Part 02

Code an isometric adventure game: AmazeBalls

Pygame Zero in 3D. Let's make the map bigger in part two of this three-part tutorial

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
- Tiled (free map editor) **mapeditor.org**
- An image manipulation program such as GIMP, or images available from **magpi.cc/fPBrhM**
- The latest version of Pygame Zero (1.2)

In this second part of our tutorial on using Pygame Zero to create a 3D isometric game, we will start from where we left off in the last part and look at ways to make our 3D area larger and easier to edit. This will mean using a map editor called Tiled, which is free to download and use, to make your own 3D maps. We'll learn how to create a simple map and export it from Tiled in a data format called JSON. We will then import it into our game and code our draw function to scroll around the map area when the player moves. Lots to do, so let's get started.

01 Tools for the job

In the previous part of this tutorial, we made our map data by writing a two-dimensional list of zeroes and ones which represented either a floor block or a wall block. The player was able to move onto any block that was a zero in the data. This time we are going to get a map-editing app to do the work for us instead of typing the data in. First, we need to get Tiled installed. You can find the Tiled homepage at **mapeditor.org**, where options are given to support the developer if you like what they are creating.

02 Getting Tiled

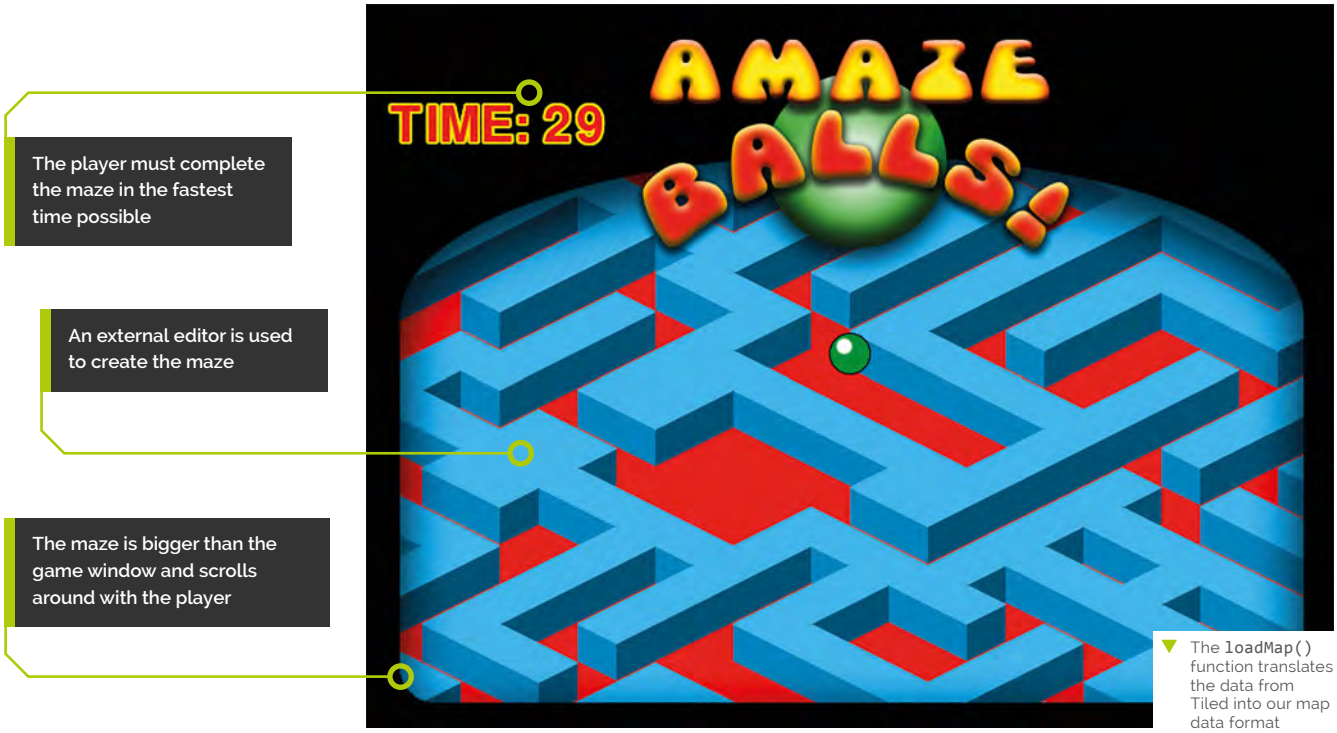
Tiled can be used on many different systems, including PCs and Mac computers. Also, more importantly for us, it works really well on Raspbian and Raspberry Pi. It's also super-easy to install. All you need to do is open up a Terminal window, then make sure you're online and are up-to-date by typing `sudo apt-get update`. You may need to type `sudo apt-get upgrade` too, depending on how long you've left your Pi without updates. When those have run, just type `sudo apt-get install tiled`, hit **RETURN**, and you should see Tiled being installed. When it's done, a new Tiled icon will appear in your Graphics submenu.

