Part 01

CODE AN ISOMETRIC ADVENTURE GAME: AMAZEBALLS

Pygame Zero in 3D. Yes it's possible and we will show you how in this three-part maze game tutorial

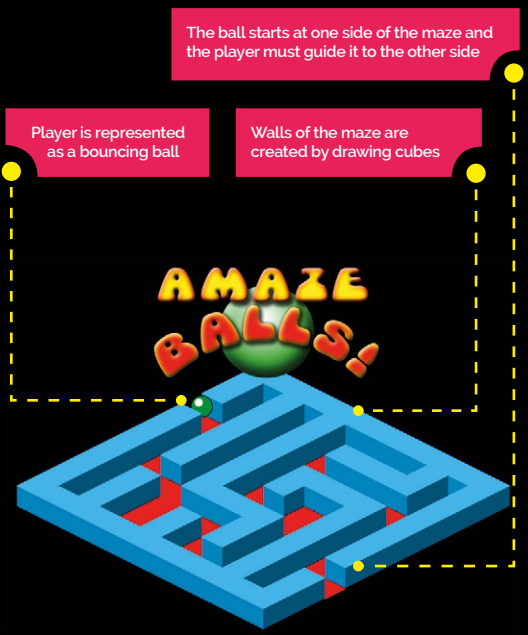
In this series so far, we have learnt how to use Pygame Zero to quickly create games. In this three-part tutorial, we will use several more new techniques to create a 3D maze game. The style of 3D graphics we'll be using is called isometric. That means that our display will be made of regular cubes that have a slightly false perspective because the display is actually made of 2D images. This technique was used in many eighties games such as Knight Lore, Alien 8, and my own series, ArcVenture. In this first part, we'll build the basic map using data lists, and create a bouncing ball character for the player to move around the maze.

01 Welcome to the third dimension

Pygame Zero was not written with 3D games in mind, but sometimes you just have to push the boundaries a little and see how far you can take an idea. This method of drawing a game area is not strictly speaking a 3D display, but it does look 3D and we can do a lot with this technique. We are going to make a map from list data and build it up out of cubes. We will then put a bouncing ball into the game area for the player to move around using the keyboard. We'll start the ball at one side of the maze and when the player guides it to the other side, the game is complete.

02 Map-making

As usual in this series, we start with importing our `pgzrun` module and don't forget



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

to call `pgzrun.go()` right at the end of the code. The default window size should suit our purposes. We are going to make a two-dimensional list of numbers that will represent our map. Each number will represent one square on the map and we will make our map twelve squares wide and twelve squares high. See **figure1.py** to see how we define this list, which we will call `mapData`. You will also see another variable called `mapInfo` that defines the width and height of our map in a structure called a dictionary.

03 A is for aardvark

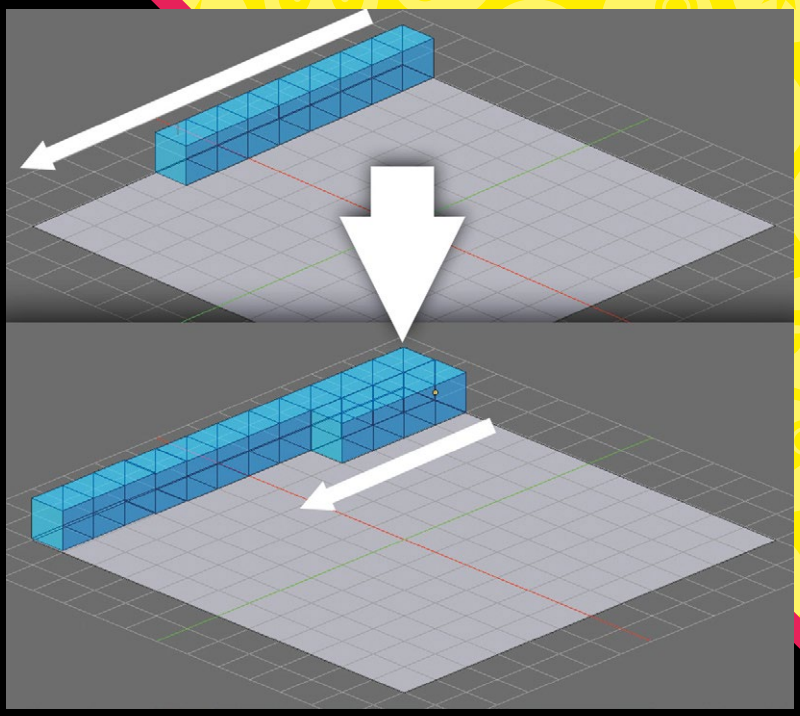
A dictionary in Python is a very useful data structure. Rather than just storing a list of values or strings, we can give each of these values a label. Looking at our `mapInfo` dictionary, we can see that there are two values declared; the first has the label 'width' and the second 'height'. To read the values back, we just need to write `mapInfo["width"]`, which would give us the value 12 in this case. Dictionaries are a very useful structure to gather together several variables that are all associated and can be referred to by their labels.

