Part 06

Code your own Pac-Man game

Pac-Man captured the hearts and pocket money of many young people in the eighties. Since then, it has made its way onto just about every computer system and console

MAKER

Mark Vanstone

Educational software author from the nineties, author of the ArcVenture series, disappeared into the corporate software wasteland. Rescued by the Raspberry Pi!

magpi.cc/YiZnxl

@mindexplorers

You'll Need

- Raspbian Jessie or newer
- An image manipulation program such as GIMP, or images available from **magpi.cc/nBSXKz**
- The latest version of Pygame Zero (1.2)
- USB joystick or gamepad (optional)

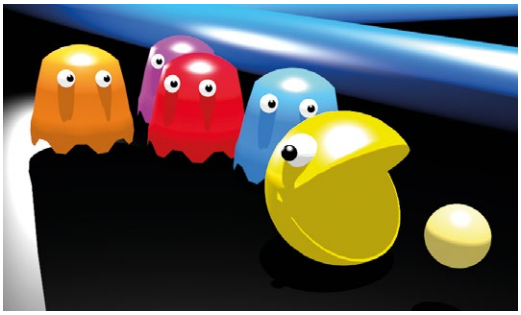
The concept of *Pac-Man* is quite simple. Pac-Man eats dots in a maze to score points. Avoid the ghosts unless you have just eaten a power-up, in which case ghosts are tasty. In this series we have gradually introduced new elements of Pygame Zero and also concepts around writing games. This is the first instalment in a two-part tutorial which will show you some more tricks to writing arcade games with Pygame Zero. We will also use some more advanced programming concepts to make our games even better. In this first part, we will put together the basics of the *Pac-Man* game and introduce the concept of adding extra Python modules to our program.

01 Let's get stuck in

As with the more recent episodes of this series, let's jump straight in, assuming that we have our basic Pygame Zero setup done. Let's set our window size to `WIDTH = 600` and `HEIGHT = 660`. This will give us room for a roughly square maze and a header area for some game information. We can get our gameplay area set up straight away by blitting two graphics – 'header' and 'colourmap' – to `0,0` and `0,80` respectively in the `draw()` function. You can make these graphics yourself or you can use ours, which can be found at **magpi.cc/nBSXKz**.

02 It's amazing

The original game had a very specific layout to the maze, but many different ones have appeared in later versions. The one we will be using is very similar to the original, but you can make your own design if you want. If you make your own, you'll also have to make two more maps (we'll come to those in a bit) which help with the running of the game. The main things about the map is that it has a central area where the ghosts start from and it doesn't have any other closed-in areas that the ghosts are likely to get trapped in (they can be a bit stupid sometimes).



▲ Because Pac-Man didn't involve shooting things, the Japanese designers thought it would appeal more to girls

03 Hmmm, pizza

Our next challenge is to get a player actor moving around the maze. For some unknown reason, the game's creator, Toru Iwatani, decided to make the main character a pizza that ate dots. Well, the eighties were a bit strange and that seemed perfectly reasonable at the time. We'll need two frames for our character: one with the mouth open and one with it closed. We can create our player actor near the top of the code using `player = Actor("pacman_o")`. This will create the actor with the mouth-open graphic. We will then set the actor's location in an `init()` function, as in previous programs.

04 Modulify to simplify

We can get our player onto the play area by setting `player.x = 290` and `player.y = 570` in the `init()` function and then call `player.draw()` in the `draw()` function, but to move the player character we'll need to get some input from the player. Previously we have used keyboard and mouse input, but this time we are going to have the option of joystick or gamepad input. Pygame Zero doesn't currently directly support gamepads, but we are going to borrow a bit of the Pygame module to get this working. We are also going to make a separate Python module for our input.

05 It's a joystick.init

Setting up a new module is easy. All we need to do is make a new file, in this case **gameinput.py**, and in our main program at the top, write `import gameinput`. In this new file we can import the Pygame functions we need with `from pygame import joystick, key` and `from pygame.locals import *`. We can then initialise the Pygame joystick object (this also includes gamepads) by typing `joystick.init()`. We can find out how many joysticks or gamepads are connected by using `joystick_count = joystick.get_count()`. If we find any joysticks connected, we need to initialise them individually – see **figure1.py**.