03 Cartography

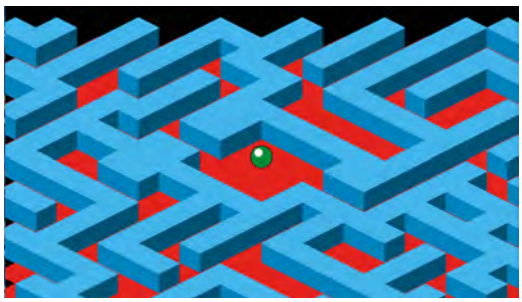
Previously our map was 12 blocks by 12, which all fitted on the screen at once. With our map editor we can make a much bigger map. It could be huge, but for the moment let's stick to a grid of 30 by 30 blocks. You may want to download our ready-made map and blocks from **magpi.cc/fPBrhM**. If you load in the map, you should see a blue and red maze, a bit like in part one but much bigger. This time, though, we have added a new block to indicate the finish point in the maze. You should be able to scroll around to see the whole map.

04 Exporting the data

Have a play around with the map editor; there is some great documentation on the website. When you have familiarised yourself with how it works, we can think about the task of getting the map data into our game. To export the data, go to Export in the File menu and when the dialogue box opens up, asking 'export as...', find a suitable place (perhaps a subdirectory called **maps**) to save the map (perhaps as **map1**), but in the drop-down labelled 'Save as type', select 'Json map files (*.json)'. This type of file (pronounced 'Jay-son', short for JavaScript Object Notation) can be viewed in a text editor.



▼ The `loadMap()` function translates the data from Tiled into our map data format



▲ When we have rewritten our `drawMap()` function, we will see some jagged edges around the extremities of the drawing area

05 JSON and the Argonauts

If we open the JSON file, we will see a load of curly and square brackets with words and numbers spread all over the place, but before you proclaim 'It's all Greek to me!', let's have a look at some of the elements so that we can understand the structure of the data. If you are familiar with the JavaScript language, you'll recognise that the curly brackets { and } are used to contain blocks of code or data, and square brackets [and] are used for lists of data. Look at the element called 'layers' and you will see data describing our map.

06 Loading the data

For this game we don't actually need all the data that is in the JSON file, but we can load it all in and just use the bits we need. Let's make

figure1.py

```
001. import json
002. import os
003.
004. def loadmap(mp):
005.     with open(mp) as json_data:
006.         d = json.load(json_data)
007.     mapdata = {"width":d["width"], "height":d["height"]}
008.     rawdata = d["layers"][0]["data"]
009.     mapdata["data"] = []
010.     for x in range(0, mapdata["width"]):
011.         st = x*mapdata["width"]
012.         mapdata["data"].append(
     rawdata[st:st+mapdata["height"]])
013.
014.     tileset = "maps/" + d["tilesets"][0]["source"].replace(
     ".tsx",".json")
015.     with open(tileset) as json_data:
016.         t = json.load(json_data)
017.
018.     mapdata["tiles"] = t["tiles"]
019.     for tile in range(0,len(mapdata["tiles"])):
020.         path = mapdata["tiles"][tile]["image"]
021.         mapdata["tiles"][tile]["image"] =
     os.path.basename(path)
022.         mapdata["tiles"][tile]["id"] =
     mapdata["tiles"][tile]["id"]+1
023.     return mapdata
```

