**MARK VANSTONE**
Educational software author from the nineties, author of the ArcVenture series, disappeared into the corporate software wasteland. Rescued by the Raspberry Pi!
**technovisualeducation.co.uk**
**twitter.com/mindexplorers**

PYGAME ZERO PART 03

# THE SCRAMBLED CAT GAME

A challenging game with minimal code using Pygame Zero Actors. In this third tutorial of the series, we introduce several new game programming techniques

## You'll Need

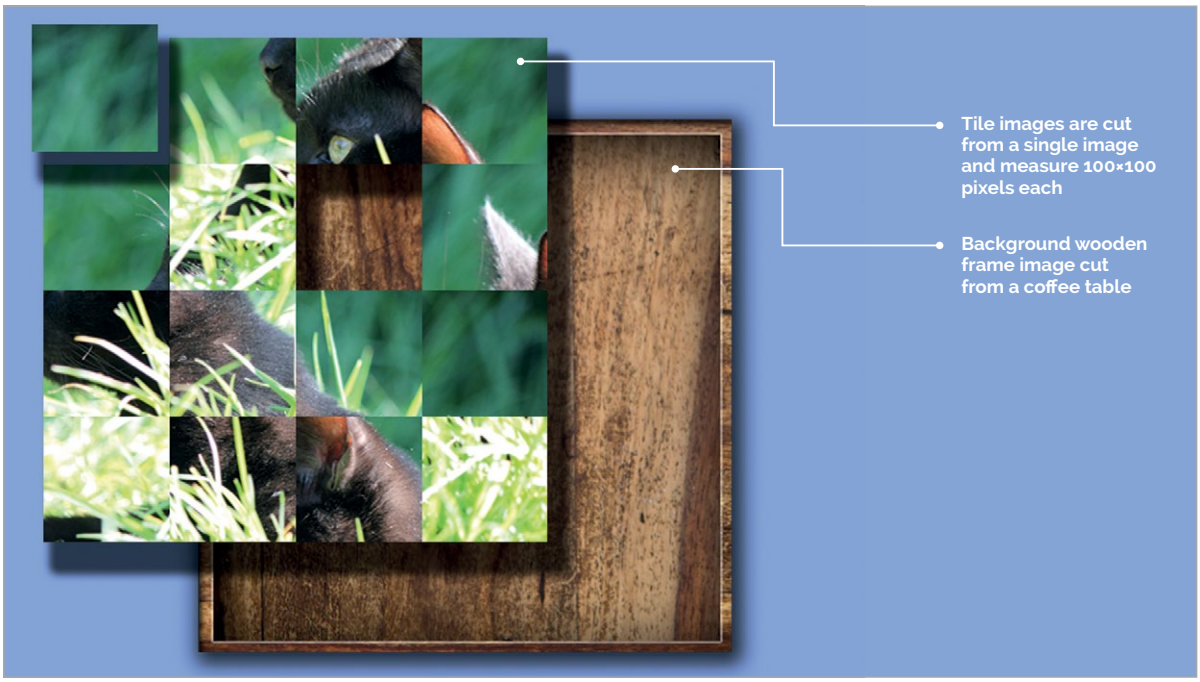
- Raspbian Jessie or newer
- An image manipulation program such as GIMP, or images from **magpi.cc/xdgVLJ**
- The latest version (1.2) of Pygame Zero
- A lot of patience to play the game

When your author first came across this type of game, it was in the form of a plastic frame with numbered tiles and was handed to him by his grandmother to keep him quiet for a while. It's an infuriating game much like a 2D forerunner to the Rubik's Cube, where the player must move jumbled up tiles around the frame to put them back into the correct order, except that there is only one spare space to move tiles into and you seem to end up with an ever more jumbled collection of tiles. In this version, the author's cat, Widdy, has very kindly donated himself to be scrambled.