06 Checking the input

We can now write a function in our gameinput module to check input from the player. If we define the function with `def checkInput(p):` we can get the x axis of a joystick using `joyin.get_axis(0)` and the y axis by using `joyin.get_axis(1)`. The numbers that are returned from these calls will be between -1 and +1, with 0 being the central position. We can check to see if the values are over 0.8 or under -0.8, as, depending on the device, we may not actually see -1 or 1 being returned. You may like to test this with your gamepad or joystick to see what range of values are returned.

07 Up, down, left, or right

The variable `p` that we are passing into our `checkInput()` function will be the player actor. We

figure1.py

```
001. # gameinput Module
002.
003. from pygame import joystick, key
004. from pygame.locals import *
005.
006. joystick.init()
007. joystick_count = joystick.get_count()
008.
009. if(joystick_count > 0):
010.     joyin = joystick.Joystick(0)
011.     joyin.init()
012.     # For the purposes of this tutorial
013.     # we are only going to use the first
014.     # joystick that is connected.
```

Top Tip

Modules

Using separate modules means not only is your code easier to follow, but it's easier for a team to work on.

figure2.py

```
001. def checkInput(p):
002.     global joyin, joystick_count
003.     xaxis = yaxis = 0
004.     if joystick_count > 0:
005.         xaxis = joyin.get_axis(0)
006.         yaxis = joyin.get_axis(1)
007.     if key.get_pressed()[K_LEFT] or xaxis < -0.8:
008.         p.angle = 180
009.         p.movex = -20
010.     if key.get_pressed()[K_RIGHT] or xaxis > 0.8:
011.         p.angle = 0
012.         p.movex = 20
013.     if key.get_pressed()[K_UP] or yaxis < -0.8:
014.         p.angle = 90
015.         p.movey = -20
016.     if key.get_pressed()[K_DOWN] or yaxis > 0.8:
017.         p.angle = 270
018.         p.movey = 20
```

figure3.py

```
001. # inside update() function
002.
003.     if player.movex or player.movey:
004.         inputLock()
005.         animate(player, pos=(player.x + player.
     movex, player.y + player.movey), duration=1/SPEED,
     tween='linear', on_finished=inputUnLock)
006.
007. # outside update() function
008.
009. def inputLock():
010.     global player
011.     player.inputActive = False
012.
013. def inputUnLock():
014.     global player
015.     player.movex = player.movey = 0
016.     player.inputActive = True
```

can test each of the directions of the joystick at the same time as the keyboard and then set the player angle (so that it points in the correct direction for movement) and also how much it needs to move. We'll set these by saying (for example, if the left arrow is pressed or the joystick is moved to the left) `if key.get_pressed()[K_LEFT] or xaxis < -0.8:` and then `p.angle = 180` and `p.movex = -20`. See **figure2.py** for the full `checkInput()` function.

▲ You can plug a gamepad or joystick into one of the USB ports on your Raspberry Pi

08 Get a move on!

Now we have our input function set up, we can call it from the `update()` function. Because this function is in a different module, we need to prefix it with the module name. In the `update()` function we write `gameinput.checkInput(player)`. After this function has been called, if there has been any input, we should have some variables set in the player actor that we can use to move. We can say `if player.movex or player.movey:` and then use the `animate()` function to move by the amount specified in `player.movex` and `player.movey`.

09 Hold your horses

The way we have the code at the moment means that any time there is some input, we fire off a new animation. This will soon mean that layers of animation get called over the top of each other, but what we want is for the animation to run and then start looking for new input. To do this we need an input locking system. We can call an input lock function before the move and then wait for the animation to finish before unlocking to look for more input. Look at **figure3.py** to see how we can make this locking system.

10 You can't just move anywhere

Now, here comes the interesting bit. We want our player actor to move around the maze, but at the moment it will go though the walls and even off the screen. We need to restrict the movement only to the corridors of the maze. There are several different ways we could do this, but for this game we're going to have an image map marking the

areas that the player actor can move within. The map will be a black and white one, showing just the corridors as black and the walls as white. We will then look at the map in the direction we want to move and see if it is black; if it is, we can move.

