Code Project: ball

PART 4 In the final tutorial of this series, **Mike Saunders** takes a lighter look at coding with a simple but addictive game...

Our expert

Mike Saunders recently found a copy of *ZEUS* for the ZX Spectrum and has been beavering away with Z80 assembly language. **http://mikeos.berlios.de**

Over the last three issues we've put our programming fingers to good use with an IRC bot, a sysadmin tool and a flash card program – all fairly serious stuff: so let's have some fun with a game in this final project. Now, most modern games take thousands of man-hours to create, not to mention an army of artists and musicians, but there's still some scope for solitary hacker to write something entertaining. After all, it didn't take a team of 500 coders and a Hollywood movie-set budget to create *Tetris* – Alexey Pajitnov managed pretty well on his own (until various filthy capitalist running-dogs of the West ran off with his idea, of course...) As with last month's project, we're going to use Python and PyGame as the foundations for our program. incidentally, there are three Pygame implementations of Tetris: see **www.pygame.org/tags/tetris** for more.

If this is the first issue of **LXF** you've picked up – and you've never written a line of Python code before – you'll be pleasantly surprised by how easy it is to understand; Python code is famous throughout the programming world for being very much self-explanatory. Or if you're familiar with other programming languages such as C or PHP, you'll applaud Python's simplicity. For instance, code blocks are marked by indentation rather than curly braces – see this program:

```
def saystuff(somestring):
    print "String passed: ", somestring

saystuff("Wowzers")
```

If you're new to Python, make sure that you have the language installed (most distros install it by default, but otherwise it will be available in your distro's package manager). Enter the above code into a text editor and save it as **test.py** in your home directory. Then fire up a terminal and enter:

```
python test.py
```

All being well, Python will interpret the code and spit out a line of text. Our sample program here merely sets up a subroutine called **saystuff**, which prints whatever text string is passed to it. You can see that the code of the subroutine is indented with a tab. Execution proper begins with the first call to **saystuff**, which passes the string **Wowzers** to be printed. It's as easy as that – you're almost ready to get coding.

But one last thing: you'll also need the Pygame modules for this tutorial. Pygame provides an extra layer on top of Python, linking with SDL and letting you display images and output sound effects in your programs. It's very widely available and will almost certainly be available in your distro's repositories, but otherwise see the **Development** section of our DVD. (If you followed last month's code project, you'll already have PyGame installed!)

Circular capers

Although game genres vary enormously, in terms of the underlying mechanics, most games involving sprites (image objects) moving around follow a similar code path:

1. Set up the screen, graphics, score counters *etc.*
2. Start loop to finish when player quits/dies.
3. Draw graphic objects on screen.
4. Get input from user (*eg* keyboard or mouse).
5. Check game logic (*eg* player hit enemy?)
6. Update graphic objects accordingly.
7. Go back to 3.

Now, we're going to write a little game that has several balls bouncing around the screen, and the player's objective is to avoid colliding the mouse pointer with them. Sounds easy? Well, if we add some randomness to the ball movements – *ie* they don't always move at the same speed – then it suddenly becomes a lot trickier. You can't just hold the mouse pointer in the bottom-left of the screen, say, because a ball could zoom down there at any moment. Throughout this, a counter keeps track of how many seconds you stay 'alive'. It's a very basic concept, but it demands good mouse dexterity and a laser-like focus on the screen.

But first things first: let's work out how to get a single ball bouncing around the screen. How does the ball know when to reverse direction? Fortunately, there's a very easy method to

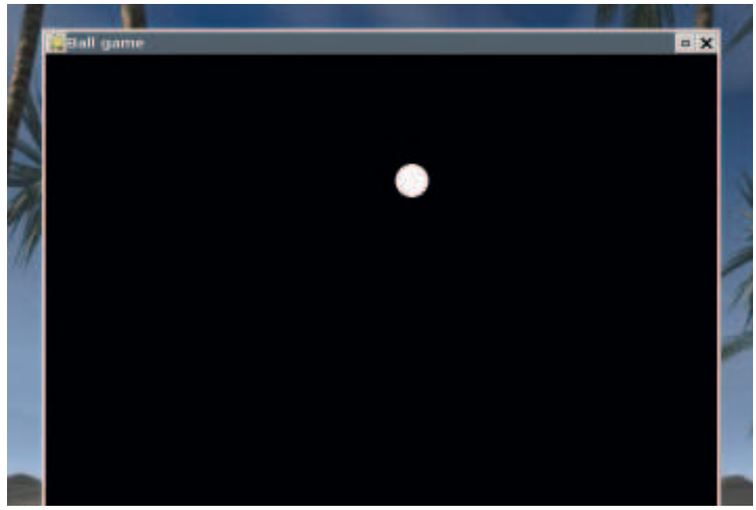
» **Last month** Part three: revise in style with your very own flashcard application.

game!

achieve this: we have two variables that we use to alter the ball movement. For each loop of the game engine, we add these numbers to the ball's position. If the ball is moving right, for instance, then it's because we're adding 1 to its horizontal position each loop. If the ball hits the right edge of the screen, we start adding -1 to its horizontal position, thereby moving it to the left.

