foodx,foody)
        for n in snake:
            if food_coor == n:
                foodx = random.randint(2,29)
                foody = random.randint(2,29)
                food_coor = (foodx,foody)
        food = convert(foodx,foody)
        pixels[food] = [60,7,10]
        pixels.show()
```

```
else:
    #turn off
    offy = snake[0][0]
    offx = snake[0][1]
    black = convert(int(offy), int(offx))
    pixels[black] = [0,0,0]
    snake.pop(0)
    pixels.show()
    time.sleep(speed)

#dying by hitting border
if new_head[0] == 1 or new_head[0] == 30:
    pixels[head_coor] = headcolour2
    pixels.show()
    time.sleep(1.5)
    for body in snake:
        bodyy = body[0]
        bodyx = body[1]
        n = convert(int(bodyy), int(bodyx))
        pixels[n] = [0,0,0]
    pixels[head_coor] = [5,0,15]
    pixels.show()
    start = False
    death = True
    continue


#dying by hitting self
if new_head in snake[0:-1]:
    pixels[head_coor] = headcolour2
    pixels.show()
    time.sleep(1.5)
    for body in snake:
        bodyy = body[0]
        bodyx = body[1]
        n = convert(int(bodyy), int(bodyx))
        pixels[n] = [0,0,0]
    pixels.show()
    start = False
    death = True
    continue
```

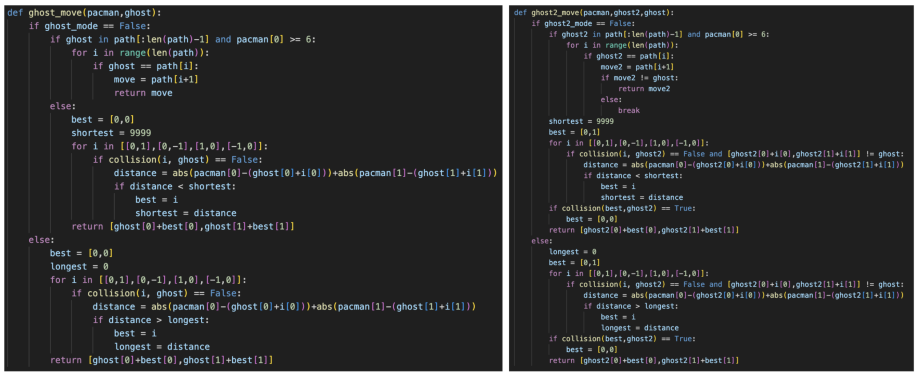
Figure 2: Code snippet of the implementation for the directional control logic. This handles upward movement, including collision detection against the boundaries and the snake's own body, as well as the conditional removal of the tail segment upon food consumption.

the snake's current position or the static 'SNAKE !' title displayed on the LED matrix at startup.

2.2 Pac-Man

For Pac-Man, a modular approach was adopted, dividing the playfield into a grid of 3x3 pixel 'cells'. Maze walls are defined by lists of cell coordinates; the `walls_display()` function iterates through these lists, illuminating a 3-by-3 block for each wall cell and selectively deactivating specific LEDs to form passageways. Characters, including Pac-Man and the ghosts, are rendered as 3x3 sprites. The `pacmanchar()` function dynamically adjusts the illuminated pixels within his sprite to simulate an opening mouth oriented towards his direction of travel. The ghost AI was the most demanding aspect. The red ghost employs a shortest-path algorithm, evaluating adjacent cells to minimise distance to Pac-

Man. A 'scared' mode inverts this logic, prompting evasion. Differential logic was implemented for the blue ghost to prevent uniform movement patterns.



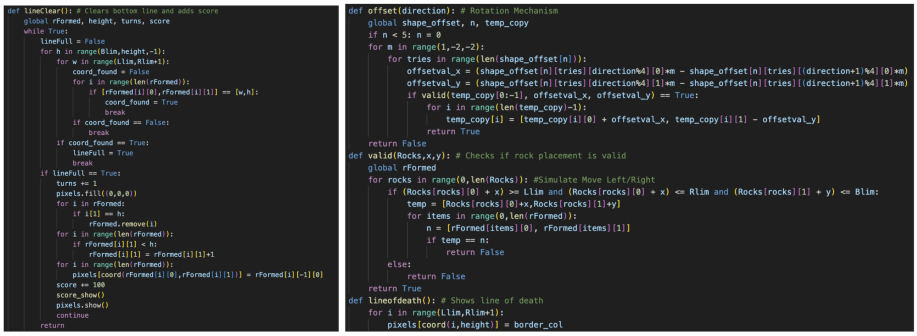
```
def ghost_move(pacman,ghost):
    if ghost_mode == False:
        if ghost in path[:len(path)-1] and pacman[0] >= 6:
            for i in range(len(path)):
                if ghost == path[i]:
                    move = path[i+1]
                    return move
        else:
            best = [0,0]
            shortest = 9999
            for i in [[0,1],[0,-1],[1,0],[-1,0]]:
                if collision(i, ghost) == False:
                    distance = abs(pacman[0]-(ghost[0]+i[0]))+abs(pacman[1]-(ghost[1]+i[1]))
                    if distance < shortest:
                        best = i
                        shortest = distance
            return [ghost[0]+best[0],ghost[1]+best[1]]
    else:
        best = [0,0]
        longest = 0
        for i in [[0,1],[0,-1],[1,0],[-1,0]]:
            if collision(i, ghost) == False:
                distance = abs(pacman[0]-(ghost[0]+i[0]))+abs(pacman[1]-(ghost[1]+i[1]))
                if distance > longest:
                    best = i
                    longest = distance
        return [ghost[0]+best[0],ghost[1]+best[1]]
```