04 Mapping the map

Now that we have some data for our map, we need to define some images that are going to be used to draw the map. We can make a new list of map blocks by writing `mapBlocks = ["map1c","map2c"]`. What this will do is define that when we see a 0 in the `mapData` list, it means draw the first image in the list, which is **map1c**. If we see a 1 in the data, then use image **map2c**. For the moment, we'll just stick to two different map blocks. The first will represent the floor; the second, the walls.

05 Show me the map

Next is to get our map data translated into a visual map we can see on the screen. We'll set up the basics in our Pygame Zero `draw()` function and then call a `drawMap()` function that we'll write to do the work. In **figure2.py**, you'll see that we fill the screen with black first, then draw our map. The `drawMap()` function, although short, may look a bit complicated, so let's go through it slowly as you'll need to understand what is happening here.



▲ The blocks build back to front, one row at a time, to produce the 3D effect

▼ Our basic Pygame Zero framework and our map data definitions

figure1.py

DOWNLOAD THE FULL CODE:

magpi.cc/DVNCvv

Language: **Python**

```
001. import pgzrun
002.
003. mapData = [[1,1,1,0,1,1,1,1,1,1,1,1],
004.            [1,0,0,0,0,0,0,0,0,0,0,1],
005.            [1,1,1,1,1,1,1,0,1,1,1,1],
006.            [1,0,0,0,0,0,0,0,0,0,0,1],
007.            [1,1,1,1,1,1,1,1,0,0,0,1],
008.            [1,0,0,0,0,0,0,1,0,1,1,1],
009.            [1,0,1,0,1,1,0,1,0,0,0,1],
010.            [1,0,1,0,1,0,0,1,1,1,0,1],
011.            [1,0,1,0,1,0,0,0,0,0,0,1],
012.            [1,1,1,0,1,1,1,1,1,1,1,1],
013.            [1,0,0,0,0,0,0,0,0,0,0,1],
014.            [1,1,1,1,1,1,1,1,0,1,1,1]]
015.
016. mapInfo = {"width":12, "height":12}
017.
018. pgzrun.go()
019.
```

figure2.py

```
001.  OFFSETX = 368
002.  OFFSETY = 200
003.  mapBlocks = ["map1c","map2c"]
004.  mapHeight = [0,32]
005.
006.  def draw(): # Pygame Zero draw function
007.      screen.fill((0, 0, 0))
008.      drawMap()
009.
010.  def drawMap():
011.      for x in range(0, 12):
012.          for y in range(0, 12):
013.              screen.blit(mapBlocks[mapData[x][y]], ((x*32)-
      (y*32)+OFFSETX,
014.                          (y*16)+(x*16)+OFFSETY -
      mapHeight[mapData[x][y]]))
015.
```

