



# GAME DEV COOKBOOK

Recreate mechanics and ideas from dozens of  
classic video games using Python and Pygame Zero

FROM THE MAKERS OF **Wireframe** MAGAZINE

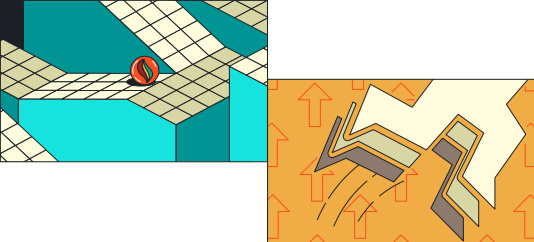
# Wireframe

Join us as we lift the lid  
on video games



Visit [wfmag.cc](http://wfmag.cc) to learn more





## Editorial

### Editor

Ryan Lambie

Email [ryan.lambie@raspberrypi.com](mailto:ryan.lambie@raspberrypi.com)

### Features Editor

Aaron Potter

Email [aaron.potter@raspberrypi.com](mailto:aaron.potter@raspberrypi.com)

### Sub-Editors

David Higgs, Vel Ilic

## Design

### Head of Design

Jack Willis

### Designers

Sara Parodi, Natalie Turner

## Contributors

Jon Bailes, Alexander Chatziioannou, Antony de Fault, Damiano Gerli, Craig Grannell, Shaun Hughes, Kim Justice, Phil King, Lewis Packwood, Nic Reuben, Paul Rose, Mark Vanstone, Howard Scott Warshaw, Jack Yarwood

## Publishing

### Publishing Director

Russell Barnes

Email [russell@raspberrypi.com](mailto:russell@raspberrypi.com)

### Director of Communications

Liz Upton

### CEO

Eben Upton

## Advertising

### Commercial Manager

Charlie Milligan

Email [charlotte.milligan@raspberrypi.com](mailto:charlotte.milligan@raspberrypi.com)

Tel +44 (0)7725 368887

## Distribution

Seymour Distribution Ltd

2 East Poultry Ave, London EC1A 9PT

Tel +44 (0)207 429 4000

## Subscriptions

Unit 6, The Enterprise Centre, Kelvin Lane, Manor Royal, Crawley, West Sussex, RH10 9PE

### To subscribe

Call 01293 312192 or visit [wfmag.cc/subscribe](http://wfmag.cc/subscribe)

### Subscription queries

[wireframe@subscriptionhelpline.co.uk](mailto:wireframe@subscriptionhelpline.co.uk)



This magazine is printed on paper sourced from sustainable forests and the printer operates an environmental management system which has been assessed as conforming to ISO 14001.

Wireframe magazine is published by Raspberry Pi (Trading) Ltd, Maurice Wilkes Building, St. John's Innovation Park, Cowley Road, Cambridge, CB4 0DS. The publisher, editor, and contributors accept no responsibility in respect of any omissions or errors relating to goods, products or services referred to or advertised in the magazine. Except where otherwise noted, content in this magazine is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0). ISSN: 2631-6722 (print), 2631-6730 (online).



Learning to program can be an intimidating proposition at the best of times, but if there's one way of making the process feel a bit friendlier, it's learning to code by making games. It was something magazine publishers understood in the eighties, when it was quite common to see long listings that would have to be painstakingly typed in, one line at a time. The results were frequently mixed – a jerky Space Invaders clone here, an off-brand version of Pac-Man there – but the process of typing those listings in (and often debugging them) helped a generation of computer users understand the basics of programming.

Source Code, one of the regular features you'll find in Wireframe magazine, works on a similar principle – but because it's the 21st century and the internet exists, you don't have to type everything in if you don't want to. Head to our Github ([wfmag.cc/git](https://wfmag.cc/git)) and you'll find an archive of every project we've ever published, plus assets, all ready to download.

You won't find vast swathes of code here, either. Instead, the idea of Source Code is to highlight a particular mechanic in a classic game, such as the pioneering boss battle in Phoenix or the falling rocks of Boulder Dash, and show you how to quickly recreate it in Python and Pygame Zero. Python is an approachable, eminently readable programming language by itself, while Pygame Zero provides an intuitive wrap-around library that allows beginners to start making games within just a few minutes. With the listings and tutorials in this digital book, you'll soon be able to move sprites around the screen, create simple physics simulations for everything from spaceships to ricocheting pool balls, or create the beginnings of a match-three puzzler.

Once you're feeling a bit more confident, you could even combine the projects published here to create something entirely new: how about a 2D platformer where you have to jump over pits while shooting away at the waves of enemies spiralling down from above? Or a driving game where you have to knock balls into goals for extra points? Start digging into the wealth of mechanics and snippets in this book, and you may just come up with the next indie hit. And, if you're really bitten by the Source Code bug, the projects published here are only a sample of what you'll find in the Wireframe archives. And, if you subscribe to the magazine ([wfmag.cc/subscribe](http://wfmag.cc/subscribe)), you'll find a new entry in the Source Code series every single month.

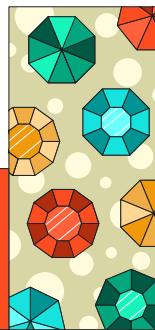
Enjoy!

**Ryan Lambie**

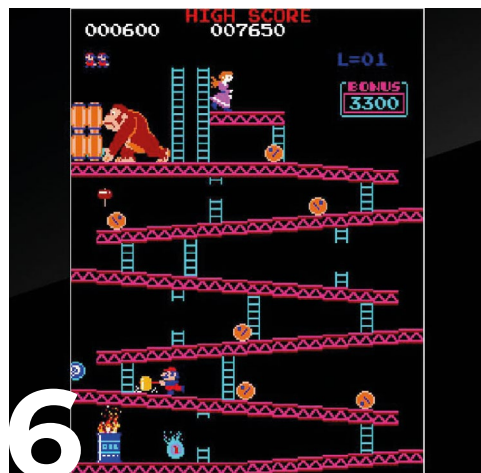
Editor, Wireframe magazine



CONTINUE?  
▶ YES  
NO



# Contents



6

## Platformer

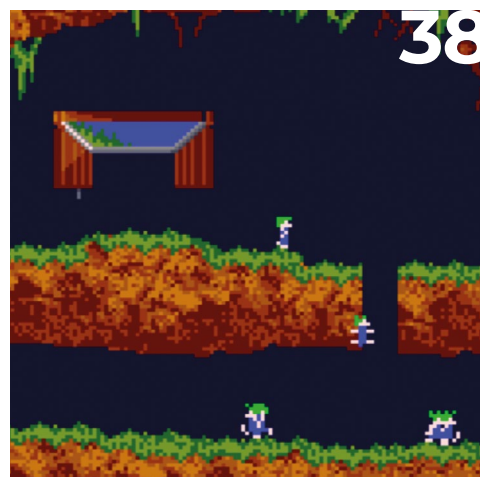
- 06. Donkey Kong**  
Replicate the arcade hit's barrel rolling physics
- 08. Super Mario Bros.**  
Re-create the Italian plumber's running and jumping action
- 12. Pitfall**  
Make swinging ropes to propel players over deadly pits
- 14. Manic Miner**  
Remake the ZX Spectrum classic's crumbling platforms

## Shooter

- 16. Space Invaders**  
Add disintegrating shields to your shoot-'em-up
- 18. Galaxian**  
Make your own dive-bombing alien attack patterns
- 26. R-Type**  
Use modular sprites to create an iconic whipping tail
- 30. Asteroids**  
The 1979 game's spaceship thruster motion detailed



26



38

## Puzzler

- 34. Pipemania**  
Create a network of pipes before the water starts to flow
- 36. Columns**  
Make a match-three puzzler in just a few lines of code
- 38. Lemmings**  
Create an army of obedient, path-following critters
- 40. Boulder Dash**  
Code a mining game and accompanying level editor



## Sport

- 50. Super Sprint**  
Make a top-down racer with AI-controlled rival cars
- 52. Side Pocket**  
Pot the balls in an homage to an arcade pool game
- 60. Hyper Sports**  
Gun down clay pigeons in a shooting minigame
- 62. Rally-X**  
Add a handy mini-map to your top-down racer

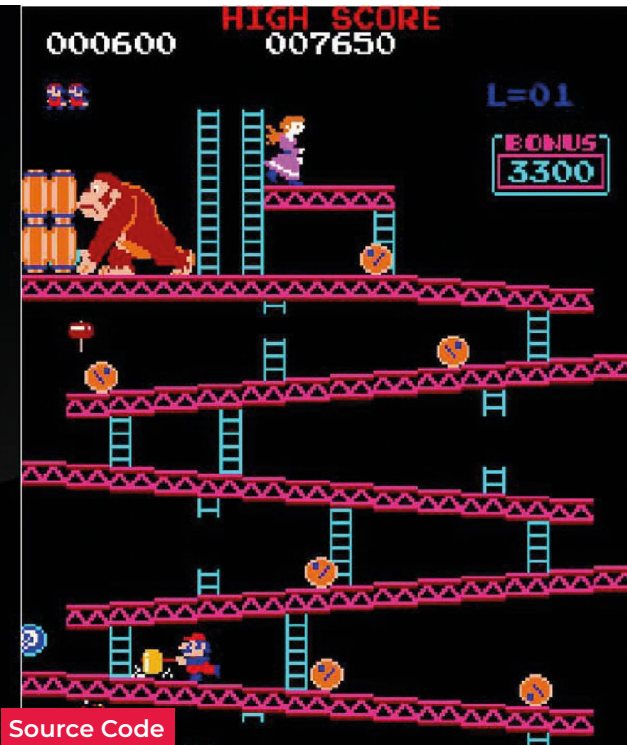


## Action/adventure

- 64. Lunar Lander**  
A Python homage to Atari's engrossing arcade staple
- 68. Frogger**  
Code a simple remake of Konami's road-crossing gem
- 76. Bomberman**  
Re-create the series' unforgettable explosions
- 84. Gauntlet**  
How to get four players dungeon crawling at the same time

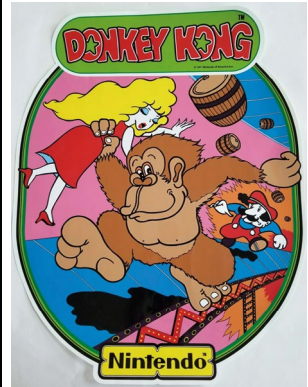
## Final touches

- 90. High score**  
An arcade-style table you can add to your own games
- 92. Continue?**  
Create game states and rules for moving between them



◀ Released in 1981, *Donkey Kong* was one of the most important games in Nintendo's history.

▼ It's fair to say Mario's changed quite a bit since this outing.



# Code your own Donkey Kong barrels



**AUTHOR**  
MARK VANSTONE

Replicate the physics of barrel rolling – straight out of the classic Donkey Kong

**D**onkey Kong first appeared in arcades in 1981, and starred not only the titular angry ape, but also a bouncing, climbing character called Jumpman – who later went on to star in Nintendo's little-known series of *Super Mario* games. *Donkey Kong* featured four screens per level, and the goal in each was to avoid obstacles and guide Mario (sorry, Jumpman) to the top of the screen to rescue the hapless Pauline. Partly because the game was so ferociously difficult from the beginning, *Donkey Kong*'s

first screen is arguably the most recognisable today: Kong lobes an endless stream of barrels, which roll down a network of crooked girders and threaten to knock Jumpman flat.

*Donkey Kong* may have been a relentlessly tough game, but we can recreate

one of its most recognisable elements with relative ease. We can get a bit of code running with Pygame Zero – and a couple of functions borrowed from Pygame – to make barrels react to the platforms they're on, roll down in the direction of a slope, and fall off the end onto the next platform. It's a very

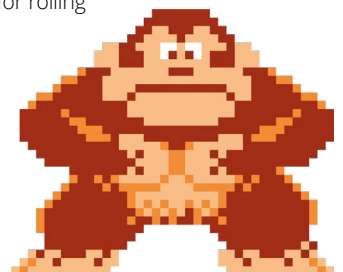
**“It's a very simple physics simulation using an invisible bitmap”**

simple physics simulation using an invisible bitmap to test where the platforms are and which way they're sloping. We also have some ladders which the barrels randomly either roll past or sometimes use to descend to the next platform below.

Once we've created a barrel as an Actor, the code does three tests for its platform position on each update: one to the bottom-left of the barrel, one bottom-centre, and

one bottom-right. It samples three pixels and calculates how much red is in those pixels. That tells us how much platform is under the barrel in each position. If the platform is tilted right, the number will be higher on the left, and the barrel must move to the right. If tilted left, the number will be higher on the right, and the barrel must move left. If there is no red under the centre point, the barrel is in the air and must fall downward.

There are just three frames of animation for the barrel rolling (you could add more for a smoother look): for rolling right, we increase the frame number stored with the barrel Actor; for rolling to the left, we decrease the frame number; and if the barrel's going down







Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag24](https://wfmag.cc/wfmag24)

# Rolling barrels in Python

Here's Mark's code snippet, which recreates *Donkey Kong's* rolling barrels in Python. To get it running on your system, you'll first need to install Pygame Zero – you can find full instructions at [wfmag.cc/pgzero](https://wfmag.cc/pgzero)

```
# Donkey Kong Barrels
from random import randint
from pygame import image, Color
import math

barrels = []
platformMap = image.load('images/map.png')
spacer = 0

def draw():
    screen.blit("background", (0, 0))
    for b in range(len(barrels)):
        if onScreen(barrels[b].x, barrels[b].y):
            barrels[b].draw()

def update():
    global spacer
    if randint(0,100) == 1 and spacer < 0:
        makeBarrel()
        spacer = 100
    spacer -= 1
    for b in range(len(barrels)):
        x = int(barrels[b].x)
        y = int(barrels[b].y)
        if onScreen(x,y):
            testcol1 = testPlatform(x-16,y+16,0)
            testcol2 = testPlatform(x,y+16,0)
            testcol3 = testPlatform(x+16,y+16,0)
            move = 0
            if testcol1 > testcol3: move = 1
```

```
            if testcol3 > testcol1: move = -1
            barrels[b].x += move
            barrels[b].frame += move * 0.1
            if move != 0: barrels[b].frame += move * 0.1
            else: barrels[b].frame += 0.1
            if barrels[b].frame >= 4: barrels[b].frame = 1
            if barrels[b].frame < 1: barrels[b].frame = 3.9
            testladder = platformMap.get_at((x,y+32))
            if testladder[2] == 255:
                if randint(0,150) == 1:
                    barrels[b].y += 20
            if testcol2 == 0: barrels[b].y += 1
            frame = str(math.floor(barrels[b].frame))
            if testPlatform(x,y+16,2) > 0:
                barrels[b].image = "bfrfront" + frame
            else:
                barrels[b].image = "bfrside" + frame

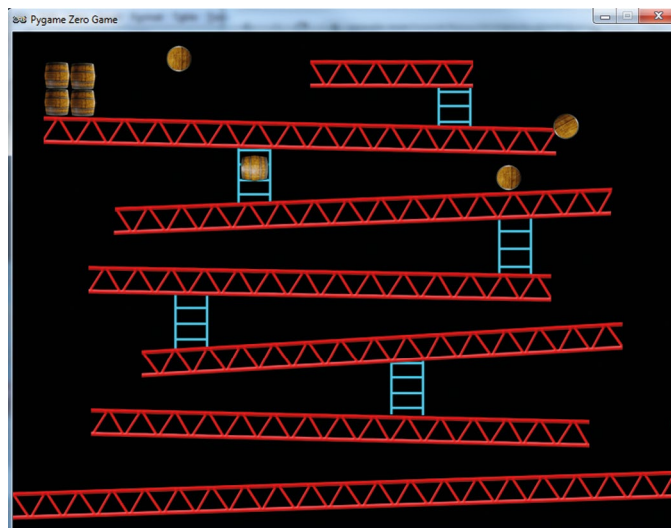
def onScreen(x,y):
    return x in range(16,784) and y in range(16,584)

def makeBarrel():
    barrels.append(Actor('bfrfront1', center=(200, 30)))
    barrels[len(barrels)-1].frame = 1

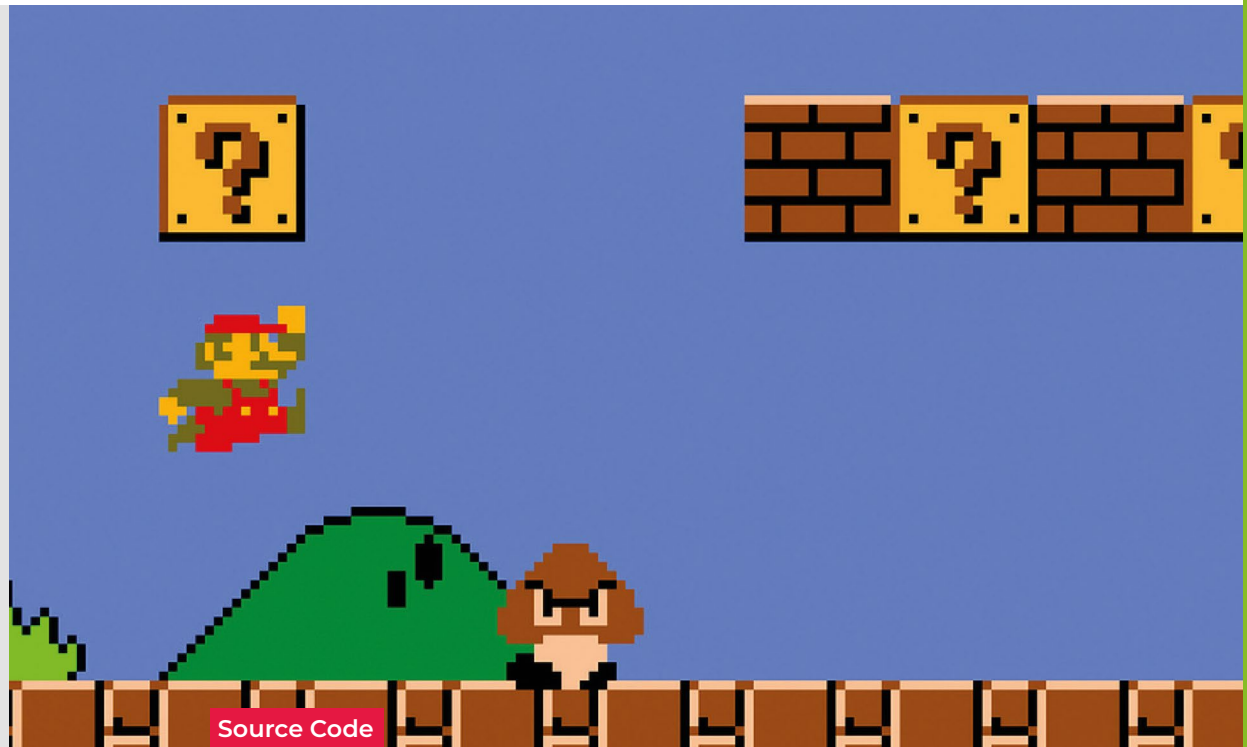
def testPlatform(x,y,col):
    c = 0
    for z in range(3):
        rgb = platformMap.get_at((x,y+z))
        c += rgb[col]
    return c
```

a ladder, we use the front-facing images for the animation. The movement down a ladder is triggered by another test for the blue component of a pixel below the barrel. The code then chooses randomly whether to send the barrel down the ladder.

The whole routine will keep producing more barrels and moving them down the platforms until they reach the bottom. Again, this is a very simple physics system, but it demonstrates how those rolling barrels can be recreated in just a few lines of code. All we need now is a jumping player character (which could use the same invisible map to navigate up the screen) and a big ape to sit at the top throwing barrels, then you'll have the makings of your own fully featured *Donkey Kong* tribute. 🐼



◀ Our *Donkey Kong* tribute up and running in Pygame Zero. The barrels roll down the platforms and sometimes the ladders.



Source Code

# Super Mario-style jumping physics

Learn how to create your own Super Mario-style running and jumping action in Python



AUTHOR  
RIK CROSS

**B**efore writing any code, it's best to decide on the rules of your world. Will you allow your player to double-jump, or change direction in mid-air? There are no right or wrong answers to these questions, but it pays to plan ahead. We'll be using Python and Pygame Zero to code the game world, but the ideas are transferable to other languages.

The first thing we need to do is create a player and some platforms to jump on! As Pygame Zero has built-in support for collision detection between game Actors and rectangles, we've stored the platforms as a list of rectangles with varying dimensions.

As vertical and horizontal motion are perpendicular to each other, they can be considered independently. Horizontal motion will involve moving the player to the left or right by updating the player's x-coordinate if the arrow keys are pressed (and the player is within the screen bounds).

This can be improved later, but we'll keep this simple initially to concentrate on the vertical motion. Before writing the code for vertical motion, let's look at the physics:

- Acceleration is the rate of change of velocity. Vertical acceleration is due to gravity, and is a positive value (i.e. acting downwards). Gravity will be stored as a global constant, as it acts on all game objects.
- Velocity is the rate of change of position. Initially the player's vertical velocity will be 0, as the player is at rest. When the player jumps, the velocity will be set to a negative value (i.e. acting upwards).

Both of these values change with respect to time, which for the sake of simplicity can be thought of as increasing with each frame. There's nothing special about the values chosen for gravity and jump velocity – these can be tinkered with to suit. You can also adjust the height and gaps between the platforms to increase your game's challenge.

In each frame, the following algorithm is used to update the player's vertical position:

- Add the acceleration value to the velocity value
- Add the velocity value to the position value

## COLLISION DETECTION

The next thing to fix is that the player's velocity (and therefore position) isn't yet affected by colliding with a platform. One way to do this is to calculate the player's new position, but only move the player to the new position if they don't hit a platform. If there's a collision, then the player isn't moved, and its velocity is set to 0. No collision means the player is free to move to the new position. Making the player jump is a matter of setting the player's vertical velocity to the predefined jump velocity. However, the player should only be allowed to jump if there's a collision. This means that the player is touching a platform. ☺





Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag7](https://wfmag.cc/wfmag7)

# Jumping physics in Python

Here's a code snippet that illustrates Rik's platform-jumping physics in Python. To get it running on your system, you'll first need to install Pygame Zero – you can find full instructions at [wfmag.cc/XVileD](https://wfmag.cc/XVileD)

```
# define screen size
WIDTH = 800
HEIGHT = 800
# define a colour
MAROON = 128,0,0
# vertical acceleration
GRAVITY = 0.2

# a list of platforms, each a rectangle in the form ((x,y)
(w,h))
platforms = [
    Rect((0,780),(800,20)),
    Rect((200,700),(100,100)),
    Rect((400,650),(100,20)),
    Rect((600,600),(100,20))
]

# create a player and define initial vertical velocity
player = Actor('player',(50,450), anchor=('left','top'))
player.w = 20
player.h = 20
# define initial and jump velocities
player.y_velocity = 0
player.jump_velocity = -7

def update():
    # horizontal movement
    # calculate new horizontal position if arrow keys are
    pressed
    if keyboard.left and player.x > 0:
        player.x -= 2
    if keyboard.right and player.x < 780:
        player.x += 2

    # vertical movement
    # temporary variable to store new y position
    newy = player.y

    # acceleration is rate of change of velocity
    player.y_velocity += GRAVITY
    # velocity is rate of change of position
    newy += player.y_velocity

    # create a rectangle for the new y position
    newplayerpositiony =
    Rect((player.x,newy),(player.w,player.h))

    # check whether the new player position collides with a
    platform
    y_collision = False
    for p in platforms:
```

```
        y_collision = newplayerpositiony.colliderect(p) or
y_collision

    # player no longer has vertical velocity if colliding with
    platform
    if y_collision:
        player.y_velocity = 0
    # only allow the player to move if it doesn't collide with
    any platforms
    else:
        player.y = newy

    # pressing space sets negative vertical velocity only if
    player is on ground
    if keyboard.space and y_collision:
        player.y_velocity = player.jump_velocity

def draw():
    screen.clear()

    # draw platforms
    for p in platforms:
        screen.draw.filled_rect(p,MAROON)

    # draw player
    player.draw()
```

## MAKING IMPROVEMENTS

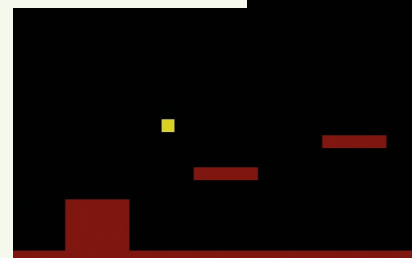
We've fixed one potential bug, by only allowing the player to jump if they're touching a platform. If you run the code on the left, though, you'll notice a few other bugs:

- The player can jump if they're touching any platform, even if they're underneath. This can be fixed by only counting collisions below the player, by comparing the player and platform's y-coordinates.
- There's no horizontal collision detection, so the player can walk through platforms. This can be fixed with horizontal collision detection.
- If the player collides with a platform at high velocity, they'll stop just short of the platform and then drop slowly to the ground. One way of fixing this is to calculate the

distance between the player and this platform, and move the player so they're on top of the platform they would have collided with.

These bugs have been fixed in a second version of the code, [jump\\_physics\\_improved.py](#), also available in the GitHub repository link above.

♥ Like *Super Meat Boy*, but without the meat.



➤ Players must change the colour of every cube to complete the level.

✓ The cabinet employed a diagonal joystick to move Q\*bert around.

Source Code



▲ It was probably just as well, considering how popular the game would become, that Jeff Lee went with Q\*bert instead of his initial idea: *Snots And Boogers*.

## Recreate Q\*bert's cube-hopping action

AUTHOR  
MARK VANSTONE

Code the mechanics of an eighties arcade hit

**L**ate in 1982, a funny little orange character with a big nose landed in arcades. The titular Q\*bert's task was to jump around a network of cubes arranged in a pyramid formation, changing the colours of each as they went. Once the cubes were all the same colour, it was on to the next level; to make things more interesting, there were enemies like Coily the snake, and objects which helped Q\*bert: some froze enemies in their tracks, while floating discs provided a lift back to the top of the stage.

Q\*bert was designed by Warren Davis and Jeff Lee at the American company Gottlieb, and soon became such a smash hit that, the following year, it was already being ported to most of the home computer platforms available at the time. New versions and remakes continued to appear for years afterwards, with a mobile phone version appearing in 2003. Q\*bert was by far Gottlieb's most popular game, and after several changes in company ownership, the firm is now part of Sony's catalogue –

Q\*bert's main character even made its way into the 2015 film, *Pixels*.

Q\*bert uses isometric-style graphics to draw a pseudo-3D display – something we can easily replicate in Pygame Zero by using a single cube graphic with which we make a pyramid of Actor objects. Starting with seven cubes on the bottom row, we can create a simple double loop to create the pile of cubes. Our Q\*bert character will be another Actor object which we'll position at the top of the pile to start. The game screen can then be displayed in the `draw()` function by looping through our 28 cube Actors and then drawing Q\*bert.

We need to detect player input, and for this we use the built-in keyboard object and check the cursor keys in our `update()` function. We need to make Q\*bert move from cube to cube so we can move the Actor 32 pixels on the x-axis and 48 pixels on the y-axis. If we do this in steps of 2 for x and 3 for y, we will have Q\*bert on the next cube in 16 steps. We can also change his image to point in the right direction depending

on the key pressed in our `jump()` function. If we use this linear movement in our `move()` function, we'll see the Actor go in a straight line to the next block. To add a bit of bounce to Q\*bert's movement, we add or subtract (depending on the direction) the values in the `bounce[]` list. This will make a bit more of a curved movement to the animation.

Now that we have our long-nosed friend jumping around, we need to check where he's landing. We can loop through the cube positions and check whether Q\*bert is over each one. If he is, then we change the image of the cube to one with a yellow top. If we don't detect a cube under Q\*bert, then the critter's jumped off the pyramid, and the game's over. We can then do a quick loop through all the cube Actors, and if they've all been changed, then the player has completed the level. So those are the basic mechanics of jumping around on a pyramid of cubes. We just need some snakes and other baddies to annoy Q\*bert – but we'll leave those for you to add. Good luck! 🍀



# Bouncing between cubes in Python

Here's Mark's code for a Q\*bert-style, cube-hopping platform game. To get it running on your system, you'll need to install Pygame Zero – full instructions are available at [wfmag.cc/pgzero](http://wfmag.cc/pgzero).

```
# Q*bert

WIDTH = HEIGHT = 500

gameState = 0
blocks = []
qbert = Actor('qbert2', center=(250, 80))
qbert.movex = qbert.movey = qbert.frame = count = 0;
bounce = [-6,-4,-2,-1,0,0,0,0,0,0,0,1,2,4,6]

for r in range(0, 7):
    for b in range(0, 7-r):
        blocks.append(Actor('block0', center=(60+(b*64)+(r*32),
400-(r*48))))

def draw():
    screen.blit("background", (0, 0))
    for b in range(0, 28): blocks[b].draw()
    if gameState == 0 or (gameState == 1 and count%4 == 0):
qbert.draw()
    if gameState == 2 : screen.draw.text("YOU CLEARED THE
LEVEL!", center = (250, 250), owidth=0.5, ocolor=(255,255,255),
color=(255,0,255) , fontsize=40)

def update():
    global gameState, count
    if gameState == 0:
        if qbert.movex == 0 and qbert.movey == 0 :
            if keyboard.left: jump(32,48,3)
            if keyboard.right: jump(-32,-48,1)
            if keyboard.up: jump(-32,48,0)
            if keyboard.down: jump(32,-48,2)
        if qbert.movex != 0 : move()
        count += 1;

def move():
    if qbert.movex > 0 :
        qbert.x -=2
        qbert.movex -=2
    if qbert.movex < 0 :
        qbert.x +=2
        qbert.movex +=2
    if qbert.movey > 0 :
        qbert.y -=3 - bounce[qbert.frame]
        qbert.movey -=3
    if qbert.movey < 0 :
        qbert.y +=3 + bounce[qbert.frame]
        qbert.movey +=3
    qbert.frame +=1
    if qbert.movex == 0 :
        checkBlock()
```

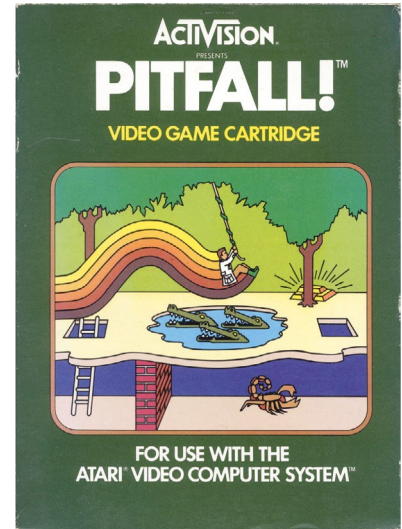
```
def checkBlock():
    global gameState
    block = -1
    curBlock = 0
    numSelected = 0
    for r in range(0, 7):
        for b in range(0, 7-r):
            x = 60+(b*64)+(r*32) -2
            y = 400-(r*48) -32
            if qbert.x == x and qbert.y == y :
                block = curBlock
                blocks[block].image = "block1"
                curBlock +=1
    if block == -1 : gameState = 1
    for b in range(0, 28):
        if blocks[b].image == "block1" : numSelected += 1
    if numSelected == 28 : gameState = 2

def jump(x,y,d):
    qbert.movex = x
    qbert.movey = y
    qbert.image = "qbert"+str(d)
    qbert.frame = 0
```

Pygame Zero Game



Our homage to Gottlieb's classic Q\*bert game. Try not to fall into the terrifying void.



Source Code

▲ Designed by David Crane, *Pitfall!* was released for the Atari 2600 and published by Activision in 1982.

◀ Our homage to the classic *Pitfall!* Atari game. Can you add some rolling logs and other hazards?

# Swing into action with an **homage to Pitfall!**



AUTHOR  
MARK VANSTONE

Grab onto ropes and swing across chasms in our Python rendition of an Atari 2600 classic

**W**hether it was because of the design brilliance of the game itself or because *Raiders of the Lost Ark* had just hit the box office, *Pitfall* Harry became a popular character on the Atari 2600 in 1982. His hazardous attempts to collect treasure struck a chord with eighties gamers, and saw *Pitfall!*, released by Activision, sell over four million copies. A sequel, *Pitfall II: The Lost Caverns* quickly followed the next year, and the game was ported to several other systems, even making its way to smartphones and tablets in the 21st century.

The game itself is a quest to find 32 items of treasure within a 20-minute time limit. There are a variety of hazards for Pitfall Harry to navigate around and over, including rolling logs, animals, and holes in the ground. Some of these holes can be

jumped over, but some are too wide and have a convenient rope swinging from a tree to aid our explorer in getting to the other side of the screen. Harry must jump towards the rope as it moves towards him and then hang on as it swings him over the pit, releasing his grip at the other end to land safely back on firm ground.

For this code sample, we'll concentrate on the rope swinging (and catching) mechanic. Using Pygame Zero, we can get our basic display set up quickly. In this case, we can split the background into three layers: the background, including the back of the pathway and the tree trunks, the treetops, and the front of the pathway. With these layers we can have a rope swinging with its pivot point behind the leaves of the trees, and, if Harry gets a jump wrong, it will look like he falls down the hole in the ground. The order in which we draw these

to the screen is background, rope, tree-tops, Harry, and finally the front of the pathway.

Now, let's get our rope swinging. We can create an Actor and anchor it to the centre and top of its bounding box. If we rotate it by changing the angle property of the Actor, then it will rotate at the top of the Actor rather than the mid-point. We can make the rope swing between -45 degrees and 45 degrees by increments of 1, but if we do this, we get a rather robotic sort of movement. To fix this, we add an 'easing' value which we can calculate using a square root to make the rope slow down as it reaches the extremes of the swing.

Our Harry character will need to be able to run backwards and forwards, so we'll need a few frames of animation. There are several ways of coding this, but for now, we can take the x coordinate and



## Swing when you're winning

Here's Mark's code snippet, which gets a swinging rope and a jumping adventurer running in Python. To get it working on your system, you'll need to install Pygame Zero – full instructions are available at [wfmag.cc/pgzero](http://wfmag.cc/pgzero).

```
# Pitfall!
import math

rope = Actor('rope', midtop=(400, 110), anchor=('center', 'top'))
harry = Actor('harry', (80, 290))
harry.attached = False
harry.jump = 0
harry.onground = True
swing = -1

def draw():
    screen.blit("background", (0, 0))
    rope.draw()
    screen.blit("trees", (0, 0))
    harry.draw()
    screen.blit("platform", (0, 335))
    if harry.x > 550 and harry.y < 300: screen.draw.text("You
made it over!", center=(400, 560), owidth=0.5, ocolor=(0, 0, 255),
color=(255, 255, 255), fontsize=40)

def update():
    global swing
    if rope.angle < -45:
        rope.angle = -45
        swing = 1
    if rope.angle > 45:
        rope.angle = 45
        swing = -1
    easing = (7 - (math.sqrt(abs(rope.angle))))/3
    rope.angle += swing * easing
    oldx = harry.x
    harry.onground = False
    if (harry.y > 289 and harry.y < 293) or (harry.y > 468 and
harry.y < 471): harry.onground = True

    if harry.x > 260 and harry.x < 540 and harry.y > 290 and harry.y
< 470 : harry.onground = False
    if keyboard.right and (harry.onground == True or harry.jump > 0):
        harry.x += 2
        harry.image = "harry"+str(int((harry.x/20)%4))
    if keyboard.left and (harry.onground == True or harry.jump > 0):
        harry.x -= 2
        harry.image = "harry"+str(int((harry.x/20)%4))+ "r"
    if harry.jump > 0:
        harry.y -= 2
        harry.jump -= 1
        harry.image = "harry0"
    else:
        if harry.y < 290 and harry.jump == 0 and harry.attached ==
False:
            harry.y += 2
            elif harry.jump == 0 and harry.x > 255 and harry.x < 540 and
harry.y < 470:
                harry.y += 2

        if oldx == harry.x and harry.jump == 0 : harry.image = "harry"
        if harry.collidepoint (rope.left, rope.bottom) and rope.angle < 25:
            harry.attached = True
        if harry.attached == True:
            harry.image = "harryrope"
            harry.y = rope.bottom + 32
            harry.x = rope.x + (rope.angle * 2.7) - 12

def on_key_down(key):
    if key == keys.SPACE:
        if harry.y == 290 or harry.attached == True:
            harry.jump = 30
            harry.attached = False
            if harry.y > 450: harry.pos = (80, 290)
```

work out which frame to display as the x value changes. If we have four frames of running animation, then we would use the %4 operator and value on the x coordinate to give us animation frames of 0, 1, 2, and 3. We use these frames for running to the right, and if he's running to the left, we just mirror the images. We can check to see if Harry is on the ground or over the pit, and if he needs to be falling downward, we add to his y coordinate. If he's jumping (by pressing the **SPACE** bar), we reduce his y coordinate.

We now need to check if Harry has reached the rope, so after a collision, we

check to see if he's connected with it, and if he has, we mark him as attached and then move him with the end of the rope until the player presses the **SPACE** bar and he can jump off at the other side. If he's swung far enough, he should land safely and not fall down the pit. If he falls, then the player can have another go by pressing the **SPACE** bar to reset Harry back to the start.

That should get Pitfall Harry over one particular obstacle, but the original game had several other challenges to tackle – we'll leave you to add those for yourselves. 🐹

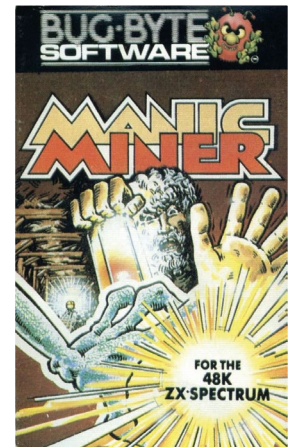


▲ In one of the earliest platformers, Pitfall Harry swings from the trees to avoid falling into deadly pits.





[Source Code](#)



▲ *Manic Miner's* cover art was wonderfully lo-fi.

◀ Our homage to the classic *Manic Miner*.