11 Testing the map

To be able to test the colour of a part of an image, we need to borrow a few functions from Pygame again. We'll also put our map functions in a separate module. So make a new Python file and call it **gamemaps.py** and in it we'll write `from pygame import image, Color`.

We must also load in our movement map, which we need to do in the Pygame way: `moveimage = image.load('images/pacmanmovemap.png')`. Then all we need to do is write a function to check that the direction of the player is valid. See **figure4.py** for this function.

12 Using the movemap

To use this new module, we need to `import gamemaps` at the top of our main code file and then, before we animate the player (but after we have checked for input), we can call `gamemaps.checkMovePoint(player)`,which will zero the `movex` and `movey` variables of the player if the move is not possible. So now we should find that the player actor can only move inside the corridors. We do have one special case that you may have noticed in **figure4.py**, and that is because there is one corridor where the player can move from one side of the screen to the other.

13 You spin me round

There is one more aspect to the movement of the player actor, and that is the animation. As Pac-Man moves, the mouth opens and shuts and points in the direction of the movement. The mouth opening and closing is easy enough: we have an image for open and one for closed and alternate between the two. For pointing in the correct direction, we can rotate the player actor. Unfortunately, this has a slight problem that Pac-Man will be upside-down when moving left. So we just need to have one version that is switched the other way round. See **figure5.py** for a function that sorts out all of this.

figure4.py

```
001. # gamemaps module
002. from pygame import image, Color
003. moveimage = image.load('images/pacmanmovemap.png')
004.
005. def checkMovePoint(p):
006.     global moveimage
007.     if p.x+p.movex < 0: p.x = p.x+600
008.     if p.x+p.movex > 600: p.x = p.x-600
009.     if moveimage.get_at((int(p.x+p.movex), int(p.y+p.
     movey-80))) != Color('black'):
010.         p.movex = p.movey = 0
```

figure5.py

```
001. def getPlayerImage():
002.     global player
003.     # we need to import datetime at the top of our code
004.     dt = datetime.now()
005.     a = player.angle
006.     # this next line will give us a number between
007.     # 0 and 5 depending on the time and SPEED
008.     tc = dt.microsecond%(500000/SPEED)/(100000/SPEED)
009.     if tc > 2.5 and (player.movex != 0 or player.movey
     !=0):
010.         # this is for the closed mouth images
011.         if a != 180:
012.             player.image = "pacman_c"
013.         else:
014.             # reverse image if facing left
015.             player.image = "pacman_cr"
016.     else:
017.         # this is for the open mouth images
018.         if a != 180:
019.             player.image = "pacman_o"
020.         else:
021.             player.image = "pacman_or"
022.     # set the angle on the player actor
023.     player.angle = a
```

Top Tip

Pygame

Pygame Zero is based on Pygame, but if you want to use some of the Pygame functions, best to do it in a separate module to avoid confusion.

figure6.py

```
# This goes in the main code file.

def initDots():
    global pacDots
    pacDots = []
    a = x = 0
    while x < 30:
        y = 0
        while y < 29:
            if gamemaps.checkDotPoint(10+x*20, 10+y*20):
                pacDots.append(Actor("dot",(10+x*20,
90+y*20)))
                pacDots[a].status = 0
                a += 1
            y += 1
        x += 1

# This goes in the gamemaps module file.

dotimage = image.load('images/pacmandotmap.png')

def checkDotPoint(x,y):
    global dotimage
    if dotimage.get_at((int(x), int(y))) ==
Color('black'):
        return True
    return False

# This bit goes in the draw() function.

    pacDotsLeft = 0
    for a in range(len(pacDots)):
        if pacDots[a].status == 0:
            pacDots[a].draw()
            pacDotsLeft += 1
        if pacDots[a].collidepoint((player.x, player.y)):
            pacDots[a].status = 1
    # if there are no dots left, the player has won
    if pacDotsLeft == 0: player.status = 2
```

14 Spot on

So when we have put in a call to `getPlayerImage()` just before we draw the player actor, we should have Pac-Man moving around, chomping and pointing in the correct direction. Now we need something to chomp. We are going to create a set of dots at even spacings along most of the corridors. An easy way to do this is to use a similar technique that we're using for testing where the corridors are. If we make an image map of the places the dots need to go and loop over the whole map, only placing dots where it is black, we can get the desired effect.