06 Building blocks

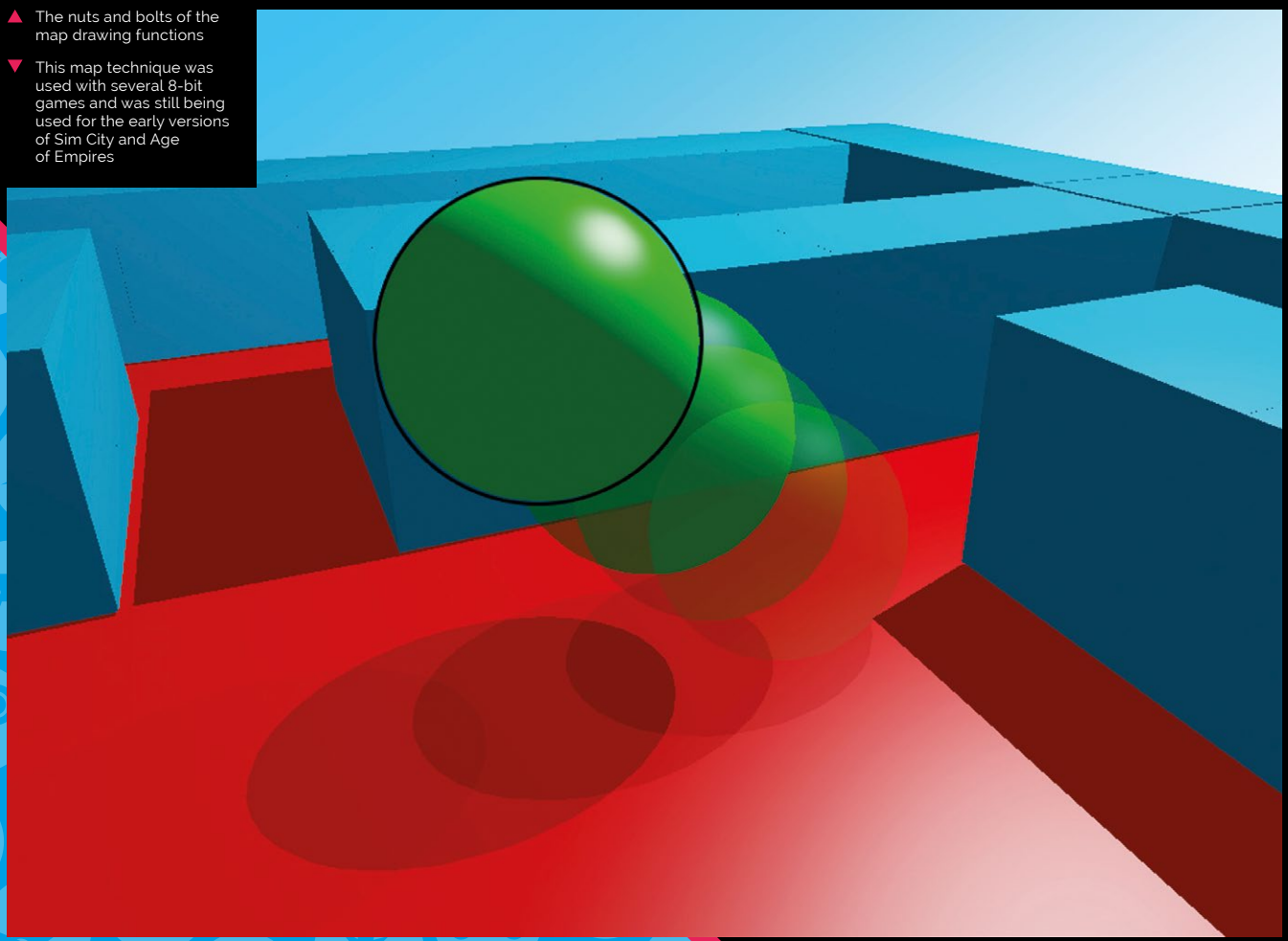
We are going to use a nested loop to run through our data and display the map. The first loop is for the x direction and inside that we have another loop for the y direction. The x and y in this case do not refer to screen pixel coordinates but to the data blocks. Now imagine that we are going to turn our data 45 degrees so that we draw our blocks diagonally down the screen. We do this with a bit of maths to translate the x and y positions in our data to locations on the screen.

07 An amazing view

To draw each block, we use the Pygame Zero `screen.blit()` function, which loads an image and draws it at the specified coordinates on the screen. So the first parameter of this function is the name

▲ The nuts and bolts of the map drawing functions

▼ This map technique was used with several 8-bit games and was still being used for the early versions of Sim City and Age of Empires



of the image. We obtain this by getting the value from `mapData[x][y]` and use `mapBlocks` to give us an image name. Now we calculate the screen coordinates. The width of each block is 64 pixels, but because we are printing diagonally we only want to move 32 pixels sideways for each block so that they overlap each other. In the down direction we move 16 pixels for each block.