**AUTHOR**  
**MARK VANSTONE**

# Remake Manic Miner's collapsing platforms

Traverse a crumbly cavern in our homage to a Spectrum classic

One of the most iconic games on the Sinclair ZX Spectrum featured a little man called Miner Willy, who spent his days walking and jumping from platform to platform collecting the items needed to unlock the door on each screen. *Manic Miner's* underground world featured caverns, processing plants, killer telephones, and even a forest featuring little critters that looked suspiciously like Ewoks.

Written by programmer Matthew Smith and released by Bug-Byte in 1983, the game became one of the most successful titles on the Spectrum. Smith was only 16 when he wrote *Manic Miner* and even constructed his own hardware to speed up the development process, assembling the code on a TRS-80 and then downloading it to the Spectrum with his own hand-built interface. The success of *Manic Miner* was then closely followed by *Jet Set Willy*, featuring the same character, and although they were originally written for the Spectrum, the games very soon made it onto just about every home


computer of the time.

Both *Manic Miner* and *Jet Set Willy* featured unstable platforms which crumbled in Willy's wake, and it's these we're going to try to recreate this month.

In this Pygame Zero example, we need three frames of animation for each of the two directions of movement. As we press the arrow keys we can move the Actor left and right, and in this case, we'll decide which frame to display based on a `count` variable, which is incremented each time our `update()` function runs. We can create platforms from a two-dimensional data list representing positions on the screen with 0 meaning a blank space, 1 being a solid platform, and 2 a collapsible platform. To set these up, we run through the list and make Actor objects for each platform segment.

For our `draw()` function, we can blit a background graphic, then Miner Willy, and then our platform blocks. During our `update()` function, apart from checking key presses, we also need to do some gravity calculations. This will mean that if Willy isn't standing on a platform or jumping, he'll start

to fall towards the bottom of the screen. Instead of checking to see if Willy has collided with the whole platform, we only check to see if his feet are in contact with the top. This means he can jump up through the platforms but will then land on the top and stop. We set a variable to indicate that Willy's standing on the ground so that when the **SPACE** bar is pressed, we know if he can jump or not. While we're checking if Willy's on a platform, we also check to see if it's a collapsible one, and if so, we start a timer so that the platform moves downwards and eventually disappears. Once it's gone, Willy will fall through. The reason we have a delayed timer rather than just starting the platform heading straight down is so that Willy can run across many tiles before they collapse, but his way back will quickly disappear. The disappearing platforms are achieved by changing the image of the platform block as it moves downward.

As we've seen, there were several other elements to each *Manic Miner* screen, such as roaming bears that definitely weren't from *Star Wars*, and those dastardly killer telephones. We'll leave you to add those... 





Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag49](https://wfmag.cc/wfmag49)

# Crumbly platforms in Python

Here's Mark's code for a *Manic Miner* screen, complete with collapsing platforms. To get it working on your system, you'll need to install Pygame Zero – full instructions are available at [wfmag.cc/pgzero](https://wfmag.cc/pgzero).

```
# Manic Miner

HEIGHT = 400
willy = Actor('willyr0',(400,300))
willy.direction = "r"
willy.jump = 0
willy.onground = False
count = 0
platforms = [[1,1,0,0,0,0,1,1,0,0,2,2,2,1,1,1,1,0,0,0,0,0],
              [0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,1],
              [1,1,1,0,0,0,2,2,2,2,0,0,0,0,1,1,1,0,0,0,0,0],
              [0,0,1,1,0,0,0,0,0,0,0,0,1,1,2,2,0,0,1,1,1,0],
              [1,1,0,0,1,1,0,0,0,2,2,2,0,0,0,0,0,0,0,0,1,1],
              [0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0]]
platformActors = []
for r in range(len(platforms)):
    for c in range(len(platforms[r])):
        if(platforms[r][c] != 0): platformActors.
append(Actor('platform'+str(platforms[r][c])+"1", (70+(c*30), 1
20+(r*40))))
        platformActors[len(platformActors)-1].status = 0

def draw():
    screen.blit("background", (0, 0))
    willy.draw()
    drawPlatforms()

def update():
    global count
    willy.image = "willy"+ willy.direction + "0"
    if keyboard.left:
        moveWilly(-1,0)
        willy.direction = "l"
        willy.image = "willyl"+ str(int(count/8)%3)
        pass
    if keyboard.right:
        moveWilly(1,0)
        willy.direction = "r"
        willy.image = "willyr"+ str(int(count/8)%3)
        pass
    checkGravity()
    count += 1

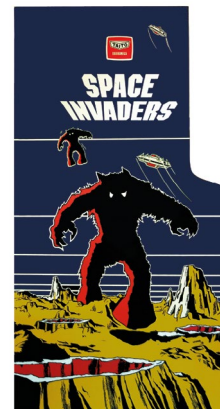
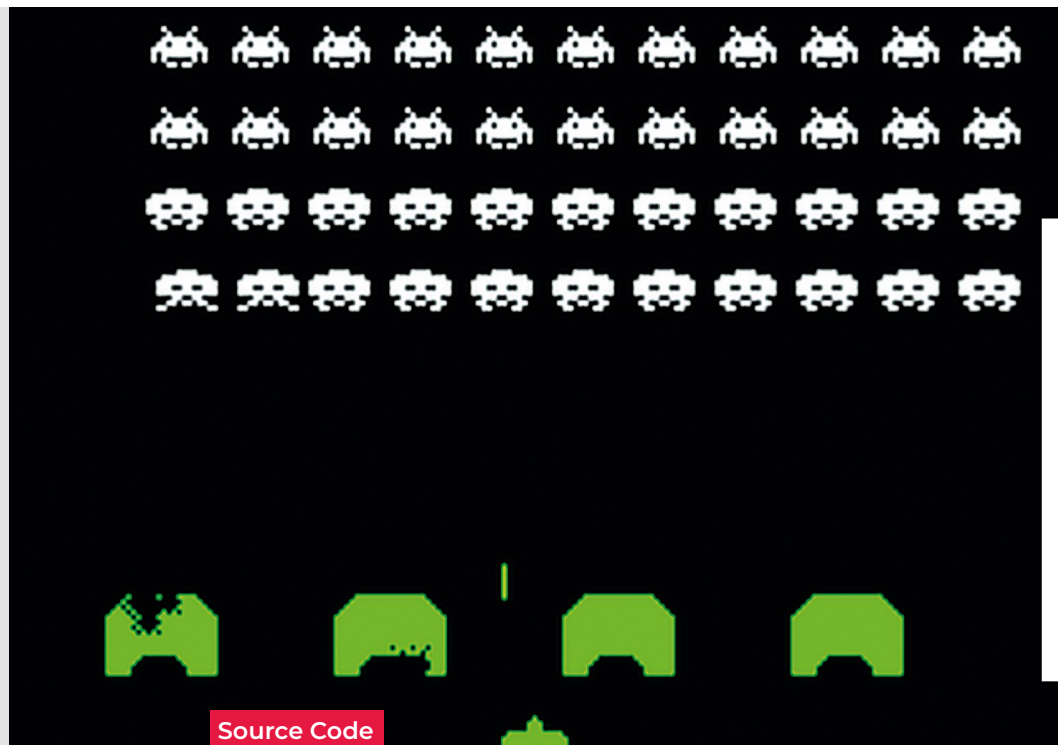
def on_key_down(key):
    if key.name == "SPACE":
        if willy.onground == True:
            willy.jump = 40

def drawPlatforms():
    for p in range(len(platformActors)):
        if platformActors[p].status != -1:
            platformActors[p].draw()
```

```
def moveWilly(x,y):
    if willy.x+x < 730 and willy.x+x > 70:
        willy.x += x

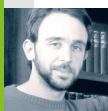
def checkGravity():
    if willy.jump > 0:
        willy.y -=2
        willy.jump -=1
    if willy.y < 320:
        willy.onground = False
        for p in range(len(platformActors)):
            frame = int(platformActors[p].image[-1])+1
            if platformActors[p].status > 0 :
                platformActors[p].status -= 1
                if platformActors[p].status == 0 :
                    platformActors[p].y += 1
                    if frame > 8 :
                        platformActors[p].status = -1
                    else:
                        platformActors[p].image =
"platform2"+str(frame)
                        platformActors[p].status = 30
                    if((willy.x > platformActors[p].x-20 and willy.x <
platformActors[p].x+20) and willy.y+20 == platformActors[p].y-
14+(frame-1) and platformActors[p].status != -1):
                        willy.onground = True
                        if platformActors[p].image[8] == "2":
                            if platformActors[p].status == 0 :
                                platformActors[p].status = 30
                                if willy.onground == False:
                                    willy.y += 1
                                else:
                                    willy.onground = True
```





Source Code

# Space Invaders' disintegrating shields



AUTHOR  
ANDREW GILLETT

They add strategy to a genre-defining shooter.  
Andrew lifts the lid on Space Invaders' shields

**R**eleased in 1978, *Space Invaders* introduced ideas so fundamental to video games that it's hard to imagine a time before them. And it did this using custom-made hardware which by today's standards is unimaginably slow.

*Space Invaders* ran on an Intel 8080 CPU operating at 2MHz. With such meagre processing power, merely moving sprites around the screen was a struggle. In modern 2D games, at the start of each frame the entire screen is reset, then all objects are displayed.

For *Space Invaders*' hardware, this process would have been too slow. Instead, each time a sprite needs to move, the game first erases the sprite from the screen, then redraws it in the new position. The game also updates only one alien per frame – which leads to the effect of the aliens moving faster when there are fewer of them. These

techniques cut down the number of pixels which need to be updated each frame, from nearly 60,000 to around a hundred.

One of *Space Invaders*' most notable features is its four shields. These provide shelter from enemy fire, but deteriorate after repeated hits. The player can take advantage of the shields' destructible nature

**“Space Invaders introduced ideas fundamental to video games”**

– by repeatedly firing at the same place on a shield's underside, a narrow gap can be created which can then be used to take out enemies. (Of course, the player can also be shot through the same gap.)

The system of updating only the minimum necessary number of pixels works well

as long as there's no need for objects to overlap. In the case of the shields, though, what happens when objects do overlap is fundamental to how they work. Whenever a shot hits something, it's replaced by an explosion sprite. A few frames later, the explosion sprite is deleted from the screen. If the explosion sprite overlapped with a shield, that part of the shield is also deleted.

The code to the right displays four shields, and then bombards them with a series of shots which explode on impact. I'm using sprites which have been scaled up by ten, to make it easier to see what's going on.

We first create two empty lists – one to hold details of any **shots** on screen, as well as explosions. These will be displayed on the screen every frame. Each entry in the **shots** list will be a dictionary data structure containing three values: a position, the sprite to be displayed, and whether the shot is in 'exploding' mode – in which case



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag9](https://wfmag.cc/wfmag9)

# Disintegrating shields in PYTHON

Here's a code snippet that shows Andrew's *Space Invaders*-style disintegrating shields working in Python. To get it running on your system, you'll first need to install Pygame Zero – you can find full instructions at [wfmag.cc/XVIIeD](https://wfmag.cc/XVIIeD)

```
from random import randint

WIDTH, HEIGHT = 1200, 700 # Dimensions of the screen (pixels)
shots, to_delete, first_frame = [], [], True

def create_random_shot():
    shots.append({'pos': [randint(0, (WIDTH-images.shot.get_
width())/10)*10, 0],
        'sprite': images.shot,
        'exploding': False})

# A shot will be created in random position every half second
clock.schedule_interval(create_random_shot, 0.5) # Try
reducing number to 0.1!

def draw():
    global first_frame, to_delete
    if first_frame:
        for x in range(50, WIDTH, 300):
            screen.blit(images.shield, [x, 500])
            first_frame = False

    for item in to_delete:
        screen.blit(item['sprite'], item['pos'])
        to_delete = [] # Clear list

    for shot in shots:
        screen.blit(shot['sprite'], shot['pos'])

def update(dt):
    # Step backwards through shots list; avoids errors that occur

    # when deleting items from the list during the for loop
    for i in range(len(shots)-1, -1, -1):
        shot = shots[i]
        if shot['exploding']:
            shot['timer'] -= 1
            if shot['timer'] <= 0:
                to_delete.append({'pos': shot['pos'], 'sprite': images.explode_black})
                del shots[i]
            else:
                # Before moving shot, add the current position to the
                to_delete.append({'pos': shot['pos'].
                copy(), 'sprite': images.shot_black})
                shot['pos'][1] += 20 # Move down the screen
                # Do collision detection based on the centre of the
                sprite
                half_width = shot['sprite'].get_width() // 2 # // =
                integer divide
                half_height = shot['sprite'].get_height() // 2
                if shot['pos'][1]+half_height >= HEIGHT:
                    del shots[i] # Gone off bottom of screen
                else:
                    # Hit something? If so change to exploding sprite
                    collide_check_pos = (shot['pos'][0]+half_width,
                    shot['pos'][1]+half_height)
                    if screen.surface.get_at(collide_check_pos) !=
                    (0,0,0):
                        shot['sprite'] = images.explode
                        shot['exploding'] = True
                        shot['timer'] = 5
```

it's displayed in the same position for a few frames before being deleted. The second list, **to\_delete**, is for sprites which need to be deleted from the screen. For simplicity, I'm using separate copies of the shot and explosion sprites where the white pixels have been changed to black (the other pixels in these sprites are set as transparent).

The function **create\_random\_shot** is called every half second. The combination of dividing the maximum value by ten, choosing a random whole number between zero and the maximum value, and then multiplying the resulting random number by ten, ensures that the chosen X coordinate is a multiple of ten.

In the draw function, we first check to see if it's the first frame, as we only want to

display the shields on that frame. The **screen.blit** method is used to display sprites, and Pygame Zero's **images** object is used to specify which sprite should be displayed. We then display all sprites in the **to\_delete** list, after which we reset it to being an empty list. Finally we display all sprites in the **shots** list.

In the update function, we go through all sprites in the **shots** list, in reverse order.

♥ *Space Invaders*-style shields running in Pygame Zero – watching them gradually disintegrate is oddly soothing.



Going through the list backwards avoids problems that can occur when deleting items from a list inside a **for** loop. For each shot, we first check to see if it's in 'exploding' mode. If so, its timer is reduced each frame – when it hits zero we add the shot to the **to\_delete** list, then delete it from **shots**.

If the item is a normal shot rather than an explosion, we add its current position to **to\_delete**, then update the shot's position to move the sprite down the screen. We next check to see if the sprite has either gone off the bottom of the screen or collided with something. Pygame's **get\_at** method gives us the colour of a pixel at a given position. If a collision occurs, we switch the shot into 'exploding' mode – the explosion sprite will be displayed for five frames. 🗨



➤ Aliens swoop down towards the player, bombing as they go. Back in 1979, this was a big step forward from Taito's *Space Invaders*.

Source Code



AUTHOR  
MARK VANSTONE

# Recreate Galaxian's iconic **attack patterns**

Blast dive-bombing aliens in our salute to Namco's classic

**H**ot on the heels of the original *Space Invaders*, *Galaxian* emerged as a rival space shooter in 1979. Released by Namco, *Galaxian* brought new colour and unpredictable motion to the alien enemy, who would swoop down on the defending player. *Galaxian* was so popular in arcades that Namco released a sequel, *Galaga*, two years later – that game complicated the attack patterns even more. It's difficult to say how many ports and clones have been made of *Galaxian*, as there are several versions of similar games for almost every home platform.

The player's role in *Galaxian* is similar to *Space Invaders*, in that they pilot a ship and need to destroy a fleet of aliens. With *Galaxian*, however, the aliens have a habit of breaking formation and swooping down towards the player's ship, and dive-bombing it. The aim is to destroy all the enemy ships and move on to the next wave. The subsequent waves of enemies get more difficult as the player progresses. For this sample, we're going to look at that swooping mechanic, and make the bare nuts and bolts of a *Galaxian* game with Pygame Zero.

First, *Galaxian* has a portrait display, so we can set the play area's width and height to be 600 and 800 respectively. Next, we can create a scrolling backdrop of stars using a bitmap that we blit to the screen and move downwards every update. We need a second blit of the stars to fill in the space that the first one leaves as it scrolls down, and we could also have another static background image behind them, which will provide a sense of depth.

Next, we set up the player ship as an Actor, and we'll capture the left and right arrow keys in the `update()` function to move the ship left and right on the screen. We can also fire off a bullet with the **SPACE** bar, which will travel up the screen until it hits an alien or goes off the top of the screen. As in the original *Galaxian*, you can only shoot one bullet at a time, so we only need one Actor for this.

The aliens are arranged in rows and move left and right across the screen together. We'll stick to just one type of alien for this sample, but draw two rows of them. You could add extra types and any number of rows. When we create the alien Actors, we can also add a status flag, and we need to determine which side of the row they're

on as when they break formation, the two sides fly in opposite directions. In this case, there'll be four aliens on the left of each row and four on the right. Once they're set up in a list, we can iterate through the list on each update and move them backwards and forwards. While we're moving our aliens, we can also check to see if they've collided with a bullet or the player ship. If the collision is with a bullet, the alien cycles through a few frames of an explosion using the status flag, and then, when their status reaches five, they're no longer drawn. If the collision is with the player, then the player dies and the game's over. We can also check a random number to see if the alien will start a bombing run; if so, we set the status to one, which will start calls to the `flyAlien()` function. This function checks which side the alien's on and starts changing the alien's angle, depending on the side. It also alters the x and y coordinates, depending on the angle. We've written this section in longhand for clarity, but this could be collapsed down a bit with the use of some multiplier variables for the x coordinates and the angles.

There we have it: the basics of *Galaxian*. Can you flesh it out into a full game? 🙄



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag50](https://wfmag.cc/wfmag50)

# Massive attack

Here's Mark's dive-bombing *Galaxian* code. To get it working on your system, you'll need to install Pygame Zero – full instructions are available at [wfmag.cc/pgzero](https://wfmag.cc/pgzero).

```
# Galaxian
from random import randint
WIDTH = 600
HEIGHT = 800

bullet = Actor('bullet', center=(0, -10))
ship = Actor('ship', center=(300, 700))
backY = count = gameover = 0
aliens = []
for a in range(0, 8):
    aliens.append(Actor('alien', center=(200+(a*50),200)))
    aliens[a].status = 0
    aliens[a].side = int(a/4)

for a in range(0, 8):
    aliens.append(Actor('alien', center=(200+(a*50),250)))
    aliens[a+8].status = 0
    aliens[a+8].side = int(a/4)

def draw():
    screen.blit("background", (0, 0))
    screen.blit("stars", (0, backY))
    screen.blit("stars", (0, backY-800))
    bullet.draw()
    drawAliens()
    if gameover != 1 or (gameover == 1 and count%2 == 0): ship.
draw()

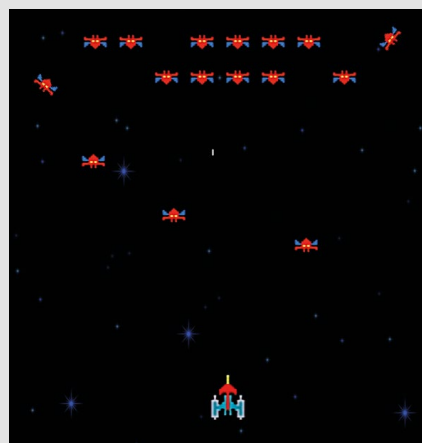
def update():
    global backY, count
    count += 1
    if gameover == 0:
        backY += 0.2
        if backY > 800: backY = 0
        if bullet.y > -10: bullet.y -= 5
        if keyboard.left and ship.x > 50 : ship.x -= 4
        if keyboard.right and ship.x < 550 : ship.x += 4
        if keyboard.space :
            if bullet.y < 0: bullet.pos = (ship.x,700)
            updateAliens()

def drawAliens():
    for a in range(0, 16):
        if aliens[a].status < 5 : aliens[a].draw();

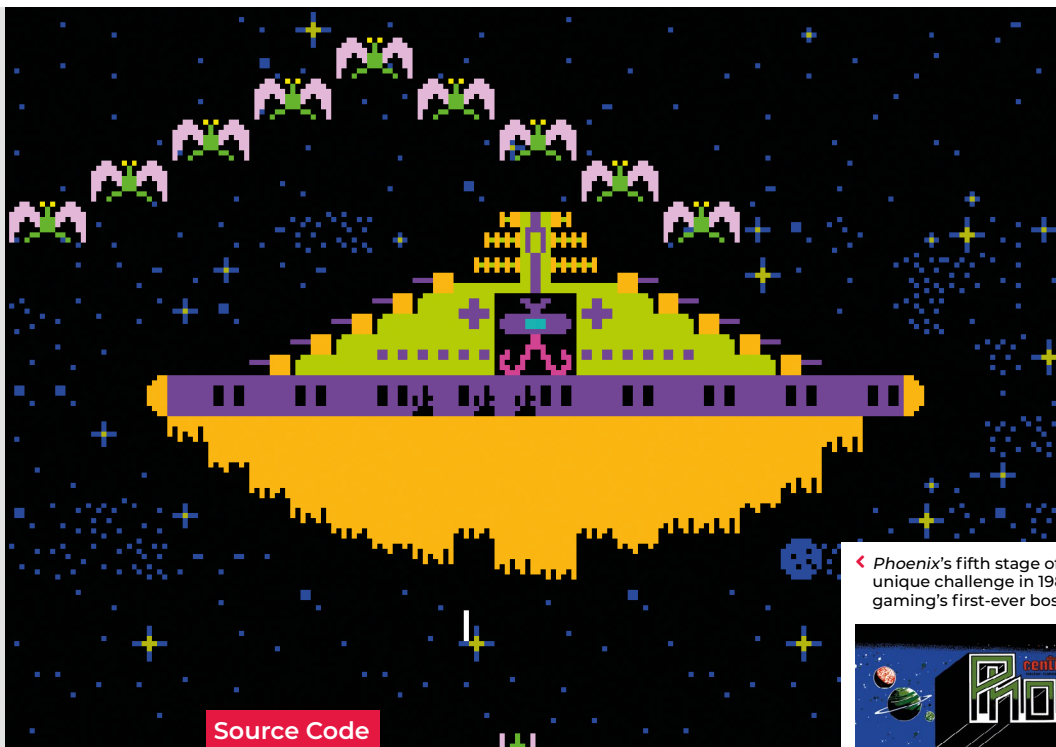
def updateAliens():
    global gameover
    for a in range(0, 16):
        aliens[a].image = "alien0"
        if count%30 < 15 : aliens[a].image = "alien1"
        if count%750 < 375:
            aliens[a].x -=0.4
```

```
else:
    aliens[a].x +=0.4
    if aliens[a].collidepoint(bullet.pos) and aliens[a].
status < 2:
    aliens[a].status = 2
    bullet.y = -10
    if aliens[a].colliderect(ship) : gameover = 1
    if randint(0,1000) == 1 and aliens[a].status == 0 :
aliens[a].status = 1
    if aliens[a].status == 1 : flyAlien(a)
    if aliens[a].status > 1 and aliens[a].status < 5:
        aliens[a].image = "alien" + str(aliens[a].status)
        aliens[a].status += 1

def flyAlien(a):
    if aliens[a].side == 0:
        if aliens[a].angle < 180 :
            aliens[a].angle += 2
            aliens[a].x -= 1
            if aliens[a].angle < 90: aliens[a].y -= 1
        if aliens[a].angle >= 90 :
            aliens[a].y += 2
        if aliens[a].angle >= 180 :
            aliens[a].angle = 180
            aliens[a].x += 1
    else:
        if aliens[a].angle > -180 :
            aliens[a].angle -= 2
            aliens[a].x += 1
            if aliens[a].angle > -90: aliens[a].y -= 1
        if aliens[a].angle <= -90 :
            aliens[a].y += 2
        if aliens[a].angle <= -180 :
            aliens[a].angle = -180
            aliens[a].x -= 1
```



◀ Our homage to the classic *Galaxian*, with angry aliens that love to break formation.



Source Code

◀ *Phoenix's* fifth stage offered a unique challenge in 1980: one of gaming's first-ever boss battles.



AUTHOR  
MARK VANSTONE

# Code a **Phoenix-style** mothership battle

It was one of gaming's first boss battles. Mark shows you how to recreate the mothership from 1980's *Phoenix*

**F**irst released in 1980, *Phoenix* was something of an arcade pioneer. The game was the kind of post-*Space Invaders* fixed-screen shooter that was ubiquitous at the time: players moved their ship from side to side, shooting at a variety of alien birds of different sizes and attack patterns. The enemies moved swiftly, and the player's only defence was a temporary shield which could be activated when the birds swooped and strafed the lone defender. But besides

all that, *Phoenix* had a few new ideas of its own: not only did it offer five distinct stages, but it also featured one of the earliest examples of a boss battle – its heavily armoured alien mothership, which required accurate shots to its shields before its weak spot could be exposed.

To recreate *Phoenix's*

boss, all we need is Pygame Zero. We can get a portrait style window with the **WIDTH** and **HEIGHT** variables and throw in some parallax stars (an improvement on the original's static backdrop) with some blitting in the **draw()** function. The parallax effect is created by having a static background

**“The mothership is made up of several Actor objects which move together”**

of stars with a second (repeated) layer of stars moving down the screen.

The mothership itself is made up of several Actor objects which move together down the screen towards the player's spacecraft, which can be moved right and left using the mouse. There's the main body of the mothership, in the centre is the alien that we want to shoot, and then we have two sets of moving shields. In this

example, rather than have all the graphics dimensions in multiples of eight (as we always did in the old days), we will make all our shield blocks 20 by 20 pixels, because computers simply don't need to work in multiples of eight any more. The first set of shields is the purple rotating bar around the middle of the ship. This is made up of 14 Actor blocks which shift one place to the right each time they move. Every other block has a couple of portal windows which makes the rotation obvious, and when a block moves off the right-hand side, it is placed on the far left of the bar.

The second set of shields are in three yellow rows (you may want to add more), the first with 14 blocks, the second with ten blocks, and the last with four. These shield blocks are fixed in place but share a behaviour with the purple bar shields, in that when they are hit by a bullet, they change to a damaged version. There are four levels of damage before







Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag26](https://wfmag.cc/wfmag26)

# Phoenix in Python

Here's Mark's code snippet, which recreates that pioneering boss battle in Python. To get it running on your system, you'll first need to install Pygame Zero – you can find full instructions at [wfmag.cc/pgzero](https://wfmag.cc/pgzero).

```
WIDTH = 600
HEIGHT = 800

mothership = Actor('mothership', center=(300, 100))
bullet = Actor('bullet', center=(0, -10))
alien = Actor('aliendude', center=(300, 110))
ship = Actor('ship', center=(300, 700))
barShield = []
lowerShield = []
backY = count = mothership.frame = gameover = 0
for b in range(0, 14):
    barShield.append(Actor('bar1'+str(b%2),
center=(310+((b-7)*20), 140)))
    lowerShield.append(Actor('shield1',
center=(310+((b-7)*20), 160)))
    barShield[b].frame = lowerShield[b].frame = 1
for b in range(0, 10):
    lowerShield.append(Actor('shield1',
center=(310+((b-5)*20), 180)))
    lowerShield[b + 14].frame = 1
for b in range(0, 4):
    lowerShield.append(Actor('shield1',
center=(310+((b-2)*20), 200)))
    lowerShield[b + 24].frame = 1

def draw():
    screen.blit("background", (0, 0))
    screen.blit("stars", (0, backY))
    screen.blit("stars", (0, backY-800))
    mothership.draw()
    if gameover != 1 or (gameover == 1 and count%2 == 0):
alien.draw()
    for b in range(0, 28):
        if b < 14:
            if barShield[b].frame < 5:
                barShield[b].draw()
            if lowerShield[b].frame < 5:
                lowerShield[b].draw()
        bullet.draw()
    if gameover != 2 or (gameover == 2 and count%2 == 0):
ship.draw()
```

```
def update():
    global backY, count, gameover
    count += 1
    if gameover == False:
        backY += 0.2
        if backY > 800: backY = 0
        mothership.y += 0.1
        mothership.frame = int(count/10)%14
        alien.y = mothership.y + 10
        for b in range(0, 28):
            if b < 14:
                x = ((mothership.frame+b)-7)*20
                if x >= 140: x -= 280
                barShield[b].y += 0.1
                barShield[b].x = (mothership.x+10)+ x
                if barShield[b].frame < 5 and barShield[b].
colliderect(bullet):
                barShield[b].frame += 1
                if barShield[b].frame < 5:
                    barShield[b].image =
"bar"+str(barShield[b].frame)
                    bullet.y = -10
                    lowerShield[b].y += 0.1
                    if lowerShield[b].frame < 5 and lowerShield[b].
colliderect(bullet):
                    lowerShield[b].frame += 1
                    if lowerShield[b].frame < 5:
                        lowerShield[b].image =
"shield"+str(lowerShield[b].frame)
                        bullet.y = -10
                    if alien.colliderect(bullet): gameover = 1
                    if ship.colliderect(mothership): gameover = 2
                    if bullet.y > -10: bullet.y -= 5

def on_mouse_down(pos):
    if bullet.y < 0: bullet.pos = (ship.x, 700)

def on_mouse_move(pos):
    ^ship.x = pos[0]
```

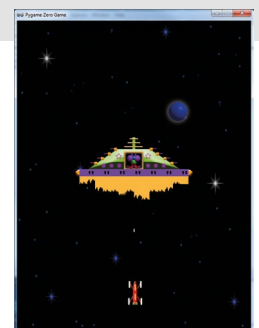
they are destroyed and the bullets can pass through. When enough shields have been destroyed for a bullet to reach the alien, the mothership is destroyed (in this version, the alien flashes).

Bullets can be fired by clicking the mouse button. Again, the original game had alien birds flying around the mothership and dive-bombing the player,

making it harder to get a good shot in, but this is something you could try adding to the code yourself.

To really bring home that eighties *Phoenix* arcade experience, you could also add in some atmospheric shooting effects and, to round the whole thing off, have an 8-bit rendition of Beethoven's *Für Elise* playing in the background. 🎵

➤ Like the original *Phoenix*, our mothership boss battle has multiple shields that need to be taken out to expose the alien at the core.





◀ Options first appeared in 1985's *Gradius*, but became a mainstay of numerous sequels and spin-offs, including the *Salamander* and *Parodius* series of games.



# Gradius' ship-following Options



AUTHOR  
RIK CROSS

Learn how to create game objects that follow the path of the main player sprite

**F**irst released by Konami in 1985, *Gradius* pushed the boundaries of the shoot-'em-up genre with its varied level design, dramatic boss fights, and innovative power-up system.

One of the most memorable of its power-ups was the Option – a small, drone-like blob that followed the player's ship and effectively doubled its firepower. By collecting more power-ups, it was possible to gather a cluster of death-dealing Options, which obediently moved wherever the player moved.

There are a few different ways of recreating *Gradius*'s sprite-following, but in this article, I'll show you a simple implementation that uses the player's 'position history' to place other following items on the screen. As always, I'll be using Python and Pygame to recreate this effect, and I'll be making use of a spaceship image created by 'pitrizzo' from [opengameart.org](http://opengameart.org).

The first thing to do is to create a spaceship and a list of 'power-up' objects. Storing the power-ups in a list allows us to perform a simple calculation on a power-up to determine its position, as you'll see later. As we'll be iterating through the power-ups stored in a list, there's no need to create

**"A small, drone-like blob that followed the player"**

a separate variable for each. Instead, we can use list comprehension to create the power-ups:

```
powerups = [Actor('powerup') for p in range(3)]
```

The player's position history will be a list of previous positions, stored as a list of (x,y) tuples. Each time the player's position changes, the new position is added to the front of the list (as the new first element).

We only need to know the spaceship's recent position history, so the list is also truncated to only contain the 100 most recent positions. Although not necessary, the following code can be added to allow you to see a selection (in this case every fifth) of these previous positions:

```
for p in previouspositions[::5]:
    screen.draw.filled_circle(p, 2,
    (255,0,0))
```

Each frame of the game, this position list is used to place each of the power-ups. In our *Gradius*-like example, we need each of these objects to follow the player's spaceship in a line, as if moving together in a single-file queue. To achieve this effect, a power-up's position is determined by its position in the power-ups list, with the first power-up in the list taking up a position nearest to the player. In Python, using `enumerate` when iterating through a list allows us to get the power-up's position



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag16](https://wfmag.cc/wfmag16)

# Sprite-following Options in Python

Here's a code snippet that creates *Gradius*-style Options in Python. To get it running on your system, you'll first need to install Pygame Zero – you can find full instructions at [wfmag.cc/pgzero](https://wfmag.cc/pgzero)

```
# set screen width and height
WIDTH = 800
HEIGHT = 800

# create spaceship and a list of 3 powerups
spaceship = Actor('spaceship', pos=(400, 400))
spaceship.speed = 4
powerups = [Actor('powerup') for p in range(3)]

# create a list of previous positions
# initially containing values to the left of the spaceship
previouspositions = [(spaceship.x - i*spaceship.speed, spaceship.y) for i in range(100)]

def update():

    global previouspositions

    # store spaceship previous position
    previousposition = (spaceship.x, spaceship.y)

    # use arrow keys to move the spaceship
    if keyboard.up:
        spaceship.y -= spaceship.speed
    if keyboard.down:
```

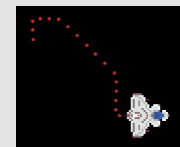
```
        spaceship.y += spaceship.speed
    if keyboard.left:
        spaceship.x -= spaceship.speed
    if keyboard.right:
        spaceship.x += spaceship.speed

    # add new position to list if the spaceship has moved
    # and ensure the list contains at most 100 positions
    if previousposition != spaceship.pos:
        previouspositions = [(spaceship.x, spaceship.y)] +
previouspositions[:99]

    # set the new position of each powerup
    for i, p in enumerate(powerups):
        newposition = previouspositions[(i+1)*20]
        p.pos = (newposition[0], newposition[1])
```

```
def draw():
    screen.clear()
    spaceship.draw()
    for p in powerups:
        p.draw()
```

✓ Plotting the  
spaceship's  
position history.



✓ Power-ups  
following a  
player sprite,  
using the player's  
position history.



in the list, which can then be used to determine which position in the player's position history to use.

```
newposition = previouspositions[(i+1)*20]
```

So, the first power-up in the list (element 0 in the list) is placed at the coordinates of the 20th  $((0+1)*20)$  position in the spaceship's history, the second power-up at the 40th position, and so on. Using this simple calculation, elements are equally-spaced along the spaceship's previous path. The only thing to be careful of here is that you have enough items in the position history for the number of items you want to follow the player!

This leaves one more question to answer; where do we place these power-ups initially, when the spaceship has no position history? There are a few different ways of solving this problem, but the simplest is just to generate a fictitious position history at the beginning of the game. As I want power-ups to be lined up behind the spaceship

initially, I again used list comprehension to generate a list of 100 positions with ever-decreasing x-coordinates.

```
previouspositions = [(spaceship.x -
i*spaceship.speed, spaceship.y) for i in
range(100)]
```

With an initial spaceship position of (400,400) and a **spaceship.speed** of 4, this means the list will initially contain the following coordinates:

```
previouspositions = [(400,400), (396,400),
(392,400), (388,400), ...]
```

Storing our player's previous position history has allowed us to create path-following power-ups with very little code. The idea of storing an object's history can have very powerful applications. For example, a paint program could store previous commands that have been executed, and include an 'undo' button that can work backwards through the commands. 🐍

## LIST COMPREHENSION

List comprehensions are a way of creating a list, using other iterables (a sequence of objects). This is really handy for creating large lists and/or lists where the elements follow a pattern. List comprehension has been used twice in this sprite-following example; for creating a list of three power-ups and for quickly creating a list of 100 previous spaceship positions.

```
>>> squares = [i*i for i in range(5)]
>>> squares
>>> [0,1,4,9,16]

>>> numbers = [2,3,4,5,6,7,8,9, 'J', 'Q', 'K', 'A']
>>> suits =
['Hearts', 'Clubs', 'Spades', 'Diamonds']
>>> playing_cards = [(n,s) for n in
numbers for s in suits]
>>> playing_cards
>>> [(2, 'Hearts'), (2, 'Clubs'), (2,
'Spades'), (2, 'Diamonds'), (3, 'Hearts'),
...]
```

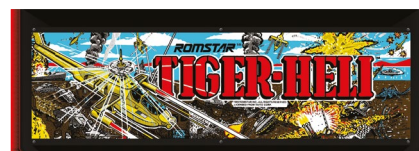
## Shooter

Tiger Heli

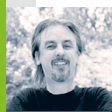


◀ Fly over the military targets, firing missiles and dropping bombs.

▼ *Tiger-Heli* was developed by Toaplan and published in Japan by Taito and by Romstar in North America.



# Recreate Tiger-Heli's bomb mechanic



AUTHOR  
MARK VANSTONE

Code an explosive homage to Toaplan's classic blaster

**R**eleased in 1985, *Tiger-Heli* was one of the earliest games from Japanese developer Toaplan: a top-down shoot-'em-up that pitted a lone helicopter against relentless waves of enemy tanks and military installations. Toaplan would go on to refine and evolve the genre through the eighties and nineties with such titles as *Truxton* and *Fire Shark*, so *Tiger-Heli* served as a kind of blueprint for the studio's legendary blasters.

*Tiger-Heli* featured a powerful secondary weapon, too: as well as a regular shot, the game's attack helicopter could also drop a deadly bomb capable of destroying everything within its blast radius. The mechanic was one that first appeared as far back as Atari's *Defender* in 1981, but Toaplan quickly made it its own, with variations on the bomb becoming one of the signatures in the studio's later games.

For our *Tiger-Heli*-style Pygame Zero code, we'll concentrate on the unique bomb aspect, but first, we need to get the basic

scrolling background and helicopter on the screen. In a game like this, we'd normally make the background out of tiles that can be used to create a varied but continuous scrolling image. For this example, though, we'll keep things simple and have one long image that we scroll down the screen and then display a copy above it. When the first image goes off the screen, we just reset the co-ordinates to display it above the second image copy. In this way, we can have an infinitely scrolling background.