15 Tasty, tasty dots

To get our dots doing their thing, we'll need to code a few things. We need to initialise actors for each dot, we need to draw each dot, and if the player eats the dot, we need to stop drawing it; **figure6.py** shows how we can do each of these jobs. We need `initDots()`, we need to add another function to **gamemaps.py** to work out where to position the dots, and we need to add some drawing code to the `draw()` function. In addition to the code in **figure6.py**, we need to add a call to `initDots()` in our `init()` function.

16 I ain't afraid of no ghosts

Now that we have our Pac-Man happily munching dots, we must introduce our villains to the mix. In the original game, the ghosts had names; in the English version they were known as Blinky, Pinky, Inky, and Clyde. They roam the maze looking for Pac-Man, starting from an enclosure in the centre of the map. We can initialise each ghost as an actor to appear at the centre of the maze and keep them in a list called `ghosts[]`. To start off with, we'll just make them move around randomly. The way we can do this is to set a random direction (`ghosts[g].dir`) for each and then keep them moving until they hit a wall.

17 Random motion

We can use the same system that we used to check player movement for the ghosts. Each time we move a ghost – `moveGhosts()` – we can get a list of which directions are available to it. If the current direction (`ghosts[g].dir`) is not available, then we randomly pick another direction until we find one that we can move in. We can also have a random occurrence of changing direction, just to make it a bit less predictable – and if the ghosts collide with each other, we could do the same. When we have moved the ghosts with the `animate()` function, we get it to count how many ghosts have finished moving. When they are all done, we can call the `moveGhosts()` function again.

Top Tip

Animations

When using the `animate()` function, it is best to use the callback function to see when it has finished, as different systems may work at different speeds.

18 Look like a ghost

The last thing to do with our ghosts is to actually draw them to the screen. We can create a function called `drawGhosts()` where we loop through the four ghosts and draw them to the screen. One of the details of the original game was that the eyes of the ghosts would follow the player; we can do this by setting the ghost image to reverse if the player is to the left of the ghost. We have numbered images so that ghost one is **ghost1.png** and ghost two is **ghost2.png**, etc. Have a look at the full **pacman1.py** program listing to see all the functions that make the ghosts work.