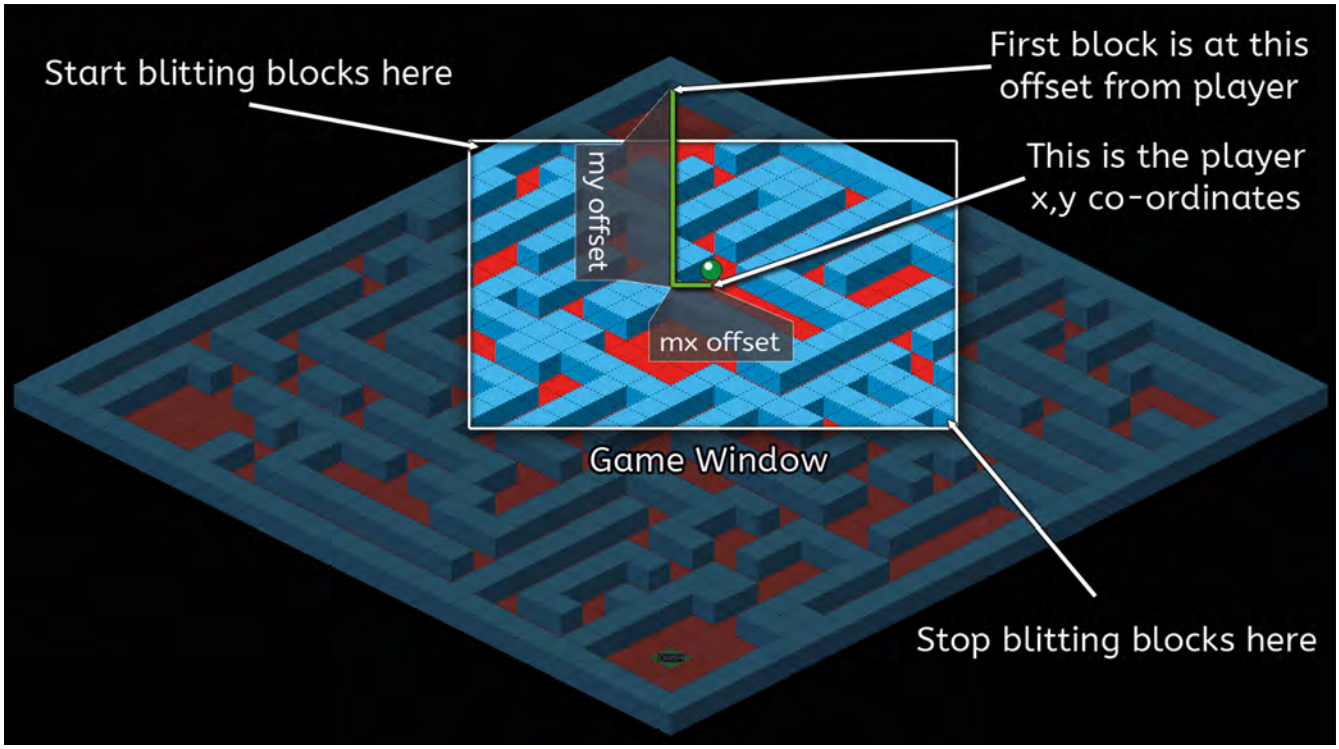


Figure 2 The coordinates that the map starts from are calculated as an offset from the player, and the maze blocks are only drawn inside the rectangle of the game window

a new module to deal with map handling, like we have done in previous tutorials (see *The MagPi* #76, **magpi.cc/76**). Let's make a new module called **map3d.py**. Python provides a module to import JSON files, so we can write `import json` at the top of our **map3d.py** file to load the module. We'll also need to use the os module for handling file paths, so import that too. Then we just need to write a function to load our map.

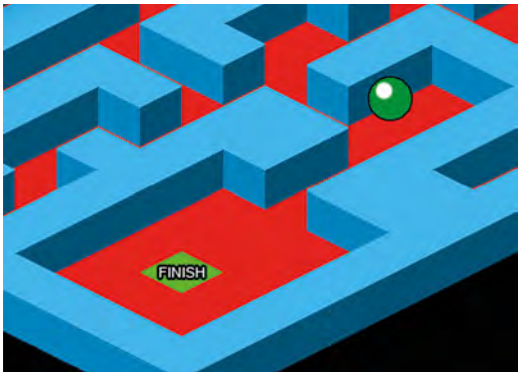
Top Tip

Looking at JSON

You can look at JSON files in any text editor, but a programming one is probably best – perhaps try Geany.

07 Getting what we need

Let's make a function called `loadmap()` and use a parameter called `mp` to pass a file location to the function. Have a look at **figure1.py** to see how we write this. You can see that we load in our map data into a variable `d` using the function `json.load()`. Then we can copy the width and height into a dictionary called `mapdata`. This dictionary will hold all the data we need by the time we get to the end of the function. Having made a temporary copy of the block data (`rawdata`), we can then loop through the values and put them in the format we want in `mapdata`.