The helicopter can be set up as an Actor with just two frames for the movement of the rotors. This should look like it's hovering above the ground, so we blit a shadow bitmap to the bottom right of the helicopter. We can set up keyboard events to move the Actor left, right, up, and down, making sure we don't allow it to go off the screen.

Now we can go ahead and set up the bombs. We can predefine a list of bomb Actors but only display them while the bombs are active. We'll trigger a bomb drop with the **SPACE** bar and set all the bombs

to the co-ordinates of the helicopter. Then, frame by frame, we move each bomb outwards in different directions so that they spread out in a pattern. You could try adjusting the number of bombs or their pattern to see what effects can be achieved. When the bombs get to frame 30, we start changing the image so that we get a flashing, expanding circle for each bomb.

It's all very well having bombs to fire, but we could really do with something to drop them on, so let's make some tank Actors waiting on the ground for us to destroy. We can move them with the scrolling background so that they look like they're static on the ground. Then if one of our bombs has a collision detected with one of the tanks, we can set an animation going by cycling through a set of explosion frames, ending with the tank disappearing.

We can also add in some sound effects as the bombs are dropped, and explosion sounds if the tanks are hit. And with that, there you have it: the beginnings of a *Tiger-Heli*-style blaster. 🎮





Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag45](https://wfmag.cc/wfmag45)

# Making bombs in Python

Here's Mark's code for a *Tiger Heli*-style shooter, complete with deadly bombs. To get it running on your system, you'll need to install Pygame Zero – full instructions can be found at [wfmag.cc/pgzero](https://wfmag.cc/pgzero).

```
# Tiger-Heli

WIDTH = 600
HEIGHT = 800
backgroundY = count = 0
heli = Actor('heli1', center=(300, 650))
bombActive = False
bombs = []
bombDirs = [(0,1),(1,1),(1,0),(0,0),(0,-1),(-1,-1),(-1,0),(-1,1),
(1,-1),(-0.5,0),(0.5,0.5),(-0.5,-0.5),(0.5,-0.5),(0,-0.5),(-0.5,0.5),(-0.5,1),
(1,-0.5),(-1,-0.5),(0.5,-1)]
for b in range(0, 18):
    bombs.append(Actor('bomb1', center=(0,0)))
    bombs[b].frame = 0
tankLocations = [(500,-250),(100,-250),(300,-500)]
tanks = []
for t in range(0,3):
    tanks.append(Actor('tank0', center=(tankLocations[t]
[0],tankLocations[t][1])))
    tanks[t].frame = 0

def draw():
    screen.blit('background',(0,backgroundY))
    screen.blit('background',(0,backgroundY-1400))
    screen.blit("helishadow"+str(count%2 + 1),(heli.x+10,heli.
y+10))
    for t in range(0,3):
        if tanks[t].frame < 10:
            tanks[t].draw()
    if bombActive == True:
        for b in range(0, 18):
            bombs[b].draw()
    heli.draw()

def update():
    global backgroundY, count,bombActive
    backgroundY += 1
    if backgroundY > 1400: backgroundY = 0
    heli.image = "heli"+str(count%2 + 1)
    if keyboard.left and heli.x > 50 : heli.x -= 2
    if keyboard.right and heli.x < 550 : heli.x += 2
    if keyboard.up and heli.y > 50 : heli.y -= 2
    if keyboard.down and heli.y < 650 : heli.y += 2
    if keyboard.space : fireBomb()
    for t in range(0,3):
        tanks[t].y = (tankLocations[t][1] + backgroundY)
        if tanks[t].y > 850: tanks[t].frame = 0
        if tanks[t].frame > 0 and tanks[t].frame < 10 :
            tanks[t].frame += 0.2
        tanks[t].image = "tank"+str(int(tanks[t].frame))
    if bombActive == True:
        for b in range(0, 18):
```

```
        bombs[b].y += 1
        bombs[b].x += bombDirs[b][0]*5
        bombs[b].y += bombDirs[b][1]*5
        bombs[b].frame += 1
        if bombs[b].frame > 30:
            bombs[b].image = "bomb"+str(bombs[b].frame-30)
            for t in range(0,3):
                if bombs[b].collidepoint(tanks[t].pos) and
tanks[t].frame == 0:
                    tanks[t].frame = 1
                    sounds.explosion.play()
            if bombs[b].frame == 40:
                bombActive = False
                count += 1

def fireBomb():
    global bombActive
    if bombActive == False :
        bombActive = True
        sounds.launch.play()
        for b in range(0, 18):
            bombs[b].frame = 1
            bombs[b].pos = heli.pos
            bombs[b].image = "bomb1"
```



< Our homage to the classic *Tiger-Heli* arcade game.



Source Code

▲ Dobkeratops: stunning in 1987, still a cool design decades later.

## R-Type's spectacular use of modular sprites

The 1987 shooter R-Type used multiple sprites to create the illusion of a much larger end-of-level boss. Here's how to recreate the effect



AUTHOR  
RYAN LAMBIE

Still images don't quite do it justice. The screen faded to an inky black; an ominous new tune began; then the monster scrolled into view. The tail appeared first, lashing up and down; then in came the rest of the abomination, with its skeletal frame, snapping jaws and floating eyeballs. It was a startling sight, particularly back in 1987, when *R-Type* was still a newcomer to the world's amusement arcades. Japanese developer Irem evidently knew that it had something special on its hands with this huge, screen-filling monster, which they called Dobkeratops: they placed it right at the end of stage one, maximising the chance that players would get to it; much of the game's promo artwork also featured the creature's hideous likeness.

Aside from the striking graphic design – which owes a debt to *Alien* artist Hans Ruedi Giger – *R-Type*'s level one boss showcased

Irem's technical ingenuity. Characters the size of Dobkeratops were vanishingly rare in eighties games, largely because hardware and memory restraints made putting them on the screen so difficult. What Irem did, though, was use several smaller sprites to create the illusion of a single, animated monster.

### “Characters the size of Dobkeratops were rare in eighties games”

Look again at Dobkeratops in action, and you can see that only three elements move: its jaw, which moves up and down; a green, parasite-like organism that emerges from its stomach (another nod to the 1979 movie, *Alien*), and most eye-catching, the long, whipping tail. The rest of the beast is essentially a static image, stored in the

hardware's memory in sections. The jaw and stomach-bursting parasite are dealt with in just twelve frames of animation altogether. Similarly, the tail actually consists of 18 relatively tiny sprites, which are programmed to whip and curl in a smoothly organic fashion, like a horrifying string of beads. Unless it was pointed out to the average player grappling with *R-Type* back in the eighties, they probably wouldn't have noticed Irem's sleight of hand.

### ORGANIC ORIGINS

Ingenious though it was, *R-Type* built its success on the foundations laid by other games. In 1985, Konami launched the seminal *Gradius* – another scrolling shooter that, like *R-Type*, saw a lone pilot fly through a succession of hostile environments, blasting aliens. While that game was in development, its director, Hiroyasu Machiguchi, said that he wanted to create a stage with an organic





^ Chained together and animated across a sinusoidal curve, these little sprites create the effect of a whipping, alien tail.

## How it all works

To illustrate how Irem created that tail, we asked Daniel Pope to come up with a piece of code in Pygame Zero – you can find this on the right. It uses the same principle that Irem came up with: multiple sprites are drawn on the screen and then animated using the sine function to create a whipping motion. “So that the components of the chain appear to stay connected, we keep the pieces at a constant distance (SEGMENT\_SIZE) from each other, and only control the angle between each piece,” Daniel explains. “Given an angle, we use the trigonometric functions `sin()` and `cos()` to work out the Cartesian (x and y) position of the segment, and each segment is positioned relative to the previous one.”

feel, since most of the other stages featured mechanical enemies. While wondering how to make something move organically on eighties hardware, Machiguchi reportedly came up with an idea: he told his designers to draw a sprite “that looks like a pachinko ball”, and use multiple instances of them to create a moving tentacle.

In a 2006 interview (translated by the good people at shmuplations.com), designer Kengo Nakamura recalls that it took two days of experimentation with this concept, but the results were immediately striking. “After about two days, I believe, we came up with the creeping movement of the tentacles,” Nakamura said. “We made each little pachinko ball in his arm move individually, and everyone was amazed at how, in a short time, the design had become so realistic, disturbing and gross.”

*Gradius*’s tentacled monsters were tucked away on stage five, and therefore less easy to reach than *R-Type*’s Dobkeratops. *Gradius* almost certainly laid the groundwork for *R-Type*, however; at least one of *R-Type*’s developers was a self-avowed fan of the game. If anything, though, *R-Type* uses the concept of organic movement through modular sprites to even greater effect; the expert sprite design, combined with that aggressively lashing tail, resulted in one of the greatest area bosses ever conceived. 🐍

## An R-Type tail in Python

Here’s a code snippet that shows an *R-Type*-esque modular tail working in Python. The code requires a pair of sprites to work: **tail\_piece.png** and **tail\_hook.png**, which you’ll find (along with the code itself) at the GitHub link on the right. The code also requires you to install Pygame Zero – you can find full instructions at [wfmag.cc/XVlleD](http://wfmag.cc/XVlleD)



Download the code from GitHub: [wfmag.cc/wfmag6](http://wfmag.cc/wfmag6)

```
from math import sin, cos

# Constants that control the wobble effect
SEGMENT_SIZE = 50 # pixels from one segment to the next
ANGLE = 2.5 # Base direction for the tail (radians)
PHASE_STEP = 0.3 # How much the phase differs in each tail piece (radians)
WOBBLE_AMOUNT = 0.5 # How much of a wobble there is (radians)
SPEED = 4.0 # How fast the wobble moves (radians per second)

# Dimensions of the screen (pixels)
WIDTH = 800
HEIGHT = 800

# The sprites we'll use.
# 10 tail pieces
tail = [Actor('tail_piece') for _ in range(10)]
# Plus a hook piece at the end
tail += [Actor('tail_hook')]

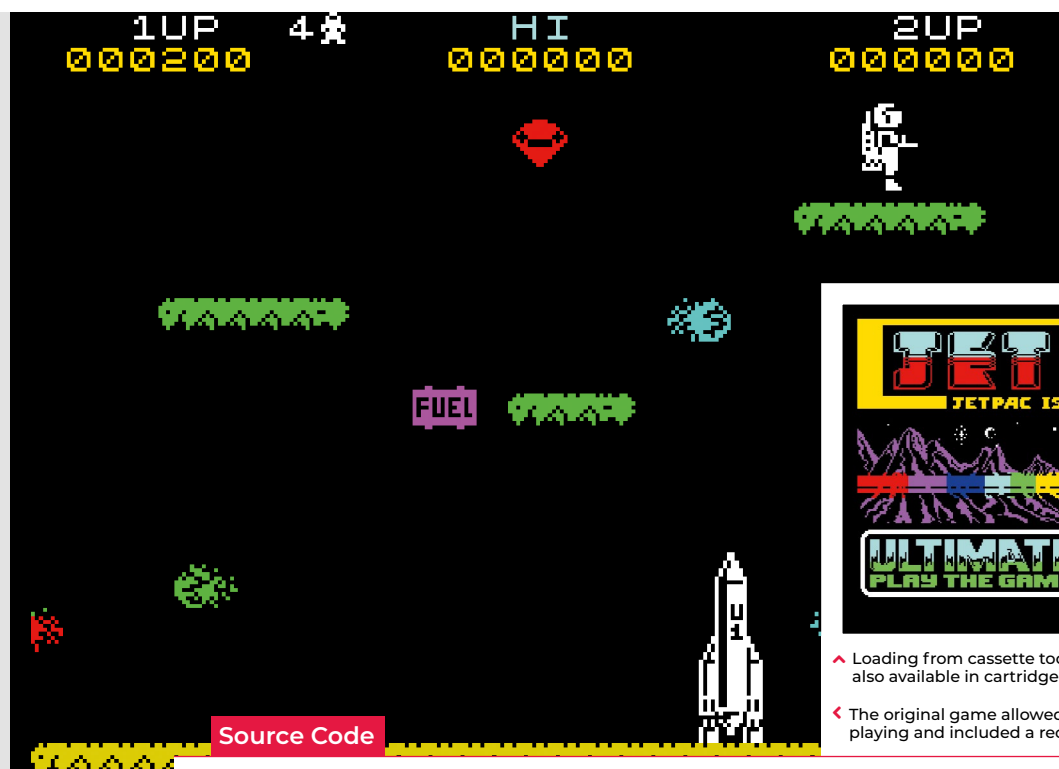
# Keep track of time
t = 0 # seconds

def draw():
    screen.clear()
    # First draw the even tail pieces
    for a in tail[::2]:
        a.draw()
    # Now draw the odd tail pieces
    for a in tail[1::2]:
        a.draw()

def update(dt):
    global t
    t += dt
    # Start at the bottom right
    x = WIDTH - SEGMENT_SIZE // 2
    y = HEIGHT - SEGMENT_SIZE // 2
    for seg, a in enumerate(tail):
        a.pos = x, y

        # Calculate an angle to the next piece which wobbles sinusoidally
        angle = ANGLE + WOBBLE_AMOUNT * sin(seg * PHASE_STEP + t * SPEED)

        # Get the position of the next piece using trigonometry
        x += SEGMENT_SIZE * cos(angle)
        y -= SEGMENT_SIZE * sin(angle)
```



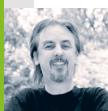
Source Code



^ Loading from cassette took a while, but Jetpac was also available in cartridge format.

< The original game allowed two players to take turns playing and included a record of the highest score.

## Code Jetpac's rocket building action



AUTHOR  
MARK VANSTONE

Pick up parts of a spaceship, fuel it up, and take off in Mark's rendition of a ZX Spectrum classic

For ZX Spectrum owners, there was something special about waiting for a game to load, with the sound of zeros and ones screeching from the cassette tape player next to the computer. When the loading screen – an image of an astronaut and Ultimate Play the Game's logo – appeared, you knew the wait was going to be worthwhile. Created by brothers Chris and Tim Stamper in 1983, *Jetpac* was one of the first hits for their studio, Ultimate Play the Game. The game features the hapless astronaut Jetman, who must build and fuel a rocket from the parts dotted around the screen, all the while avoiding or shooting swarms of deadly aliens.

This month's code snippet will provide the mechanics of collecting the ship parts and fuel to get Jetman's spaceship to take off. We can use the in-built Pygame Zero Actor objects for all the screen elements and the Actor collision routines to deal with gravity

and picking up items. To start, we need to initialise our Actors. We'll need our Jetman, the ground, some platforms, the three parts of the rocket, some fire for the rocket engines, and a fuel container. The way each Actor behaves will be determined by a set of lists. We have a list for objects with gravity,

### "Assemble a rocket, fill it with fuel, and lift off"

objects that are drawn each frame, a list of platforms, a list of collision objects, and the list of items that can be picked up.

Our `draw()` function is straightforward as it loops through the list of items in the draw list and then has a couple of conditional elements being drawn after. The `update()` function is where all the action happens: we check for keyboard input to move Jetman around, apply gravity to all the items on the gravity list, check for collisions with the platform list, pick up the next item if Jetman

is touching it, apply any thrust to Jetman, and move any items that Jetman is holding to move with him. When that's all done, we can check if refuelling levels have reached the point where Jetman can enter the rocket and blast off.

If you look at the helper functions `checkCollisions()` and `checkTouching()`, you'll see that they use different methods of collision detection, the first being checking for a collision with a specified point so we can detect collisions with the top or bottom of an actor, and the touching collision is a rectangle or bounding box collision, so that if the bounding box of two Actors intersect, a collision is registered. The other helper function `applyGravity()` makes everything on the gravity list fall downward until the base of the Actor hits something on the collide list.

So that's about it: assemble a rocket, fill it with fuel, and lift off. The only thing that needs adding is a load of pesky aliens and a way to zap them with a laser gun. ☹



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag40](https://wfmag.cc/wfmag40)

# Rocket building in Python

Here's Mark's *Jetpac* code snippet. To get it running on your system, you'll need to install Pygame Zero – you can find full instructions at [wfmag.cc/pgzero](https://wfmag.cc/pgzero).

```
import random
import time
t0 = time.clock()
jetman = Actor('jetman1',(400,500))
ground = Actor('ground',(400,550))
platform1 = Actor('platform1',(400,350))
platform2 = Actor('platform2',(200,200))
platform3 = Actor('platform3',(650,200))
rocket1 = Actor('rocket1',(520,500))
rocket2 = Actor('rocket2',(400,300))
rocket3 = Actor('rocket3',(200,150))
rocketFire = Actor('rocketfire',(521,0))
fuel = Actor('fuel',(50,-50))
gravityList = [jetman,rocket1,rocket2,rocket3,fuel]
drawList = [rocket1,rocket2,rocket3,ground,platform1,
platform2,platform3,fuel]
platformList = [ground,platform1,platform2,platform3]
collideList = [rocket1,rocket2,rocket3]
pickupList = [rocket2,rocket3,fuel,0,fuel,0,fuel,0,0]
gravity = 1.5
jetman.thrust = jetman.holding = jetman.item = gameState =
fuelLevel = timeElapsed = 0
jetman.dir = "1"
def draw():
    global timeElapsed
    screen.clear()
    for i in range(0, len(drawList)):
        drawList[i].draw()
    if gameState == 0:
        jetman.draw()
        timeElapsed = int(time.clock() - t0)
    else:
        rocketFire.draw()
        screen.draw.text("MISSION ACCOMPLISHED", center =
(400, 300), owidth=0.5, ocolor=(255,255,255), color=(0,0,255),
fontsize=80)
        screen.draw.text("TIME:"+str(timeElapsed), center= (400, 20),
owidth=0.5, ocolor=(255,255,255), color=(255,0,0), fontsize=40)
def update():
    global gameState, fuelLevel
    burn = ""
    if gameState == 0:
        if keyboard.up:
            jetman.thrust = limit(jetman.thrust+0.3,0,5)
            burn = "f"
        if keyboard.left:
            jetman.dir = "1"
            jetman.x -= 1
        if keyboard.right:
            jetman.dir = "r"
            jetman.x += 1
    applyGravity()
```

```
coll =
checkCollisions(platformList,(jetman.x,jetman.y-32))
if coll == False:
    jetman.y -= jetman.thrust
if pickupList[jetman.item] != 0:
    if checkTouching(pickupList[jetman.item], jetman):
        jetman.holding = pickupList[jetman.item]
jetman.thrust = limit(jetman.thrust-0.1,0,5)
jetman.image = "jetman" + jetman.dir + burn
if jetman.holding != 0 :
    jetman.holding.pos = jetman.pos
    if jetman.holding.x == rocket1.x and jetman.
holding.y < 440:
        jetman.holding = 0
        jetman.item += 1
    if fuel.x == rocket1.x and fuel.y+16 > rocket3.y-32 and
jetman.holding == 0:
        fuelLevel += 1
    if fuelLevel < 4:
        jetman.item += 1
    if fuelLevel < 3 :
        fuel.pos = (random.randint(50, 750),-50)
    else:
        fuel.pos = (0,650)
        gravityList[fuelLevel].image =
"rocket"+str(fuelLevel)+"f"
    if fuelLevel == 3 and jetman.x == rocket1.x and
jetman.y > rocket3.y:
        gameState = 1
    if gameState == 1:
        rocket1.y -= 1
        rocket2.y -= 1
        rocket3.y -= 1
        rocketFire.y = rocket1.y + 50
def limit(n, minn, maxx):
    return max(min(maxn, n), minn)
def checkCollisions(cList, point):
    for i in range(0, len(cList)):
        if cList[i].collidepoint(point):
            return True
    return False
def checkTouching(a1,a2):
    if a1.colliderect(a2): return True
    return False
def applyGravity():
    for i in range(0, len(gravityList)):
        if
checkCollisions(platformList,(gravityList[i].x,gravityList[i].
y+(gravityList[i].height/2))) == False and
checkCollisions(collideList,(gravityList[i].x,gravityList[i].
y+(gravityList[i].height/2))) == False:
            gravityList[i].y += gravity
```



Source Code



AUTHOR  
RIK CROSS

## Asteroids' thruster motion

Learn how to recreate the iconic player physics from Atari's arcade classic, Asteroids

**A**steroids is a space-shooter game released by Atari in 1979, first as an arcade game and later for the 2600 and other Atari consoles. The aim is to control a spaceship, stay alive and score points, by shooting asteroids as they move around the screen, breaking them into smaller and smaller pieces.

The controls for the player's spaceship were unique because you could only 'thrust' the spaceship forwards in the direction it was facing. The spaceship would then continue in this direction until it either decelerated to rest, or until the spaceship was thrust in another direction. This resulted in some unique player physics that made for simple yet addictive gameplay.

The spaceship thruster motion is achieved by making use of some trigonometry. A force applied at an angle can be broken down into its horizontal and vertical components, acting independently

at right angles to each other. When applied together, these two components have the same effect as the original force.

The horizontal and vertical components can be calculated by taking the cosine and the sine of the angle (respectively) and multiplying by the force. These values can then be used to calculate an object's position over time.

To replicate this type of player motion, you'll need two images for your spaceship; one 'normal' spaceship image, and one that shows the spaceship being thrust forward. You can either create these images yourself, or if (like me) your artistic skills are lacking, you can adapt images from a repository like [opengameart.org](http://opengameart.org).

This example uses Pygame Zero's framework, in which an angle of 0 corresponds to the spaceship facing to the right. The angle of the spaceship is then incremented as the spaceship turns anticlockwise. I've therefore

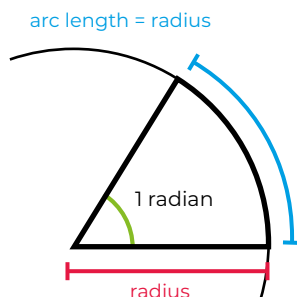
duplicated the image, rotated it so that it is facing to the right, and added flames to the rear of one of the images to show forward acceleration.

Initially, the spaceship is placed in the centre of the screen, with an angle of 0. The spaceship is also given a value for acceleration, as well as horizontal and vertical speeds. As I want the spaceship to be stationary to begin with, both of the values for speed are initially set to 0.

Pressing the 'up' arrow key will apply acceleration to the ship, in the direction that it is currently facing. I'll start by first changing the spaceship's image, so that it appears to be thrusting forward when the 'up' arrow key is pressed.

Spaceship motion is achieved by splitting its acceleration into horizontal and vertical components, and applying each to the corresponding speed variable.

When working with angles, it's often preferred to use radians instead of



^ One radian is the angle made by an arc of equal length to a circle's radius.

degrees. One radian is defined as the angle made by an arc whose length is equal to the radius of a circle. One radian corresponds to about 57 degrees, and there are  $2\pi$  radians in a circle. When using radians, not only are commonly used angles convenient fractions of  $\pi$ , but calculations in radians are less likely to introduce rounding errors.

These updated horizontal and vertical speeds are then used to update the spaceship's position on the screen. Notice that the vertical speed is actually subtracted from the spaceship's position, due to the fact that the 'y' coordinate increases as a sprite moves down the screen.

The 'left' and 'right' arrow keys are used to rotate the spaceship. Because Pygame Zero resets a sprite's angle when its image is changed, notice that the angle is saved to a temporary `new_angle` variable, before being updated and re-applied to the spaceship.

Once the basic ship movement's been implemented, there are many adaptations that can be made. Firstly, the spaceship's acceleration value could be modified, to allow for faster or slower motion around the screen. It's worth trying out different values until you settle on something that feels right. The spaceship could also be made to decelerate when not being thrust forward, to mirror the original game. This deceleration could happen either when the 'up' arrow key isn't being pressed, or when the 'down' arrow key is pressed. As a more difficult challenge, you could even try 'wrapping' the spaceship's movement, so that it appears on the opposite edge of the screen if it travels too far in any particular direction. 🧐

## Asteroids thruster motion in Python

Here's that *Asteroids* thruster code in full. To get it running on your system, you'll first need to install Pygame Zero – you can find full instructions at [wfmag.cc/pgzero](http://wfmag.cc/pgzero)

```
import math

# set screen width and height
WIDTH = 800
HEIGHT = 800

# create a new spaceship, using the 'spaceship.png' image
spaceship = Actor('spaceship')
# place the spaceship in the centre of the screen, facing right
spaceship.center = (WIDTH/2, HEIGHT/2)
spaceship.angle = 0
# set an acceleration for the spaceship
spaceship.ACCELERATION = 0.02
# initially the spaceship is stationary
spaceship.x_speed = 0
spaceship.y_speed = 0

def update():
    # save the spaceship's current angle,
    # as changing the actor's image resets the angle to 0
    new_angle = spaceship.angle

    # rotate left on left arrow press
    if keyboard.left:
        new_angle += 2

    # rotate right on right arrow press
    if keyboard.right:
        new_angle -= 2

    # accelerate forwards on up arrow press
    # and change displayed image
    if keyboard.up:
        spaceship.image = 'spaceship_thrust'
        spaceship.x_speed += math.cos(math.radians(new_angle)) * spaceship.ACCELERATION
        spaceship.y_speed += math.sin(math.radians(new_angle)) * spaceship.ACCELERATION
    else:
        spaceship.image = 'spaceship'

    # set the new angle
    spaceship.angle = new_angle
    # use the x and y speed to update the spaceship position
    # subtract the y speed as coordinates go from top to bottom
    spaceship.x += spaceship.x_speed
    spaceship.y -= spaceship.y_speed

def draw():
    screen.clear()
    spaceship.draw()
```



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag4](https://wfmag.cc/wfmag4)



# Subscribe today



## 3 ISSUES FOR £5

[wfmag.cc/3for5](http://wfmag.cc/3for5)

Subscribe  
for 12 months  
**FROM JUST £45**



### Subscriber benefits

- **Free delivery**  
Get it fast and for free
- **Exclusive offers**  
Take advantage of our offers and discounts
- **Great savings**  
Save money on the cover price compared to stores

### Special offer:

**3 issues for £5**

- **Low initial cost**
- **Free delivery**

3 for £5 offer only available in UK

### Subscribe for 6 months

<b>£25</b> (UK)	<b>£30</b> (USA)
<b>£35</b> (EU)	<b>£40</b> (RoW)

### Subscribe for 12 months

**wfmag.cc/subscribe**

<b>£45</b> (UK)	<b>£55</b> (USA)
<b>£70</b> (EU)	<b>£80</b> (RoW)

Offers and prices are subject to change at any time



Digital subscriptions from

**£1.99**



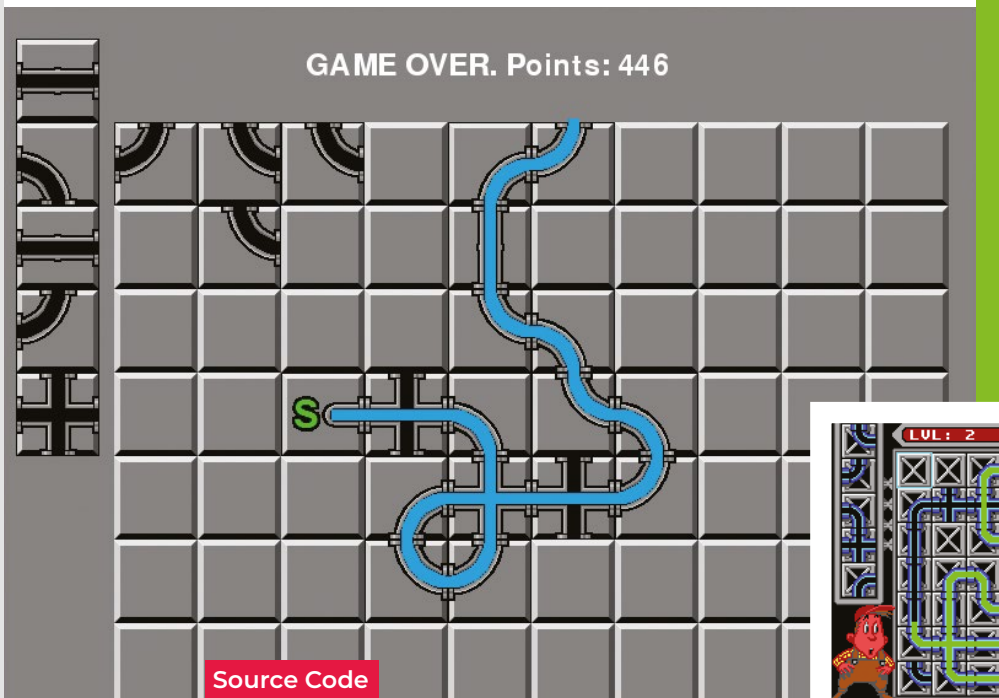
Available on the  
**App Store**



GET IT ON  
**Google Play**

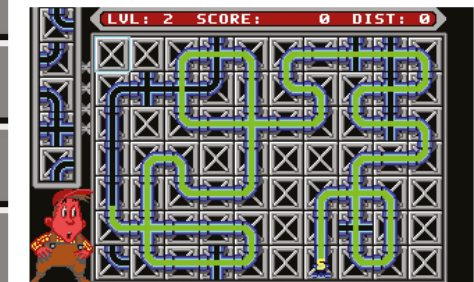
Visit **wfmag.cc/subscribe** or call **01293 312192** to order

Subscription queries: **wireframe@subscriptionhelpline.co.uk**



◀ Our *Pipe Mania* homage. Build a pipeline before the water escapes, and see if you can beat your own score.

▼ *Pipe Mania*'s design is so effective, it's appeared in various guises elsewhere – even as a minigame in *BioShock*.



## Code your own Pipe Mania puzzler



AUTHOR  
JORDI SANTONJA

Create a network of pipes before the water starts to flow in our re-creation of a classic puzzler

**P**ipe Mania, also called *Pipe Dream* in the US, is a puzzle game developed by The Assembly Line in 1989 for Amiga, Atari ST, and PC, and later ported to other platforms, including arcades. The player must place randomly generated sections of pipe onto a grid. When a counter reaches zero, water starts to flow and must reach the longest possible distance through the connected pipes.

Let's look at how to recreate *Pipe Dream* in Python and Pygame Zero. The variable **start** is decremented at each frame. It begins with a value of **60\*30**, so it reaches zero after 30 seconds if our monitor runs at 60 frames per second. In that time, the player can place tiles on the grid to build a path. Every time the user clicks on the grid, the last tile from **nextTiles** is placed on the play area and a new random tile appears at the top of the next tiles. **randint(2,8)** computes a random value between 2 and 8. **grid** and **nextTiles** are lists of tile values,

from 0 to 8, and are copied to the screen in the **draw** function with the **screen.blit** operation. **grid** is a two-dimensional list, with sizes **gridWidth=10** and **gridHeight=7**. Every pipe piece is placed in **grid** with a mouse click. This is managed with the Pygame functions **on\_mouse\_move** and **on\_mouse\_down**, where the variable **pos** contains the mouse position in the window. **panelPosition** defines the position of the top-left corner of the grid in the window. To get the grid cell, **panelPosition** is subtracted from **pos**, and the result is divided by **tileSize** with the integer division **//**. **tileMouse** stores the resulting cell element, but it is set to **(-1,-1)** when the mouse lies outside the grid.

The **images** folder contains the PNGs with the tile images, two for every tile: the graphical image and the path image. The **tiles** list contains the name of every tile, and adding to it **\_block** or **\_path** obtains the name of the file. The values stored in **nextTiles** and **grid** are the indexes of the elements in **tiles**.

The image **waterPath** isn't shown to the user, but it stores the paths that the water is going to follow. The first point of the water path is located in the starting tile, and it's stored in **currentPoint**. **update** calls the function **CheckNextPointDeleteCurrent**, when the water starts flowing. That function finds the next point in the water path, erases it, and adds a new point to the **waterFlow** list. **waterFlow** is shown to the user in the **draw** function.

**pointsToCheck** contains a list of relative positions, offsets, that define a step of two pixels from **currentPoint** in every direction to find the next point. Why two pixels? To be able to define the 'cross' tile, where two lines cross each other. In a 'cross' tile the water flow must follow a straight line, and this is how the only points found are the next points in the same direction. When no next point is found, the game ends and the score is shown: the number of points in the water path, **playState** is set to **0**, and no more updates are done. 🕒



# Pipe-wrangling in Python

Here's Jordi's code for a *Pipe Mania*-style puzzler. To get it working on your system, you'll need to install Pygame Zero – full instructions are available at [wfmag.cc/pgzero](http://wfmag.cc/pgzero).

```
# Pipe Mania
from pygame import image, Color, Surface
from random import randint

gridWidth, gridHeight = 10, 7
grid = [[0 for x in range(gridWidth)] for y in range(gridHeight)]
tileSize = 68
panelPosition = (96, 96)
numberNextTiles = 5
nextTiles = [randint(2, 8) for y in range(numberNextTiles)]
nextTilesPosition = (16, 28)
tileMouse = (-1, -1)

tiles = ['empty', 'start',
         'hori', 'vert', 'cross',
         'bottomleft', 'bottomright',
         'topleft', 'topright']

pathTiles = [image.load('images/' + tiles[i] + '_path.png') for i in
              range(1,9)]

waterPath = Surface((gridWidth*tileSize, gridHeight*tileSize))
waterPath.fill(Color('black'))
grid[3][2] = 1 # start tile
waterPath.blit(pathTiles[0], (2 * tileSize, 3 * tileSize))
currentPoint = (2 * tileSize + 43, 3 * tileSize + 34)
waterFlow = []
start = 60*30 # 30 seconds

playState = 1

pointsToCheck = [(2, 0), (0, 2), (-2, 0), (0, -2),
                 (2, 1), (1, 2), (-2, 1), (1, -2),
                 (2, -1), (-1, 2), (-2, -1), (-1, -2),
                 (2, -2), (2, 2), (-2, 2), (-2, -2)]

def draw():
    screen.blit('background', (0,0))
    for x in range(gridWidth):
        for y in range(gridHeight):
            screen.blit(tiles[grid[y][x]] + '_block', (
                panelPosition[0] + x * tileSize,
                panelPosition[1] + y * tileSize))
    for y in range(numberNextTiles):
        screen.blit(tiles[nextTiles[y]] + '_block', (
            nextTilesPosition[0],
            nextTilesPosition[1] + y * tileSize))
    for point in waterFlow:
        screen.blit('water', point)
    if playState == 1:
        if tileMouse[0] >= 0 and tileMouse[1] >= 0:
            screen.blit(tiles[nextTiles[-1]] + '_block', (
                panelPosition[0] + tileMouse[0] * tileSize,
                panelPosition[1] + tileMouse[1] * tileSize))
```

```
if start > 0:
    screen.draw.text("Start in "
+ str(start // 60), center=(400, 50), fontsize=35)
else:
    screen.draw.text("GAME OVER. Points: "
+ str(len(waterFlow)), center=(400, 50), fontsize=35)

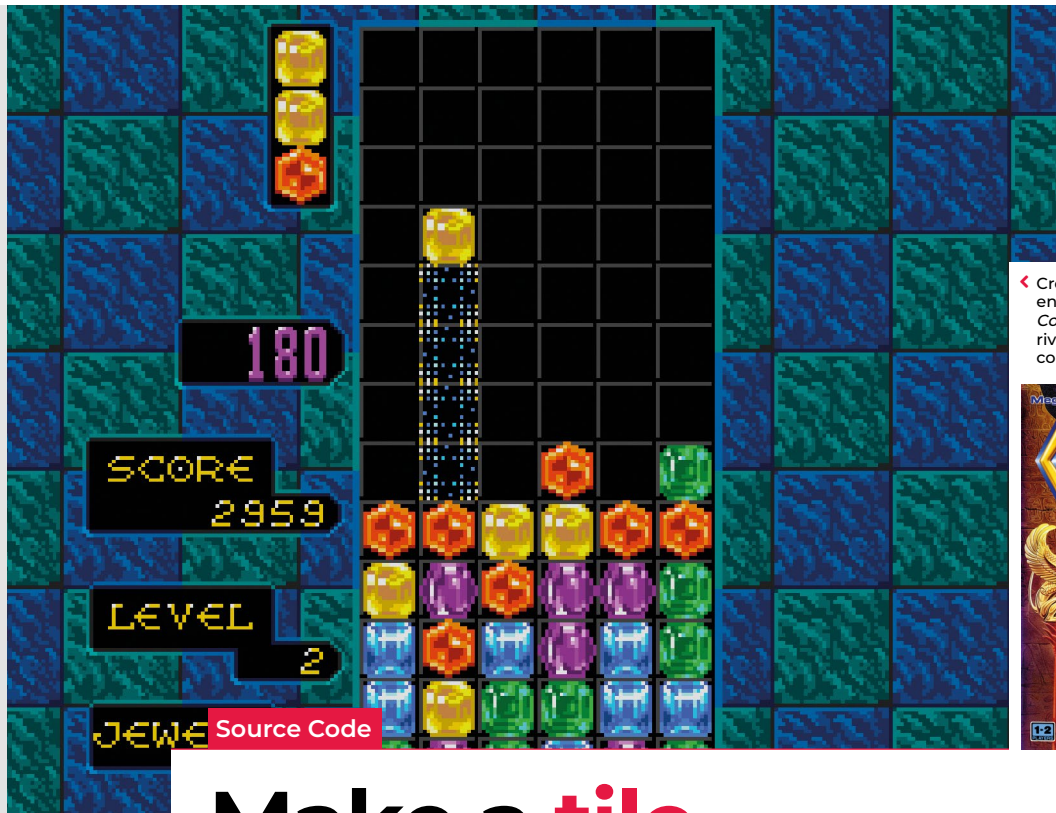
def update():
    global start, playState
    if start > 0:
        start -= 1
    elif playState == 1:
        if not CheckNextPointDeleteCurrent():
            playState = 0

def CheckNextPointDeleteCurrent():
    global currentPoint
    for point in pointsToCheck:
        newPoint = (currentPoint[0] + point[0], currentPoint[1]
+ point[1])
        if newPoint[0] < 0 or newPoint[1] < 0 or newPoint[0] >=
gridWidth*tileSize or newPoint[1] >= gridHeight*tileSize:
            return False # goes outside the screen
        if waterPath.get_at(newPoint) != Color('black'):
            waterPath.set_at(newPoint, Color('black'))
            waterFlow.append((newPoint[0] + panelPosition[0] - 4,
newPoint[1] + panelPosition[1] - 4))
            currentPoint = newPoint
            return True
    return False # no next point found

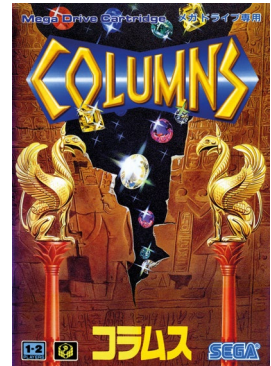
def on_mouse_down(pos):
    if playState == 1 and tileMouse[0] >= 0 and tileMouse[1] >=
0:
        if grid[tileMouse[1]][tileMouse[0]] != 1: # not start
tile
            grid[tileMouse[1]][tileMouse[0]] = nextTiles[-1]
            waterPath.fill(Color('black'), (tileMouse[0] *
tileSize, tileMouse[1] * tileSize, tileSize, tileSize))
            waterPath.blit(pathTiles[nextTiles[-1] - 1],
(tileMouse[0] * tileSize, tileMouse[1] * tileSize))
            for i in reversed(range(numberNextTiles - 1)):
                nextTiles[i + 1] = nextTiles[i]
                nextTiles[0] = randint(2, 8)

def on_mouse_move(pos):
    global tileMouse
    if playState == 1:
        tileMouse = ((pos[0] - panelPosition[0])//tileSize,
(pos[1] - panelPosition[1])//tileSize)
        if pos[0] < panelPosition[0] or pos[1] < panelPosition[1]
or tileMouse[0] >= gridWidth or tileMouse[1] >= gridHeight:
            tileMouse = (-1, -1) # mouse outside panel
```





◀ Created by Hewlett-Packard engineer Jay Geertsens, *Columns* was Sega's sparkly rival to Nintendo's all-conquering *Tetris*.