## >STEP-01

### Decisions, decisions

When first approaching a sliding tile game, our first thought would be to have a matrix variable (or two-dimensional list) representing the frame with its tiles. You would then have a process function to handle the tiles moving around, changing the values in each of the matrix positions. However, Pygame Zero has a few tricks up its sleeve which work very well in this circumstance. Pygame Zero Actors are your friends and can save you a huge wedge of coding time. Let's get stuck in and set



You can find suitable images for games around the house. For this game we used a cat and a coffee table

## WRITING YOUR PROGRAM STRUCTURE

When creating your own programs, you can start by writing a structure of empty functions before writing the detailed code. This can help to visualise the way the program will work.

your program up with a few standard Pygame Zero basics, as in **Figure 1**.

## >STEP-02

### Getting the groundwork done

Before we get into the visuals, you may notice that there are several functions in **Figure 1** with just the keyword **pass** in them. In Python we are not allowed to have an empty function, so if we write **pass** in it, it means that the function should do nothing. This may sound a bit strange but it means that we can build the basic structure of our program before we start thinking about the finer details. Now that we have our structure, we can add some background graphics. If you have downloaded the graphics from our GitHub link (**magpi.cc/xdgVlJ**), we can add a background colour with **`screen.fill(red, green, blue)`** and then 'blit' a frame using **`screen.blit('board', (150, 50))`** in the **`draw()`** function.

## >STEP-03

### A night on the tiles

You may have noticed the rather stylish wooden frame that is now displayed if you run the program. It was carved out of the author's coffee table for this tutorial and will serve as the holder for the tiles we are about to create. All we need to do to create the tiles is to define a list to hold the tile Actors and then a function to create and arrange the new Actors in the frame. We can do this with a double loop to set the x and y co-ordinates of each tile. So first, define the tile list near the top of the code using **`tileList = []`** and then a function to make the tiles. Have a look at **Figure 2** (overleaf) to see how we can do that. We can call our **`makeTiles()`** function at the end of the code, just before **`pgzrun.go()`**.

## Figure1.py

Language

>PYTHON

**DOWNLOAD:**
**magpi.cc/xSnagK**

```
01. import pgzrun
02. WIDTH = 800
03. HEIGHT = 600
04. gameStatus = 0
05.
06. def draw(): # Pygame Zero draw function
07.     pass
08.
09. def update(): # Pygame Zero update function
10.     pass
11.
12. def on_mouse_down(pos):
13.     pass
14.
15. pgzrun.go()
```

**Figure 1** The basic setup for our Pygame Zero program

## >STEP-04

### Stating the obvious

One technique that we have used in both previous tutorials in this series is to use a variable to keep track of the overall state of the game, and this game is no different. We need to know if the player is allowed to interact with the tiles or if they have completed the puzzle and a few other things too. For this we can define a global variable **`gameStatus = 0`** near the top

**COLLISION TESTING**

There are several ways to test collisions. For our border tests, we could have used border Actors and tested collision with them instead of using a mathematical check.

of our program. In this game we will use the value 0 to allow the player to interact with the tiles, then 1 to show that a tile is being moved, 2 will mean that we are preparing the game (more on this later), and 3 will show that the player has completed the puzzle.

## >STEP-05

### Point and click

In this game we will be using two different types of user input to give the player a choice of how to control the tiles. Players will be able to click on a tile to get it to move into the spare space, but we will provide the option to use the arrow keys too. These two input methods work quite differently, so let's look at mouse input first. Pygame Zero provides a function **on_mouse_down(pos)** so that we can capture a mouse click. We then compare the co-ordinates in the variable **pos** to the tiles to see if they collide. If they do, then the player has clicked on a tile.