We have added a new tile for the finish. When the player moves onto this block, the maze is solved

08 The tileset connection

One bit of information that we need is where to find the details about which images to use on each block. This is included as a value called 'tilesets'. In this case we will assume that we only have one tileset defined, so we can read it in and go and find our block images. Except there is a slight crinkle in the plan. Our map data refers to the tileset file as a .tsx file. What we need to do is go back to Tiled and export our tileset from the tileset editor as a JSON file. Then, when we import it, we just switch the .tsx extension for .json.

09 A night on the tiles

Once we have loaded in our tileset data as a JSON file, we can then loop through the tiles and get the names of each of our block images. You will note that we add one to the id value to match the values that Tiled has exported. When we have all that data in our `mapdata` dictionary, we can pass that back to our main program by writing `return mapdata`. Going back to our main

program, we'll need to add an import for our map3d module at the top of the code and then, before our `draw()` function, we can write `mapData = map3d.loadmap("maps/map1.json")` instead of our list of zeroes and ones.

10 Thinking big

In part one of this tutorial, our maze was 12×12 blocks, which just fitted nicely into the game area. Now we have a 30×30 maze which, if we draw it all, will go off the sides of the game window. We need to be able to scroll the map around the screen as the player moves through the maze. The way we can do this is to keep the bouncing ball in the centre of the game area and, as the player moves, scroll the map. So, in effect, what we are saying is that we are going to draw the map relative to the player rather than relative to the game window.

11 It's all relative

To draw our map, we are going to use the same basic loops (x and y) as before, but we will start drawing our map based on the coordinates we calculated for the player screen x and y coordinates. With that starting screen position, we loop in a range that is either side of the player in both directions. See **Figure 2** for a visual explanation of what we are doing in these loops. In simple terms, what we are saying is: 'Start drawing the map from coordinates that will make the player appear in the centre of the window. Then only draw the blocks that are visible in the window.' See **figure3.py** for how we have changed the `drawMap()` function to do this.

12 Extra functions

You will see in **figure3.py** that we have a couple of new functions that we have not defined yet. The first is `onMap()`, which we pass an x and a y coordinate to. These are block locations which we test to make sure that the coordinates we are asking for are actually on our map, otherwise we will get an error. If the x or y are less than 0 or greater than the width (or height) of the map, then we can ignore it. The other function is `findData()`. This function finds the data associated with a tile of a given id. Look at **figure4.py** (overleaf) to see how these functions are written.

figure3.py

```
001. def drawMap():
002.     psx = OFFSETX
003.     psy = OFFSETY-32
004.     mx = psx - player["sx"]
005.     my = psy - player["sy"]+32
006.
007.     for x in range(player["x"]-12, player["x"]+16):
008.         for y in range(player["y"]-12, player["y"]+16):
009.             if onMap(x,y):
010.                 b = mapData["data"][y][x]
011.                 td = findData(mapData["tiles"], "id", b)
012.                 block = td["image"]
013.                 bheight =  td["imageheight"]-34
014.                 bx = (x*32)-(y*32) + mx
015.                 by = (y*16)+(x*16) + my
016.                 if -32 <= bx < 800 and 100 <= by < 620:
017.                     screen.blit(block, (bx, by - bheight))
018.                 if x == player["x"] and y == player["y"]:
019.                     screen.blit("ball"+str(player["frame"]),
    (psx, psy))
```

▲ The updated drawMap() function

map3d.py

Language: **Python**

```
001. # 3dmap module for AmazeBalls
002. import json
003. import os
004.
005. def loadmap(mp):
006.     with open(mp) as json_data:
007.         d = json.load(json_data)
008.     mapdata = {"width":d["width"], "height":d["height"]}
009.     rawdata = d["layers"][0]["data"]
010.     mapdata["data"] = []
011.     for x in range(0, mapdata["width"]):
012.         st = x*mapdata["width"]
013.         mapdata["data"].append(rawdata[st:st+mapdata["height"]])
014.
015.     tileset = "maps/" + d["tilesets"][0]["source"].replace(
    ".tsx",".json")
016.     with open(tileset) as json_data:
017.         t = json.load(json_data)
018.
019.     mapdata["tiles"] = t["tiles"]
020.     for tile in range(0,len(mapdata["tiles"])):
021.         path = mapdata["tiles"][tile]["image"]
022.         mapdata["tiles"][tile]["image"] = os.path.basename(path)
023.         mapdata["tiles"][tile]["id"] = mapdata["tiles"][tile]["id"]+1
024.     return mapdata
```

figure4.py

```
001.  def onMap(x,y):
002.      if 0 <= x < mapData["width"] and 0 <= y < mapData["height"]:
003.          return True
004.      return False
005.
006.  def findData(lst, key, value):
007.      for i, dic in enumerate(lst):
008.          if dic[key] == value:
009.              return dic
010.      return -1
```