# Make a tile-matching game



AUTHOR  
RIK CROSS

Rik shows you how to code your own Columns-style tile-matching puzzler

**T**ile-matching games began with *Tetris* in 1984 and the less famous *Chain Shot!* the following year. The genre gradually evolved through games like *Dr. Mario*, *Columns*, *Puyo Puyo*, and *Candy Crush Saga*. Although their mechanics differ, the goals are the same: to organise a board of different-coloured tiles by moving them around until they match.

Here, I'll show how you can create a simple tile-matching game using Python and Pygame. In it, any tile can be swapped with the tile to its right, with the aim being to make matches of three or more tiles of the same colour. Making a match causes the tiles to disappear from the board, with tiles dropping down to fill in the gaps.

At the start of a new game, a board of randomly generated tiles is created. This is made as an (initially empty) two-dimensional array, whose size is determined by the values of **rows**

and **columns**. A specific tile on the board is referenced by its row and column number.

We want to start with a truly random board, but we also want to avoid having any matching tiles. Random tiles are added to each board position, therefore, but replaced if a tile is the same as the one above or to its left (if such a tile exists).

In our game, two tiles are 'selected' at any one time, with the player pressing the arrow keys to change those tiles. A **selected** variable keeps track of the row and column of the left-most selected tile, with the other tile being one column to the right of the left-most tile. Pressing **SPACE** swaps the two selected tiles, checks for matches, clears any matched tiles, and fills any gaps with new tiles.

A basic 'match-three' algorithm would simply check whether any tiles on the board have a matching colour tile on either side, horizontally or vertically. I've opted for something a little more convoluted, though,

as it allows us to check for matches on any length, as well as track multiple, separate matches. A **currentmatch** list keeps track of the (x,y) positions of a set of matching tiles. Whenever this list is empty, the next tile to check is added to the list, and this process is repeated until the next tile is a different colour. If the **currentmatch** list contains three or more tiles at this point, then the list is added to the overall **matches** list (a list of lists of matches!) and the **currentmatch** list is reset. To clear matched tiles, the matched tile positions are set to **None**, which indicates the absence of a tile at that position. To fill the board, tiles in each column are moved down by one row whenever an empty board position is found, with a new tile being added to the top row of the board.

The code provided here is just a starting point, and there are lots of ways to develop the game, including a scoring system and animation to liven up your tiles. 🐍





Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag25](https://wfmag.cc/wfmag25)

# Match-three in Python

Here's Rik's code snippet, which creates a simple match-three game in Python. To get it running on your system, you'll first need to install Pygame Zero – you can find full instructions at [wfmag.cc/pgzero](https://wfmag.cc/pgzero)

```
from random import randint

WHITE = 255,255,255

boardx = 40
boardy = 40
tilesize = 40
columns = 8
rows = 12
numberoftiles = 9

WIDTH = (boardx * 2) + (tilesize * columns)
HEIGHT = (boardy * 2) + (tilesize * rows)

tiles = [[1] * columns for j in range(rows)]
for r in range(rows):
    for c in range(columns):
        tiles[r][c] = randint(1, numberoftiles-1)
        while (r>0 and tiles[r][c] == tiles[r - 1][c]) or (c > 0
and tiles[r][c] == tiles[r][c - 1]):
            tiles[r][c] = randint(1, numberoftiles - 1)

selected = [0,0]

def checkmatches():
    matches = []
    for c in range(columns):
        currentmatch = []
        for r in range(rows):

            if currentmatch == [] or tiles[r][c] == tiles[r - 1][c]:
                currentmatch.append((r,c))
            else:
                if len(currentmatch) >= 3:
                    matches.append(currentmatch)
                    currentmatch = [(r,c)]
                if len(currentmatch) >= 3:
                    matches.append(currentmatch)
    for r in range(rows):
        currentmatch = []
        for c in range(columns):
            if currentmatch == [] or tiles[r][c] == tiles[r][c - 1]:
                currentmatch.append((r,c))
            else:
                if len(currentmatch) >= 3:
                    matches.append(currentmatch)
                    currentmatch = [(r,c)]
                if len(currentmatch) >= 3:
                    matches.append(currentmatch)

    return matches
```

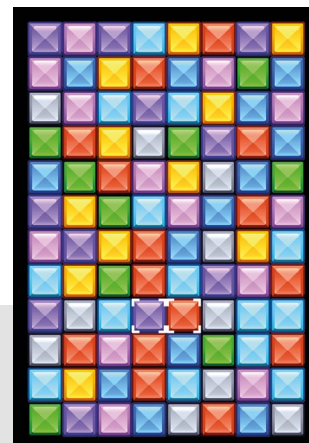
```
def clearmatches(matches):
    for match in matches:
        for position in match:
            tiles[position[0]][position[1]] = None

def fillboard():
    for c in range(columns):
        for r in range(rows):
            if tiles[r][c] == None:
                for rr in range(r,0,-1):
                    tiles[rr][c] = tiles[rr - 1][c]
                tiles[0][c] = randint(1, numberoftiles - 1)
                while tiles[0][c] == tiles[1][c] or (c > 0 and
tiles[0][c] == tiles[0][c-1]) or (c<columns-1 and tiles[0][c]
== tiles[0][c+1]):
                    tiles[0][c] = randint(1, numberoftiles - 1)

def on_key_up(key):
    if key == keys.LEFT:
        selected[0] = max(0,selected[0] - 1)
    if key == keys.RIGHT:
        selected[0] = min(selected[0] + 1,columns - 2)
    if key == keys.UP:
        selected[1] = max(0,selected[1] - 1)
    if key == keys.DOWN:
        selected[1] = min(selected[1] + 1,rows - 1)
    if key == keys.SPACE:
        tiles[selected[1]][selected[0]], tiles[selected[1]]
[selected[0] + 1] = tiles[selected[1]][selected[0] + 1],
tiles[selected[1]][selected[0]]
        matches = checkmatches()
        clearmatches(matches)
        fillboard()

def draw():
    screen.clear()
    for r in range(rows):
        for c in range(columns):
            screen.
            blit(str(tiles[r][c]),
(boardx + (c * tilesize),
boardy + (r * tilesize)))
            screen.
            blit('selected',(boardx+
(selected[0] * tilesize),
boardy + (selected[1] *
tilesize)))
```

➤ A board consisting of 12 rows and 8 columns of tiles. Pressing **SPACE** will swap the 2 selected tiles (outlined in white), and in this case, create a match of red tiles vertically.





◀ The original *Lemmings*, first released for the Amiga, quickly spread like a virus to just about every computer and console of the day.



Source Code

# Path-following Lemmings

Learn how to create your own obedient lemmings that follow any path put in front of them



AUTHOR  
RIK CROSS

**L**emmings is a puzzle-platformer, created at DMA Design, and first became available for the Amiga in 1991. The aim is to guide a number of small lemming sprites to safety, navigating traps and difficult terrain along the way. Left to their own devices, the lemmings will simply follow the path in front of them, but additional 'special powers' given to lemmings allow them to (among other things) dig, climb, build, and block in order to create a path to freedom (or to the next level, anyway).

I'll show you a simple way (using Python and Pygame) in which lemmings can be made to follow the terrain in front of them. The first step is to store the level's terrain information, which I've achieved by using a two-dimensional list to store the colour of each pixel in the background 'level' image. In my example, I've used the 'Lemcraft' tileset by Matt Hackett (of Lost Decade Games) – taken from [opengameart.org](http://opengameart.org) – and used the

'Tiled' software ([mapeditor.org](http://mapeditor.org)) to stitch the tiles together into a level.

The algorithm we then use can be summarised as follows: check the pixels immediately below a lemming. If the colour of those pixels isn't the same as the background colour, then the lemming is falling.

**"Left to their own devices, the lemmings will follow the path in front of them"**

In this case, move the lemming down by one pixel on the y-axis. If the lemming isn't falling, then it's walking. In this case, we need to see whether there is a non-ground, background-coloured pixel in front of the lemming for it to move onto. If a pixel is found in front of the lemming (determined by its direction) that is low enough to get to (i.e. lower than its `climbheight`), then the lemming moves forward on the x-axis by one pixel, and upwards on the y-axis to the new ground level. However, if no

suitable ground is found to move onto, then the lemming reverses its direction.

The above algorithm is stored as a lemming's `update()` method, which is executed for each lemming, each frame of the game. The sample `level.png` file can be edited, or swapped for another image altogether. If using a different image, just remember to update the level's `BACKGROUND_COLOUR` in your code, stored as a (red, green, blue, alpha) tuple. You may also need to increase your lemming `climbheight` if you want them to be able to navigate a climb of more than four pixels.

There are other things you can do to make a full *Lemmings* clone. You could try replacing the yellow-rectangle lemmings in my example with pixel-art sprites with their own walk cycle animation (see my article in issue #14) or give your lemmings some of the special powers they'll need to get to safety, achieved by creating flags that determine how lemmings interact with the terrain around them. ☺



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag17](https://wfmag.cc/wfmag17)

# Path-following critters in Python

Here's a code snippet that will send path-following creatures roaming around your screen. To get it running, you'll first need to install Pygame Zero – you can find full instructions at [wfmag.cc/pgzero](https://wfmag.cc/pgzero)

```
from time import sleep
from PIL import Image

# screen size
HEIGHT=800
WIDTH=800

# level information
level_image = 'level'
BACKGROUND_COLOUR = (114,114,201,255)

# store the colour of each pixel in the 'level' image
img = Image.open('images/level.png')
pixels = [[img.getpixel((x, y)) for y in range(HEIGHT)] for x
in range(WIDTH)]

# a list to keep track of the lemmings
lemmings = []
max_lemmings = 10
start_position = (100,100)
# a timer and interval for creating new lemmings
timer = 0
interval = 10

# returns 'True' if the pixel specified is 'ground'
# (i.e. anything except BACKGROUND_COLOUR)
def groundatposition(pos):
    # ensure position contains integer values
    pos = (int(pos[0]),int(pos[1]))
    # get the colour from the 'pixels' list
    if pixels[pos[0]][pos[1]] != BACKGROUND_COLOUR:
        return True
    else:
        return False

class Lemming(Actor):
    def __init__(self, **kwargs):
    super().__init__(image='lemming', pos=start_position,
    anchor=('left','top'), **kwargs)
    self.direction = 1
    self.climbheight = 4
    self.width = 10
    self.height = 20

    # update a lemming's position in the level
    def update(self):
        # if there's no ground below a lemming (check both
        corners), it is falling
        bottomleft = groundatposition((self.pos[0],self.
        pos[1]+self.height))
        bottomright = groundatposition((self.pos[0]+(self.
        width-1), self.pos[1]+self.height))
        if not bottomleft and not bottomright:
```

```
        self.y += 1
        # if not falling, a lemming is walking
        else:
            height = 0
            found = False
            # find the height of the ground in front of a
            # lemming up to the maximum height a lemming
            # can climb
            while (found == False) and (height <= self.
            climbheight):
                # the pixel 'in front' of a lemming
                # will depend on the direction it's
                # traveling
                if self.direction == 1:
                    positioninfront = (self.pos[0]+self.width,
                    self.pos[1]+(self.height-1)-height)
                else:
                    positioninfront = (self.pos[0]-1, self.
                    pos[1]+(self.height-1)-height)
                if not groundatposition(positioninfront):
                    self.x += self.direction
                    # rise up to new ground level
                    self.y -= height
                    found = True

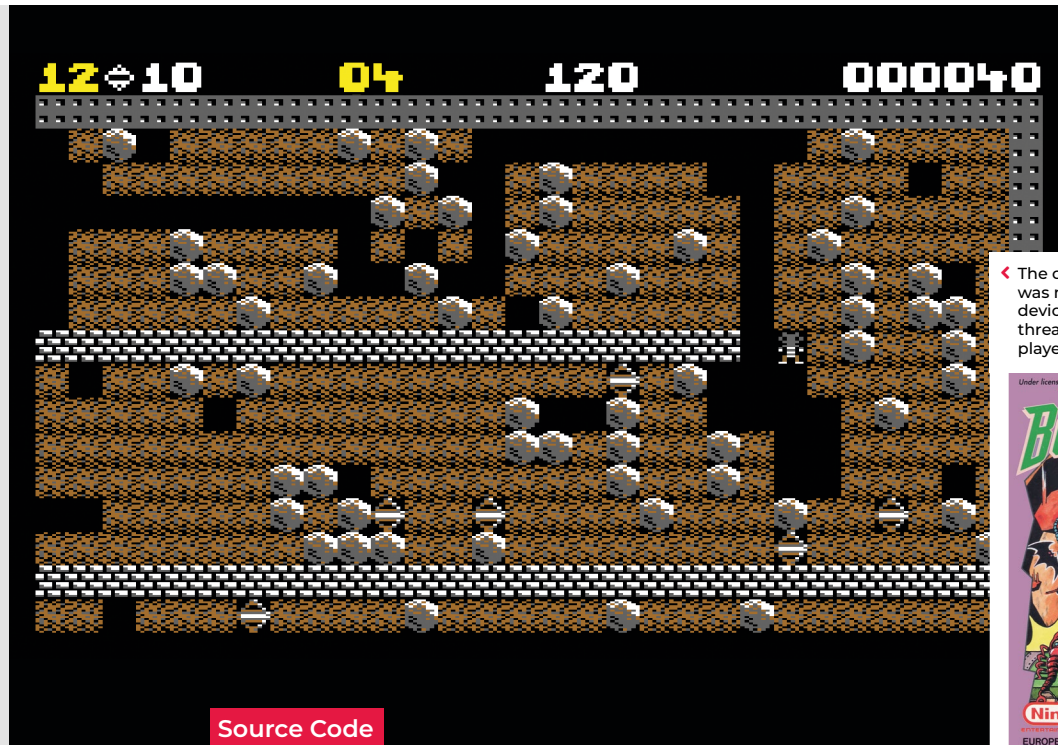
                height += 1
            # turn the lemming around if the ground in front
            # is too high to climb
            if not found:
                self.direction *= -1

    def update():
        global timer
        # increment the timer and create a new
        # lemming if the interval has passed
        timer += 0.1
        if timer > interval and len(lemmings) < max_lemmings:
            timer = 0
            lemmings.append(Lemming())
        # update each lemming's
        position in the level
        for i in lemmings:
            i.update()

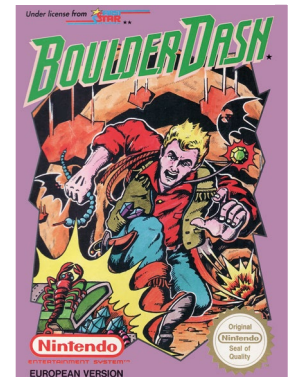
    def draw():
        screen.clear()
        # draw the level
        screen.blit(level_image,(0,0))
        # draw lemmings
        for i in lemmings:
            i.draw()
```

✓ Sprites cling to the ground below them, navigating uneven terrain, and reversing direction when they hit an impassable obstacle.





< The original *Boulder Dash* was marked out by some devious level design, which threatened to squash the player at every turn.



Source Code

# Code a Boulder Dash mining game



AUTHOR  
MARK VANSTONE

Dig through the caves to find gems – but watch out for falling boulders

**B**oulder Dash first appeared in 1984 for the Commodore 64, Apple II, and the Atari 400/800. It featured an energetic gem collector called Rockford who, thanks to some rather low-resolution graphics, looked a bit like an alien. His mission was to tunnel his way through a series of caves to find gems while avoiding falling rocks dislodged by his digging. Deadly creatures also inhabited the caves which, if destroyed by dropping rocks on them, turned into gems for Rockford to collect.

The ingenious level designs were what made *Boulder Dash* so addictive. Gems had to be collected within a time limit to unlock the exit, but some were positioned in places that would need planning to get to, often using the physics of falling boulders to block or clear areas. Of course, the puzzles got increasingly tough as the levels progressed.

Written by Peter Liepa and Chris Gray,

*Boulder Dash* was published by First Star Software, which still puts out new versions of the game to this day. Due to its original success, *Boulder Dash* was ported to all kinds of platforms, and the years since have seen no fewer than 20 new iterations of *Boulder Dash*, and a fair few clones, too.

We're going to have a look at the boulder physics aspect of the game, and make a simple level where Rockford can dig out some gems and hopefully not get flattened under an avalanche of rocks. Writing our code in Pygame Zero, we'll automatically create an 800 by 600-size window to work with. We can make our game screen by defining a two-dimensional list, which, in this case, we will fill with soil squares and randomly position the rocks and gems. Each location in the list matrix will have a name: either **wall** for the outside boundary, **soil** for the diggable stuff, **rock** for a round, moveable boulder, **gem** for a collectable item, and finally, **rockford** to symbolise our hero.

We can also define an Actor for Rockford, as this will make things like switching images and tracking other properties easier.

Our **draw()** function is just a nested loop to iterate through the list matrix and blit to the screen whatever is indicated in each square. The Rockford Actor is then drawn over the top. We can also keep a count of how many gems have been collected and provide a congratulatory message if all of them are found. In the **update()** function, there are only two things we really need to worry about: the first being to check for keypresses from the player and move Rockford accordingly, and the second to check rocks to see if they need to move.

Rockford is quite easy to test for movement, as he can only move onto an empty square – a soil square or a gem square. It's also possible for him to push a boulder if there's an empty space on the other side. For the boulders, we need to first test if there's an empty space below it, and





Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag30](https://wfmag.cc/wfmag30)

# Tumbling rocks in Python

Here's Mark's code snippet, which creates some falling *Boulder Dash* rocks – and an intrepid explorer – in Python. To get it running on your system, you'll need to install Pygame Zero – you can find full instructions at [wfmag.cc/pgzero](https://wfmag.cc/pgzero)

```
import random

rockford = Actor('rockford-1', center=(60, 100))
gameState = count = 0
items = [[] for _ in range(14)]
gems = collected = 0
for r in range(0, 14):
    for c in range(0, 20):
        itype = "soil"
        if(r == 0 or r == 13 or c == 0 or c == 19): itype = "wall"
        elif random.randint(0, 4) == 1: itype = "rock"
        elif random.randint(0, 20) == 1:
            itype = "gem"
            gems += 1
        items[r].append(itype)
items[1][1] = "rockford"

def draw():
    screen.fill((0,0,0))
    if gems == collected: infoText("YOU COLLECTED ALL THE GEMS!")
    else: infoText("GEMS : "+ str(collected))
    for r in range(0, 14):
        for c in range(0, 20):
            if items[r][c] != "" and items[r][c] != "rockford":
                screen.blit(items[r][c], ((c*40), 40+(r*40)))
    if gameState == 0 or (gameState == 1 and count%4 == 0):
        rockford.draw()

def update():
    global count
    mx = my = 0
    if count%10 == 0:
        for r in range(13, -1, -1):
            for c in range(19, -1, -1):
                if items[r][c] == "rockford":
                    if keyboard.left: mx = -1
                    if keyboard.right: mx = 1

                    if keyboard.up: my = -1
                    if keyboard.down: my = 1
                    if items[r][c] == "rock": testRock(r,c)
                    rockford.image = "rockford"+str(mx)
                    if gameState == 0: moveRockford(mx,my)
                    count += 1

def infoText(t):
    screen.draw.text(t, center = (400, 20), owidth=0.5,
        ocolor=(255,255,255), color=(255,0,255), fontsize=40)

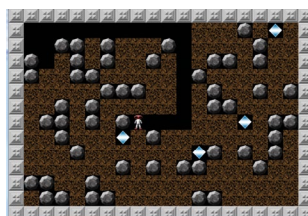
def moveRockford(x,y):
    global collected
    rx, ry = int((rockford.x-20)/40), int((rockford.y-40)/40)
    if items[ry+y][rx+x] != "rock" and items[ry+y][rx+x] != "wall":
        if items[ry+y][rx+x] == "gem": collected +=1
        items[ry][rx], items[ry+y][rx+x] = "", "rockford"
        rockford.pos = (rockford.x + (x*40), rockford.y + (y*40))
        if items[ry+y][rx+x] == "rock" and y == 0:
            if items[ry][rx+(x*2)] == "":
                items[ry][rx], items[ry][rx+(x*2)], items[ry+y][rx+x] = "", "rock", "rockford"
                rockford.x += x*40

def testRock(r,c):
    if items[r+1][c] == "":
        moveRock(r,c,r+1,c)
    elif items[r+1][c] == "rock" and items[r+1][c-1] == "" and items[r][c-1] == "":
        moveRock(r,c,r+1,c-1)
    elif items[r+1][c] == "rock" and items[r+1][c+1] == "" and items[r][c+1] == "":
        moveRock(r,c,r+1,c+1)

def moveRock(r1,c1,r2,c2):
    global gameState
    items[r1][c1], items[r2][c2] = "", items[r1][c1]
    if items[r2+1][c2] == "rockford": gameState = 1
```

if so, the boulder must move downwards. We also test to see if a boulder is on top of another boulder – if it is, the top boulder can roll off and down onto a space either to the left or the right of the one beneath.

There's not much to add to this snippet of code to turn it into a playable game of *Boulder Dash*. See if you can add a timer, some monsters, and, of course, some puzzles for players to solve on each level. 🐹



▲ Our homage to *Boulder Dash* running in Pygame Zero. Dig through the caves to find gems – while avoiding death from above.

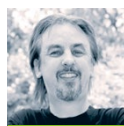
## BOTTOMS UP

An important thing to notice about the process of scanning through the list matrix to test for boulder movement is that we need to read the list from the bottom upwards; otherwise, because the boulders move downwards, we may end up testing a boulder multiple times if we test from the beginning to the end of the list. Similarly, if we read the list matrix from the top down, we may end up moving a boulder down and then when reading the next row, coming across the same one again, and moving it a second time.



# Source Code DX: make a Boulder Dash level editor in Python

In a Source Code special, Mark shows you how to create an entire Boulder Dash construction kit from scratch



**AUTHOR**  
**MARK VANSTONE**

Mark Vanstone is the technical director of TechnoVisual, author of the nineties educational game series, *ArcVenture*, and after all this time, still can't resist game coding. [education.technovisual.co.uk](http://education.technovisual.co.uk)



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag56](https://wfmag.cc/wfmag56)

# BOULDER DASH™

**B**oulder Dash was a popular video computer game in the mid-eighties, and in Wireframe issue 30 ([wfmag.cc/30](https://wfmag.cc/30)), we showed you how to code your own miniature remake.

This time, we'll expand on that program to make a level editor, which you can then use to design your own puzzles for other players to navigate.

Before you get started, be sure to have a look through that previous Source Code article to familiarise yourself with how the program works – you can get the code and assets for it from that issue's GitHub: [wfmag.cc/wfmag30](https://wfmag.cc/wfmag30). As a quick reminder of how the game works, it saw intrepid hero Rockford dig his way through underground caves to find gems, all the while avoiding the falling rocks. Those rocks not only fall downwards

if there's nothing to hold them up, but they'll also roll down onto other rocks if there's nothing to the left or right of them. Rockford's controlled with the cursor keys, and the aim is to collect all the gems to complete the level.

We'll continue writing our code in Pygame Zero based on the original program, but to incorporate an editor section, we change our window size to add an extra 200 pixels to the width. As before, our game screen is defined by a two-dimensional list which, in our original program, we filled with random items. For this version, we'll set the play area to a 'default' layout of all soil blocks with wall blocks around the outside to stop Rockford from going off the screen. Each location in the list matrix has a name: either **wall** for the outside boundary, **soil** for the diggable stuff, **rock** for a round, moveable boulder, **gem** for a collectable jewel, and finally **rockford** to denote our hero. Rockford is also defined as an Actor, as this makes things like switching images and tracking other properties easier.

The first thing to do to our program is to add a switch to turn the editor on or off. To do this, we'll define a variable called **editorState**. If this variable's set to True, we open the program with an extra 200 pixels on the right-hand side of the play area using **WIDTH = 1000**. When this area is



► The original *Boulder Dash* had its own level editor called the Construction Kit.



◀ When we first load the program, we fill the game area with just soil and a wall border.

shown, we'll need to draw all the elements we need to see in the editor. First, to keep things tidy, we make a function called `drawEditor()` which will be called from the `draw()` function. We probably want to display a title for the area – something like 'EDITOR' will do the job – and then underneath, print 'ON' if the `editorState` variable is True. We can toggle the `editorState` variable with a key press such as the `SPACE` bar. We do this by defining the `on_key_down(key)` function and then capturing the key press value. If the `SPACE` bar is pressed, we set the `editorState` to the opposite of what it is. The code below shows you how we do this.

**“We need to be able to change the blocks that are in the play area”**

```
def on_key_down(key):
    global editorState
    if key == keys.SPACE and WIDTH > 800:
        editorState = not editorState
```

To write an editor for the game, we need to be able to change the blocks that are in the play area. So here's the plan: we'll create a visual list of the available blocks which are clickable. When the block's clicked on, it becomes the currently selected item. Then if we click in the play area,

it will change the block to be the item we've selected. All we need to do to change what's in the play area is to change the item name in the items list and the display will update. We'll be using a small set of blocks for this example, but you can add new ones yourself. If you want to create new images for any new blocks, you'll

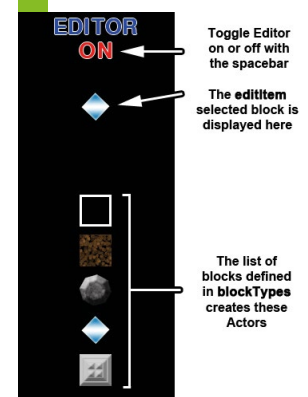
need an image editing program. If you don't have Photoshop, you can create new graphics with programs like GIMP or even Microsoft Paint.

There are also free online alternatives such as Photopea, which can export graphics as a PNG file – this is what you'll need for any new blocks. The block images are 40 pixels by 40 pixels in size and should be saved in the images directory alongside your program file.

## BLOCK FALL

To create our visual list of blocks, we can create a list of Actors which will mean we can test them for mouse clicks. We'll call the list `blockTypes` and define them at the start of the program. Notice how we can define a screen position for them so we can have them neatly arranged in a line. If we had more blocks then we could arrange them in a grid format. You can add your own ➔

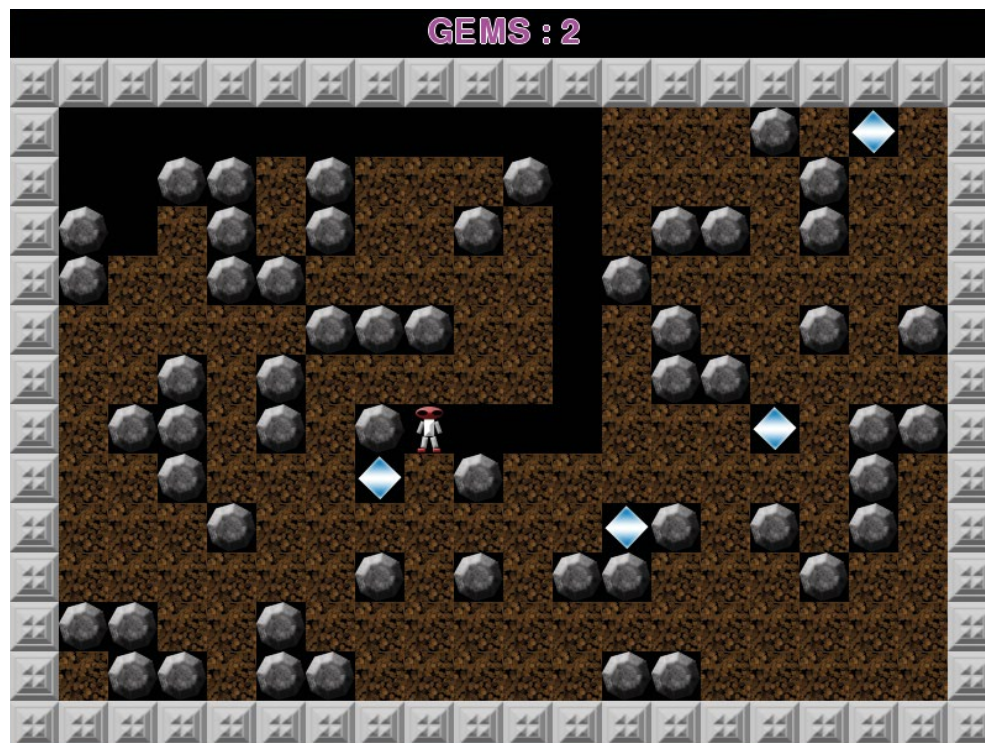
▼ The editor panel. This could start to look a lot busier once you start adding your own block types.



► Our earlier Source Code Boulder Dash game from issue 30 had randomly generated blocks.

## EARTH SHAKING

If you owned a ZX Spectrum in the early 1990s, you may have encountered one of the best *Boulder Dash* clones ever made: *Earth Shaker*, programmed by Michael Batty and given away as a cassette on the cover of Your Sinclair magazine. With large scrolling levels and a complex array of additional block types – such as a Gravity Stick, which made rocks and other items float for a period – it was a slick and polished iteration of a familiar classic. A few months later, Your Sinclair published a level editor, which, like the one introduced here, allowed readers to design, save, and play their own maps in the original game. You had to type in reams of BASIC, though, but thanks to the wonders of the internet, you can now download it from World of Spectrum: [wfmag.cc/earthshaker](http://wfmag.cc/earthshaker).



blocks here if you want to expand the game, and you can define new blocks in the same way with their name being the file name of the image without .png at the end. To display our list on the screen, we'll need to add a `draw()` command for each item on the list. See the code below to learn how the list is defined and how our `drawEditor()` function is shaping up.

```
blockTypes = [
    Actor('blank', center=(900, 250)),
    Actor('soil', center=(900, 300)),
    Actor('rock', center=(900, 350)),
    Actor('gem', center=(900, 400)),
    Actor('wall', center=(900, 450))
]

def drawEditor():
    screen.draw.text("EDITOR", center = (900, 20),
        owidth=0.5, ocolor=(255,255,255), color=(0,0,255),
        , fontsize=40)
    if editorState: screen.draw.text("ON", center
        = (900, 50), owidth=0.5, ocolor=(255,255,255),
        color=(255,0,0) , fontsize=40)
    for b in range(0, len(blockTypes)):
        blockTypes[b].draw()
```

We should now see the editor panel (with the 'ON' indicator) and a column of blocks. Next, we

need to make them clickable. To do this, we'll need to define an `on_mouse_down(pos)` function. In this function, we'll check we're in the right `editorState` (True) and then check each of the `blockTypes` on the list with the `collidepoint(pos)` function to see if the mouse down event was over the block in the editor section. If it was, then we can set a variable to represent the currently selected item called `editItem`. This variable will be defined at the top of the program and set as the name of the block that was clicked. As things stand, we won't have any visual indicator of which block is currently selected, so we can remedy this by drawing a copy of the `editItem` block in the editor above the list with `screen.blit(editItem,(880,100))`.

We should now have an editor with a list of blocks which can be clicked to set the currently selected item, which is then displayed above the list. Once we've selected a block, we then want to be able to place it in the game area so it changes the map. To do this, we need to check the mouse click position to see if it's over the play area. Then we need to work out which square on the map has been clicked and change that item in the data to be our `editItem` value. Each of the blocks on the map are 40 pixels by 40 pixels, so we can find the position we need in the items list by dividing the mouse position by 40. However, the game area's displayed starting at 40 pixels down the



screen (to give room for information prompts), so we subtract 40 from the mouse y position before we do the division. The code below shows you how this calculation and testing for clicks on the blocks in the editor is written in the `on_mouse_down(pos)` function.

```
def on_mouse_down(pos):
    global editItem
    if editorState:
        c = int(pos[0]/40)
        r = int((pos[1]-40)/40)
        if r > 0 and r < 14 and c > 0 and c < 20:
            if editItem != "blank":
                items[r][c] = editItem
            else : items[r][c] = ""
        else:
            for b in range(0, len(blockTypes)):
                if blockTypes[b].collidepoint(pos):
                    editItem = blockTypes[b].image
```

## GRID LOCKED

If you've added some extra blocks to the list, you should be able to select and place them on the map at this point. If you want them to behave differently than other blocks in the game, though, you'll need to add some code. The code you write will depend what you want the blocks to do. For example, if you wanted to add a fire block which will sizzle Rockford if he walks over it, you'd need to put some code to test the block directly under Rockford to see if it's a fire block; if it is, set `gameState` to 1. You'd need to put that code in the `moveRockford(x,y)` function. If you want to add extra blocks, have a look at the full listing at the end of this tutorial to see how different types of blocks are dealt with in that function.

So now we should be able to go from a default play area with just soil, border wall, and Rockford to generating a set of boulders and walls with gems for the player to collect without getting squished.

There are many ways to arrange the boulders and walls to make it difficult for the player to get the gems without clearing the soil or moving boulders in the right order. (Of course, you'll need to make sure that it's actually possible to collect the gems.)

Once you've laid out some blocks on the game area, you can test the level by hitting the **SPACE**

**"If you want to expand  
the game, you can  
define new blocks"**



^ Repton: a lot like Boulder Dash, but set in a posh boarding school. (We may have made that last bit up.)

bar to switch the editor mode off and then start moving Rockford around the play area to see how the boulders react. The only problem with this situation at the moment is that as we move Rockford around, he's changing the map we have made. The rocks start moving, the gems get collected, and the only way to get the original map back is to make it again in the editor. What we need is a way of saving and loading maps. Let's make a couple of buttons to load and save maps, then. We can position these down at the

bottom of the editor.

They'll be Actors and respond to a mouse click like our blocks, but when clicked, we'll call functions `saveMap()` and

`loadMap()`. There are several ways we can save data from our program; if the data was more complicated, we might want to look at the JSON format to save our game maps, or we could use a comma-separated text format. In this case, though, a really effective way of saving this data is to use a library called Pickle. This provides data serialisation functions, which means translating structured data into and from a suitable file format. By opening a file and then calling `pickle.dump`, we can take a lot of the headache out of saving our maps. Conversely, when we want to load our map back in, we just open the file and call `pickle.load` and the data is read back into our items list. Have a look at the code overleaf to see the basics of our save and load functions. ➡

```
import pickle

def loadMap():
    global items
    with open('mymap.map', 'rb') as fp:
        items = pickle.load(fp)

def saveMap():
    with open('mymap.map', 'wb') as fp:

        pickle.dump(items, fp)
```

## LOAD AND SAVE

You'll see that we're using a fixed file name for saving the map. If you wanted to have the user change the file name, you might want to have a look at the `filedialog` part of the `tkinter` library to provide a load or save dialog window to enable choosing a file name for your map. For the purposes of this article though, we will stick with a fixed file name for our map.

Currently when our save function is called, there's no feedback to the user that anything has happened, which may be a bit disconcerting for some. It's probably wise to add some messages into our routines, then: we'll want to have a confirmation that the file has been saved or if there was a problem saving it. We can make a simple messaging system by having a global variable `editorMessage` and a countdown variable

`editorMessageCount` to display the message for a period of time and then stop displaying it. If we set the `editorMessage` variable to something like 'MAP SAVED' and the `editorMessageCount` variable to 200, then we can check to see if this variable is greater than zero in the `drawEditor()` function, then, if it is, display the message on the screen using `screen.draw.text()`. After displaying the text, we decrement the `editorMessageCount` variable by 1. This will mean that after 200 cycles of the `draw()` function, our message will disappear.

What if the saving operation failed? There could be all kinds of reasons why this might happen, and it's always a good policy to check when files are loaded or saved that the data transfer actually happened. To check that our file save didn't encounter an error, we can use a `try:` and then an `except IOError:` structure. Underneath the `try:` command we open our file, use `Pickle` to dump the data to the file, and then set our `editorMessage` to confirm the file's saved. Then we use the `except IOError:` command, and under that, we set our `editorMessage` to display an error message. This means that if an error occurs while saving, we'll see an error message; otherwise, we'll see the confirmation and know that our map file has been saved. Look at the code below to see the updated `saveMap()` function with error checking.