## Figure2.py

```
01. def makeTiles():
02.     global tileList
03.     xoffset = 251
04.     yoffset = 151
05.     x = y = c = 0
06.     while y < 4:
07.         while x < 4:
08.             if(c < 15):
09.                 tileList.append(Actor("img"+str(c),
    pos = (xoffset+(x*100),yoffset+(y*100))))
10.             c += 1
11.             x += 1
12.         x = 0
13.         y += 1
```

**Figure 2** The `makeTiles()` function. We loop from 0–3 using y and inside the loop from 0–3 using x

## Figure3.py

**Figure 3** Using a locking system while player inputs are being processed

```
01. def on_mouse_down(pos):
02.     if (gameStatus == 0):
03.         doLock()
04.         for t in range(15):
05.             if tileList[t].collidepoint(pos):
06.                 m = moveTile(tileList[t])
07.                 if(m != False):
08.                     animate(tileList[t],on_
    finished=releaseLock, pos=(tileList[t].x+m[1],
    tileList[t].y+m[2]))
09.                     return True
10.         releaseLock()
11.
12. def releaseLock():
13.     global gameStatus
14.     gameStatus = 0
15.
16. def doLock():
17.     global gameStatus
18.     gameStatus = 1
```

## >STEP-06

### Which tile?

To check which tile was clicked, we can use a **for** loop. We know how many tiles we have (15), so we can say **for t in range(15):**, where the variable **t** will be our counter. We can then check each tile to see if it has been clicked, with **if tileList[t].collidepoint(pos):**. While we are dealing with moving the tile, it's a good idea to stop the player from interacting with the game, otherwise we may get several tiles trying to move at once. We can change the **gameStatus** variable to 1 to lock the input. See **Figure 3** to see how this works in code.

## >STEP-07

### Let's get things moving

You may notice in **Figure 3**, in the middle of the **for** loop there is a call to **moveTile(tileList[t])**. This is a function that will determine which way a tile can move. We are going to use the **colliderect()** method of our tile Actors to test all directions. If no collision is detected in a certain direction then we know that we can move the tile that way. We will also include a condition to test that the tiles cannot move outside the frame, so we will be using two types of collision detection: one with the built-in Pygame Zero Actor objects and one with a boundary check.

## >STEP-08

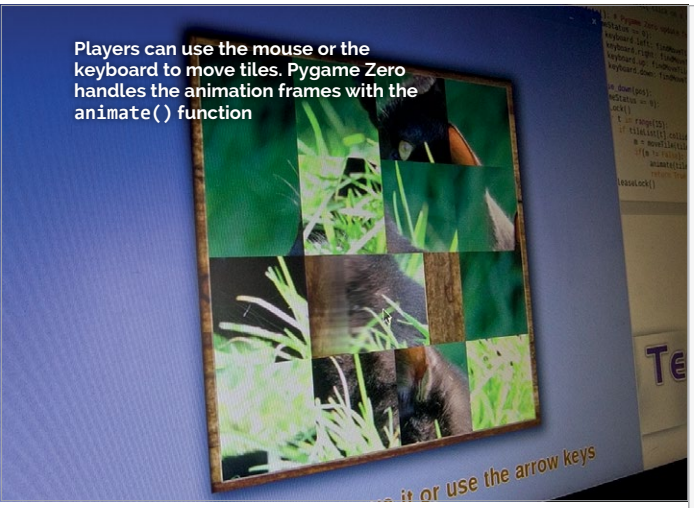
### Repel borders!

Sorry for that terrible pun, but first we need to stop the tiles being allowed through the frame border. We can do this with a simple **if** condition for each direction. So for the right border we would say **if(tile.x < borderRight):** and then we would do our Actor collision test. We will need a border test for each direction. What we are actually doing in the **moveTile()** function is saying: "Can we move the tile right, or left, or up, or down?" If the answer to any of those is yes (and it can only be one of them or none), then tell us which way to move.

## >STEP-09

### Tiles to the left, tiles to the right

There is a cunning plan you can use to test if a tile can move in a certain direction without seeing it move. If you add 1 to the x co-ordinate of the tile and then test for a collision – we are going to use a function called **checkCollide(tile)** – then we will know if a tile is to the right of our test tile. We then set the tile back to its original position by subtracting 1 from the x co-ordinate, and all this happens before the Pygame




Players can use the mouse or the keyboard to move tiles. Pygame Zero handles the animation frames with the `animate()` function

Zero **`draw()`** function is called again so you never see anything move. We can also do the same test for moving right, up, and down.