Makes sense? If you're unsure how this works, here's a Python program to demonstrate it in action. You can find this code in the **Magazine/CodeProject** section of the DVD as **ball1.py**. To run this program, you'll also need an image called **ball.png** alongside the code, which is a 32x32 pixel image containing a white (filled) circle on a black background. It's on the DVD, or you can make it in seconds via *Gimp* – create a new 32x32 pixel image, fill it with black, use the circle select tool to create a new circle selection and fill that with white. Save it as **ball.png** in the same directory as **ball1.py** and run it by entering **python ball1.py**.

```
from pygame import *              # Use PyGame's functionality!

ballpic = image.load('ball.png')

done = False

ballx = 0 # Ball position variables
bally = 0
ballxmove = 1
ballymove = 1

init()          # Start PyGame
screen = display.set_mode((640, 480))      # Give us a nice window
display.set_caption('Ball game')           # And set its title

while done == False:
    screen.fill(0)    # Fill the screen with black (colour 0)
    screen.blit(ballpic, (ballx, bally))    # Draw ball
    display.update()

    time.delay(1)       # Slow it down!

    ballx = ballx + ballxmove    # Update ball position
    bally = bally + ballymove

    if ballx > 600:       # Ball reached screen edges?
        ballxmove = -1
    if ballx < 0:
        ballxmove = 1
    if bally > 440:
        ballymove = -1
    if bally < 0:
        ballymove = 1

    for e in event.get():    # Check for ESC pressed
        if e.type == KEYUP:
            if e.key == K_ESCAPE:
                done = True
```



› Our prototype ball-bouncing program – not much to look at yet, but seeing this on your screen will indicate that you've got the basics working.

Let's step through this. In the first line, we tell Python that we want to use routines from the PyGame library. Then we load the ball image we created before and store it in an object called **ballpic**, and create a true/false variable to determine when the game has finished.

The following four lines are hugely important: these declare variables which control the position and movement of the ball. **ballx** and **bally** store the location (in pixels) of the ball in our game window – **0,0** being the top-left, and **640,480** being the bottom-right pixel. **ballxmove** and **ballymove** store the numbers we add to the ball for each game loop; we set them to 1 initially, so that when the game starts, 1 is added to **ballx** and **bally** each loop, thereby moving the ball down and to the right. So, our ball starts in the top-left of the screen, and starts moving diagonally down-right when the program starts.

Next, we set up a new Pygame window and start the main game loop, filling (clearing) the screen with black and drawing our ball at its current position (code comments are denoted with **#** characters). The following code chunk determines how the ball is going to move:

```
    ballx = ballx + ballxmove
    bally = bally + ballymove

    if ballx > 600:
        ballxmove = -1
    if ballx < 0:
        ballxmove = 1
    if bally > 440:
        ballymove = -1
    if bally < 0:
        ballymove = 1
```

»

» In the first two lines, we update the ball's horizontal (**x**) and vertical (**y**) position by adding the two movement variables. If **ballxmove** and **ballymove** are 1, then the ball will move 1 pixel right and 1 pixel down each game loop. But then the following **if** statements check to see if the ball is near the edge of the screen, and if so, change **ballxmove** and **ballymove** accordingly. If, for instance, the ball position is over 600 pixels horizontally, then it should 'bounce' off and start moving left – so we start adding -1 to its position (effectively subtracting 1).

With just a few lines of code, we've managed to create the impression that the ball is bouncing around the screen – not bad! The final lines of this program set up a keyboard handler, so that you can quit the game by pressing the Esc key at any moment.

Quick tip

Having problems with your PyGame project? Keep track of variables by printing them out to the terminal with a simple **print** statement. For instance, if something weird is happening with the ball movement in our game, you can easily find out what's wrong by printing out the **xmove** or **ymove** variables – put a **print <variable name>** in the main game loop and you can watch it change in the terminal window when running the game.