```
def ghost2_move(pacman,ghost2,ghost):
    if ghost2_mode == False:
        if ghost2 in path[:len(path)-1] and pacman[0] >= 6:
            for i in range(len(path)):
                if ghost2 == path[i]:
                    move2 = path[i+1]
                    if move2 != ghost:
                        return move2
                    else:
                        break
        shortest = 9999
        best = [0,1]
        for i in [[0,1],[0,-1],[1,0],[-1,0]]:
            if collision(i, ghost2) == False and [ghost2[0]+i[0],ghost2[1]+i[1]] != ghost:
                distance = abs(pacman[0]-(ghost2[0]+i[0]))+abs(pacman[1]-(ghost2[1]+i[1]))
                if distance < shortest:
                    best = i
                    shortest = distance
        if collision(best,ghost2) == True:
            best = [0,0]
        return [ghost2[0]+best[0],ghost2[1]+best[1]]
    else:
        longest = 0
        best = [0,1]
        for i in [[0,1],[0,-1],[1,0],[-1,0]]:
            if collision(i, ghost2) == False and [ghost2[0]+i[0],ghost2[1]+i[1]] != ghost:
                distance = abs(pacman[0]-(ghost2[0]+i[0]))+abs(pacman[1]-(ghost2[1]+i[1]))
                if distance > longest:
                    best = i
                    longest = distance
        if collision(best,ghost2) == True:
            best = [0,0]
        return [ghost2[0]+best[0],ghost2[1]+best[1]]
```

Figure 3: Implementation of ghost character behaviours. Defines the movement logic for the red and blue ghosts, including their pursuit algorithms during standard play and their evasive patterns when the 'scared mode' is activated following cherry consumption.

2.3 Tetris

The seven standard Tetromino shapes are stored in `shape_spawner`, each defined by a set of relative coordinates and an associated colour. Settled blocks are tracked within the `rFormed` list. The `valid()` function is invoked prior to every movement or rotation of the current piece (`rMoving`) to prevent overlap with `rFormed` or violation of the grid boundaries. Rotation logic was engineered to perform coordinate transformations around a central point, supplemented by an `offset()` function to implement 'wall kick' functionality, adjusting the piece's position if the raw rotation results in an illegal state. Line clearing is executed by the `lineClear()` function, which scans the grid, removes complete rows from `rFormed`, shifts all superior rows downward, and increments the score accordingly.



```
def lineClear(): # Clears bottom line and adds score
    global rFormed, height, turns, score
    while True:
        lineFull = False
        for h in range(Blim,height,-1):
            for w in range(Llim,Rlim+1):
                coord_found = False
                for i in range(len(rFormed)):
                    if [rFormed[i][0],rFormed[i][1]] == [w,h]:
                        coord_found = True
                        break
                if coord_found == False:
                    break
            if coord_found == True:
                lineFull = True
                break
        if lineFull == True:
            turns += 1
            pixels.fill((0,0,0))
            for i in rFormed:
                if i[1] == h:
                    rFormed.remove(i)
            for i in range(len(rFormed)):
                if rFormed[i][1] < h:
                    rFormed[i][1] = rFormed[i][1]+1
            for i in range(len(rFormed)):
                pixels[coord(rFormed[i][0],rFormed[i][1])] = rFormed[i][-1][0]
            score += 100
            score_show()
            pixels.show()
            continue
        return
```

```
def offset(direction): # Rotation Mechanism
    global shape_offset, n, temp_copy
    if n < 5: n = 0
    for m in range(1,-2,-2):
        for tries in range(len(shape_offset[n])):
            offsetval_x = (shape_offset[n][tries][direction%4][0]*m - shape_offset[n][tries][(direction+1)%4][0]*m)
            offsetval_y = (shape_offset[n][tries][direction%4][1]*m - shape_offset[n][tries][(direction+1)%4][1]*m)
            if valid(temp_copy[0:-1], offsetval_x, offsetval_y) == True:
                for i in range(len(temp_copy)-1):
                    temp_copy[i] = [temp_copy[i][0] + offsetval_x, temp_copy[i][1] - offsetval_y]
                return True
    return False
def valid(Rocks,x,y): # Checks if rock placement is valid
    global rFormed
    for rocks in range(0,len(Rocks)): #Simulate Move Left/Right
        if (Rocks[rocks][0] + x) >= Llim and (Rocks[rocks][0] + x) <= Rlim and (Rocks[rocks][1] + y) <= Blim:
            temp = [Rocks[rocks][0]+x,Rocks[rocks][1]+y]
            for items in range(0,len(rFormed)):
                n = [rFormed[items][0], rFormed[items][1]]
                if temp == n:
                    return False
        else:
            return False
    return True
def lineofdeath(): # Shows line of death
    for i in range(Llim,Rlim+1):
        pixels[coord(i,height)] = border_col
```

Figure 4: Code snippet of the core Tetris mechanics implementation. The valid() function prevents piece overlap with settled blocks or boundary violations; offset() provides 'wall kick' functionality by adjusting piece position after rotation; lineClear() scans for complete rows, removes them, and shifts remaining blocks downward; and lineofdeath() visually indicates the maximum height threshold for piece placement.

Dear Judges,

I apologise for the delay in submission. I have included all necessary documents below. Please note that any textfiles linked in the code have not been included below for simplicity. They are simply to list the highscores, and one can open a standard text file, save it to the corresponding file name, and it will function properly. Please also note that if you would like this to run, the code must be run with the equipment on hand. The code will run smoothly, without errors, so long as the necessary equipment is present. I appreciate the extension granted and hope you enjoy my project.