▲ The functions to test if a coordinate is inside the map area – `onMap()` – and `findData()`, which finds tile data for map drawing

figure5.py

```
001.  def doMove(p, x, y):
002.      if onMap(p["x"]+x, p["y"]+y):
003.          mt =
      mapData["data"][p["y"]+y][p["x"]+x]
004.          if mt == 1:
005.              p.update({"queueX":x,
      "queueY":y, "moveDone":False})
006.              if mt == 3:
007.                  mazeSolved = True
```

▲ The updated `doMove()` function

13 Masking the edges

If we draw our map now, we have lost that nice diamond shape map. And if we move the player down the map, we get a jagged edge at the top and blocks popping in and out of view as our `drawMap()` function decides which ones to draw in the range. We can tidy this up by overlaying a frame that obscures the edges of the printed area. We do this by having an image which covers the whole window but has a transparent cut-out area where we want the map blocks to be shown. We blit this frame graphic after we have called `drawMap()` in our `draw()` function.

14 I can't move!

Now that we have our map drawing, if you are adding in code following on from part one, you may notice that we can't move the bouncing ball any more. That's because the data we have loaded is a slightly different format and has floor blocks as id 1 and walls as id 2, so at the moment our `doMove()` function is thinking we are surrounded by walls (which were id 1 in the last part). We need to change our `doMove()` function to accommodate the new data format. Have a look at **figure5.py** to see what we need to write.

15 Finding the exit

Now that we have tidied up our display and got our ball moving again, we'll need to change a few of the default values that we start with. In the last part, we had the player starting at x = 0 and y = 3. We will need to change those values to 3 and 3 in the player data at the top of the code to be suitable for this map. We'll also want to change the `OFFSETY` constant to 300 to move the map a little further down the screen. We should now be able to guide the bouncing ball around the maze towards the bottom of the screen, where we should find the finish tile.

16 Over the finish line

If we try to move onto the finish block, we won't be able to, as our `doMove()` function is only detecting blocks we can move to as id 1. So we need to add another condition. Instead of only testing `mt` for 1, let's make that line if `mt == 1` or `mt == 3`: (because the id of the finish tile is 3). We can then add a variable to set if the player moves onto the finish, by adding inside this condition: `if mt == 3: mazeSolved = True`. We'll also need to declare `mazeSolved` as global inside `doMove()` and set its initial value to `False` at the top of our program.

17 Time is running out

Let's add a timer to the game. When the player reaches the finish (`mazeSolved is True`), we can stop the timer and display a message. So, first we make a timer variable at the top of our program with `timer = 0` and then, right at the end of the code, just before `pgzrun.go()`, we can use the Pygame Zero clock function called `schedule_interval()`. If we write `clock.schedule_interval(timerTick, 1.0)`, then the function `timerTick()` will be called once every second.

18 The clock struck one

So, all we need to do now is define the `timerTick()` function. We'll need to check if `mazeSolved` is `False` and add 1 to our `timer` variable if it is. Then we can add a `screen.draw.text` line to our `draw()` function to display the timer value and if `mazeSolved` is `True`, we can add some more text to say the maze has been solved and how many seconds it took. See the full program listing for how to write the code for those bits.

In the next instalment, we are going to add some baddies to contend with and, just for good measure, we can throw in some dynamite!