Ball game 2.0

All good and well so far – we have our basic game structure in place. Now we want to add more balls, and also detect if the mouse pointer is colliding with any of them. For the former, we're going to set up an array of 'dictionary' entries to keep track of the balls. This gives us a huge amount of flexibility: we can have as many balls as we want, instead of limiting ourselves to **ball0**, **ball1**, **ball2** etc. Dictionaries are a doddle in Python:

```
mydict = {'Bach': 100, 'Handel': 75, 'Vivaldi': 90}

print mydict['Vivaldi']
```

Here, we associate three words with numbers, and then print out the value contained in **'Vivaldi'**, which is 90. We'll use a dictionary to store the X, Y, X movement and Y movement values of our balls – a bit like a struct in C. But whereas C bogs us down in memory management turmoil, in Python we can create loads of ball objects without any hassle at all, giving them their own individual dictionary entries.

The final thing we need to think about is collision detection. How do we tell when the mouse pointer has collided with a ball? Logically, it seems sanest to go through the position of every ball and compare them with the mouse pointer location. But we have a trick up our sleeve: the balls are white, and the background is black, so why not just detect when the mouse pointer is over a white pixel? That only takes one line of code, and is very fast...

Here's the code, which you can find as **ball2.py** in the **Magazine/CodeProject** section of the DVD, along with the **ball.png** picture we created earlier (it's exactly the same).

```
from pygame import *
import random

ballpic = image.load('ball.png')
ballpic.set_colorkey((0,0,0))

numballs = 10
delay = 5

done = False

balls = []

for count in range(numballs):
    balls.append(dict)
    balls[count] = {'x': 0, 'y': 0, 'xmove': random.randint(1, 2),
'ymove': random.randint(1, 2)}

init()
screen = display.set_mode((640, 480))
display.set_caption('Ball game')
event.set_grab(1)

while done == False:
    screen.fill(0)

    for count in range(numballs):
        screen.blit(ballpic, (balls[count]['x'], balls[count]['y']))

    display.update()

    time.delay(delay)

    for count in range(numballs):
        balls[count]['x'] = balls[count]['x'] + balls[count]['xmove']
        balls[count]['y'] = balls[count]['y'] + balls[count]['ymove']

    for count in range(numballs):
        if balls[count]['x'] > 620:
            balls[count]['xmove'] = random.randint(-2, 0)
        if balls[count]['x'] < -10:
            balls[count]['xmove'] = random.randint(0, 2)
        if balls[count]['y'] > 470:
            balls[count]['ymove'] = random.randint(-2, 0)
        if balls[count]['y'] < -10:
            balls[count]['ymove'] = random.randint(0, 2)

    for e in event.get():
        if e.type == KEYUP:
            if e.key == K_ESCAPE:
                done = True

    if screen.get_at((mouse.get_pos())) == (255, 255, 255, 255):
        done = True

print "You lasted for", time.get_ticks()/1000, "seconds!"
```

The general concepts behind this code are the same as before, but there's lots of new juicy code to explore. Near the top, where we load our ball image, we also set its **colorkey** to (**0,0,0**)' which is the RGB (Red/Green/Blue) value for black. What we're saying here is: set all black pixels in our ball picture to be transparent. This is important when we have many balls bouncing around, as we want them to overlap gracefully, and not have unsightly black

I want pretty pictures!

The final version of our game is hardly a *tour de force* on the graphical front, but we can spruce it up a bit by adding a background image. However, it's important to remember how we're detecting where the balls are – we're looking for white pixels. So your background image shouldn't contain any fully white (**255,255,255 RGB**) pixels, otherwise the game will end if you mouse over them!

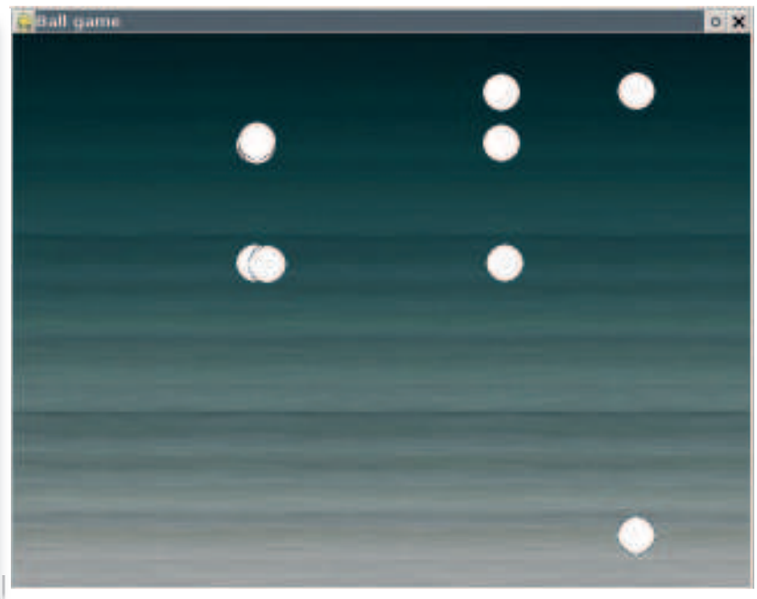
Find a picture and resize it to 640x480. If you suspect there are any white pixels in the image, you can always lower its brightness in *Gimp* which will eliminate such problems. Save this picture alongside **ball2.py** and call it **background.jpg**. Now, in **ball2.py**, enter the following code beneath the early **ballpic.set_colorkey** line:

```
backdrop = image.load('background.jpg')
```