08 Diamonds are forever

To get our screen x coordinate, we multiply the data x value by 32 and subtract that from the data y value times 32. That gives us a screen x coordinate starting from 0. We then add a predefined offset to move the starting block to the middle of the screen. We do the same to get the screen y coordinate but use a multiplier of 16 for the data x and y, add an offset to move the start block down the screen a bit, and take into account that some blocks are taller than others using `mapHeight`. When this runs, we'll see twelve rows and twelve columns drawn in a diamond shape.

> We'll make a dictionary to hold all the data we need to know about the player

09 Balls

Now it's time to make our player character, which in this case will be represented by a bouncing ball. We'll start by getting the ball positioned and moving around the map first. We'll make a dictionary to hold all the data we need to know about the player. See **figure3.py** for how we define the player data. The `x` and `y` values are where the player is in the block data map. We will start the player in column(`x`) 0 and row(`y`) 3. The `frame` value will tell us which frame of animation to show. The `sx` and `sy` values are the actual screen coordinates where the ball will be drawn.

10 Blitting the player

Leaving aside the other values in our player dictionary for the moment, the other piece of code in **figure3.py** goes inside our `drawMap()` loops just after we blit the blocks. The code says "if the block we are currently dealing with is the block that the player is on, work out the screen coordinates (using the same calculation as the blocks) and draw the ball to the screen." We only work out the screen coordinates of the ball once, as we'll now make a `doMove()` function that will handle the player screen coordinates from here.

11 On the move

Our `doMove()` function will introduce a couple of new Python techniques and some more of the player dictionary data. We pass `doMove()` three parameters. The first is the player dictionary structure (so we can read and write player values), and then the x and y change we want to make in block units. The `x` and `y` will be 0, 1 or -1 and represent a movement in our map block data. The first bit of `doMove()` will check to make sure that the block we are moving to is within the bounds of our map. This is a shorthand way of comparing several values and in simple terms is *0 <= x < width*, which means x needs to be between 0 and width minus 1. See **figure4.py** for the actual code to use.

Data formatting

If you are using hand-coded data, it's a good idea to format it in a way you can read it easily.

▼ Defining the player data structure and drawing the player character to the screen

figure3.py

```
001. # This code is near the top of our program
002.
003. player = {"x":0, "y":3, "frame":0, "sx":0, "sy":0,
004.           "moveX":0, "moveY":0, "queueX":0, "queueY":0,
005.           "moveDone":True, "movingNow":False,
     "animCounter":0}
006.
007.
008. # This code goes in the drawMap() function inside the y loop
009.
010.             if x == player["x"] and y == player["y"]:
011.                 if player["sx"] == 0:
012.                     player["sx"] = (x*32)-(y*32)+OFFSETX
013.                     player["sy"] = (y*16)+(x*16)+OFFSETY-32
014.                 screen.blit("ball"+str(player["frame"]),
     (player["sx"], player["sy"]))
015.
016.
```

figure4.py

```
001.  def update(): # Pygame Zero update function
002.      global player
003.      if player["moveDone"] == True:
004.          if keyboard.left:
005.              doMove(player, -1, 0)
006.          if keyboard.right:
007.              doMove(player, 1, 0)
008.          if keyboard.up:
009.              doMove(player, 0, -1)
010.          if keyboard.down:
011.              doMove(player, 0, 1)
012.      updateBall(player)
013.
014.  def doMove(p, x, y):
015.      if 0 <= (p["x"]+x) < mapInfo["width"] and 0 <=
      (p["y"]+y) < mapInfo["height"]:
016.          if mapData[p["x"]+x][p["y"]+y] == 0:
017               p.update({"queueX":x, "queueY":y,
      "moveDone":False})
018.
```

▲ Getting keyboard input and responding by setting up the data for moving the player character

12 Watch your step

After we have checked the player movement will be inside the map area, we can test to see if the movement will be to a floor block (value 0 in our data). We just need to check the value in `mapData` and we are good to go. The next line of code is a clever way of changing several values in a dictionary. We use the dictionary `p.update()` function to set `queueX`, `queueY`, and `moveDone` values all at the same time. The reason we are queueing the movement rather than just moving is because we may already be moving and we want to wait until the end of the previous move.

> "The reason we are queueing the movement rather than just moving is because we may already be moving"

Dictionaries

Organising data in dictionaries makes it easier to understand what the data is used for. It's also a good stepping stone towards using object-oriented programming (OOP).