## >STEP-10

### Collision course

We need to write a function that will test to see if a collision has happened when we moved our tile 1 unit (pixel). The **`checkCollide()`** function loops through the tile list and checks to see if any of the tiles are colliding with the tile that the player has clicked on. We just loop through the tile list and use the **`colliderect()`** method to test for collision (and also make sure we are not testing the tile that has been clicked) and if there is, return **`True`**. If no collision was detected then the function will exit with a **`False`** return value. Have a look at **Figure 4** to see how we test a border and check for tile collision.

## >STEP-11

### Very animated

We will need to add tests for left, up, and down to our **`moveTile()`** function; when that is done, it will return what is known as a tuple. This is a return value with several parts: the direction as a string, the x offset to move the tile, and the y offset to move the tile. This tuple gets sent back to our **`on_mouse_down()`** function and if it's not **`False`** then we know we can move the tile based on the return values. We now call the **`animate()`** function. This is a Pygame Zero function to move an Actor from one place to another.

## >STEP-12

### Locking player interaction

While we are animating the tile moving, we don't want the player to be able to click on any other tiles. We have used our **`doLock()`** function to change the **`gameStatus`** to 1 so no mouse clicks are reacted to. When the animation has finished, we want the player to make their next move, so in our call to **`animate()`** we include **`on_finished=releaseLock`**. This will call the function **`releaseLock()`** which will set the **`gameStatus`** back to 0. This will mean that the player can click on another tile. You will notice from **Figure 3** that if the mouse click is not on a tile then the lock is released at the end of the function.

## Figure4.py

**Figure 4** An example of testing border collision and then using the Actor collision detection

```
01. def moveTile(tile):
02.     borderRight = 551
03.     rValue = False
04.     if(tile.x < borderRight): # can we go right?
05.         tile.x += 1
06.         if(not checkCollide(tile)): rValue = "right",
    100, 0
07.         tile.x -= 1
08.
09.     return rValue
10.
11. def checkCollide(tile):
12.     for t in range(15):
13.         if tile.colliderect(tileList[t]) and tile !=
    tileList[t]: return True
14.     return False
```

## >STEP-13

### Using the arrows

The second way that we can capture the player's input is to use the arrow keys. This form of input relies on the fact that if an arrow key is pressed (for example the up arrow), there is only one tile that is able to move in that direction – or alternatively, no tiles are able to move in that direction. So all we need to do is work out which tile can move in the direction of the arrow pressed. We are going to do this with a function called **`findMoveTile(moveDirection)`** and we pass it the direction of movement that we want it to look for.

## >STEP-14

### Scanning the keyboard

First, we must check if the player is pressing one of the arrow keys. We do this in our **`update()`** function. We want to check if the **`gameStatus`** is 0 before processing anything; if it is, we can check the left arrow key by writing **`if keyboard.left: findMoveTile("left")`**. We then write a similar line of code for each of the other arrow keys, passing a different string to our **`findMoveTile()`** function. In the latter function we use the same **`moveTile()`** function as we did with the mouse click input, but this time we check for the direction that has been passed by the keyboard press check.

## >STEP-15

### Which tile is it?

Now all we need to do is write the **`findMoveTile()`** function that scans through the tile Actors and

### MULTI-PURPOSE FUNCTIONS

Try to write your functions so that they can be used in different ways, like our `moveTile()` function. If there is less code to write, there is less code to debug.

### WATCH OUT FOR MULTIPLE INPUT

When using the Pygame Zero keyboard object to test for key presses, it doesn't return just a single event. It will continuously read as True while the player holds the key, which may not be what you are expecting.