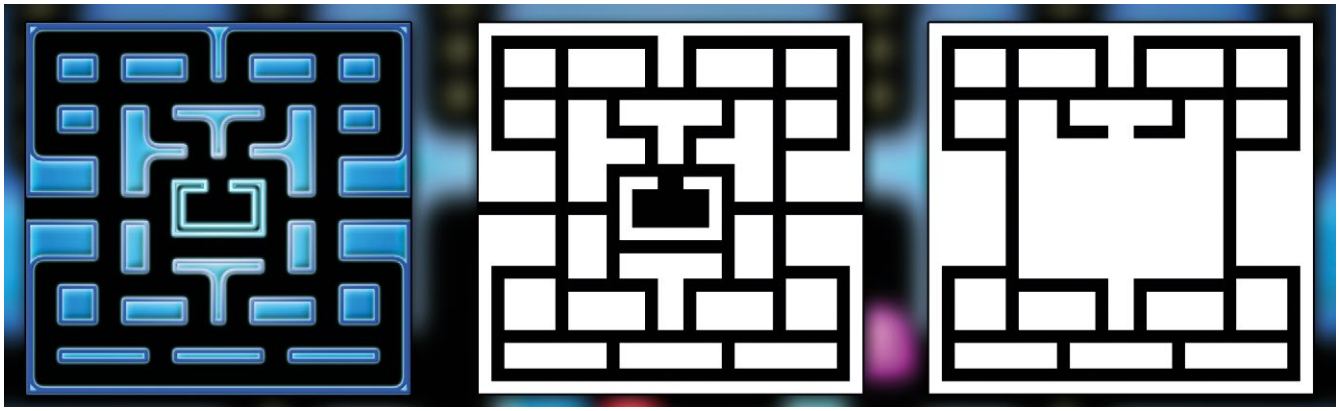
19 Game over

Of course, we need to deal with the end-of-the-game conditions and, as before, we can use a status variable. In this case we have previously set `player.status = 2` if the player wins. We can check to see if a ghost collides with the player and set `player.status = 1`. Then we just need to display some text in the `draw()` function based on this variable. And that's it for part one. In the next part we'll be giving the ghosts more brains, adding levels, lives, and power-ups – and adding some sweet, soothing music and sound effects.

gameinput.py

Language: **Python 3**

```
001. # gameinput Module
002.
003. from pygame import joystick, key
004. from pygame.locals import *
005.
006. joystick.init()
007. joystick_count = joystick.get_count()
008.
009. if(joystick_count > 0):
010.     joyin = joystick.Joystick(0)
011.     joyin.init()
012.
013. def checkInput(p):
014.     global joyin, joystick_count
015.     xaxis = yaxis = 0
016.     if joystick_count > 0:
017.         xaxis = joyin.get_axis(0)
018.         yaxis = joyin.get_axis(1)
019.     if key.get_pressed()[K_LEFT] or xaxis < -0.8:
020.         p.angle = 180
021.         p.movex = -20
022.     if key.get_pressed()[K_RIGHT] or xaxis > 0.8:
023.         p.angle = 0
024.         p.movex = 20
025.     if key.get_pressed()[K_UP] or yaxis < -0.8:
026.         p.angle = 90
027.         p.movey = -20
028.     if key.get_pressed()[K_DOWN] or yaxis > 0.8:
029.         p.angle = 270
030.         p.movey = 20
```



Three maps are used: one which we see, one to check possible movements, and one to check where dots are to be placed

Colour Map | Movement Map | Dot Location Map

gamemaps.py

Language: **Python 3**

```
# gamemaps module

from pygame import image, Color
moveimage = image.load('images/pacmanmovemap.png')
dotimage = image.load('images/pacmandotmap.png')

def checkMovePoint(p):
    global moveimage
    if p.x+p.movex < 0: p.x = p.x+600
    if p.x+p.movex > 600: p.x = p.x-600
    if moveimage.get_at((int(p.x+p.movex), int(p.y+p.movey-80))) != Color('black'):
        p.movex = p.movey = 0

def checkDotPoint(x,y):
    global dotimage
    if dotimage.get_at((int(x), int(y))) == Color('black'):
        return True
    return False

def getPossibleDirection(g):
    global moveimage
    if g.x-20 < 0:
        g.x = g.x+600
    if g.x+20 > 600:
        g.x = g.x-600
    directions = [0,0,0,0]
    if g.x+20 < 600:
        if moveimage.get_at((int(g.x+20), int(g.y-80))) == Color('black'): directions[0] = 1
    if g.x < 600 and g.x >= 0:
        if moveimage.get_at((int(g.x), int(g.y-60))) == Color('black'): directions[1] = 1
    if g.x-20 >= 0:
        if moveimage.get_at((int(g.x-20), int(g.y-80))) == Color('black'): directions[2] = 1
    if g.x < 600 and g.x >= 0:
        if moveimage.get_at((int(g.x), int(g.y-100))) == Color('black'): directions[3] = 1
    return directions
```