13 The update

Staying with **figure4.py**, we can see how to get our movement controls from the keyboard. In the Pygame Zero function `update()` we look for the cursor keys being pressed and if so, call our `doMove()` function with suitable movement parameters. When we have checked for movement, we then call a function `updateBall()` which will do all the heavy lifting of animating the ball and moving it from one block to the next. You will notice that before we check the keyboard, we make sure that we are ready for more input by checking the player dictionary value `moveDone`, which we set to `False` in `doMove()`.

14 Sequencing the animation

All we have left to do now is get the ball to move from one block to the next, but if we get things in the wrong sequence we can end up with the ball being drawn in front of blocks that it is meant to be behind, or behind blocks that it should be in front of. At this stage we can bring in the changing frames of the animation to make the ball bounce up and down. We also want to make the ball move smoothly from one place to another, so the smaller the movement from one `draw()` to the next, the better.

15 You've been framed

Let's start with the animation frames for the bouncing ball. We have eight frames named **ball0.png** up to **ball7.png**. If we just increase the frame value in our player dictionary each time we call `updateBall()` and when we get to 8 set the value back to 0, our `drawMap()` function will take care of drawing the animation in a loop. The only problem with this is that if we run the animation at this speed, the ball is bouncing very fast so we need to slow it down. For this we use another value from our player dictionary, `animCounter`. With this value, we count every four frames and on the fourth frame we add one to the `frame` value.

16 Frame by frame

We need to time the movement of the ball with the correct frames so that it looks like it's

bouncing smoothly while moving. We want to wait until `frame = 4` before starting the move. At that point we move our `queueX` and `queueY` values into `moveX` and `moveY`, and set `movingNow` to `True`. When `movingNow` is `True`, the `sx` and `sy` values of the player are changed. For each block, we need to move 32 pixels left or right in 1 pixel increments, and 16 pixels up or down in 0.5 pixel increments. So our move will take 32 update cycles.

17 One block to the next

For our moving ball to be displayed correctly in the drawing order of the map, we need to change the block it is located at on the correct frame of the animation. When `frame = 7` is the time to change, so at that point we update the player dictionary `x` and `y` values based on the `moveX` and `moveY` values. We then set the player `moveDone` value to `True`. This will mean that when we return to `frame = 4`, we can clear `moveX` and `moveY`, and set `movingNow` to `False` unless another move has been queued.

18 Wrapping it up

You can see from **figure5.py** how this frame sequencing works in the `updateBall()` function. You will see the check for `movingNow` first and use a separate function, `moveP()`, to change the screen coordinates of the player. Then we do all the logic around sequencing actions to the current frame. You'll also see a check to see if the maze has been solved. We set a global variable, `mazeSolved`, to `False` at the start and if the player arrives at block 11, 8 we set the variable to `True` and display a suitable message in `draw()`.

19 Level one complete

So that's the first part of this tutorial. We have looked at creating a 3D-looking map from data and 2D images, and how to move an animated character around the map. This game format has a lot of possibilities and in the next episode we'll look at making the map larger, editing the map data in an external editor, and loading the data from a separate file.

figure5.py

```
001. def updateBall(p):
002.     global mazeSolved
003.     if p["movingNow"]:
004.         if p["moveX"] == -1: moveP(p,-1,-0.5)
005.         if p["moveX"] == 1: moveP(p,1,0.5)
006.         if p["moveY"] == -1: moveP(p,1,-0.5)
007.         if p["moveY"] == 1: moveP(p,-1,0.5)
008.     p["animCounter"] += 1
009.     if p["animCounter"] == 4:
010.         p["animCounter"] = 0
011.         p["frame"] += 1
012.         if p["frame"] > 7:
013.             p["frame"] = 0
014.         if p["frame"] == 4:
015.             if p["moveDone"] == False:
016.                 if p["queueX"] != 0 or p["queueY"] !=0:
017.                     p.update({"moveX":p["queueX"],
    "moveY":p["queueY"], "queueX":0, "queueY":0,
    "movingNow": True})
018.             else:
019.                 p.update({"moveX":0, "moveY":0,
    "movingNow":False})
020.                 if p["x"] == 11 and p["y"] == 8:
021.                     mazeSolved = True
022.
023.         if p["frame"] == 7 and p["moveDone"] == False and
    p["movingNow"] == True:
024.             p["x"] += p["moveX"]
025.             p["y"] += p["moveY"]
026.             p["moveDone"] = True
```