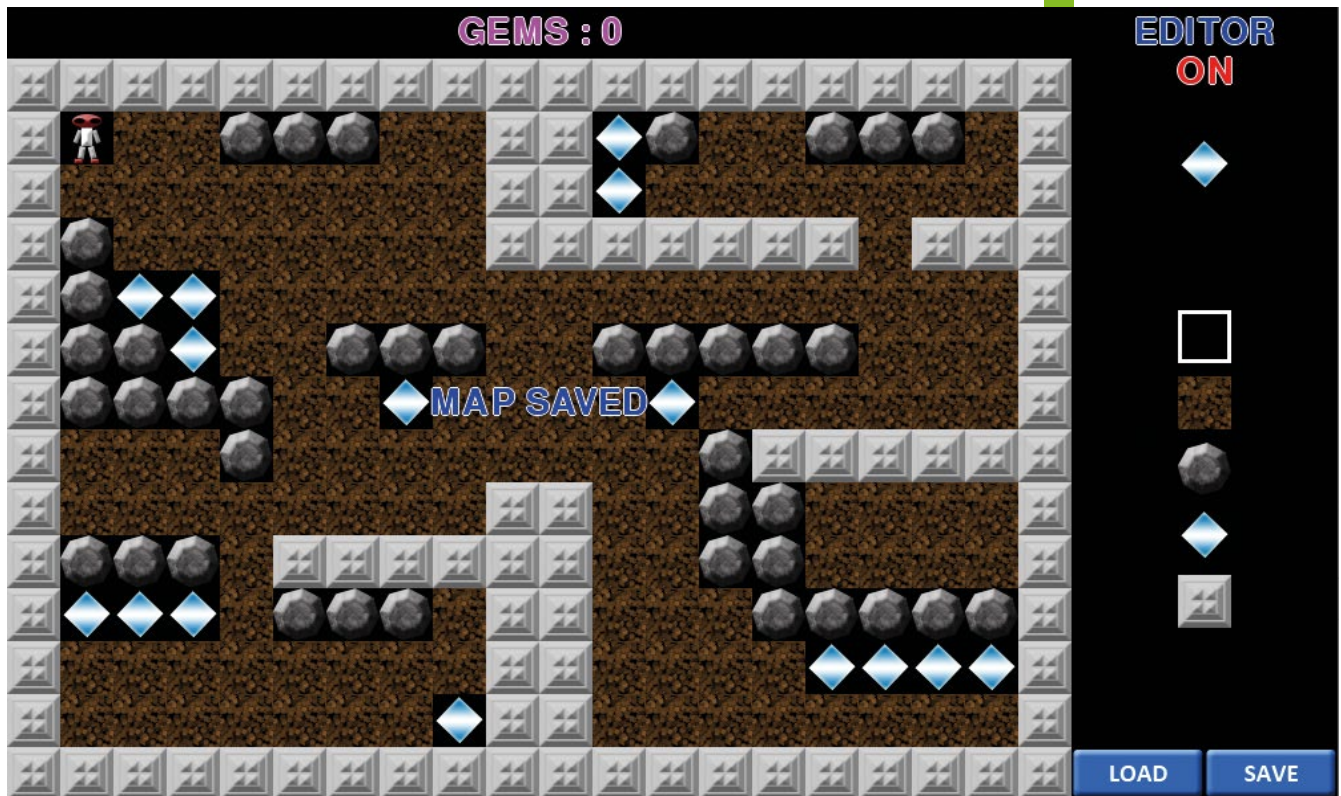
```
def saveMap():
    global editorMessage, editorMessageCount
    try:
        with open('mymap.map', 'wb') as fp:
            pickle.dump(items, fp)
        editorMessage = "MAP SAVED"
        editorMessageCount = 200
    except IOError:
        editorMessage = "ERROR SAVING MAP"
        editorMessageCount = 200
```

Our save function is now complete, so we'll turn our attention to how we load the map back in. We have the basics of the loading routine using the `pickle.load()` function, but what happens if we haven't created a map yet? The loading routine would fail and we wouldn't have any map data to work from. We can use the same technique we used with the `saveMap()` function to catch an error if it can't load the file. By using `try:` and `except IOError:` again, we can display a message to say the map has loaded if no error

♥ *Earth Shaker* was one of the best *Boulder Dash* clones on the ZX Spectrum.







^ The finished editor.  
See how devious you can  
make your own puzzles!

occurs, and if the map isn't loaded, fill our items list with the default map layout (just soil and boundary walls) and display a message to say that the default map has been loaded. Having put all this in place, we can then add a call to `loadMap()` when the program first runs. If we have an existing map file, the program will load it, and if not, it will load the default map. This means we don't need to generate the default items list at the beginning of the program as the `loadMap()` function will do it for us.

## FILE HANDLING

Now we have an editor that will automatically load the last map we saved or make a default map, allowing us to edit all the blocks in the game area, save the map, and then test it with Rockford. If we test our puzzle layout and find that Rockford gets squished, then at the moment all we can do is close the program and restart it to get back to the saved map. That's going to get very tedious if we're testing over and over. What we need is a reset key. We can check for the escape key in the `on_key_down()` function, and when that's detected, we need to set our `gems`, `collected`, and `gameState` variables all to zero, redefine the Rockford

Actor to be back in the top corner, and then call `loadMap()`. This will set everything back to the way it first loads in.

Our editor's nearly finished now, with just one more thing to do. When we've made and tested our fiendish map, we'll want to challenge our friends, family, or random passers-by to solve it. In other words, we want to let them play the game

without the editor section. All we need to do is change the `editorState` at the top of the program to `False` (this will mean the editor section will not be shown) and add a new variable, `editorEnabled` (also set to `False`), which we will check before letting the `SPACE` bar switch modes. The game is then playable by a non-editing user.

You now have a fully functional *Boulder Dash* editor! Have a look at the full listing to see how everything fits together. You could, of course, expand this to add more block items for Rockford to deal with, or enable multiple levels by loading in different maps as the player completes each one. You could add more tools for the editor, such as file load and save dialogs so you can choose the file names you use for your maps, but we'll leave you to have fun adding those extra features. ☺

**“Now we have an editor  
that will load the last  
map we saved”**

## Boulder Builder

Here's Mark's code for a full-featured *Boulder Dash* construction kit. To get it running on your system, you'll need to install Pygame Zero. Full instructions are available at [wfmag.cc/pgzero](http://wfmag.cc/pgzero).

```
# Boulder Dash Editor
import pgzrun
import pickle

editorState = True
editorEnabled = True

if editorState:
    WIDTH = 1000

gameState = count = 0
editItem = "blank"
editorMessage = ""
editorMessageCount = 0

blockTypes = [
    Actor('blank', center=(900, 250)),
    Actor('soil', center=(900, 300)),
    Actor('rock', center=(900, 350)),
    Actor('gem', center=(900, 400)),
    Actor('wall', center=(900, 450))
]

loadButton = Actor('load', center=(850, 580))
saveButton = Actor('save', center=(950, 580))
items = [[] for _ in range(14)]
gems = collected = 0
rockford = Actor('rockford-1', center=(60, 100))

def draw():
    screen.fill((0,0,0))
    if gems == 0 and collected > 0: infoText("YOU COLLECTED ALL THE GEMS!")
    else: infoText("GEMS : "+ str(collected))
    for r in range(0, 14):
        for c in range(0, 20):
            if items[r][c] != "" and items[r][c] != "rockford":
                screen.blit(items[r][c], ((c*40), 40+(r*40)))
    if gameState == 0 or (gameState == 1 and count%4 == 0):
        rockford.draw()
        drawEditor()

def update():
    global count, gems
    mx = my = 0
    if count%10 == 0:
        gems = 0
```

```
for r in range(13, -1, -1):
    for c in range(19, -1, -1):
        if items[r][c] == "gem":
            gems += 1
        if items[r][c] == "rockford":
            if keyboard.left: mx = -1
            if keyboard.right: mx = 1
            if keyboard.up: my = -1
            if keyboard.down: my = 1
        if items[r][c] == "rock": testRock(r,c)
        rockford.image = "rockford"+str(mx)
        if gameState == 0 and editorState == False:
            moveRockford(mx,my)
            count += 1

def on_mouse_down(pos):
    global editItem
    if editorState:
        c = int(pos[0]/40)
        r = int((pos[1]-40)/40)
        if loadButton.collidepoint(pos): loadMap()
        if saveButton.collidepoint(pos): saveMap()
        if r > 0 and r < 14 and c > 0 and c < 20:
            if editItem != "blank":
                items[r][c] = editItem
            else : items[r][c] = ""
        else:
            for b in range(0, len(blockTypes)):
                if blockTypes[b].collidepoint(pos):
                    editItem = blockTypes[b].image

def on_key_down(key):
    global editorState, gameState, rockford, collected, gems
    if key == keys.SPACE and editorEnabled:
        editorState = not editorState
    if key == keys.ESCAPE:
        gems = collected = gameState = 0
        rockford = Actor('rockford-1', center=(60, 100))
        loadMap()

def infoText(t):
    screen.draw.text(t, center = (400, 20), owidth=0.5,
        ocolor=(255,255,255), color=(255,0,255) , fontsize=40)

def moveRockford(x,y):
    global collected
```

```

    rx, ry = int((rockford.x-20)/40), int((rockford.y-40)/40)
    if items[ry+y][rx+x] != "rock" and items[ry+y][rx+x] !=
"wall":
        if items[ry+y][rx+x] == "gem": collected +=1
        items[ry][rx], items[ry+y][rx+x] = "", "rockford"
        rockford.pos = (rockford.x + (x*40), rockford.y + (y*40))
    if items[ry+y][rx+x] == "rock" and y == 0:
        if items[ry][rx+(x*2)] == "":
            items[ry][rx], items[ry][rx+(x*2)], items[ry+y][rx+x]
= "", "rock", "rockford"
            rockford.x += x*40

def testRock(r,c):
    if items[r+1][c] == "":
        moveRock(r,c,r+1,c)
    elif items[r+1][c] == "rock" and items[r+1][c-1] == "" and
items[r][c-1] == "":
        moveRock(r,c,r+1,c-1)
    elif items[r+1][c] == "rock" and items[r+1][c+1] == "" and
items[r][c+1] == "":
        moveRock(r,c,r+1,c+1)

def moveRock(r1,c1,r2,c2):
    global gameState
    items[r1][c1], items[r2][c2] = "", items[r1][c1]
    if items[r2+1][c2] == "rockford": gameState = 1

def drawEditor():
    global editorMessageCount
    screen.draw.text("EDITOR", center = (900, 20), owidth=0.5,
ocolor=(255,255,255), color=(0,0,255), fontsize=40)
    if editorState: screen.draw.text("ON", center = (900, 50),
ocolor=(255,255,255), color=(255,0,0), fontsize=40)
    for b in range(0, len(blockTypes)):
        blockTypes[b].draw()
    if editItem != "":
        screen.blit(editItem,(880,100))
    loadButton.draw()
    saveButton.draw()
    if editorMessageCount > 0:
        screen.draw.text(editorMessage, center = (400, 300),
ocolor=(255,255,255), color=(0,0,255), fontsize=40)
        editorMessageCount -= 1

def loadMap():
    global items, rockford, editorMessage, editorMessageCount
    try:

```

```

        with open ('mymap.map', 'rb') as fp:
            items = pickle.load(fp)
            editorMessage = "MAP LOADED"
            editorMessageCount = 200
        except IOError:
            editorMessage = "DEFAULT MAP LOADED"
            editorMessageCount = 200
        for r in range(0, 14):
            for c in range(0, 20):
                itype = "soil"
                if(r == 0 or r == 13 or c == 0 or c == 19): itype
= "wall"
                items[r].append(itype)
                items[1][1] = "rockford"

def saveMap():
    global editorMessage, editorMessageCount
    try:
        with open('mymap.map', 'wb') as fp:
            pickle.dump(items, fp)
            editorMessage = "MAP SAVED"
            editorMessageCount = 200
        except IOError:
            editorMessage = "ERROR SAVING MAP"
            editorMessageCount = 200

loadMap()
pgzrun.go()

```



^ Enter the code shown here (or download it from our GitHub) and you'll be designing your own cunning stages in no time.



Source Code



▲ The original arcade machine had three steering wheels and three accelerator pedals.

# Recreate Super Sprint's top-down racing



AUTHOR  
MARK VANSTONE

Making player and computer-controlled cars race round a track isn't as hard as it sounds

**D**ecades before the advent of more realistic racing games like *Sega Rally* or *Gran Turismo*, Atari produced a string of popular arcade racers, beginning with *Gran Trak 10* in 1974 and gradually updated via the *Sprint* series, which appeared regularly through the seventies and eighties. By 1986, Atari's *Super Sprint* allowed three players to compete at once, avoiding obstacles and collecting bonuses as they careened around the track.

The original arcade machine was controlled with steering wheels and accelerator pedals, computer-controlled cars added to the racing challenge. Tracks were of varying complexity, with some featuring flyover sections and shortcuts, while oil slicks and tornadoes provided obstacles to avoid. If a competitor crashed really badly, a new car would be airlifted in by helicopter.

So how can we make our own *Super Sprint*-style racing game with Pygame Zero?

To keep this example code short and simple, I've created a simple track with a few bends. In the original game, the movement of the computer-controlled cars would have followed a set of coordinates round the track, but as computers have much more memory now, I have used a bitmap guide

**"This snippet shows how you can get a top-down racing game working"**

for the cars to follow. This method produces a much less predictable movement for the cars as they turn right and left based on the shade of the track on the guide.

With Pygame Zero, we can write quite a short piece of code to deal with both the player car and the automated ones, but to read pixels from a position on a bitmap, we need to borrow a couple of objects directly from Pygame: we import the Pygame image

and Color objects and then load our guide bitmaps. One is for the player to restrict movement to the track, and the other is for guiding the computer-controlled cars around the track.

The cars are Pygame Zero Actors, and are drawn after the main track image in the `draw()` function. Then all the good stuff happens in the `update()` function. The player's car is controlled with the up and down arrows for speed, and the left and right arrows to change the direction of movement. We then check to see if any cars have collided with each other. If a crash has happened, we change the direction of the car and make it reverse a bit. We then test the colour of the pixel where the car is trying to move to. If the colour is black or red (the boundaries), the car turns away from the boundary.

The car steering is based on the shade of a pixel's colour read from the guide bitmap. If it's light, the car will turn right, if it's dark,



# Top-down racing in Python

Here's a code snippet that creates a *Super Sprint*-style racer in Python. To get it running on your system, you'll first need to install Pygame Zero – you can find full instructions at [wfmag.cc/pgzero](http://wfmag.cc/pgzero)

```
import math
from random import randint
from pygame import image, Color

controlimage1 = image.load('images/guide1.png')
controlimage2 = image.load('images/guide2.png')
cars = []

for c in range(4):
    cars.append(Actor('car'+str(c), center=(400, 70+(30*c))))
    cars[c].speed = 0

def draw():
    screen.blit("track", (0, 0))
    for c in range(4):
        cars[c].draw()

def update():
    if keyboard.up: cars[0].speed += .15
    if keyboard.down: cars[0].speed -= .15
    if(cars[0].speed != 0):
        if keyboard.left: cars[0].angle += 2
        if keyboard.right: cars[0].angle -= 2
    for c in range(4):
        crash = False
        for i in range(4):
            if cars[c].collidepoint(cars[i].center) and c != i:
                crash = True
                cars[c].speed = -(randint(0,1)/10)
        if crash:
            newPos = calcNewXY(cars[c].center, 2, math.
radians(randint(0,360)-cars[c].angle))
            else:
                newPos = calcNewXY(cars[c].center, cars[c].
speed*2, math.radians(180-cars[c].angle))
            if c == 0:
```

```
        ccol = controlimage1.get_
at((int(newPos[0]),int(newPos[1])))
        else:
            ccol = controlimage2.get_
at((int(newPos[0]),int(newPos[1])))
            if cars[c].speed != 0:
                if ccol != Color('blue') and ccol != Color('red'):
                    cars[c].center = newPos
            else:
                if c > 0:
                    if ccol == Color('blue'):
                        cars[c].angle += 5
                    if ccol == Color('red'):
                        cars[c].angle -= 5
                    cars[c].speed = cars[c].speed/1.1
                if c > 0 and cars[c].speed < 1.8+(c/10):
                    cars[c].speed += randint(0,1)/10
                if crash:
                    cars[c].angle += ((ccol[1]-
136)/136)*(2.8*cars[c].speed)
                else:
                    cars[c].angle -= ((ccol[1]-
136)/136)*(2.8*cars[c].speed)
                else:
                    cars[c].speed = cars[c].speed/1.1

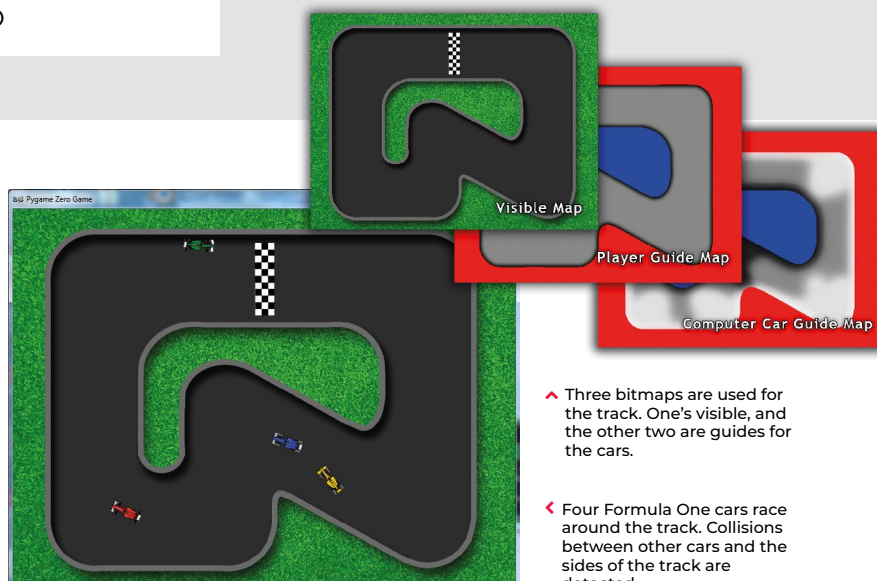
def calcNewXY(xy,speed,ang):
    newx = xy[0] - (speed*math.cos(ang))
    newy = xy[1] - (speed*math.sin(ang))
    return newx, newy
```



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag21](http://wfmag.cc/wfmag21)

the car will turn left, and if it's mid-grey, the car continues straight ahead. We could make the cars stick more closely to the centre by making them react quickly, or make them more random by adjusting the steering angle more slowly. A happy medium would be to get the cars mostly sticking to the track but being random enough to make them tricky to overtake.

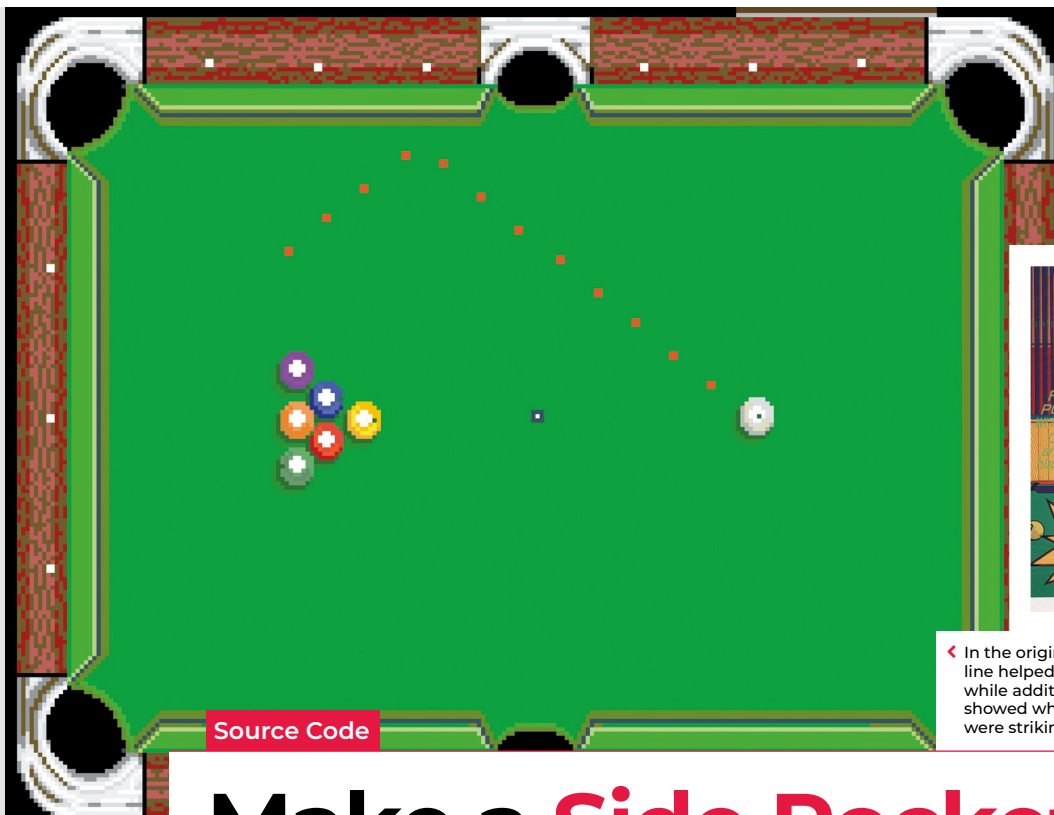
Our code will need a lot of extra elements to mimic Atari's original game, but this short snippet shows how easily you can get a top-down racing game working in Pygame Zero. 🐍



➤ Three bitmaps are used for the track. One's visible, and the other two are guides for the cars.

◀ Four Formula One cars race around the track. Collisions between other cars and the sides of the track are detected.





◀ In the original *Side Pocket*, the dotted line helped the player line-up shots, while additional functions on the UI showed where and how hard you were striking the cue ball.

Source Code

# Make a Side Pocket pool game



**AUTHOR**  
MAC BOWLEY

Recreate the arcade pool action of Data East's *Side Pocket*. Mac has the code

**C**reated by Data East in 1986, *Side Pocket* was an arcade pool game that challenged players to sink all the balls on the table and achieve a minimum score to progress. As the levels went on, players faced more balls in increasingly difficult locations on the table.

Here, I'll focus on three key aspects from *Side Pocket*: aiming a shot, moving the balls, and handling collisions for balls and pockets. This project is great for anyone who wants to dip their toe into 2D game physics. I'm going to use the Pygame's built-in collision

system as much as possible, to keep the code readable and short wherever I can. Before thinking about aiming and moving balls, I need a table to play on. I created both a border and a play area sprite using [piskelapp.com](http://piskelapp.com); originally, this was one

**“Before I think about aiming and moving balls, I need a table to play on”**

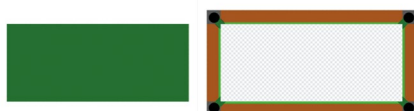
sprite, and I used a `rect` to represent the play area (see **Figure 1**). Changing to two sprites and making the play area an actor made all the collisions easier to handle and made everything much easier to place. For the balls, I made simple 32×32 sprites in varying colours. I need to be able to keep track of some information about each ball on the table, such as its position, a sprite, movement, and whether it's been pocketed or not – once a ball's pocketed, it's removed

from play. Each ball will have similar functionality as well – moving and colliding with each other. The best way to do this is with a class: a blueprint for each ball that I will make copies of when I need a new ball on the table.

```
class Ball:
    def __init__(self, image, pos):
        self.actor = Actor(image,
            center=pos, anchor=("center", "center"))
        self.movement = [0, 0]
        self.pocketed = False

    def move(self):
        self.actor.x += self.movement[0]
        self.actor.y += self.movement[1]
        if self.pocketed == False:
            if self.actor.y < playArea.
                top + 16 or self.actor.y > playArea.
                bottom-16:
                self.movement[1] = -self.
                    movement[1]
```

✓ **Figure 1:** Our table with separate border. You could add some detail to your own table, or even adapt a photograph to make it look even more realistic.





Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag36](https://wfmag.cc/wfmag36)

```

        self.actor.y =
clamp(self.actor.y, playArea.top+16,
playArea.bottom-16)
        if self.actor.x < playArea.
left+16 or self.actor.x > playArea.
right-16:
            self.movement[0] = -self.
movement[0]
            self.actor.x =
clamp(self.actor.x, playArea.left+16,
playArea.right-16)
        else:
            self.actor.x += self.
movement[0]
            self.actor.y += self.
movement[1]
            self.resistance()

def resistance(self):
    # Slow the ball down
    self.movement[0] *= 0.95
    self.movement[1] *= 0.95

    if abs(self.movement[0]) +
abs(self.movement[1]) < 0.4:
        self.movement = [0, 0]

```

The best part about using a class is that I only need to make one piece of code to move a ball, and I can reuse it for every ball on the table. I'm using an array to keep track of the ball's movement – how much it will move each frame. I also need to make sure it bounces off the sides of the play area if it hits them. I'll use an array to hold all the balls on the table. To start with, I need a cue ball:

```

balls = []
cue_ball = Ball("cue_ball.png",
(WIDTH//2, HEIGHT//2))
balls.append(cue_ball)

```

## AIMING THE SHOT

In *Side Pocket*, players control a dotted line that shows where the cue ball will go when they take a shot. Using the joystick or arrow buttons rotated the shot and moved the line, so players could aim to get the balls in the pockets (see **Figure 2** overleaf). To achieve this, we have to dive into our first bit of maths, converting a rotation in degrees to a pair of x and y movements. I decided my rotation would be at 0 degrees

when pointing straight up; the player can then press the right and left arrow to increase or decrease this value.

Pygame Zero has some built-in attributes for checking the keyboard, which I'm taking full advantage of.

```

shot_rotation = 270.0 # Start pointing
up table
turn_speed = 1
line = [] # To hold the points on my line
line_gap = 1/12
max_line_length = 400

def update():
    global shot_rotation

    ## Rotate your aim
    if keyboard[keys.LEFT]:
        shot_rotation -= 1 * turn_speed
    if keyboard[keys.RIGHT]:
        shot_rotation += 1 * turn_speed

    # Make the rotation wrap around
    if shot_rotation > 360:
        shot_rotation -= 360
    if shot_rotation < 0:
        shot_rotation += 360

```

At 0 degrees, my cue ball's movement should be 0 in the x direction and -1 in y. When the rotation is 90 degrees, my x movement would be 1 and y would be zero; anything in between should be a fraction between the two numbers. I could use a lot of 'if-elses' to set this, but an easier way is to use **sin** and **cos** on my angle – I **sin** the rotation to get my x value and **cos** the rotation to get the y movement.

```

# The in-built functions need radian
rot_radians = shot_rotation * (math.
pi/180)

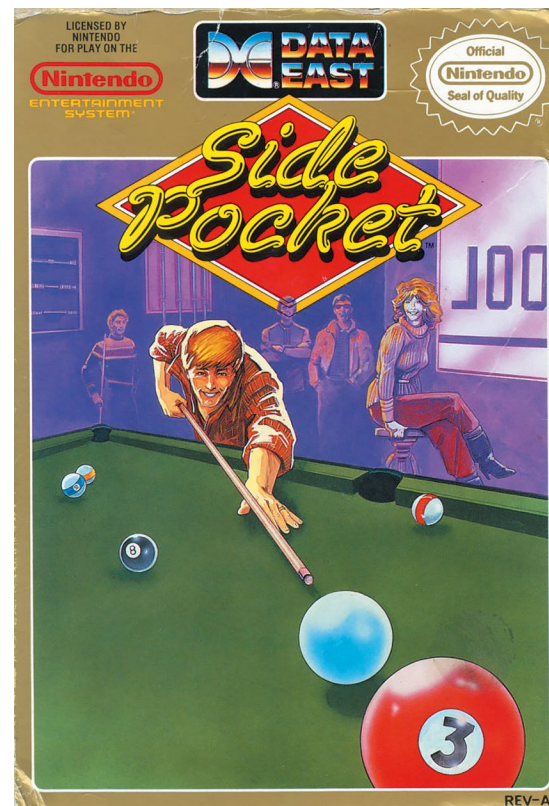
x = math.sin(rot_rads)
y = -math.cos(rot_rads)
if not shot:
    current_x = cue_ball.actor.x
    current_y = cue_ball.actor.y
    length = 0
    line = []
    while length < max_line_length: ➔

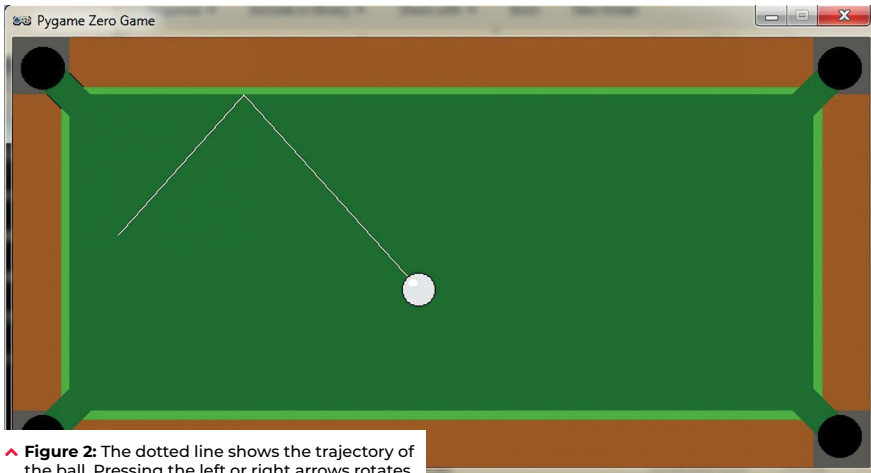
```



➤ *Side Pocket* was a big hit for Data East in the mid-eighties, and spawned a whole string of ports and spin-offs.

➤ The NES *Side Pocket* was a solid conversion. It was even ported back to arcades as an adult-themed spin-off from the main series.





▲ **Figure 2:** The dotted line shows the trajectory of the ball. Pressing the left or right arrows rotates the aim.

```
hit = False
if current_y < playArea.top
or current_y > playArea.bottom:
    y = -y
    hit = True
if current_x < playArea.left
or current_x > playArea.right:
    x = -x
    hit = True
if hit == True:
    line.append((current_x-
(x*line_gap), current_y-(y*line_gap)))
    length += math.sqrt(((x*line_
gap)**2)+(y*line_gap)**2)
    current_x += x*line_gap
    current_y += y*line_gap
    line.append((current_x-(x*line_
gap), current_y-(y*line_gap)))
```

I can then use those x and y co-ordinates to create a series of points for my aiming line.

## SHOOTING THE BALL

To keep things simple, I'm only going to have a single shot speed – you could improve this design by allowing players to load up a more powerful shot over time, but I won't do that here.

```
shot = False
ball_speed = 30

...

## Inside update
## Shoot the ball with the space bar
if keyboard[keys.SPACE] and not shot:
    shot = True
    cue_ball.momentum = [x*ball_
speed, y*ball_speed]
```

When the shot variable is **True**, I'm going to move all the balls on my table – at the beginning, this is just the cue ball – but this code will also move the other balls as well when I add them.

```
# Shoot the ball and move all the balls
on the table
else:
    shot = False
    balls_pocketed = []
    collisions = []
    for b in range(len(balls)):
        # Move each ball
        balls[b].move()
        if abs(balls[b].momentum[0])
+ abs(balls[b].momentum[1]) > 0:
            shot = True
```

Each time I move the balls, I check whether they still have some movement left. I made a **resistance** function inside the **ball** class that will slow them down.

## COLLISIONS

Now for the final problem: getting the balls to collide with each other and the pockets. I need to add more **balls** and some **pocket** actors to my game in order to test the collisions.

```
balls.append(Ball("ball_1.png", (WIDTH//2
- 75, HEIGHT//2)))
balls.append(Ball("ball_2.png", (WIDTH//2
- 150, HEIGHT//2)))

pockets = []
pockets.append(Actor("pocket.png",
toleft=(playArea.left, playArea.top),
```

**“Now for the final problem: getting the balls to collide with each other and the pockets”**

```
anchor=("left", "top"))
# I create one of these actors for each
pocket, they are not drawn
```

Each ball needs to be able to collide with the others, and when that happens, the direction and speed of the balls will change. Each ball will be responsible for changing the direction of the ball it has collided with, and I add a new function to my **ball** class:

```
def collide(self, ball):
    collision_normal = [ball.actor.x
- self.actor.x, ball.actor.y - self.
actor.y]
    ball_speed = math.sqrt(collision_
normal[0]**2 + collision_normal[1]**2)
    self_speed = math.sqrt(self.
momentum[0]**2 + self.momentum[1]**2)
    if self.momentum[0] == 0 and
self.momentum[1] == 0:
        ball.momentum[0] = -ball.
momentum[0]
        ball.momentum[1] = -ball.
momentum[1]
    elif ball_speed > 0:
        collision_normal[0] *= 1/
ball_speed
        collision_normal[1] *= 1/
ball_speed
        ball.momentum[0] = collision_
normal[0] * self_speed
        ball.momentum[1] = collision_
normal[1] * self_speed
```

When a collision happens, the other ball should move in the opposite direction to the collision. This is what allows you to line-up slices and knock balls diagonally into the pockets. Unlike the collisions with the edges, I can't just reverse the x and y movement. I need to change its direction, and then give it a part of the current ball's speed. Above, I'm using a **normal** to find the direction of the collision. You can think of this as the direction to the other ball as they collide.

## HANDLING COLLISIONS

I need to add to my `update` loop to detect and store the collisions to be handled after each set of movement.

```
# Check for collisions
for other in balls:
    if other != b and b.actor:
        colliderect(other.actor):
            collisions.append((b, other))
    # Did it sink in the hole?
    in_pocket = b.actor.
    collidelistall(pockets)
    if len(in_pocket) > 0 and b.pocketed
== False:
        if b != cue_ball:
            b.movement[0] = (pockets[in_
pocket[0]].x - b.actor.x) / 20
            b.movement[1] = (pockets[in_
pocket[0]].y - b.actor.y) / 20
            b.pocket = pockets[in_
pocket[0]]
            balls_pocketed.append(b)
        else:
            b.x = WIDTH//2
            b.y = HEIGHT//2
```

First, I use the `colliderect()` function to check if any of the balls collide this frame – if they do, I add them to a list. This is so I handle all the movement first and then the collisions. Otherwise, I'm changing the momentum of balls that haven't moved yet. I detect whether a pocket was hit as well; if so, I change the momentum so that the ball heads towards the pocket and doesn't bounce off the walls anymore.

When all my balls have been moved, I can handle the collisions with both the other balls and the pockets:

```
for col in collisions:
    col[0].collide(col[1])
if shot == False:
    for b in balls_pocketed:
        balls.remove(b)
```

And there you have it: the beginnings of an arcade pool game in the *Side Pocket* tradition. You can get the full code and assets from [wfmag.cc/wfmag36](http://wfmag.cc/wfmag36), and you can find some suggestions for improving and expanding the game in the box on the right. 🍷

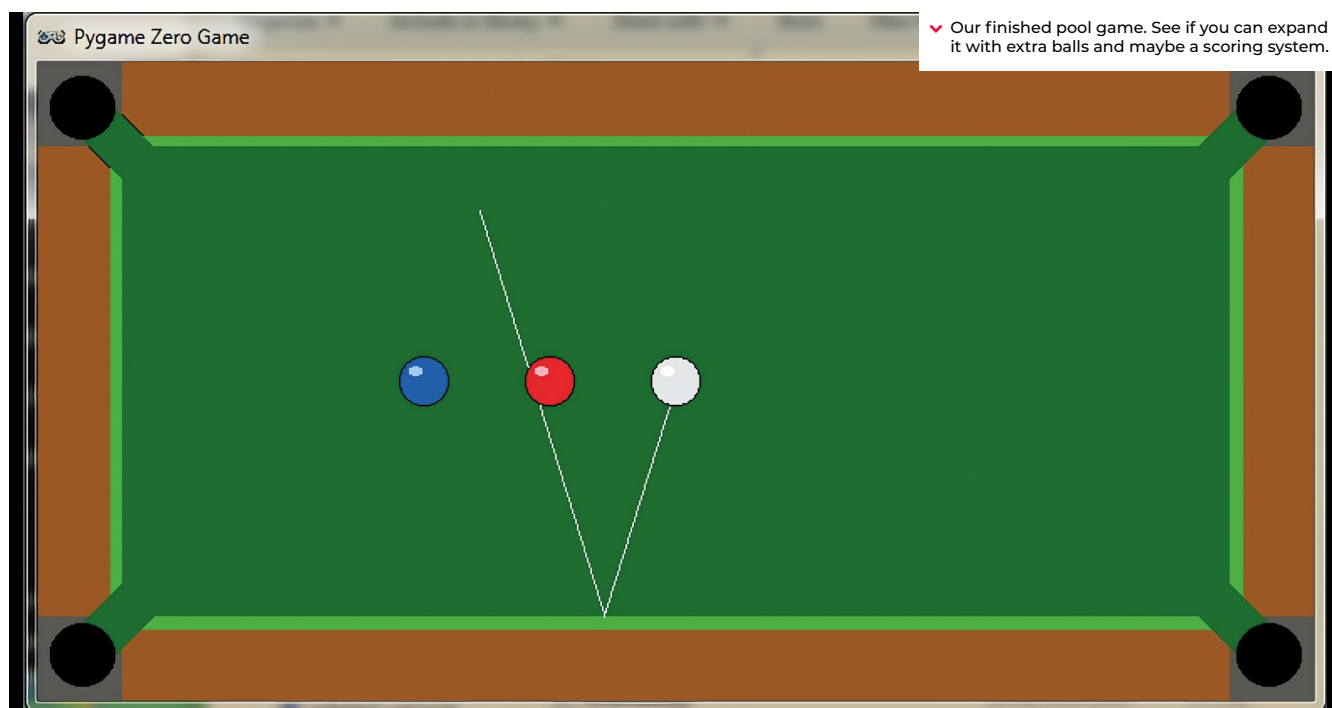


## WHAT NEXT?

If you wanted to improve your pool game, there are a few things you could do...