pacman1.py

Language: **Python 3**

```
import pgzrun
import gameinput
import gamemaps
from random import randint
from datetime import datetime
WIDTH = 600
HEIGHT = 660

player = Actor("pacman_o") # Load in the player Actor image
SPEED = 3

def draw(): # Pygame Zero draw function
    global pacDots, player
    screen.blit('header', (0, 0))
    screen.blit('colourmap', (0, 80))
    pacDotsLeft = 0
    for a in range(len(pacDots)):
        if pacDots[a].status == 0:
            pacDots[a].draw()
            pacDotsLeft += 1
        if pacDots[a].collidepoint((player.x, player.y)):
            pacDots[a].status = 1
    if pacDotsLeft == 0: player.status = 2
    drawGhosts()
    getPlayerImage()
    player.draw()
    if player.status == 1: screen.draw.text("GAME OVER", center=(300, 434), owidth=0.5, ocolor=(255,255,255), color=(255,64,0) , fontsize=40)
    if player.status == 2: screen.draw.text("YOU WIN!", center=(300, 434), owidth=0.5, ocolor=(255,255,255), color=(255,64,0) , fontsize=40)

def update(): # Pygame Zero update function
    global player, moveGhostsFlag, ghosts
    if player.status == 0:
        if moveGhostsFlag == 4: moveGhosts()
        for g in range(len(ghosts)):
            if ghosts[g].collidepoint((player.x, player.y)):
                player.status = 1
                pass
        if player.inputActive:
            gameinput.checkInput(player)
            gamemaps.checkMovePoint(player)
            if player.movex or player.movey:
                inputLock()
                animate(player, pos=(player.x + player.movex, player.y + player.movey), duration=1/SPEED, tween='linear', on_finished=inputUnLock)

def init():
```

DOWNLOAD THE FULL CODE:
magpi.cc/jefHFU

```
    global player
    initDots()
    initGhosts()
    player.x = 290
    player.y = 570
    player.status = 0
    inputUnLock()

def getPlayerImage():
    global player
    dt = datetime.now()
    a = player.angle
    tc = dt.microsecond%(500000/SPEED)/(100000/SPEED)
    if tc > 2.5 and (player.movex != 0 or player.movey
!=0):
        if a != 180:
            player.image = "pacman_c"
        else:
            player.image = "pacman_cr"
    else:
        if a != 180:
            player.image = "pacman_o"
        else:
            player.image = "pacman_or"
    player.angle = a

def drawGhosts():
    for g in range(len(ghosts)):
        if ghosts[g].x > player.x:
            ghosts[g].image = "ghost"+str(g+1)+"r"
        else:
            ghosts[g].image = "ghost"+str(g+1)
        ghosts[g].draw()

def moveGhosts():
    global moveGhostsFlag
    dmoves = [(1,0),(0,1),(-1,0),(0,-1)]
    moveGhostsFlag = 0
    for g in range(len(ghosts)):
        dirs = gamemaps.getPossibleDirection(ghosts[g])
        if ghostCollided(ghosts[g],g) and randint(0,3)
== 0: ghosts[g].dir = 3
        if dirs[ghosts[g].dir] == 0 or randint(0,50) ==
0:
            d = -1
            while d == -1:
                rd = randint(0,3)
                if dirs[rd] == 1:
                    d = rd
            ghosts[g].dir = d
        animate(ghosts[g], pos=(ghosts[g].x
+ dmoves[ghosts[g].dir][0]*20, ghosts[g].y +
dmoves[ghosts[g].dir][1]*20), duration=1/SPEED,
tween='linear', on_finished=flagMoveGhosts)

def flagMoveGhosts():
    global moveGhostsFlag
    moveGhostsFlag += 1

def ghostCollided(ga,gn):
    for g in range(len(ghosts)):
        if ghosts[g].colliderect(ga) and g != gn:
            return True
    return False

def initDots():
    global pacDots
    pacDots = []
    a = x = 0
    while x < 30:
        y = 0
        while y < 29:
            if gamemaps.checkDotPoint(10+x*20, 10+y*20):
                pacDots.append(Actor("dot",(10+x*20,
90+y*20)))
                pacDots[a].status = 0
                a += 1
            y += 1
        x += 1

def initGhosts():
    global ghosts, moveGhostsFlag
    moveGhostsFlag = 4
    ghosts = []
    g = 0
    while g < 4:
        ghosts.append(Actor("ghost"+str(g+1)
,(270+(g*20), 370)))
        ghosts[g].dir = randint(0, 3)
        g += 1

def inputLock():
    global player
    player.inputActive = False

def inputUnLock():
    global player
    player.movex = player.movey = 0
    player.inputActive = True

init()
pgzrun.go()
```