Top Tip

Drawing order

Remember that in the `draw()` function, things are drawn in the order you call them, so always draw the things you want on top last.

amazeballs2.py

DOWNLOAD THE FULL CODE:
magpi.cc/fPBrhM

Language: **Python**

```
001. import pgzrun
002. import map3d
003.
004. player = {"x":3, "y":3, "frame":0, "sx":0, "sy":96,
005.           "moveX":0, "moveY":0, "queueX":0, "queueY":0,
006.           "moveDone":True, "movingNow":False,
     "animCounter":0}
007. OFFSETX = 368
008. OFFSETY = 300
009. timer = 0
010. mazeSolved = False
011.
012. mapData = map3d.loadmap("maps/map1.json")
013.
014. def draw(): # Pygame Zero draw function
015.     screen.fill((0, 0, 0))
016.     drawMap()
017.     screen.blit('title', (0, 0))
018.     screen.draw.text("TIME: "+str(timer) , topleft=(
     20, 80), owidth=0.5, ocolor=(255,255,0),
     color=(255,0,0) , fontsize=60)
019.     if mazeSolved:
020.         screen.draw.text("MAZE SOLVED in " + str(timer)
     + " seconds!" , center=(400, 450), owidth=0.5,
     ocolor=(0,0,0), color=(0,255,0) , fontsize=60)
021.
022.
023. def update(): # Pygame Zero update function
024.     global player, timer
025.     if player["moveDone"] == True:
026.         if keyboard.left: doMove(player, -1, 0)
027.         if keyboard.right: doMove(player, 1, 0)
028.         if keyboard.up: doMove(player, 0, -1)
029.         if keyboard.down: doMove(player, 0, 1)
030.     updateBall(player)
031.
032. def timerTick():
033.     global timer
034.     if not mazeSolved:
035.         timer += 1
036.
037. def drawMap():
038.     psx = OFFSETX
039.     psy = OFFSETY-32
040.     mx = psx - player["sx"]
041.     my = psy - player["sy"]+32
042.
043.     for x in range(player["x"]-12, player["x"]+16):
044.         for y in range(player["y"]-12, player["y"]+16):
045.             if onMap(x,y):
046.                 b = mapData["data"][y][x]
047.                 td = findData(mapData["tiles"], "id", b)
048.                 block = td["image"]
049.                 bheight =  td["imageheight"]-34
050.                 bx = (x*32)-(y*32) + mx
051.                 by = (y*16)+(x*16) + my
052.                 if -32 <= bx < 800 and 100 <= by < 620:
053.                     screen.blit(block, (bx, by -
     bheight))
054.                 if x == player["x"] and y ==
     player["y"]:
055.                     screen.blit(
     "ball"+str(player["frame"]), (psx, psy))
056.
057. def findData(lst, key, value):
058.     for i, dic in enumerate(lst):
059.         if dic[key] == value:
060.             return dic
061.     return -1
062.
063. def onMap(x,y):
064.     if 0 <= x < mapData["width"] and 0 <= y <
     mapData["height"]:
065.         return True
066.     return False
067.
068. def doMove(p, x, y):
069.     global mazeSolved
070.     if onMap(p["x"]+x, p["y"]+y):
071.         mt = mapData["data"][p["y"]+y][p["x"]+x]
072.         if mt == 1 or mt == 3:
073.             p.update({"queueX":x, "queueY":y,
     "moveDone":False})
074.             if mt == 3:
075.                 mazeSolved = True
076.
077. def updateBall(p):
078.     if p["movingNow"]:
079.         if p["moveX"] == -1: moveP(p,-1,-0.5)
080.         if p["moveX"] == 1: moveP(p,1,0.5)
081.         if p["moveY"] == -1: moveP(p,1,-0.5)
082.         if p["moveY"] == 1: moveP(p,-1,0.5)
083.     p["animCounter"] += 1
084.     if p["animCounter"] == 4:
085.         p["animCounter"] = 0
086.         p["frame"] += 1
087.         if p["frame"] > 7:
088.             p["frame"] = 0
089.         if p["frame"] == 4:
090.             if p["moveDone"] == False:
091.                 if p["queueX"] != 0 or p["queueY"] !=0:
092.                     p.update({"moveX":p["queueX"],
     "moveY":p["queueY"], "queueX":0, "queueY":0,
     "movingNow": True})
093.                 else:
094                      p.update({"moveDone":True, "moveX":0,
     "moveY":0, "movingNow":False})
095.
096.         if p["frame"] == 7 and p["moveDone"] == False
     and p["movingNow"] == True:
097.             p["x"] += p["moveX"]
098.             p["y"] += p["moveY"]
099.             p["moveDone"] = True
100.
101. def moveP(p,x,y):
102.     p["sx"] += x
103.     p["sy"] += y
104.
105. clock.schedule_interval(timerTick, 1.0)
106. pgzrun.go()
```