So now we have our background picture in memory, ready to use. But we need to display it on the screen every loop, so further down in **ball2.py,** replace the **screen.fill(0)** line with this:

```
screen.blit(backdrop, (0,0))
```

This draws the background image before the balls. Note that if it's quite a complex image (*eg* lots of colours), then this extra blitting process will slow the game down slightly – but you can tweak the ball speeds and **delay** variable to compensate for that.



› **This is more like it! Multiple circle mayhem demands lightning reactions and pixel-perfect accuracy with the mouse pointer...**

corners drawn on other balls. So, only the white pixels of our balls will be displayed.

The following **numballs** and **delay** variables affect the difficulty of the game. As you'd expect, **numballs** controls the number of balls in play, whereas **delay** is the time (in milliseconds) that the game should pause each loop. You can leave these as-is for now – but if you fancy more of a challenge, you can up the number of balls and reduce the delay.

Our **balls = []** line sets up a new array of ball objects, and in typical Python fashion, we're not limited to the number of objects (nor do we have to define the number straight away). The

```
for count in range(numballs):
```

line sets up a loop which runs **numball** (10) times, adding new dictionary objects to the **balls** array and giving them starting values – top-left of screen, and random movement down-right. The **1, 2** in the random number generator means 'any number between 1 and 2 (inclusive)'. So we have 10 balls, all starting off with random speeds.

Next, we set up the screen as before, and add an **event.set_grab(1)** line which constrains the mouse pointer in the game window; it'd be too easy if you could move the mouse pointer outside! Then we have our main game loop. As before, we fill the screen with black, and then use another **for** loop to display (blit) all our balls to the screen.

After updating the screen and delaying (so that it runs at the same speed on all machines), we again traverse through our array of balls, updating their positions with our movement variables. Each ball has its own copy of **xmove** and **ymove** in its dictionary entry, so they can all move independently. Following this is the game logic, which determines if the balls have reached the window edges. This time, we've tweaked the values so that the balls can go slightly off-screen (remember, they're 32x32 pixels). This is vital for the gameplay: it means you can't just move the mouse cursor into a corner and never get hit! The balls now reach every part of the screen, so you have to keep moving the mouse.

The final three lines of code are new: **screen.get_at()** returns the pixel colour value at the specified position, which we set as the mouse pointer with **mouse.get_pos()**. We say: 'if the pixel colour at the mouse position is white (**255,255,255**), then **done = True** so the **while** main game loop will end.

Finally, we print the number of seconds for which the player survived – **time.get_ticks()** returns it in milliseconds, so we divide it by 1000 before displaying it.

Finishing up

Not bad for 55 lines of code, is it? As mentioned before, you can increase the difficulty of the game by raising the value of **numballs** at the top – the default of 10 is tricky enough, but if you think you're dextrous enough, try knocking it up to 15 or 20 for some finger-twistingly frantic gameplay. There are many other aspects of the game you can fiddle with too, such as altering the random numbers used in the main game logic (if ball has hit screen edge) section.

Pygame is chock-full of features to play around with, and, using a handful of lines of code, you can add sound effects or even a background music track to the game. **www.pygame.org/docs/** has some fantastically in-depth documentation to help users explore its functionality, along with a reference to all the routines used in this tutorial. Having programmed in countless languages and environments, from Amiga Blitz Basic to C#-SDL on Mono/.NET, I can safely say that Pygame is one of the most blissfully easy game programming setups around – it's the perfect way to concretise any game ideas that are floating around in your head. Good luck! LXF

Take it further!

That's the end of this coding project series, but we've clubbed together all our best ideas and printed a special magazine jam-packed with more programming projects. It's called *Code It!*, it costs £9.99 and it includes two dozen hands-on coding projects for programmers of all levels. It's on sale as you read this, but be quick – it's going to sell out quickly!

» **Next month** Try your hand at building an *rsync* server to save bandwidth.