1. Add more balls, and arrange them in challenging ways.
2. Implement a scoring system that increases with each ball being pocketed.
3. Give the player lives; a certain amount of shots before they have to start again.

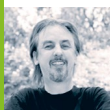
Best of luck and happy developing!





Source Code

# Code your own pinball video game



AUTHOR  
MARK VANSTONE

Get flappers flapping and balls bouncing off bumpers

There are so many pinball video games that it's become a genre in its own right. For the few of you who haven't encountered pinball for some reason, it originated as an analogue arcade machine where a metal ball would be fired onto a sloping play area and bounce between obstacles. The player operates a pair of flippers by pressing buttons on each side of the machine, which will in turn ping the ball back up the play area to hit obstacles and earn points. The game ends when the ball falls through the exit at the bottom of the play area.

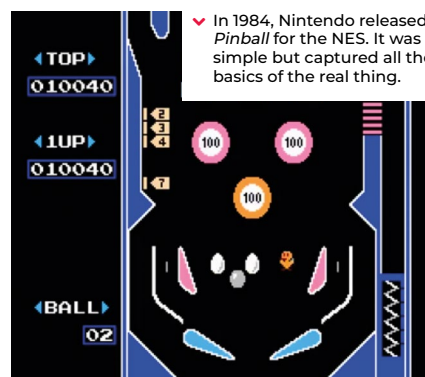
Video game developers soon started

trying to recreate pinball, first with fairly rudimentary graphics and physics, but with increasingly greater realism over time – if you look at Nintendo's *Pinball* from 1984, then, say, *Devil's Crush* on the Sega Mega Drive in 1990, and then 1992's *Pinball Dreams* on PC, you can see how radically the genre evolved in just a few years. In this month's Source Code, we're going to put together a very simple rendition of pinball in Pygame Zero. We're not going to use any complicated maths or physics systems, just a little algebra and trigonometry.

Let's start with our background. We need an image which has barriers around the outside for the ball to bounce off, and a gap at the bottom for the ball to fall through. We also want some obstacles in the play area and an entrance at the side for the ball to enter when it's first fired. In this case, we're going to use our background as a collision map, too, so we need to design it so that all the areas that the ball can move in are black.

Next, we need some flippers. These are defined as Actors with a pivot anchor position set near the larger end, and are positioned near the bottom of the play area. We detect left and right key presses and rotate the angle of the flippers by 20 degrees within a range of -30 to +30 degrees. If no key is pressed, then the flipper drops back down. With these elements in place, we have our play area and an ability for the player to defend the exit.

All we need now is a ball to go bouncing around the obstacles we've made. Defining the ball as an Actor, we can add a direction and a speed parameter to it. With these values set, the ball can be moved using a bit of trigonometry. Our new x-coordinate will move by the sin of the ball direction multiplied by the speed, and the new



✓ In 1984, Nintendo released *Pinball* for the NES. It was simple but captured all the basics of the real thing.

y-coordinate will move by the cos of the ball direction multiplied by speed. We need to detect collisions with objects and obstacles, so we sample four pixels around the ball to see if it's hit anything solid. If it has, we need to make the ball bounce.

If you wanted more realistic physics, you'd calculate the reflection angle from the surface which has been hit, but in this case, we're going to use a shortcut which will produce a rough approximation. We work out what direction the ball is travelling in and then rotate either left or right by a quarter of a turn until the ball no longer collides with a wall. We could finesse this calculation further to create a more accurate effect, but we'll keep it simple for this sample. Finally, we need to add some gravity. As the play area is tilted downwards, we need to increase the ball speed as it travels down and decrease it as it travels up.

All of this should give you the bare bones of a pinball game. There's lots more you could add to increase the realism, but we'll leave you to discover the joys of normal vectors and dot products... 🧐



✓ Here it is: your own pinball game in less than 100 lines of code.



# Play the silver ball

Here's Mark's code for a simple pinball video game. To get it running on your system, you'll first need to install Pygame Zero – full instructions can be found at [wfmag.cc/pgzero](http://wfmag.cc/pgzero).



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag53](http://wfmag.cc/wfmag53)

```
# Pinball
import pgzrun
import math
import random
from pygame import image, Color

WIDTH = 600
HEIGHT = 800
collisionMap = image.load('images/background.png')
flipperLeft = Actor('flipperl', center=(210, 660), anchor=(20, 20))
flipperLeft.angle = -30
flipperRight = Actor('flipperrr', center=(390, 660), anchor=(112, 20))
flipperRight.angle = 30

def init():
    global gamestate, ball
    ball = Actor('ball', center=(560, 310))
    ball.speed = 5 + random.randint(0, 7)
    ball.dir = 4 + ((random.randint(0, 10)/10)-0.5)
    gamestate = 0

def draw():
    screen.blit("background", (0, 0))
    flipperLeft.draw()
    flipperRight.draw()
    if gamestate == 0 or random.randint(0, 1) == 1: ball.draw()

def update():
    if gamestate == 0:
        if keyboard.left:
            flipperLeft.angle = limit(flipperLeft.angle+20, -30, 30)
        else:
            flipperLeft.angle = limit(flipperLeft.angle-20, -30, 30)
        if keyboard.right:
            flipperRight.angle = limit(flipperRight.angle-20, -30, 30)
        else:
            flipperRight.angle = limit(flipperRight.angle+20, -30, 30)
        moveBall()
        checkBounce()
    else:
        if keyboard.space: init()

def moveBall():
    global gamestate
    ball.x += ball.speed * math.sin(ball.dir)
    ball.y += ball.speed * math.cos(ball.dir)
    if ball.x > 570 or ball.y > 760: gamestate = 1
```

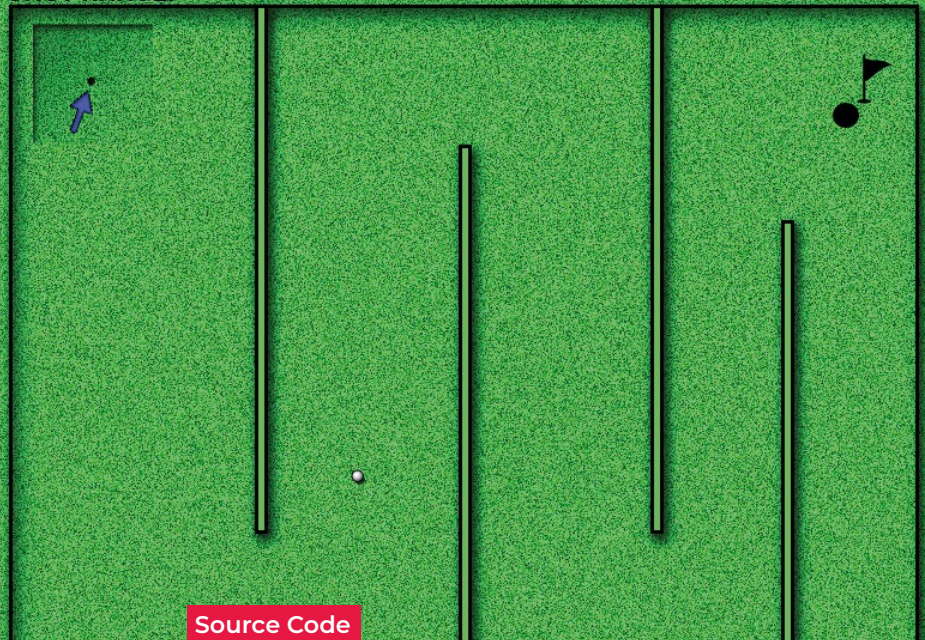
```
def checkBounce():
    global score
    d = math.degrees(ball.dir)%360
    inc = -1.5
    if d > 90 and d < 270:
        inc = 1.5
        ball.speed -= 0.03
        if ball.speed < 0: ball.dir = 0
    else:
        if ball.speed < 10: ball.speed += 0.04
    if flipperRight.collidepoint(ball.pos):
        ball.dir = 4 + (flipperRight.angle/50)
        if keyboard.right :
            ball.speed += 0.3
            moveBall()
        if inc == 1.5:
            ball.dir = 0
            moveBall()
    if flipperLeft.collidepoint(ball.pos):
        ball.dir = 3 + (flipperLeft.angle/50)
        if keyboard.left :
            ball.speed += 0.3
            moveBall()
        if inc == 1.5:
            ball.dir = 0
            moveBall()
    rgb = collisionCheck()
    while rgb != Color("black"):
        ball.dir += inc
        moveBall()
        rgb = collisionCheck()

def collisionCheck():
    r = 22
    cl = [(0, -r), (r, 0), (0, r), (-r, 0)]
    for t in range(4):
        rgb = collisionMap.get_at((int(ball.x)+cl[t][0],
int(ball.y)+cl[t][1]))
        if rgb != Color("black"):
            return rgb
    return rgb

def limit(n, minn, maxx):
    return max(min(maxn, n), minn)

init()
pgzrun.go()
```

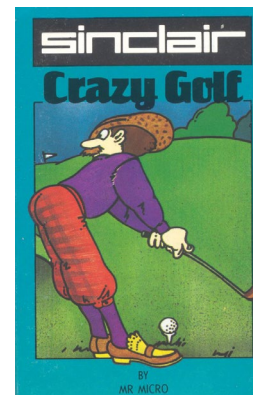
SHOT RANGE:



Source Code

◀ The pointer's angle is rotated using degrees, but we'll use radians for our ball direction as it will simplify our movement and bounce calculations.

✓ The game was called *Crazy Golf* on the cover...



AUTHOR  
MARK VANSTONE

# Code a Spectrum-style Crazy Golf game

Putt the ball around irrational obstacles in our take on golf

**F**irst released by Mr. Micro in 1983 – then under the banner of Sinclair Research – *Krazy Golf* was, confusingly, also called *Crazy Golf*. The loading screen featured the *Krazy* spelling, but on the cover, it was plain old *Crazy Golf*. Designed for the ZX Spectrum, the game provided nine holes and a variety of obstacles to putt the ball around. *Crazy Golf* was released at a time when dozens of other games were hitting the Spectrum market, and although it was released under the Sinclair name and reviewed in magazines such as *Crash*, it didn't make much impact. The game itself employed a fairly rudimentary control system, whereby the player selects the angle of the shot at the top left of the screen, sets the range via a bar along the top, and then presses the **RETURN** key to take the shot.

If you've been following our Source Code articles each month, you will have seen the pinball game where a ball bounces off various surfaces. In that example, we

used a few shortcuts to approximate the bounce angles. Here, we're only going to have horizontal and vertical walls, so we can use some fairly straightforward maths to calculate more precisely the new angle as the ball bounces off a surface. In the original game, the ball was limited to only 16 angles, and the ball moved at the same speed regardless of the strength of the shot. We're going to improve on this a bit so that there's more flexibility around the shot angle; we'll also get the ball to start moving fast and then reduce its speed until it stops.

To make this work, we need to have a way of defining whether an obstruction is horizontal or vertical, as the calculation is different for each. We'll have a background graphic showing the course and obstacles, but we'll also need another map to check our collisions. We need to make a collision map that just has the obstacles on it, so we need a white background; mark all the horizontal surfaces red and all the vertical surfaces blue. As we move the ball around the screen (in much the same way as our

pinball game) we check to see if it has collided with a surface by sampling the colours of the pixels from the collision map. If the pixel's blue, we know that the ball has hit a vertical wall; if it's red, the wall's horizontal. We then calculate the new angle for the ball. If we mark the hole as black, then we can also test for collision with that – if the ball's in the hole, the game ends.

We have our ball bouncing mechanism, so now we need our user interaction system. We'll use the left and right arrow keys to rotate our pointer, which designates the direction of the next shot. We also need a range-setting gizmo, which will be shown as a bar at the top of the screen. We can make that grow and shrink with the up and down arrows. Then when we press the **RETURN** key, we transfer the pointer angle and the range to the ball and watch it go. We ought to count each shot so that we can display a tally to the player once they've putted the ball into the hole. From this point, it's a simple task to create another eight holes – and then you'll have a full crazy golf game! 🏌️

# Crazy Golf in Python

Here's Mark's code for a nifty top-down golf game. To get it running on your system, you'll need to install Pygame Zero. Full instructions are available at [wfmag.cc/pgzero](http://wfmag.cc/pgzero).



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag54](http://wfmag.cc/wfmag54)

```
# Crazy Golf
import pgzrun
import math
from pygame import image, Color

collisionMap = image.load('images/collision.png')
pointer = Actor('pointer', center=(90,85))
pointer.angle = 0
ball = Actor('ball', center=(100,150))
ball.speed = ball.dir = 0
gamestate = shots = 0
shotrange = 300

def draw():
    screen.blit("background", (0, 0))
    ball.draw()
    pointer.draw()
    screen.draw.filled_
    rect(Rect((180,5),(shotrange,10)),(255,0,0))
    screen.draw.text("SHOT RANGE:", topleft = (20,
2),color=(0,0,0) , fontsize=28)
    if gamestate == 1 : screen.draw.text("YOU SUNK THE BALL IN
" + str(shots) + " STROKES", center = (400, 300), owidth=0.5,
ocolor=(255,255,0), color=(255,0,0) , fontsize=50)

def update():
    global shotrange
    if gamestate == 0:
        if keyboard.left:
            pointer.angle += 5
        if keyboard.right:
            pointer.angle -= 5
        if keyboard.up:
            shotrange = limit(shotrange + 10, 0, 600)
        if keyboard.down:
            shotrange = limit(shotrange - 10, 0, 600)
        checkBounce()
        moveBall()
        ball.speed = limit(ball.speed-0.01, 0, 10)

def on_key_down(key):
    if gamestate == 0:
        if key.name == "RETURN": hitBall(pointer.
angle,shotrange/100)

def hitBall(a,s):
    global shots
    ball.speed = s
    ball.dir = math.radians(a)
```

```
shots += 1

def moveBall():
    ball.x += ball.speed * math.sin(ball.dir)
    ball.y += ball.speed * math.cos(ball.dir)

def checkBounce():
    global gamestate
    rgb = collisionCheck()
    if rgb == Color("black"):
        gamestate = 1

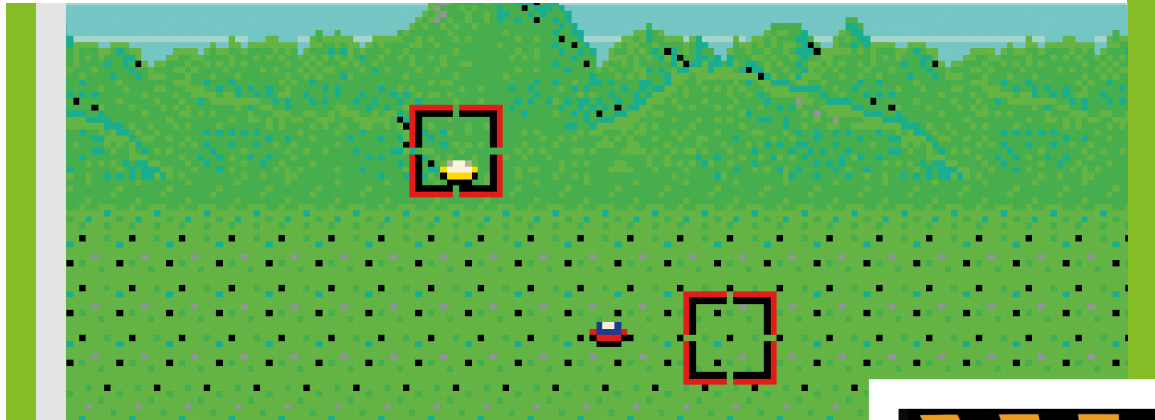
def collisionCheck():
    r = 4
    cl = [(0,-r),(r,0),(0,r),(-r,0)]
    for t in range(4):
        rgb = collisionMap.get_at((int(ball.x)+cl[t]
[0],int(ball.y)+cl[t][1]))
        if rgb != Color("white"):
            if rgb == Color("blue"):
                ball.dir = (2*math.pi - ball.dir)%(2*math.pi)
            if rgb == Color("red"):
                ball.dir = (3*math.pi - ball.dir)%(2*math.pi)
    return rgb

def limit(n, minn, maxx):
    return max(min(maxn, n), minn)

pgzrun.go()
```



^ ...but weirdly, the loading screen spelled the name as Krazy Golf. The early games industry was strange.



✓ Like its predecessor, *Track & Field*, *Hyper Sports* had two run buttons and one action button per player.



^ *Hyper Sports*' Japanese release was tied in with the 1984 Summer Olympics.

◀ When the clay targets appear, the player uses the left and right buttons to shoot either the left or right target respectively.



Source Code

# Code Hyper Sports' shooting minigame

Gun down the clay pigeons in our re-creation of a classic minigame from Konami's *Hyper Sports*



AUTHOR  
MARK VANSTONE

Konami's sequel to its 1983 arcade hit, *Track & Field*, *Hyper Sports* offered seven games – or events – in which up to four players could participate. Skeet shooting was perhaps the most memorable game in the collection, and required just two buttons: fire left and fire right. The display showed two target sights, and each moved up and down to come into line with the next clay disc's trajectory. When the disc was inside the red target square, the player pressed the fire button, and if their timing was correct, the clay disc exploded. Points were awarded for being on target, and every now and then, a parrot flew across the screen, which could be gunned down for a bonus.

To make a skeet shooting game with Pygame Zero, we need a few graphical elements. First, a static background of hills and grass, with two clay disc throwers each side of the screen, and a semicircle where

our shooter stands – this can be displayed first, every time our `draw()` function is called. We can then draw our shooter (created as an Actor) in the centre near the bottom of the screen. The shooter has three images: one central while no keys are pressed, and two for the directions left and right when the player presses the left or right keys. We also need to have two square target sights to the left and right above the shooter, which we can create as Actors.

To make the clay targets, we create an array to hold disc Actor objects. In our `update()` function we can trigger the creation of a new disc based on a random number, and once created, start an animation to move it across the screen in front of the shooter. We can add a shadow to the discs by tracking a path diagonally across the screen so that the shadow appears at the correct Y coordinate regardless of the disc's height – this is a simple way of giving our

game the illusion of depth. While we're in the `update()` function, looping around our disc object list, we can calculate the distance of the disc to the nearest target sight frame, and from that, work out which is the closest.

When we've calculated which disc is closest to the right-hand sight, we want to move the sight towards the disc so that their paths intersect. All we need to do is take the difference of the Y coordinates, divide by two, and apply that offset to the target sight. We also do the same for the left-hand sight. If the correct key (left or right arrows) is pressed at the moment a disc crosses the path of the sight frame, we register a hit and cycle the disc through a sequence of exploding frames. We can keep a score and display this with an overlay graphic so that the player knows how well they've done.

And that's it! You may want to add multiple players and perhaps a parrot bonus, but we'll leave that up to you. 🦜





Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag35](https://wfmag.cc/wfmag35)

# Skeet shooting in Python

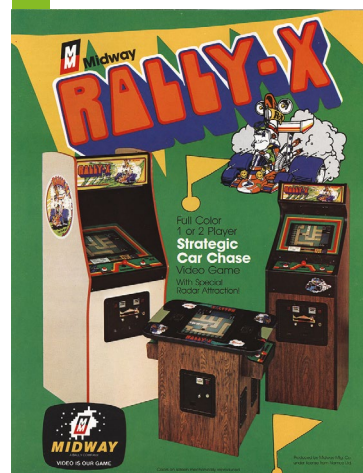
Here's Mark's code snippet, which creates a skeet shooting game in Python. To get it running on your system, you'll need to install Pygame Zero – you can find full instructions at [wfmag.cc/pgzero](https://wfmag.cc/pgzero).

```
from random import randint
gameState = shootTimer = score = 0
shooter = Actor('shooter', center=(400, 450))
frameLeft = Actor('frame', center=(320, 350))
frameRight = Actor('frame', center=(480, 350))
skeets = []
def draw():
    screen.blit("background", (0, 0))
    if gameState == 0:
        for s in range(len(skeets)):
            if skeets[s].x > 0 and skeets[s].x < 800 and skeets[s].
frame < 4:
            skeets[s].draw()
            screen.blit("shadow", (skeets[s].x-20, 400-(skeets[s].
life/2)))
            shooter.draw()
            frameLeft.draw()
            frameRight.draw()
    else:
        screen.draw.text("ROUND OVER", center = (400, 300),
owidth=0.5, ocolor=(255,255,255), color=(0,255,0) , fontsize=80)
        screen.blit("overlay", (0, 0))
        screen.draw.text("SCORE:"+str(score), center = (400, 550),
owidth=0.5, ocolor=(255,255,255), color=(0,0,255) , fontsize=80)
        screen.draw.text("PYGAME ZERO SKEET SHOOT", center = (400, 55),
owidth=0.5, ocolor=(255,255,255), color=(255,0,0) , fontsize=60)
def update():
    global shootTimer, gameState
    if gameState == 0:
        if len(skeets) == 100: gameState = 1
        if randint(0,100) == 1: makeSkeet(700)
        if randint(0,100) == 2: makeSkeet(100)
        if shootTimer == 0:
            shooter.image = "shooter"
        else: shootTimer -= 1
        for s in range(len(skeets)):
            skeets[s].life += 1
            if skeets[s].frame > 0 and skeets[s].frame < 4:
                skeets[s].image = "skeet"+str(skeets[s].frame)
                skeets[s].frame += 1
            if skeets[s].x < 320 and skeets[s].dir == "right":
                skeets[s].distToLeftTarget = 320 - skeets[s].x
            else: skeets[s].distToLeftTarget = 999
            if skeets[s].x > 480 and skeets[s].dir == "left":
                skeets[s].distToRightTarget = skeets[s].x - 480
            else: skeets[s].distToRightTarget = 999
            targetLeft = getNearestSkeetY("left")
            if targetLeft > 0: frameLeft.y += (targetLeft-
frameLeft.y)/2
            targetRight = getNearestSkeetY("right")
            if targetRight > 0: frameRight.y += (targetRight-
frameRight.y)/2
```

```
def on_key_down(key):
    global shootTimer
    if (shootTimer == 0):
        if key.name == "LEFT":
            shooter.image = "shooter_l"
            shootTimer = 10
            checkShot("left")
        if key.name == "RIGHT":
            shooter.image = "shooter_r"
            shootTimer = 10
            checkShot("right")
def makeSkeet(st):
    skeets.append(Actor('skeet', center=(st, 370)))
    s = len(skeets)-1
    skeets[s].frame = 0
    skeets[s].life = 0
    skeets[s].distToLeftTarget = 999
    skeets[s].distToRightTarget = 999
    endpoint = 800
    skeets[s].dir = "right"
    if st > 400:
        endpoint = 0
        skeets[s].dir = "left"
    animate(skeets[len(skeets)-1], duration=3, pos=(endpoint,
randint(-200,250)))
def getNearestSkeetY(leftorright):
    y = 0
    dist = 999
    for s in range(len(skeets)):
        if leftorright == "right":
            if(skeets[s].distToRightTarget < dist):
                dist = skeets[s].distToRightTarget
                y = skeets[s].y
        if leftorright == "left":
            if(skeets[s].distToLeftTarget < dist):
                dist = skeets[s].distToLeftTarget
                y = skeets[s].y
    return y
def checkShot(leftorright):
    global score
    sounds.shot.play()
    for s in range(len(skeets)):
        if leftorright == "right":
            if skeets[s].collidepoint((frameRight.x, frameRight.y))
and skeets[s].frame == 0:
                score += 1000
                skeets[s].frame = 1
        if leftorright == "left":
            if skeets[s].collidepoint((frameLeft.x, frameLeft.y)) and
skeets[s].frame == 0:
                score += 1000
                skeets[s].frame = 1
```







^ Three different cabinet styles were available for Rally-X.

< In Namco's original arcade game, the red cars chased the player relentlessly around each level. Note the handy mini-map on the right.

# Code a Rally-X-style mini-map



AUTHOR  
MARK VANSTONE

Race around using a mini-map for navigation, just like the arcade classic, Rally-X

The original *Rally-X* arcade game blasted onto the market in 1980, at the same time as *Pac-Man* and *Defender*. This was the first year that developer Namco had exported its games outside Japan thanks to the deal it struck with Midway, an American game distributor. The aim of *Rally-X* is to race a car around a maze, avoiding enemy cars while collecting yellow flags – all before your fuel runs out.

The aspect of *Rally-X* that we'll cover here is the mini-map. As the car moves around the maze, its position can be seen relative to the flags on the right of the screen. The main view of the maze only shows a section of the whole map, and scrolls as the car moves, whereas the mini-map shows the whole size of the map but without any of the maze walls – just dots where the car and flags are (and in the original, the enemy cars). In our example, the mini-map is five times smaller than the main map, so it's easy to work out the calculation to translate large map co-ordinates to mini-map co-ordinates.

To set up our *Rally-X* homage in Pygame Zero, we can stick with the default screen size of 800×600. If we use 200 pixels for the side panel, that leaves us with a 600×600 play area. Our player's car will be drawn in the centre of this area at the co-ordinates 300,300. We can use the in-built rotation of the Actor object by setting the angle property of the car. The maze scrolls depending on which direction the car is pointing, and this can be done by having a lookup table in the form of a dictionary list (**directionMap**) where we define x and y increments for each angle the car can travel. When the cursor keys are pressed, the car stays central and the map moves.

To detect the car hitting a wall, we can use a collision map. This isn't a particularly memory-efficient way of doing it, but it's easy to code. We just use a bitmap the same size as the main map which has all the roads as black and all the walls as white. With this map, we can detect if there's a wall in the direction in which the car's moving by testing the pixels directly in front of it. If a wall is

detected, we rotate the car rather than moving it. If we draw the side panel after the main map, we'll then be able to see the full layout of the screen with the map scrolling as the car navigates through the maze.

We can add flags as a list of Actor objects. We could make these random, but for the sake of simplicity, our sample code has them defined in a list of x and y co-ordinates. We need to move the flags with the map, so in each **update()**, we loop through the list and add the same increments to the x and y co-ordinates as the main map. If the car collides with any flags, we just take them off the list of items to draw by adding a **collected** variable. Having put all of this in place, we can draw the mini-map, which will show the car and the flags. All we need to do is divide the object co-ordinates by five and add an x and y offset so that the objects appear in the right place on the mini-map.

And those are the basics of *Rally-X*! All it needs now is a fuel gauge, some enemy cars, and obstacles – but we'll leave those for you to sort out... 🤖



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag43](https://wfmag.cc/wfmag43)

# Rally-X racing in Python

Here's Mark's code for a *Rally-X*-style racer, complete with mini-map. To get it working on your system, you'll need to install Pygame Zero – full instructions are available at [wfmag.cc/pgzero](https://wfmag.cc/pgzero).

```
# Rally X
from pygame import image, Color

car = Actor('car', center=(300, 300))
car.angle = 180
mapx = -100
mapy = 0
directionMap = {(0,1), (90:(1,0), 180:(0,-1), 270:(-1,0))}
speed = 5
collisionmap = image.load('images/collisionmap.png')
count = gameStatus = 0
flagsXY=[(200,1900),(300,1100),(300,300),
(400,600),(600,1600),(800,350)]
flags = []
for f in range(0, 6):
    flags.append(Actor('flag', center=(0, 0)))
    flags[len(flags)-1].collected = False

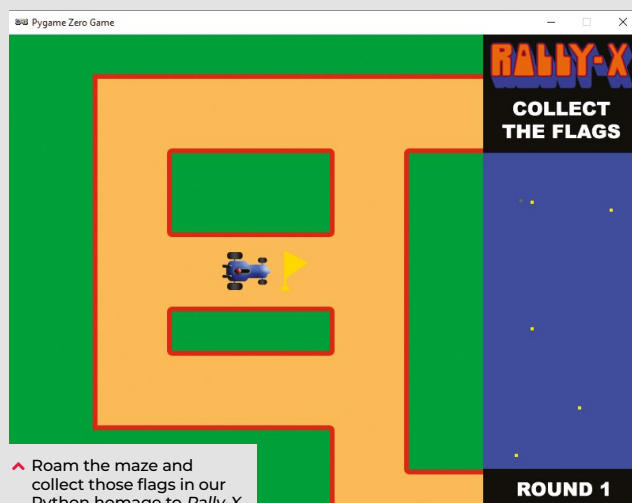
def draw():
    screen.blit("colourmap", (mapx, mapy))
    car.draw()
    for f in range(0, 6):
        if not flags[f].collected: flags[f].draw()
    screen.blit("sidepanel", (600, 0))
    drawMiniMap()
    if gameStatus == 1: screen.draw.text("YOU GOT
ALL THE FLAGS!", center = (400, 300), owidth=0.5,
ocolor=(255,255,255), color=(0,0,255), fontsize=80)

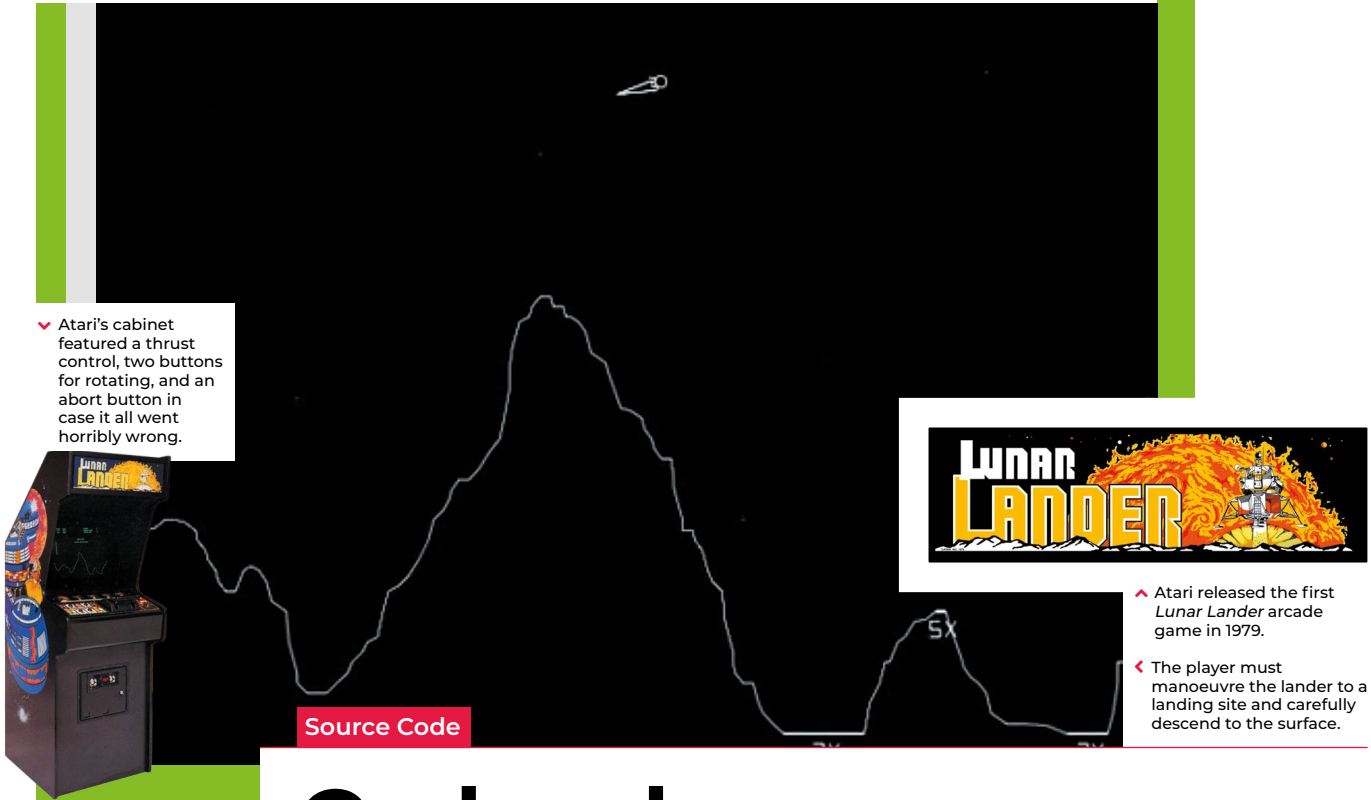
def update():
    global mapx, mapy, count, gameStatus
    if gameStatus == 0:
        checkInput()
        testmove = (int((-mapx+300) - ((directionMap[car.
angle][0]*8) * speed)), int((-mapy+300) - ((directionMap[car.
angle][1]*8) * speed)))
        if collisionmap.get_at(testmove) == Color('black'):
            mapx += directionMap[car.angle][0] * speed
            mapy += directionMap[car.angle][1] * speed
        else:
            car.angle += 90
            if car.angle == 360: car.angle = 0
            if collisionmap.get_at((int((-mapx+330), int(-
mapy+300))) == Color('white'): mapx += 1
            if collisionmap.get_at((int((-mapx+270), int(-
mapy+300))) == Color('white'): mapx -= 1
            if collisionmap.get_at((int((-mapx+300), int(-
mapy+330))) == Color('white'): mapy += 1
```

```
            if collisionmap.get_at((int((-mapx+300), int(-
mapy+270))) == Color('white'): mapy -= 1
            flagCount = 0
            for f in range(0, 6):
                flags[f].x = flagsXY[f][0]+mapx
                flags[f].y = flagsXY[f][1]+mapy
                if flags[f].collidepoint(car.pos):
                    flags[f].collected = True
                    if flags[f].collected == True: flagCount += 1
            count += 1
            if flagCount == 6: gameStatus = 1

def checkInput():
    if keyboard.left: car.angle = 90
    if keyboard.right: car.angle = 270
    if keyboard.up: car.angle = 0
    if keyboard.down: car.angle = 180

def drawMiniMap():
    carRect = Rect((658+(-mapx/5), 208+(-mapy/5)), (4, 4))
    if count%10 > 5:
        screen.draw.filled_rect(carRect, (0, 0, 0))
    else:
        screen.draw.filled_rect(carRect, (100, 100, 100))
    for f in range(0, 6):
        if not flags[f].collected:
            flagRect = Rect((600+(flagsXY[f]
[0]/5), 150+(flagsXY[f][1]/5)), (4, 4))
            screen.draw.filled_rect(flagRect, (255, 255, 0))
```





✓ Atari's cabinet featured a thrust control, two buttons for rotating, and an abort button in case it all went horribly wrong.

▲ Atari released the first *Lunar Lander* arcade game in 1979.

◀ The player must manoeuvre the lander to a landing site and carefully descend to the surface.

Source Code

## Code a homage to **Lunar Lander**



AUTHOR  
MARK VANSTONE

Shoot for the moon in Mark's version of an Atari hit

**F**irst released in 1979 by Atari, *Lunar Lander* was based on a concept created a decade earlier. The original 1969 game (actually called *Lunar*) was a text-based affair that involved controlling a landing module's thrust to guide it safely down to the lunar surface; a later iteration, *Moonlander*, created a more visual iteration of the same idea on the DEC VT50 graphics terminal.

Given that it appeared at the height of the late-seventies arcade boom, though, it was Atari's coin-op that became the most recognisable version of *Lunar Lander*, arriving just after the tenth anniversary of the Apollo 11 moon landing. Again, the aim of the game was to use rotation and thrust controls to guide your craft, and gently set it down on a suitably flat platform. The game required efficient control of the lander, and extra points were awarded for parking successfully on more challenging areas of the landscape.

The arcade cabinet was originally going to feature a normal joystick, but this was changed to a double stalked up-down lever

providing variable levels of thrust. The player had to land the craft against the clock with a finite amount of fuel with the Altitude, Horizontal Speed, and Vertical Speed readouts at the top of the screen as a guide. Four levels of difficulty were built into the game, with adjustments to landing controls and landing areas.

To write a game like *Lunar Lander* with Pygame Zero, we can replace the vector graphics with a nice pre-drawn static background and use that as a collision detection mechanism and altitude meter. If our background is just black where the Lander can fly and a different colour anywhere the landscape is, then we can test pixels using the Pygame function `image.get_at()` to see if the lander has landed. We can also test a line of pixels from the Lander down the Y-axis until we hit the landscape, which will give us the lander's altitude.

The rotation controls of the lander are quite simple, as we can capture the left and right arrow keys and increase or decrease the rotation of the lander; however, when thrust

is applied (by pressing the up arrow) things get a little more complicated. We need to remember which direction the thrust came from so that the craft will continue to move in that direction even if it is rotated, so we have a direction property attached to our lander object. A little gravity is applied to the position of the lander, and then we just need a little bit of trigonometry to work out the movement of the lander based on its speed and direction of travel.