## Figure5.py

**Figure 5** Moving tiles from the detection of a key press on the arrow keys

```
01. def update(): # Pygame Zero update function
02.     if (gameStatus == 0):
03.         if keyboard.left: findMoveTile("left")
04.         if keyboard.right: findMoveTile("right")
05.         if keyboard.up: findMoveTile("up")
06.         if keyboard.down: findMoveTile("down")
07.
08. def findMoveTile(moveDirection):
09.     doLock()
10.     for t in range(15):
11.         m = moveTile(tileList[t])
12.         if(m != False):
13.             if(m[0] == moveDirection):
14.                 animate(tileList[t],on_
    finished=releaseLock, pos=(tileList[t].x+m[1],
    tileList[t].y+m[2]))
15.                 return True
16.     releaseLock()
17.     return False
```

## Figure6.py

**Figure 6** The `scrambleCat()` function. Pass a list of simulated keystrokes to `findMoveTile()`, just like if we were pressing the keys

```
01. scrambleCountdown = 30
02. scrambleList = [2, 0, 2, 0, 3, 1, 1, 1, 3, 0, 0, 2,
    1, 2, 1, 3, 3, 0, 3, 0, 2, 0, 2, 2, 1, 3, 1, 3, 3,
    1, 2]
03.
04.
05. def scrambleCat():
06.     global gameStatus, scrambleCountdown,
    scrambleList
07.     tileDirs = ["left", "right", "up", "down"]
08.     if(scrambleCountdown > 0):
09.         mt = False
10.         while(mt == False):
11.             mt = findMoveTile(tileDirs[scrambleList[
    scrambleCountdown]])
12.             scrambleCountdown -= 1
13.         gameStatus = 2
14.     else:
15.         gameStatus = 0
```

finds the one that can move in the direction that the player has pressed. We loop though our tile list and try to move each tile. If we find it can move then we check the direction of movement. If it matches the direction that we are looking for then we have a match and we can initiate the animation to move the tile. **Figure 5** shows this whole process, from key press to animation. Note that we are locking and unlocking the **gameStatus** while this is happening, to avoid multiple moves at once.

## >STEP-16

### Time to scramble the cat

The game is not much fun if we start with a completed puzzle, so we need to add a system for mixing up the tiles. There are various ways we could do this, but we thought it might be quite nice if we start the game with the tiles being scrambled move by move. We can do this by simulating key presses using the same functions as in the previous steps. We can use a new **gameStatus** of 2 to indicate that the player can't interact, then cycle through a predefined or random set of simulated key presses until everything is jumbled up. We have used a predefined list in this case.

## >STEP-17

### How scrambled is scrambled?

For our scrambling, we can have a list of movements. If we alter the number of movements we are using then the game becomes more or less difficult. We can start at 30 and see how that works. Our scrambling function, **scrambleCat()**, calls the **findMoveTile()** function and uses the **scrambleCountdown** variable to check if we have made enough moves. After each animation, we are using a bit of a cunning trick: we always call the **releaseLock()** function after an animation so we can slip in a test to see if gameStatus is 2 and if so, call **scrambleCat()** again. See **Figure 6** for the **scrambleCat()** function.

## >STEP-18

### Have we won yet?

At some point in the game, there is a slim chance that the player will actually rearrange the tiles back into the correct order. We will need to write a check to see if this has happened each time a tile has been moved. We know that the positions of the tiles were correct before we scrambled them, so what we can do is make a list of the x and y co-ordinates of each tile when we first make the tile Actors. Then all we need to do is to compare that list with the current x and y values of our tile Actors and if they all match, we have a winner!

## >STEP-19

### Finishing touches

You can see from the full program listing the **checkSuccess()** function and how that works, and we can also add some text prompts into our **draw()** function based on the value of **gameStatus**. That's about it! We have a working Scrambled Cat game. You may want to add some features such as a score based on how many moves the player makes, or you may want to change the images. You could have scrambled egg or a scrambler bike or make the tile matrix larger for an even more difficult challenge.