To judge if the lander has been landed safely or rammed into the lunar surface, we look at the downward speed and angle of the craft as it reaches an altitude of 1. If the speed is sufficiently slow and the angle is near vertical, then we trigger the landed message, and the game ends. If the lander reaches zero altitude without these conditions met, then we register a crash. Other elements that can be added to this sample are things like a limited fuel gauge and variable difficulty levels. You might even try adding the sounds of the rocket booster noise featured on the original arcade game. 🌕



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag37](https://wfmag.cc/wfmag37)

# Lunar Lander in Python

Here's Mark's code for a simple, modern take on *Lunar Lander*. To get it running on your system, you'll need to install Pygame Zero – full instructions are available at [wfmag.cc/pgzero](https://wfmag.cc/pgzero).

```
import math
from pygame import image, Color
import time
start_time = time.time()
backgroundImage = image.load('images/background.png')
lander = Actor('lander', (50, 30))
lander.angle = lander.direction = -80
lander.thrust = 0.5
gravity = 0.8
lander.burn = speedDown = gameState = gameTime = 0

def draw():
    global gameTime
    screen.blit('background', (0, 0))
    screen.blit('space', (0, 0))
    r = lander.angle
    if(lander.burn > 0):
        lander.image = "landerburn"
    else:
        lander.image = "lander"
    lander.angle = r
    lander.draw()
    if gameState == 0:
        gameTime = int(time.time() - start_time)
        screen.draw.text("Altitude : "+ str(getAlt()),
            topleft=(650, 10), owidth=0.5, ocolor=(255, 0, 0),
            color=(255, 255, 0), fontsize=25)
        screen.draw.text("Time : "+ str(gameTime), topleft=(40,
            10), owidth=0.5, ocolor=(255, 0, 0), color=(255, 255, 0),
            fontsize=25)
        if gameState == 2:
            screen.draw.text("Congratulations \n The Eagle Has
            Landed", center=(400, 50), owidth=0.5, ocolor=(255, 0, 0),
            color=(255, 255, 0), fontsize=35)
        if gameState == 1:
            screen.draw.text("Crashed", center=(400, 50),
            owidth=0.5, ocolor=(255, 0, 0), color=(255, 255, 0), fontsize=35)

def update():
    global gameState, speedDown
    if gameState == 0:
        if keyboard.up:
            lander.thrust = limit(lander.thrust+0.01, 0, 1)
            changeDirection()
            lander.burn = 1
        if keyboard.left: lander.angle += 1
        if keyboard.right: lander.angle -= 1
        oldPos = lander.center
        lander.y += gravity
        newPos = calcNewXY(lander.center, lander.thrust, math.
            radians(90-lander.direction))
        lander.center = newPos
```

```
speedDown = newPos[1] - oldPos[1]
lander.thrust = limit(lander.thrust-0.001, 0, 1)
lander.burn = limit(lander.burn-0.05, 0, 1)
if speedDown < 0.2 and getAlt() == 1 and lander.angle >
-5 and lander.angle < 5:
    gameState = 2
if getAlt() == 0:
    gameState = 1

def changeDirection():
    if lander.direction > lander.angle: lander.direction -= 1
    if lander.direction < lander.angle: lander.direction += 1

def limit(n, minn, maxn):
    return max(min(maxn, n), minn)

def calcNewXY(xy, speed, ang):
    newx = xy[0] - (speed*math.cos(ang))
    newy = xy[1] - (speed*math.sin(ang))
    return newx, newy

def getAlt():
    testY = lander.y+8
    height = 0;
    while testPixel((int(lander.x), int(testY))) ==
    Color('black') and height < 600:
        testY += 1
        height += 1
    return height

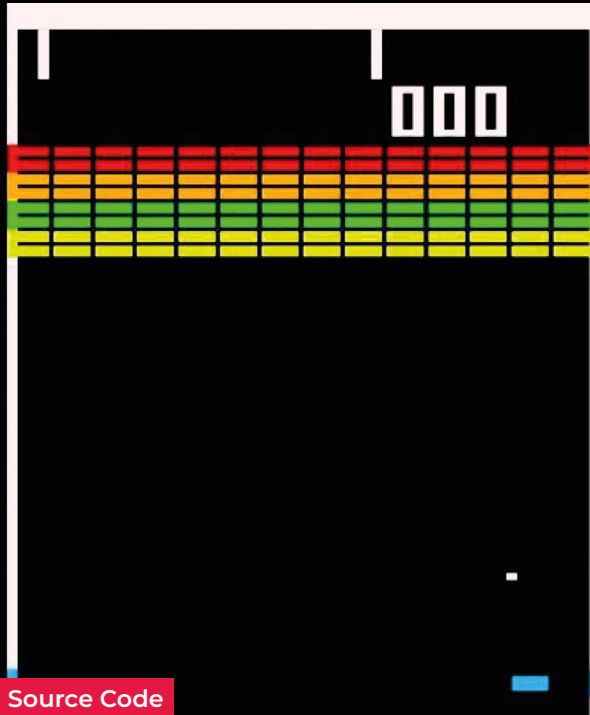
def testPixel(xy):
    if xy[0] >= 0 and xy[0] < 800 and xy[1] >= 0 and xy[1] <
    600:
        return
    backgroundImage.get_at(xy)
    else:
        return
    Color('black')
```

› Our homage to the classic *Lunar Lander*. Can you land without causing millions of dollars' worth of damage?



## ENGAGE

The direction of thrust could be done in several ways. In this case, we've kept it simple, with one directional value which gradually moves in a new direction when an alternative thrust is applied. You may want to try making an X- and Y-axis direction calculation for thrust so that values are a combination of the two dimensions. You could also add joystick control to provide variable thrust input.



▲ The original *Breakout*, designed by Nolan Bushnell and Steve Bristow, and famously built by a young Steve Wozniak.

Source Code

## Breakout's brick-breaking action

Atari's *Breakout* was one of the earliest video game blockbusters. Here's how to recreate it in Python



AUTHOR  
RYAN LAMBIE

The games industry owes a lot to the humble bat and ball. Designed by Allan Alcorn in 1972, *Pong* was a simplified version of table tennis, where the player moved a bat and scored points by ricocheting a ball past their opponent. About four years later, Atari's Nolan Bushnell and Steve Bristow figured out a way of making *Pong* into a single-player game. The result was 1976's *Breakout*, which rotated *Pong*'s action through 90 degrees and replaced the second player with a wall of bricks. Points were scored by deflecting the ball off the bat and destroying the bricks; and, as in *Pong*, the player would lose the game if the ball left the play area. *Breakout* was a hit for Atari, and remains one of those game ideas that has never quite faded from view; in the 1980s, Taito's *Arkanoid* updated the action with collectible power-ups, multiple stages with different layouts of bricks,

and additional enemies that disrupted the trajectory of the player's ball. *Breakout* had an impact on other genres, too; game designer Tomohiro Nishikado came up with the idea for *Space Invaders* by switching *Breakout*'s bat with a base that shot bullets, while its bricks became aliens that moved

**"Breakout replaced Pong's second player with a wall of bricks"**

and fired back at the player.

The code on the right, written by Daniel Pope, shows you just how easy it is to get a basic version of *Breakout* up and running in Python, using the Pygame Zero library. Like Atari's original, it draws a wall of blocks on the screen, sets a ball bouncing around, and gives the player a paddle, which can be controlled by moving the mouse left and right. The ball physics are relatively simple to grasp, too.

The ball has a velocity, *vel* – which is a vector, or a pair of numbers: *vx* for the x direction and *vy* for the y direction. The program loop checks the position of the ball and whether it's collided with a brick or the edge of the play area. If the ball hits the left side of the play area, the ball's x velocity *vx* is set to positive, thus sending it bouncing to the right. If it hits the right side, it's set to a negative number, so it moves left. Likewise when the ball hits the top or bottom of a brick, we set the sign of the y velocity *vy*, and so on for the collisions with the bat, and the top of the play area and the sides of bricks. Collisions set the sign of *vx* and *vy* but never change the magnitude. This is called a perfectly elastic collision.

To this basic framework, you could add all kinds of additional features: a 2012 talk by developers Martin Jonasson and Petri Purho, which you can watch on YouTube ([wfmag.cc/breakout](http://wfmag.cc/breakout)), shows how the *Breakout* concept can be given new life with the addition of a few modern design ideas. 🎮





Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag11](https://wfmag.cc/wfmag11)

# Bricks and balls in Python

Courtesy of Daniel Pope, here's a simple *Breakout* game written in Python. To get it running on your system, you'll first need to install Pygame Zero – you can find full instructions at [wfmag.cc/pgzero](https://wfmag.cc/pgzero)

```
import random
import colorsys
from math import copysign

WIDTH = 600
HEIGHT = 800
BALL_SIZE = 10
MARGIN = 50

BRICKS_X = 10
BRICKS_Y = 5
BRICK_W = (WIDTH - 2 * MARGIN) //
BRICKS_X
BRICK_H = 25

ball = ZRect(WIDTH / 2, HEIGHT / 2,
BALL_SIZE, BALL_SIZE)
bat = ZRect(WIDTH / 2, HEIGHT - 50, 80,
12)

bricks = []

def hsv_color(h, s, v):
    """Return an RGB color from HSV."""
    r, g, b = colorsys.hsv_to_rgb(h, s,
v)
    return r * 255, g * 255, b * 255

def reset():
    """Reset bricks and ball."""
    # First, let's do bricks
    del bricks[:]
    for x in range(BRICKS_X):
        for y in range(BRICKS_Y):
            brick = ZRect(
                (x * BRICK_W + MARGIN, y
* BRICK_H + MARGIN),
                (BRICK_W - 1, BRICK_H
- 1)
            )
            hue = (x + y) / BRICKS_X
            saturation = (y / BRICKS_Y)
* 0.5 + 0.5
            brick.highlight = hsv_
color(hue, saturation * 0.7, 1.0)
            brick.color = hsv_color(hue,
saturation, 0.8)
            bricks.append(brick)
```

```
# Now reset the ball
ball.center = (WIDTH / 2, HEIGHT /
2)
ball.vel = (random.uniform(-200,
200), 400)

# Reset bricks and ball at start
reset()

def draw():
    screen.clear()
    for brick in bricks:
        screen.draw.filled_rect(brick,
brick.color)
        screen.draw.line(brick.
bottomleft, brick.topleft, brick.
highlight)
        screen.draw.line(brick.topleft,
brick.topright, brick.highlight)

        screen.draw.filled_rect(bat, 'pink')
        screen.draw.filled_circle(ball.
center, BALL_SIZE // 2, 'white')

def update():
    # When you have fast moving objects,
like the ball, a good trick
    # is to run the update step several
times per frame with tiny time steps.
    # This makes it more likely that
collisions will be handled correctly.
    for _ in range(3):
        update_step(1 / 180)

def update_step(dt):
    x, y = ball.center
    vx, vy = ball.vel

    if ball.top > HEIGHT:
        reset()
        return

    # Update ball based on previous
velocity
    x += vx * dt
    y += vy * dt
    ball.center = (x, y)
```

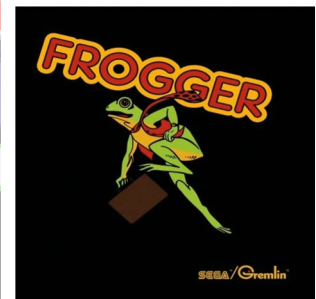
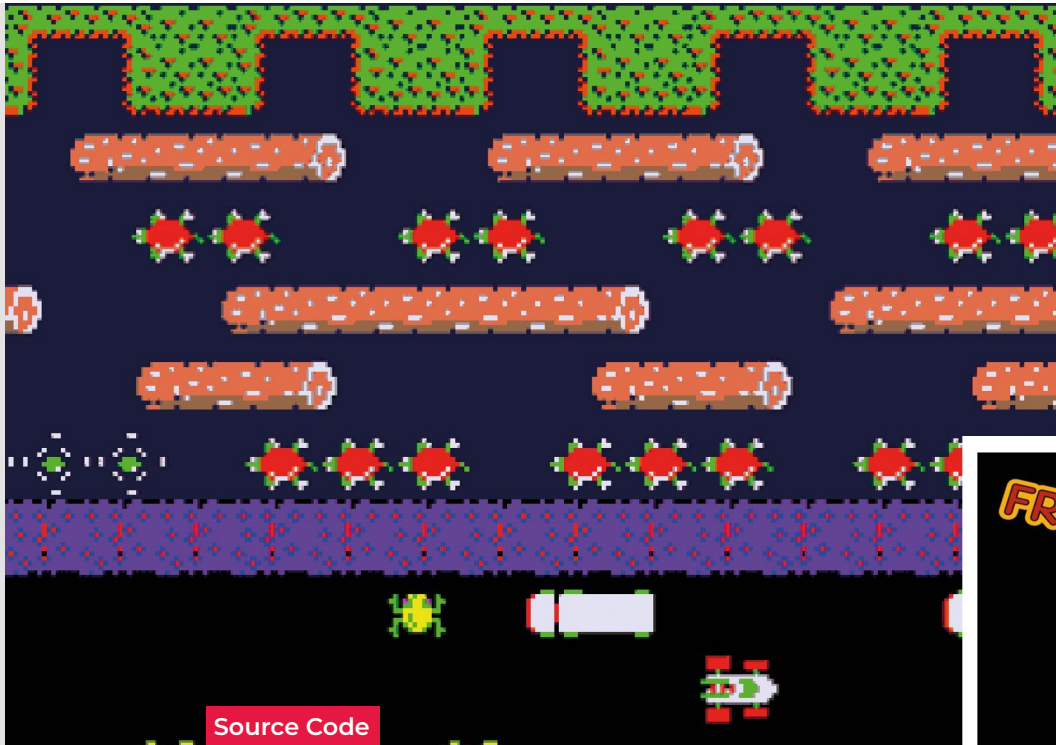
```
# Check for and resolve collisions
if ball.left < 0:
    vx = abs(vx)
    ball.left = -ball.left
elif ball.right > WIDTH:
    vx = -abs(vx)
    ball.right = 2 * (ball.right -
WIDTH)

if ball.top < 0:
    vy = abs(vy)
    ball.top = -1

if ball.colliderect(bat):
    vy = -abs(vy)
    # randomise the x velocity but
keep the sign
    vx = copysign(random.uniform(50,
300), vx)
else:
    # Find first collision
    idx = ball.collidelist(bricks)
    if idx != -1:
        brick = bricks[idx]
        # Work out what side we
collided on
        dx = (ball.centerx - brick.
centerx) / BRICK_W
        dy = (ball.centery - brick.
centery) / BRICK_H
        if abs(dx) > abs(dy):
            vx = copysign(abs(vx), dx)
        else:
            vy = copysign(abs(vy), dy)
        del bricks[idx]

# Write back updated position and
velocity
ball.center = (x, y)
ball.vel = (vx, vy)

def on_mouse_move(pos):
    x, y = pos
    bat.centerx = x
    if bat.left < 0:
        bat.left = 0
    elif bat.right > WIDTH:
        bat.right = WIDTH
```



## Code a Frogger-style road-crossing game

Save the frog from busy roads and rivers with a simple remake of Konami's classic arcade game



AUTHOR  
RIK CROSS

**W**hy did the frog cross the road? Because *Frogger* would be a boring game if it didn't. Released in 1981 by Konami, the game appeared in assorted bars, sports halls, and arcades across the world, and became an instant hit. The concept was simple: players used the joystick to move a succession of frogs from the bottom of the screen to the

top, avoiding a variety of hazards – cars, lorries, and later, the occasional crocodile. Each frog had to be safely manoeuvred to one of five alcoves within a time limit, while extra points were awarded for eating flies along the way.

Before *Frogger*, Konami mainly focused

on churning out clones of other hit arcade games like *Space Invaders* and *Breakout*; *Frogger* was one of its earliest original ideas, and the simplicity of its concept saw it ported to just about every home system available at the time. (Ironically, Konami's game would

**“We can recreate *Frogger* in just a few lines of Pygame Zero code”**

fall victim to repeated cloning by other developers.) Decades later, developers still take inspiration from it; Hipster Whale's *Crossy Road* turned *Frogger* into an endless running game; earlier this year, Konami returned to the creative well with *Frogger in Toy Town*, released on Apple Arcade.

We can recreate much of *Frogger*'s gameplay in just a few lines of Pygame Zero code. The key elements are the frog's movement, which use the arrow keys, vehicles that move across the screen, and

floating objects – logs and turtles – moving in opposite directions. Our background graphic will provide the road, river, and grass for our frog to move over. The frog's movement will be triggered from an `on_key_down()` function, and as the frog moves, we switch to a second frame with legs outstretched, reverting back to a sitting position after a short delay. We can use the inbuilt Actor properties to change the image and set the angle of rotation.

For all the other moving elements, we can also use Pygame Zero Actors; we just need to make an array for our cars with different graphics for the various rows, and an array for our floating objects in the same way.

In our `update()` function, we need to move each Actor according to which row it's in, and when an Actor disappears off the screen, set the x coordinate so that it reappears on the opposite side. Handling the logic of the frog moving across the road is quite easy; we just check for collision with each of the





Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag27](https://wfmag.cc/wfmag27)

# Frogger in Python

Here's Mark's code snippet, which recreates *Frogger* in Python. To get it running on your system, you'll first need to install Pygame Zero -- you can find full instructions at [wfmag.cc/pgzero](https://wfmag.cc/pgzero)

```
frog = Actor('frog1', center=(400, 580))
frog.direction = frog.delay = 0
frog.onBoard = -1
cars = []
floats = []
gameState = count = 0
for r in range(0, 6):
    for c in range(0, 4):
        cars.append(Actor('car'+str(r+1),
            center=((r*20)+(c*(240-(r*10))), 540-(r*40))))
        if r < 5: floats.append(Actor('float'+str(r+1),
            center=((r*20)+(c*(240-(r*10))), 260-(r*40))))

def draw():
    global count
    screen.blit("background", (0, 0))
    for c in range(0, 20):
        floats[c].draw()
    if gameState == 0 or (gameState == 1 and count%2 == 0):
        frog.draw()
    for c in range(0, 24):
        cars[c].draw()
    count += 1

def update():
    global gameState
    if gameState == 0:
        frog.onBoard = -1
        for r in range(0, 6):
            s = -1
            if r%2 == 0: s = 1
            for c in range(0, 4):
                i = (r*4)+c
```

```
cars[i].x += s
if cars[i].x > 840: cars[i].x = -40
if cars[i].x < -40: cars[i].x = 840
if cars[i].colliderect(frog): gameState = 1
if r < 5:
    floats[i].x -= s
    if floats[i].x > 880: floats[i].x = -80
    if floats[i].x < -80: floats[i].x = 880
    if floats[i].colliderect(frog):
        frog.onBoard = i
        frog.x -= s

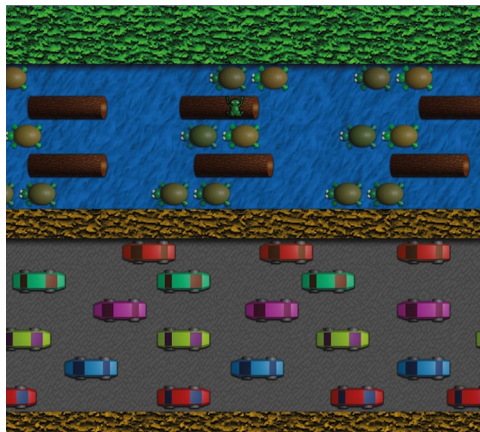
if frog.delay > 0:
    frog.delay += 1
    if frog.delay > 10:
        frog.image = "frog1"
        frog.angle = frog.direction
    if frog.y > 60 and frog.y < 270 and frog.onBoard ==
-1: gameState = 1

def on_key_down(key):
    if gameState == 0:
        if key.name == "UP": frogMove(0,-40,0)
        if key.name == "DOWN": frogMove(0,40,180)
        if key.name == "LEFT": frogMove(-40,0,90)
        if key.name == "RIGHT": frogMove(40,0,270)

def frogMove(x,y,d):
    if 800 > frog.x+x > 0: frog.x += x
    if 600 > frog.y+y > 0: frog.y += y
    frog.image = "frog2"
    frog.delay = 1
    frog.angle = frog.direction = d
```

cars, and if the frog hits a car, then we have a squashed frog. The river crossing is a little more complicated. Each time the frog moves on the river, we need to make sure that it's on a floating Actor. We therefore check to make sure that the frog is in collision with one of the floating elements, otherwise it's game over.

There are lots of other elements you could add to the example shown here: the original arcade game provided several frogs to guide to their alcoves on the other side of the river, while crocodiles also popped up from time to time to add a bit more danger. Pygame Zero has all the tools you need to make a fully functional version of Konami's hit. 🐸

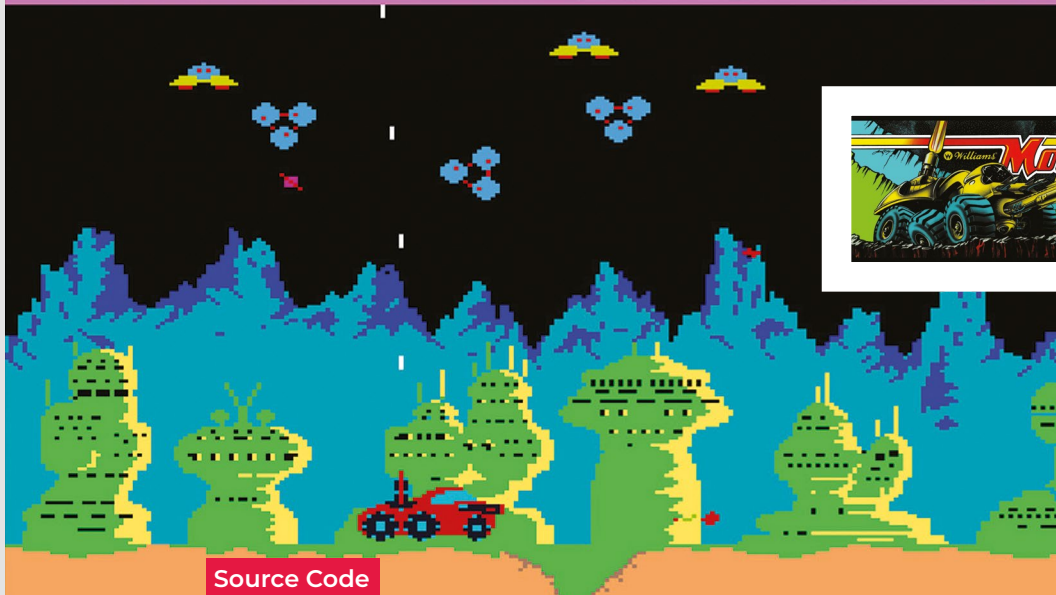


## Amphibious machines

The *Frogger* arcade machine was groundbreaking in that it used two CPUs: a pair of Z80 processors, one to run the main game and one to handle the sound. Konami used the same Z80 CPU in many of its other arcade hits in the early eighties, including Konami's *Ping Pong*, *Time Pilot*, and the surreal pigs-versus-wolves action game, *Pooyan*. Along with *Scramble*, also released in 1981, *Frogger* was by far Konami's most influential game of the period, though, with its ports selling an estimated 20 million units across all systems.

★ 017500 - Y POINT G  
1P - 001450 TIME 061

→ E T D T Z



Source Code

# Moon Patrol's illusion of depth

Classic games like Moon Patrol used parallax scrolling to create a sense of depth. Here's how you can recreate the effect for yourself



**AUTHOR**  
**RIK CROSS**

**P**arallax scrolling is an effect that can be used to give the illusion of depth in a two-dimensional world. Often in games, player motion is simulated by keeping the player in a fixed position on the screen, with platforms, backgrounds, enemies, and other objects moving across the screen, relative to the stationary player. This gives a 'window' on the game world, with the player taking centre stage.

A parallax scrolling effect is achieved by moving an object at a speed dependent on its distance from the player. Background objects (for example, clouds or distant mountains) move across the screen more slowly than objects in the foreground (such as platforms and trees). The quicker an object moves across the screen, the closer to the 'window' the object is perceived to be.

You can see this effect for yourself if you

look out of a window and move your head horizontally from side to side (you'll just have to make sure that no one is watching you!). Nearby objects will move across your field of vision more quickly than objects further away.

The effect was first used in computer

**"The idea predates computer games, and is borrowed from the 'multiplane camera', an invention used by Disney"**

games in the early 1980s, in titles such as *Moon Patrol* and *Jungle Hunt*, and in many games since. You'll notice the effect in classics such as *Super Mario World* and *Sonic the Hedgehog*, and it was used to create a creepy atmosphere in the more modern *Limbo*. It's not just used in platformers, either: nearby stars and distant galaxies have been used to create the same parallax

effect in countless space shooters.

In fact, the idea predates computer games, and is borrowed from the 'multiplane camera', an invention used by Disney and others to film cartoons in the 1930s. This achieved the effect by moving multiple layers of artwork past the camera at different speeds and distances.

To create the parallax scrolling effect for yourself, you'll first need to decide how to break up your background into a number of separate image layers that will move independently from each other.

In my example, I'll use three layers for far, mid, and near mountain ranges, but the technique will work for any number of layers – how many of them you'll need will depend on the level of detail you're aiming for.

Each layer should consist of a duplicated image, giving a layer that is exactly twice





Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag3](https://wfmag.cc/wfmag3)

## PARALLAX SCROLLING in PYTHON

Here's a parallax scrolling effect that uses three layers to create a moving background. To get it running on your system, you'll first need to install Pygame Zero – you can find full instructions at [wfmag.cc/XVzieD](https://wfmag.cc/XVzieD)

```
# set screen width and height
WIDTH = 800
HEIGHT = 400

# create the back layer
layer_back = Actor('image_back')
layer_back.topleft = 0, 0
layer_back.speed = 1

#create the middle layer
layer_middle = Actor('image_middle')
layer_middle.topleft = 0, 0
layer_middle.speed = 3

#create the front layer
layer_front = Actor('image_front')
layer_front.topleft = 0, 0
layer_front.speed = 5
```

```
#add layers to list
layers = [layer_back, layer_middle, layer_front]

def update():
    for l in layers:
        # move each layer to the left
        l.left -= l.speed
        # if the layer has moved far enough to the left
        # then reset the layers position
        if l.right <= WIDTH:
            l.left = 0

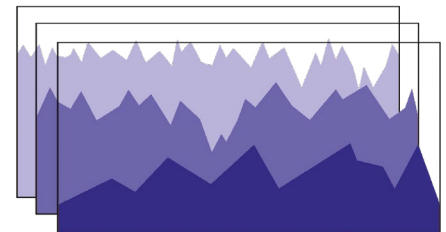
def draw():
    screen.clear()
    # draw all images in the image list
    for l in layers:
        l.draw()
```

the width of the screen. The images should contain some transparency, so that the movement of other layers behind can be seen. To avoid a 'seam' in my mountain layers, I've also made sure that the mountain height is the same at the left and right side of the image. If you find creating the layer images difficult, you can either use the images I've created for my example, or go to an open-source media repository like Open Game Art ([opengameart.org](https://opengameart.org)) and search for 'parallax' – you'll find lots of great ready-made layer sets to use.

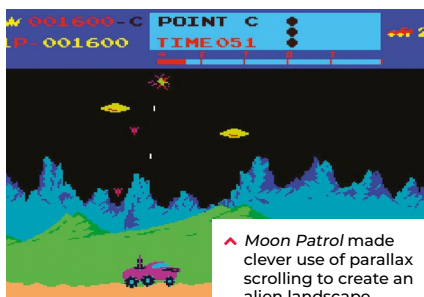
Each layer is given its own speed, which will be used when updating its position. You can tinker with the numbers; all that's important is that the speed of a layer is higher than the layers behind it. Layers start

with their left edge at the left of the screen, and every frame, layers are moved to the left according to their speed. Once the right hand edge of a layer's image has reached the right edge of the screen, it is reset to its original starting position.

Layers are then drawn to the screen in order, from back to front. And that's all there is to it! Once you've achieved this basic effect, you can play with the code and experiment with different images to see what you can create. 🎮

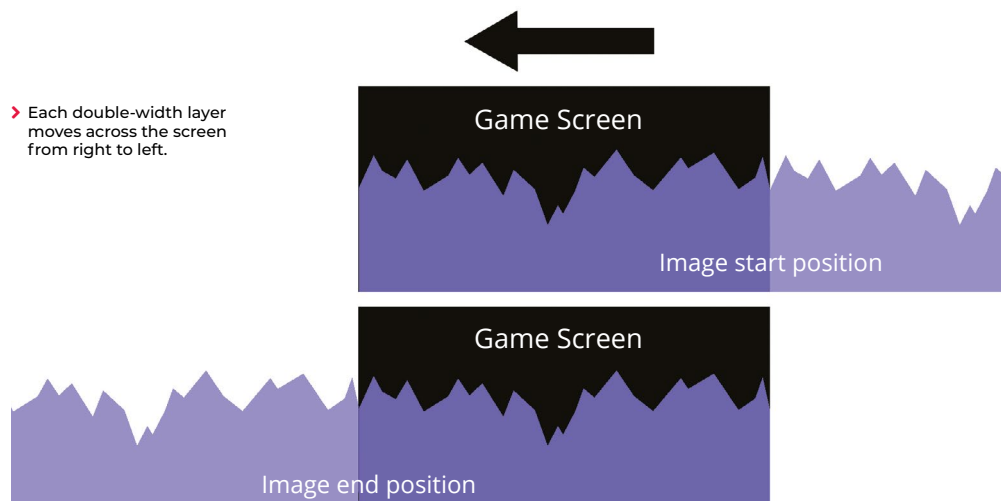


▲ The background is broken up into a number of separate layers that move independently from each other.



▲ Moon Patrol made clever use of parallax scrolling to create an alien landscape.

➤ Each double-width layer moves across the screen from right to left.



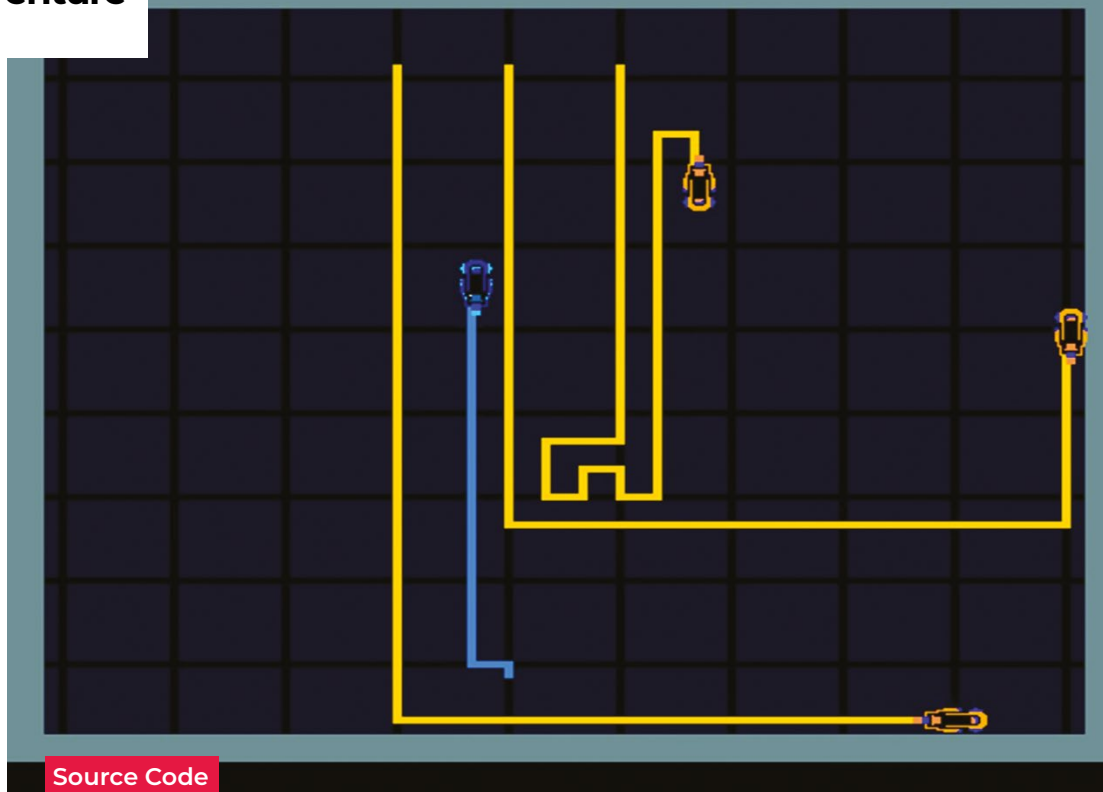
## Action/adventure

Tron

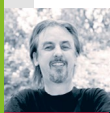


^ The *TRON* cab had two game controllers: a rotary wheel and a joystick.

> Battle against AI enemies in the original arcade classic.



Source Code



AUTHOR  
MARK VANSTONE

# Code a Light Cycle arcade minigame

Speed around an arena, avoiding walls and deadly trails

**A**t the beginning of the 1980s, Disney made plans for an entirely new kind of animated movie that used cutting-edge computer graphics. The resulting film was 1982's *TRON*, and it inevitably sparked one of the earliest tie-in arcade machines. The game featured several minigames, including one based on

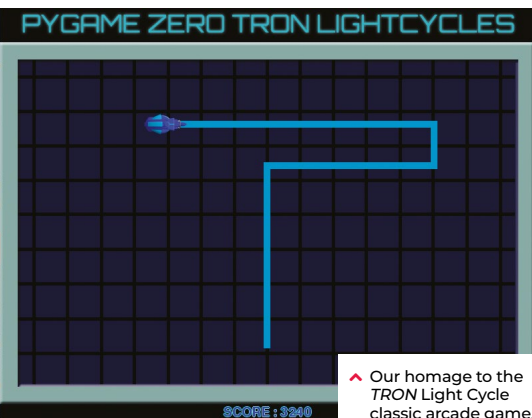
the Light Cycle section of the movie, where players speed around an arena on high-tech motorbikes, which leave a deadly trail of light in their wake. If competitors hit any walls or cross the path of any trails, then it's game over. Players progress through the twelve levels which were all named after programming languages. In the Light Cycle game, the players compete against AI players who drive yellow Light Cycles around the arena. As the levels progress, more AI Players are added.

The *TRON* game, distributed by Bally Midway, was well-received in arcades, and even won Electronic Games Magazine's (presumably) coveted Coin-operated Game of the Year gong. Although the arcade game wasn't ported to home computers at the time, several similar games – and outright clones – emerged, such as the unsightly named *Light Cycle* for the BBC Micro, Oric, and ZX Spectrum.

The *Light Cycle* minigame is essentially a variation on *Snake*, with the player leaving a trail behind them as they move around the

screen. There are various ways to code this with Pygame Zero. In this sample, we'll focus on the movement of the player Light Cycle and creating the trails that are left behind as it moves around the screen. We could use line drawing functions for the trail behind the bike, or go for a system like *Snake*, where blocks are added to the trail as the player moves. In this example, though, we're going to use a two-dimensional list as a matrix of positions on the screen. This means that wherever the player moves on the screen, we can set the position as visited or check to see if it's been visited before and, if so, trigger an end-game event.

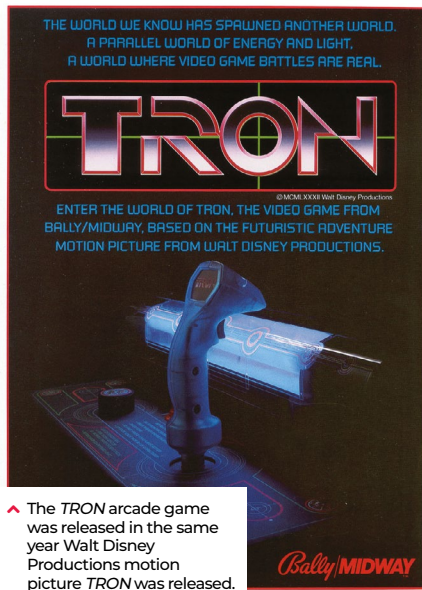
For the main `draw()` function, we first blit our background image which is the cross-hatched arena, then we iterate through our two-dimensional list of screen positions (each 10 pixels square) displaying a square anywhere the Cycle has been. The Cycle is then drawn and we can add a display of the score. The `update()` function contains code to move the Cycle and check for collisions. We use a list of directions in degrees to control



^ Our homage to the *TRON* Light Cycle classic arcade game.



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag47](https://wfmag.cc/wfmag47)



^ The TRON arcade game was released in the same year Walt Disney Productions motion picture TRON was released.

the angle the player is pointing, and another list of x and y increments for each direction. Each update we add x and y coordinates to the Cycle actor to move it in the direction that it's pointing multiplied by our speed variable. We have an `on_key_down()` function defined to handle changing the direction of the Cycle actor with the arrow keys.

We need to wait a while before checking for collisions on the current position, as the Cycle won't have moved away for several updates, so each screen position in the matrix is actually a counter of how many updates it's been there for. We can then test to see if 15 updates have happened before testing the square for collisions, which gives our Cycle enough time to clear the area. If we do detect a collision, then we can start the game-end sequence. We set the `gamestate` variable to 1, which then means the `update()` function uses that variable as a counter to run through the frames of animation for the Cycle's explosion. Once it reaches the end of the sequence, the game stops. We have a key press defined (the `SPACE` bar) in the `on_key_down()` function to call our `init()` function, which will not only set up variables when the game starts but sets things back to their starting state.