# scrambledcat.py

```
import pgzrun
WIDTH = 800
HEIGHT = 600
gameStatus = 0
tileList = []
correctList = []
scrambleCountdown = 30
scrambleList = [2, 0, 2, 0, 3, 1, 1, 1, 3, 0, 0, 2, 1, 2, 1, 3, 3, 0, 3, 0, 2, 0, 2, 2, 1, 3, 1, 3, 3, 1, 2]

def draw(): # Pygame Zero draw function
    global gameStatus
    screen.fill((141, 172, 242))
    screen.blit('board', (150, 50))
    for t in range(15):
        tileList[t].draw()
    if (gameStatus == 3): screen.draw.text("Success!", (315, 20), owidth=0.5, ocolor=(255,255,255), color=(128,64,0) , fontsize=60)
    if (gameStatus == 2): screen.draw.text("Please wait while we scramble the cat", (135, 540), owidth=0.5, ocolor=(255,255,255), color=(128,64,0) , fontsize=40)
    if (gameStatus <= 1): screen.draw.text("Click on a tile to move it or use the arrow keys", (95, 540), owidth=0.5, ocolor=(255,255,255), color=(128,64,0) , fontsize=40)

def update(): # Pygame Zero update function
    if (gameStatus == 0):
        if keyboard.left: findMoveTile("left")
        if keyboard.right: findMoveTile("right")
        if keyboard.up: findMoveTile("up")
        if keyboard.down: findMoveTile("down")

def on_mouse_down(pos):
    if (gameStatus == 0):
        doLock()
        for t in range(15):
            if tileList[t].collidepoint(pos):
                m = moveTile(tileList[t])
                if(m != False):
                    animate(tileList[t],on_finished=releaseLock, pos=(tileList[t].x+m[1], tileList[t].y+m[2]))
                    return True
        releaseLock()

def findMoveTile(moveDirection):
    doLock()
    for t in range(15):
        m = moveTile(tileList[t])
        if(m != False):
            if(m[0] == moveDirection):
                animate(tileList[t],on_finished=releaseLock, pos=(tileList[t].x+m[1], tileList[t].y+m[2]))
                return True
    releaseLock()
    return False

def releaseLock():
    global gameStatus
    if(gameStatus == 2): scrambleCat()
    else: gameStatus = checkSuccess()

def doLock():
    global gameStatus
    gameStatus = 1

def checkSuccess():
    for t in range(15):
        if(tileList[t].x != correctList[t][0] or tileList[t].y != correctList[t][1]):
            return 0
    return 3     # we have success!

def makeTiles():
    global tileList, correctList
    xoffset = 251
    yoffset = 151
    x = y = c = 0
    while y < 4:
        while x < 4:
            if(c < 15):
                tileList.append(Actor("img"+str(c), pos = (xoffset+(x*100),yoffset+(y*100))))
                correctList.append((xoffset+(x*100),yoffset+(y*100)))
            c += 1
            x += 1
        x = 0
        y += 1
    scrambleCat()

def scrambleCat():
    global gameStatus, scrambleCountdown, scrambleList
    tileDirs = ["left", "right", "up", "down"]
    if(scrambleCountdown > 0):
        mt = False
        while(mt == False):
            mt = findMoveTile(tileDirs[scrambleList[scrambleCountdown]])
            scrambleCountdown -= 1
        gameStatus = 2
    else:
        gameStatus = 0

def moveTile(tile):
    borderRight = 551
    borderLeft = 251
    borderTop = 151
    borderBottom = 451
    rValue = False
    if(tile.x < borderRight): # can we go right?
        tile.x += 1
        if(not checkCollide(tile)): rValue = "right", 100, 0
        tile.x -= 1
    if(tile.x > borderLeft): # can we go left?
        tile.x -= 1
        if(not checkCollide(tile)): rValue = "left", -100, 0
        tile.x += 1
    if(tile.y < borderBottom): # can we go down?
        tile.y += 1
        if(not checkCollide(tile)): rValue = "down", 0, 100
        tile.y -= 1
    if(tile.y > borderTop): # can we go up?
        tile.y -= 1
        if(not checkCollide(tile)): rValue = "up", 0, -100
        tile.y += 1
    return rValue

def checkCollide(tile):
    for t in range(15):
        if tile.colliderect(tileList[t]) and tile != tileList[t]: return True
    return False

makeTiles()
pgzrun.go()
```