So that's the fundamentals of the player Light Cycle movement and collision checking. To make it more like the original arcade game, why not try experimenting with the code and adding a few computer-controlled rivals? 🤖

## Light Cycles in Python

Here's Mark's code for a Light Cycle minigame straight out of *TRON*. To get it working on your system, you'll need to install Pygame Zero – full instructions are available at [wfmag.cc/pgzero](https://wfmag.cc/pgzero).

```
# TRON

speed = 3
dirs = [0,90,180,270]
moves = [(0,-1),(-1,0),(0,1),(1,0)]

def draw():
    screen.blit("background", (0, 0))
    for x in range(0, 79):
        for y in range(0, 59):
            if matrix[x][y] > 0:
                matrix[x][y] += 1
                screen.blit("dot", ((x*10)-5, (y*10)-5))
    bike.draw()
    screen.draw.text("SCORE : " + str(score), center=(400, 588), owidth=0.5,
        ocolor=(0,255,255), color=(0,0,255), fontsize=28)

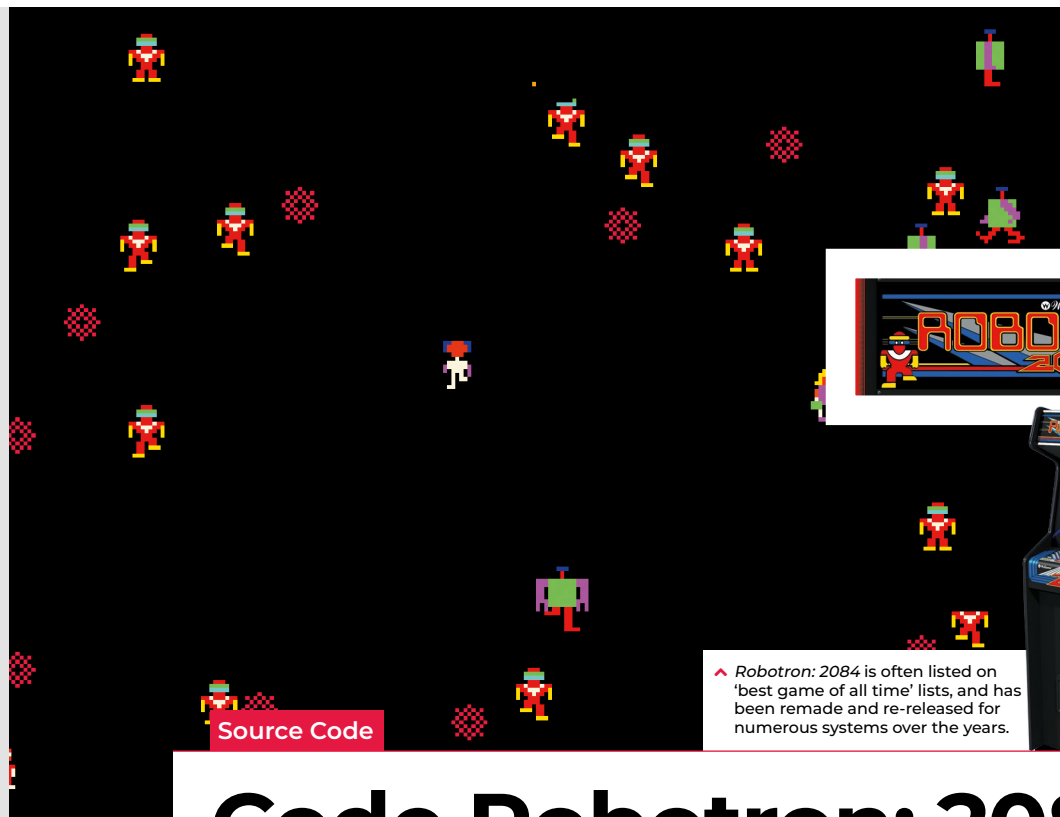
def update():
    global matrix, gamestate, score
    if gamestate == 0:
        bike.angle = dirs[bike.direction]
        bike.x += moves[bike.direction][0]*speed
        bike.y += moves[bike.direction][1]*speed
        score += 10
        if matrix[int(bike.x/10)][int(bike.y/10)] < 15 :
            matrix[int(bike.x/10)][int(bike.y/10)] += 1
        else:
            gamestate = 1
            if bike.x < 60 or bike.x > 750 or bike.y < 110 or bike.y > 525:
                gamestate = 1
    else:
        if gamestate < 18:
            bike.image = "bike"+str(int(gamestate/2))
            bike.angle = dirs[bike.direction]
            gamestate += 1

def on_key_down(key):
    if key == keys.LEFT:
        bike.direction += 1
        snapBike()
        if bike.direction == 4 : bike.direction = 0
    if key == keys.RIGHT:
        bike.direction -= 1
        snapBike()
        if bike.direction == -1 : bike.direction = 3
    if key == keys.SPACE and gamestate == 18:
        init()

def snapBike():
    bike.x = int(bike.x/10)*10
    bike.y = int(bike.y/10)*10

def init():
    global bike, matrix, gamestate, score
    bike = Actor('bike1', center=(400, 500))
    bike.direction = 0
    matrix = [[0 for y in range(60)] for x in range(80)]
    gamestate = score = 0

init()
```



[Source Code](#)

^ Robotron: 2084 is often listed on 'best game of all time' lists, and has been remade and re-released for numerous systems over the years.



AUTHOR  
MAC BOWLEY

# Code Robotron: 2084's twin-stick action

Move in one direction and fire in another with Mac's re-creation of an arcade classic

**R**eleased back in 1982, *Robotron: 2084* popularised the concept of the twin-stick shooter. It gave players two joysticks which allowed them to move in one direction while also shooting at enemies in another. Here, I'll show you how to recreate those controls using Python and Pygame. We don't have access to any sticks, only a keyboard, so we'll be using the arrow keys for movement and **WASD** to control the direction of fire.

The movement controls use a **global** variable, a few **if** statements, and two built-in Pygame functions: **on\_key\_down** and **on\_key\_up**. The **on\_key\_down** function is called when a key on the keyboard is pressed, so when the player presses the right arrow key, for example, I set the x direction of the player to be a positive 1. Instead of setting the movement to 1, instead, I'll add 1 to the direction. The **on\_key\_down** function is called when a button's released. A key being

released means the player doesn't want to travel in that direction anymore and so we should do the opposite of what we did earlier – we take away the 1 or -1 we applied in the **on\_key\_up** function.

We repeat this process for each arrow key. Moving the player in the **update()** function is the last part of my movement; I apply a move speed and then use a **playArea** rect to clamp the player's position.

Now for the aiming and rotating. When my player aims, I want them to set the direction the bullets will fire, which functions like the movement. The difference this time is that when a player hits an aiming key, I set the direction directly rather than adjusting the values. If my player aims up, and then releases that key, the shooting will stop. Our next challenge is changing this direction into a rotation for the turret. Actors in Pygame can be rotated in degrees, so I have to find a way of turning a pair of x and y directions into a rotation. To do this, I use

the math module's **atan2** function to find the arc tangent of two points. The function returns a result in radians, so it needs to be converted. (You'll also notice I had to adjust mine by 90 degrees. If you want to avoid having to do this, create a sprite that faces right by default.)

To fire bullets, I'm using a flag called 'shooting' which, when set to **True**, causes my turret to turn and fire. My bullets are dictionaries; I could have used a class, but the only thing I need to keep track of is an actor and the bullet's direction.

You can look at the **update** function and see how I've implemented a fire rate for the turret as well. You can edit the **update** function to take a single parameter, **dt**, which stores the time since the last frame. By adding these up, you can trigger a bullet at precise intervals and then reset the timer.

This code is just a start – you could add enemies and maybe other player weapons to make a complete shooting experience. 🎮





Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag38](https://wfmag.cc/wfmag38)

# Twin-stick shooting in Python

Here's Mac's code snippet. To get it running on your system, you'll need to install Pygame Zero – you can find full instructions at [wfmag.cc/pgzero](https://wfmag.cc/pgzero).

```
import pygame as pg
import math

WIDTH = 860
HEIGHT = 540

bg = pg.image.load("images/arena.png").
convert()
play_area = Rect((150, 75), (560, 390))

player = Actor("treads.png",
center=(WIDTH//2, HEIGHT//2),
anchor=('center', 'center'))
turret = Actor("turret.png",
center=(player.x, player.y),
anchor=('center', 'center'))
pl_movement = [0, 0]
pl_move_speed = 5

pl_rotation = [0, 0]
turn_speed = 5
shooting = False
bullets = []
bullet_speed = 150
fire_rate = 0.15
fire_timer = 0

def on_key_down(key, unicode):
    global shooting

    # Movement
    if key == keys.RIGHT:
        pl_movement[0] += 1
    if key == keys.LEFT:
        pl_movement[0] += -1
    if key == keys.UP:
        pl_movement[1] += -1
    if key == keys.DOWN:
        pl_movement[1] += 1
    if key == keys.D:
        pl_rotation[0] = 1
    if key == keys.A:
        pl_rotation[0] = -1
    if key == keys.W:
        pl_rotation[1] = -1
    if key == keys.S:
        pl_rotation[1] = 1
    print(pl_rotation)

def on_key_up(key):
    global shooting
```

```
# Movement
if key == keys.RIGHT:
    pl_movement[0] = 0
if key == keys.LEFT:
    pl_movement[0] = 0
if key == keys.UP:
    pl_movement[1] = 0
if key == keys.DOWN:
    pl_movement[1] = 0
if key == keys.D:
    pl_rotation[0] = 0
if key == keys.A:
    pl_rotation[0] = 0
if key == keys.W:
    pl_rotation[1] = 0
if key == keys.S:
    pl_rotation[1] = 0

def update(dt):
    global shooting, bullets, fire_timer

    # Movement every frame
    player.x += pl_movement[0] * pl_
move_speed
    player.y += pl_movement[1] * pl_
move_speed

    # Clamp the position
    if player.y - 16 < play_area.top:
        player.y = play_area.top + 16
    elif player.y + 16 > play_area.
bottom:
        player.y = play_area.bottom - 16
    if player.x - 16 < play_area.left:
        player.x = play_area.left + 16
    elif player.x + 16 > play_area.
right:
        player.x = play_area.right - 16

    turret.pos = player.pos

    if any([keyboard[keys.W],
keyboard[keys.A], keyboard[keys.S],
keyboard[keys.D]]):
        shooting = True
    else:
        shooting = False
        fire_timer = fire_rate

    if shooting == True:

        # Rotate the turret
        desired_angle = (math.atan2(-
```

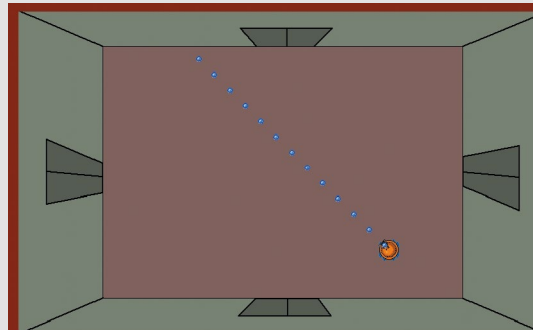
```
pl_rotation[1], pl_rotation[0]) / (math.
pi/180)) - 90
        turret.angle = desired_angle
        fire_timer += dt
        if fire_timer > fire_rate:
            bullet = {}
            bullet["actor"] =
Actor("bullet.png", center=player.pos,
anchor=('center', 'center'))
            bullet["direction"] = pl_
rotation.copy()
            bullet["actor"].x += pl_
rotation[0] * 4
            bullet["actor"].y += pl_
rotation[1] * 4
            bullets.append(bullet)
            fire_timer = 0

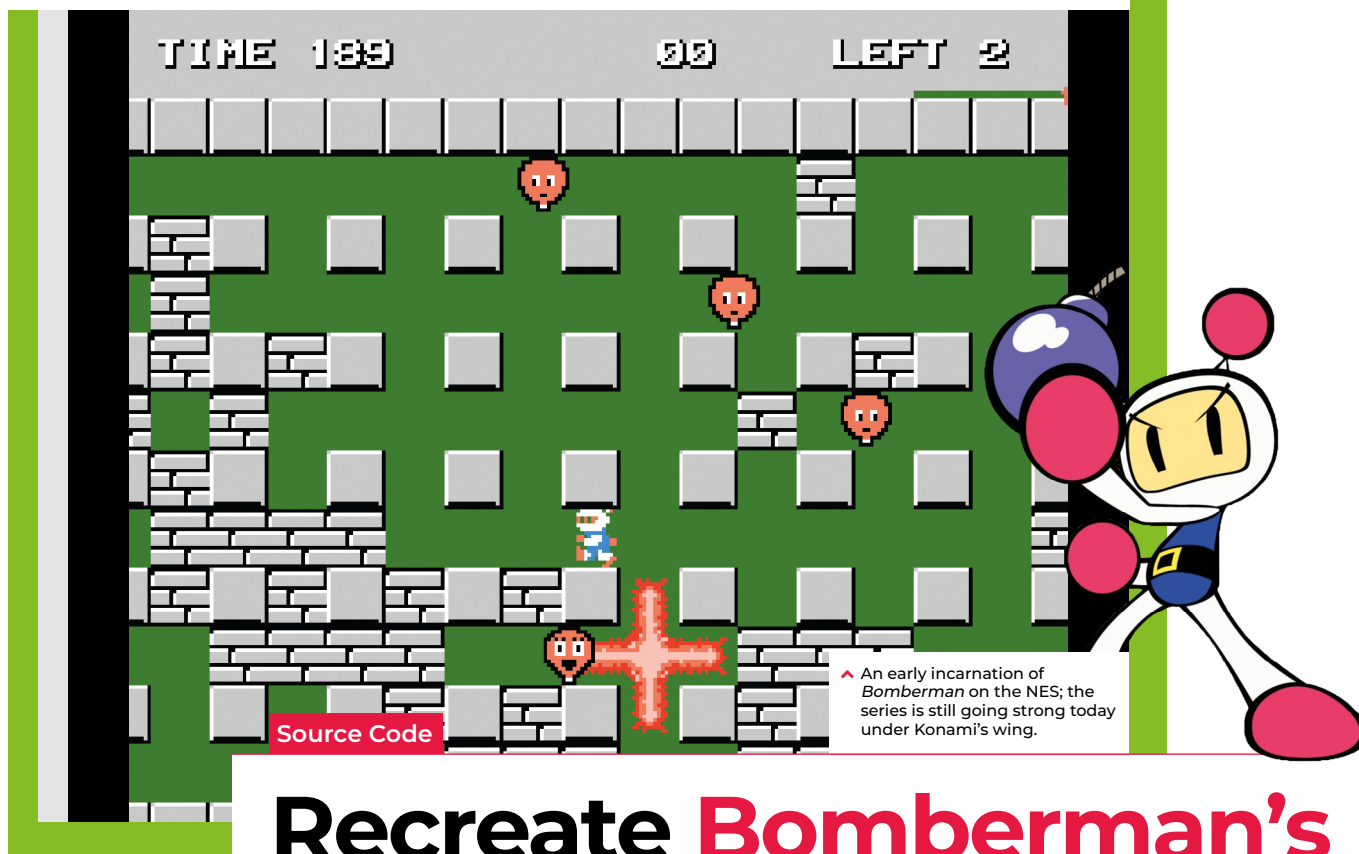
        bullets_to_remove = []
        for b in bullets:
            b["actor"].x += b["direction"]
[0] * bullet_speed * dt
            b["actor"].y += b["direction"]
[1] * bullet_speed * dt
            if not b["actor"].
colliderect(play_area):
                bullets_to_remove.append(b)

        for b in bullets_to_remove:
            bullets.remove(b)

def draw():
    screen.blit(bg, (0, 0))
    player.draw()
    turret.draw()
    for b in bullets:
        b["actor"].draw()
```

✓ The arena background and tank sprites were created in Piskel ([piskelapp.com](https://piskelapp.com)). Separate sprites for the tank allow the turret to rotate separately from the tracks.





# Recreate Bomberman's iconic explosives

Learn how to recreate the exploding bombs found in the classic Bomberman games



AUTHOR  
RIK CROSS

**B**omberman was first released in the early 1980s as a tech demo for a BASIC compiler, but soon became a popular series that's still going today. *Bomberman* sees players use bombs to destroy enemies and uncover doors behind destructible tiles. In this article, I'll show you how to recreate the bombs that explode in four directions, destroying parts of the level as well as any players in their path!

The game level is a tilemap stored as a two-dimensional array. Each tile in the map is a **Tile** object, which contains the tile type, and corresponding image. For simplicity, a tile can be set to one of five types; **GROUND**, **WALL**, **BRICK**, **BOMB**, or **EXPLOSION**. In this example code, **BRICK** and **GROUND** can be exploded with bombs, but **WALL** cannot, but of course, this behaviour can be changed.

Each **Tile** object also has a timer, which is decremented each frame of the game. When a tile's timer reaches 0, an action

is carried out, which is dependent on the tile type. **BOMB** tiles (and surrounding tiles) turn into **EXPLOSION** tiles after a short delay, and **EXPLOSION** tiles eventually turn back into **GROUND**. At the start of the game, the tilemap for the level is generated, in this case consisting of mostly **GROUND**, with some **WALL** and a couple of **BRICK** tiles. The player

**"I'll show you how to recreate the bombs that explode in four directions"**

starts off in the top-left tile, and moves by using the arrow keys. Pressing the **SPACE** key will place a bomb in the player's current tile, which is achieved by setting the **Tile** at the player's position to **BOMB**. The tile's timer is also set to a small number, and once this timer is decremented to 0, the bomb tile and the tiles around it are set to **EXPLOSION**. The bomb explodes outwards in four directions,

with a range determined by the **RANGE**, which in our code is 3. As the bomb explodes out to the right, for example, the tile to the right of the bomb is checked. If such a tile exists (i.e. the position isn't out of the level bounds) and can be exploded, then the tile's type is set to **EXPLOSION** and the next tile to the right is checked. If the explosion moves out of the level bounds, or hits a **WALL** tile, then the explosion will stop radiating in that direction. This process is then repeated for the other directions.

There's a nice trick for exploding the bomb without repeating the code four times, and it relies on the sine and cosine values for the four direction angles. The angles are 0° (up), 90° (right), 180° (down) and 270° (left). When exploding to the right (at an angle of 90°),  $\sin(90)$  is 1 and  $\cos(90)$  is 0, which corresponds to the offset direction on the x- and y-axis respectively. These values can be multiplied by the tile offset, to explode the bomb in all four directions. 🌐



# Bomberman bombs in Python

Here's Rik's example code, which recreates *Bomberman*'s bombs and four-way explosions in Python. To get it running on your system, you'll first need to install Pygame Zero – you can find full instructions at [wfmag.cc/pgzero](http://wfmag.cc/pgzero)

```
from math import cos, sin, radians

SIZE = 9
WIDTH = SIZE*45 - 5
HEIGHT = SIZE*45 - 5

# bomb range
RANGE = 3

GROUND = 0
WALL = 1
BRICK = 2
BOMB = 3
EXPLOSION = 4
# images for tile types
images = ['ground', 'wall', 'brick', 'bomb', 'explosion']

player = Actor('player')
player.mapx = 0
player.mapy = 0

# each position in tilemap is a 'Tile' with type, image, timer
class Tile():
    def __init__(self, type):
        self.set(type)
    def set(self, type):
        self.timer = 0
        self.t = type
        self.i = images[type]

tilemap = [[Tile(WALL) if x%2==1 and y%2==1 else Tile(GROUND)
for y in range(10)] for x in range(10)]
tilemap[3][2].set(BRICK)
tilemap[4][7].set(BRICK)

def on_key_down():

    newx = player.mapx
    newy = player.mapy

    if keyboard.left and player.mapx > 0:
        newx -= 1
    elif keyboard.right and player.mapx < SIZE-1:
        newx += 1
    elif keyboard.up and player.mapy > 0:
        newy -= 1
    elif keyboard.down and player.mapy < SIZE-1:
        newy += 1

    if tilemap[newx][newy].t in [GROUND, EXPLOSION]:
        player.mapx = newx
        player.mapy = newy

# space key to place bomb
if keyboard.space:
    tilemap[player.mapx][player.mapy].set(BOMB)
    tilemap[player.mapx][player.mapy].timer = 150

def update():

    for x in range(SIZE):
        for y in range(SIZE):

            tile = tilemap[x][y]

            # decrement timer
            if tile.timer > 0:
                tile.timer -= 1

            # process tile types on timer finish
            if tile.timer <= 0:

                # explosions eventually become ground
                if tile.t == EXPLOSION:
                    tile.set(GROUND)

                # bombs eventually create explosions
                if tile.t == BOMB:
                    # bombs radiate out in all 4 directions
                    for angle in range(0, 360, 90):
                        cosa = int(cos(radians(angle)))
                        sina = int(sin(radians(angle)))
                        # RANGE determines bomb reach
                        for ran in range(1, RANGE):
                            xoffset = ran*cosa
                            yoffset = ran*sina
                            if x+xoffset in range(0, SIZE) and \
                                y+yoffset in range(0, SIZE) and \
                                tilemap[x+xoffset][y+yoffset].t in
[GROUND, BRICK]:
                                tilemap[x+xoffset][y+yoffset].set(EXPLOSION)
                                tilemap[x+xoffset][y+yoffset].timer = 50
                            else:
                                break

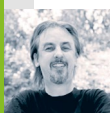
                    # remove bomb
                    tile.set(EXPLOSION)
                    tile.timer = 50

def draw():
    for x in range(SIZE):
        for y in range(SIZE):
            screen.blit( tilemap[x][y].i, (x*45, y*45) )
    # draw the player
    screen.blit( player.image, (player.mapx*45, player.mapy*45)
```



- ^ Although released to tie in with Jackie Chan's *Spartan X*, *Kung-Fu Master* was originally inspired by the Bruce Lee film, *Game of Death*.
- < Thomas battles his way through the martial arts mob to rescue his girlfriend, Sylvia.

## Code a **Kung-Fu Master** style beat-'em-up



AUTHOR  
MARK VANSTONE

Punch and kick your way through a rabble of bad dudes in a simple scrolling beat-'em-up

**K**ung-Fu Master hit arcades in 1984. Its side-scrolling action, punching and kicking through an army of knife-throwing goons, helped create the beat-'em-up genre. In fact, its designer, Takashi Nishiyama, would go on to kickstart the *Street Fighter* series at Capcom, and later start up the *Fatal Fury* franchise at SNK.

In true eighties arcade style, *Kung-Fu Master* distills the elements of a chop-socky action film to its essentials. Hero Thomas and his girlfriend are attacked, she's kidnapped, and Thomas fights his way through successive levels of bad guys to rescue her. The screen scrolls from side to side, and Thomas must use his kicks and punches to get from one side of the level to the other and climb the stairs to the next floor of the building.

To recreate this classic with Pygame Zero, we'll need quite a few frames of animation, both for the hero character and the enemies he'll battle. For a reasonable walk cycle, we'll

need at least six frames in each direction. Any fewer than six won't look convincing, but more frames can achieve a smoother effect. For this example, I've used the 3D package Poser, since it has a handy walk designer which makes generating sequences of animation much easier.

**"For a walk cycle animation, we'll need at least six frames in each direction"**

Once we have the animation frames for our characters, including a punch, kick, and any others you want to add, we need a background for the characters to walk along. The image we're using is 2000×400 pixels, and we start the game by displaying the central part so our hero can walk either way. By detecting arrow key presses, the hero can 'walk' one way or the other by moving the background left and right, while

cycling through the walk animation frames. Then if we detect a **Q** key press, we change the action string to kick; if it's **A**, it's punch. Then in our `update()` function, we use that action to set the Actor's image to the indicated action frame.

Our enemy Actors will constantly walk towards the centre of the screen, and we can cycle through their walking frames the same way we do with the main hero. To give kicks and punches an effect, we put in collision checks. If the hero strikes while an enemy collides with him, we register a hit. This could be made more precise to require more skill, but once a strike's registered, we can switch the enemy to a different status that will cause them to fall downwards and off the screen.

This sample is a starting point to demonstrate the basics of the beat-'em-up genre. With the addition of flying daggers, several levels, and a variety of bad guys, you'll be on your way to creating a Pygame Zero version of this classic game. 🐍





Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag32](https://wfmag.cc/wfmag32)

# A basic beat-'em-up basis

Here's Mark's code to make a classic 1980s-style scrolling beat-'em-up in Python. To get up and running you'll need to install Pygame Zero – full instructions are available at [wfmag.cc/pgzero](https://wfmag.cc/pgzero)

```
# Kung-Fu Master
import random

HEIGHT = 450
gameState = count = 0
bloke = Actor('walk1_0001', center=(400, 250))
blokeDir = "1"
backPos = -500
dudes = []
action = ""
actioncount = 0

def draw():
    screen.fill((0,0,0))
    screen.blit("background", (backPos, 30))
    screen.draw.text("Pygame Zero Kung-Fu Master", center =
(400, 15), owidth=1, ocolor=(255,0,0), color=(255,255,0) ,
fontsize=30)
    if gameState != 1 or (gameState == 1 and count%2 ==
0):bloke.draw()
    for d in dudes:
        d.draw()

def on_key_down(key):
    global action, actioncount
    actioncount = 10
    if gameState == 0:
        if key.name == "A": action = "punch"
        if key.name == "Q": action = "kick"

def update():
    global count, backPos, blokeDir, action, actioncount
    if gameState == 0:
        bloke.image = 'stand' + blokeDir
        if action == "punch": bloke.image = 'punch'+blokeDir
        if action == "kick": bloke.image = 'kick'+blokeDir
        if actioncount <= 0: action = ""
        if keyboard.left: moveBloke(3,"1")
        elif keyboard.right: moveBloke(-3,"r")
        if random.randint(0, 100) == 0: makeDude()
        updateDudes()
    count += 1
    actioncount -= 1

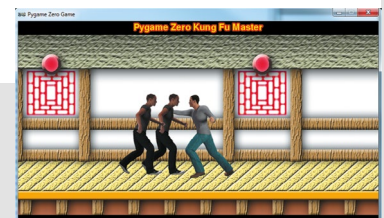
def moveBloke(x,d):
    global backPos, blokeDir
    frame = int((count%48)/8) + 1
    if backPos + x < -3 and backPos + x > -1197:
        backPos += x
        moveDudes(x)
        bloke.image = 'walk'+d+'_000'+str(frame)
        blokeDir = d
```

```
def makeDude():
    d = len(dudes)
    if random.randint(0, 1) == 0:
        dudes.append(Actor('duder_0001', center=(-50,250)))
    else:
        dudes.append(Actor('dudel_0001', center=(850, 250)))
    dudes[d].status = 0

def updateDudes():
    global gameState
    frame = int((count%48)/8) + 1
    for d in dudes:
        if (bloke.image == 'punch'+blokeDir or bloke.image ==
'kick'+blokeDir) and bloke.collidepoint((d.x, d.y)):
            d.status += 1
            if d.x <=400:
                if d.status > 10:
                    d.image = 'dudefallr'
                    d.y += 5
                else:
                    d.x += 2
                    d.image = 'duder_000'+str(frame)
            if d.x >400:
                if d.status > 10:
                    d.image = 'dudefalll'
                    d.y += 5
                else:
                    d.x -= 2
                    d.image = 'dudel_000'+str(frame)
            if d.x > 398 and d.x < 402 and d.status == 0:
                gameState = 1

def moveDudes(x):
    for d in dudes:
        d.x += x
```

> Our Kung-Fu Master homage features punches, kicks, and a host of goons to use them on.



## THE GENERATION GAME

Because we're moving the background when our hero walks left and right, we need to make sure we move our enemies with the background, otherwise they'll look as though they're sliding in mid-air – this also applies to any other objects that aren't part of the background. The number of enemies can be governed in several ways: in our code, we just have a random number deciding if a new enemy will appear during each update, but we could use a predefined pattern for the enemy generation to make it a bit less random, or we use a combination of patterns and random numbers.



^ Each of the six levels got progressively harder to navigate and had to be completed within a time limit.

v Although the designer was against it, Atari wanted the marbles to have smiley faces on them. The idea didn't make it to the game but is reflected in the game logo.



Source Code

## Code a homage to Marble Madness



AUTHOR  
MARK VANSTONE

Code the map and movement basics of  
the innovative marble-rolling arcade game

**H**itting arcades in 1984, Atari's *Marble Madness* presented a rather different control mechanism than other games of the time. The original arcade cabinet provided players with a trackball controller rather than a conventional joystick, and the aim was to guide a marble through a three-dimensional course in the fastest possible time. This meant that a player could change the angle and speed of the marble as it rolled and avoid various obstacles and baddies.

During development, designer Mark Cerny had to shelve numerous ideas for *Marble Madness*, since the hardware just wasn't able to achieve the level of detail and interaction he wanted. The groundbreaking 3D display was one idea that made it through to the finished game: its pre-rendered, ray-traced isometric levels.

*Marble Madness* was the first game to use Atari's System 1 upgradeable hardware

platform, and also boasted the first use of an FM sound chip produced by Yamaha to create its distinctive stereo music. The game was popular in arcades to start with, but interest appeared to drop off after a few months – something Cerny attributed to the fact that the game didn't take long to play.

**“The ball physics are calculated from the grey-shaded heightmap”**

*Marble Madness*'s popularity endured in the home market, though, with ports made for most computers and consoles of the time – although inevitably, most of these didn't support the original's trackball controls.

For our version of *Marble Madness*, we're going to use a combination of a rendered background and a heightmap in Pygame Zero, and write some simple physics code to simulate the marble rolling over the terrain's

flats and slopes. We can produce the background graphic using a 3D modelling program such as Blender. The camera needs to be set to Orthographic to get the forced perspective look we're after. The angle of the camera is also important, in that we need an X rotation of 54.7 degrees and a Y rotation of 45 degrees to get the lines of the terrain correct. The heightmap can be derived from an overhead view of the terrain, but you'll probably want to draw the heights of the blocks in a drawing package such as GIMP to give you precise colour values on the map.

The ball rolling physics are calculated from the grey-shaded heightmap graphic. We've left a debug mode in the code; by changing the debug variable to **True**, you can see how the marble moves over the terrain from the overhead viewpoint of the heightmap. The player can move the marble left and right with the arrow keys – on a level surface it will gradually slow down if no keys are pressed. If the marble is on a gradient



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag34](https://wfmag.cc/wfmag34)

# Rolling marbles in Python

To get Mark's code running on your system, you'll need to install Pygame Zero – all instructions are at [wfmag.cc/pgzero](https://wfmag.cc/pgzero).

```
# Marble Madness
from pygame import image

HEIGHT = 570
WIDTH = 600
gameState = 0
marble = Actor('marble', center=(300, 45))
marbleh = Actor('marbleh', center=(300, 60))
marble.dir = marble.speed = 0
heightmap = image.load('images/height45.png')
# set debug variable below to True for debug mode
debug = False

def draw():
    if(debug):
        screen.blit("height45", (0, 0))
        marbleh.draw()
    else:
        screen.blit("map", (0, 0))
        if gameState == 0:
            marble.draw()
        else:
            if gameState == 2:
                screen.draw.text("YOU WIN!", center = (300,
300), owidth=0.5, ocolor=(255,255,255), color=(0,0,255) ,
fontsize=80)
                marble.draw()
            else:
                screen.draw.text("GAME OVER!", center = (300,
300), owidth=0.5, ocolor=(255,255,255), color=(0,0,255) ,
fontsize=80)
                screen.blit("overlay", (0, 0))
```

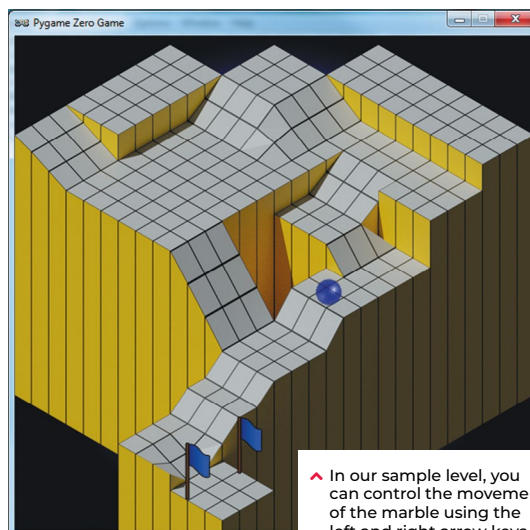
```
def update():
    if gameState == 0:
        if keyboard.left:
            marble.dir = max(marble.dir-0.1,-1)
            marble.speed = min(1,marble.speed + 0.1)
        if keyboard.right:
            marble.dir = min(marble.dir+0.1,1)
            marble.speed = min(1,marble.speed + 0.1)
        moveMarble()
        marble.speed = max(0,marble.speed - 0.01)

def moveMarble():
    global gameState
    ccol = getHeight(marbleh.x,marbleh.y)
    lcol = getHeight(marbleh.x-10,marbleh.y+10)
    rcol = getHeight(marbleh.x+10,marbleh.y+10)
    if ccol.r == 0:
        gameState = 1
    if (lcol.r < ccol.r or rcol.r < ccol.r):
        marble.y += marble.speed
        marble.speed += 0.03
    marbleh.x += marble.speed*marble.dir
    marbleh.y += marble.speed
    marble.x = marbleh.x
    marble.y = (marbleh.y*0.6)+((255-ccol.r)*1.25)
    marble.angle = marble.angle + marble.speed*marble.dir*-10
    if marble.angle > 0 : marble.angle = -50
    if marble.angle < -50 : marble.angle = 0
    if marbleh.y > 610: gameState = 2

def getHeight(x,y):
    return heightmap.get_at((int(x),int(y)))
```

on the heightmap, it will increase speed in the direction of the gradient. If the marble hits a section of black on the heightmap, it falls out of play, and we stop the game.

That takes care of the movement of the marble in two dimensions, but now we have to translate this to the rendered background's terrain. The way we do this is to translate the Y coordinate of the marble as if the landscape was all at the same level – we multiply it by 0.6 – and then move it down the screen according to the heightmap data, which in this case moves the marble down 1.25 pixels for each shade of colour. We can use an overlay for items the marble always rolls behind, such as the finish flag. And with that, we have the basics of a *Marble Madness* level. 🐍



## Module Madness

We use the image module from Pygame to sample the colour of the pixel directly under the marble on the heightmap. We also take samples from the left diagonal and the right diagonal to see if there is a change of height. We are only checking for left and right movement, but this sample could be expanded to deal with the two other directions and moving up the gradients, too. Other obstacles and enemies can be added using the same heightmap translations used for the marble, and other overlay objects can be added to the overlay graphic.



# Recreate Pang's sprite spawning mechanic



**AUTHOR**  
RIK CROSS

Rik shows you how to recreate the spawning of objects found in the balloon-bursting arcade gem, Pang

**P**rogrammed by Mitchell and distributed by Capcom, *Pang* was first released as an arcade game in 1989, but was later ported to a whole host of home computers, including the ZX Spectrum, Amiga, and Commodore 64. The aim of *Pang* is to destroy balloons as they bounce around the screen, either alone or working together with another player, in increasingly elaborate levels. Destroying a balloon can sometimes also spawn a

power-up, freezing all balloons for a short time or giving the player a better weapon with which to destroy balloons.

Initially, the player is faced with the task of destroying a small number of large balloons. However, destroying a large balloon spawns two smaller balloons, which in turn spawns

**“Pang was the game that took the sphere-hating concept to the masses”**

take in my example is to use various features of object orientation (as usual, my example code has been written in Python, using the Pygame Zero library). It's also worth mentioning that for brevity, the example code only deals with simple spawning and destroying of objects, and doesn't handle balloon movement or collision detection.

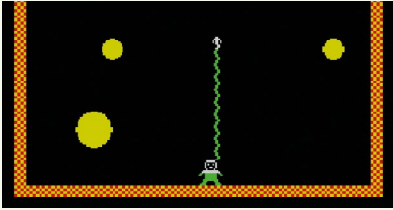
The base **Enemy** class is simply a subclass of Pygame Zero's **Actor** class, including a static **enemies** list to keep track of all enemies that exist within a level. The **Enemy** subclass also includes a **destroy()** method, which removes an enemy from the **enemies** list and deletes the object.

There are then three further subclasses of the **Enemy** class, called **LargeEnemy**, **MediumEnemy**, and **SmallEnemy**. Each of these subclasses are instantiated with a specific image, and also include a **destroy()** method. This method simply calls the same **destroy()** method of its parent **Enemy** class, but additionally creates two more objects

two smaller balloons, and so on. Each level is only complete once all balloons have been broken up and completely destroyed. To add challenge to the game, different-sized balloons have different attributes – smaller balloons move faster and don't bounce as high, making them more difficult to destroy.

There are a few different ways to achieve this game mechanic, but the approach I'll





## HIDDEN HISTORIES

*Pang* was a widely-ported hit in 1989, but its concept originates even further back in video game history. *Asteroids* pioneered a markedly similar brand of spawning, splitting enemies in 1979, but *Pang*'s clearest antecedent is the 1983 game *Cannon Ball*, released for the MSX and later the ZX Spectrum as *Bubble Buster*. Released by Hudson Soft, *Cannon Ball* is markedly similar to *Pang*: it has the little guy running around at the bottom of the screen, the dividing balloons, and even the distinctive harpoon weapon. But with its chunky graphics and minimal sound, it's easy to see why *Pang* – with its colourful characters and wealth of power-ups – was the game that took the sphere-hating concept to the masses.

nearby – with large enemies spawning two medium enemies, and medium enemies spawning two small enemies.

In the example code, initially two **LargeEnemy** objects are created, with the first object in the **enemies** list having its **destroy()** method called each time the **Space** key is pressed. If you run this code, you'll see that the first large enemy is destroyed and two medium-sized enemies are created. This chain reaction of destroying and creating enemies continues until all **SmallEnemy** objects are destroyed (small enemies don't create any other enemies when destroyed).

As I mentioned earlier, this isn't the only way of achieving this behaviour, and there are advantages and disadvantages to this approach. Using subclasses for each size of enemy allows for a lot of customisation, but could get unwieldy if much more than three enemy sizes are required. One alternative is to simply have a single **Enemy** class, with a **size** attribute. The enemy's image, the entities it creates when destroyed, and even the movement speed and bounce height could all depend on the value of the enemy size. 🐼

## Pang balloons in Python

Here's Rik's example code, which recreates *Pang*'s spawning balloons in Python. To get it running on your system, you'll first need to install Pygame Zero – you can find full instructions at [wfmag.cc/XVIIeD](http://wfmag.cc/XVIIeD)

```
class Enemy(Actor):
    # static list, to keep track of all enemies
    enemies = []
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        # add enemy to the enemies list
        self.enemies.append(self)
    def destroy(self):
        # remove self from the enemies list
        self.enemies.remove(self)
        self = None

class LargeEnemy(Enemy):
    def __init__(self, **kwargs):
        # all large-sized enemies have the same image
        super().__init__(image='large_enemy', **kwargs)
    def destroy(self):
        # spawn 2 medium-sized enemies when destroying
        m1 = MediumEnemy(pos=(self.pos[0]-40, self.pos[1]-40))
        m2 = MediumEnemy(pos=(self.pos[0]+40, self.pos[1]+40))
        super().destroy()

class MediumEnemy(Enemy):
    def __init__(self, **kwargs):
        # all medium-sized enemies have the same image
        super().__init__(image='medium_enemy', **kwargs)
    def destroy(self):
        # spawn 2 small-sized enemies when destroying
        s1 = SmallEnemy(pos=(self.pos[0]-20, self.pos[1]-20))
        s2 = SmallEnemy(pos=(self.pos[0]+20, self.pos[1]+20))
        super().destroy()

class SmallEnemy(Enemy):
    def __init__(self, **kwargs):
        # all small-sized enemies have the same image
        super().__init__(image='small_enemy', **kwargs)

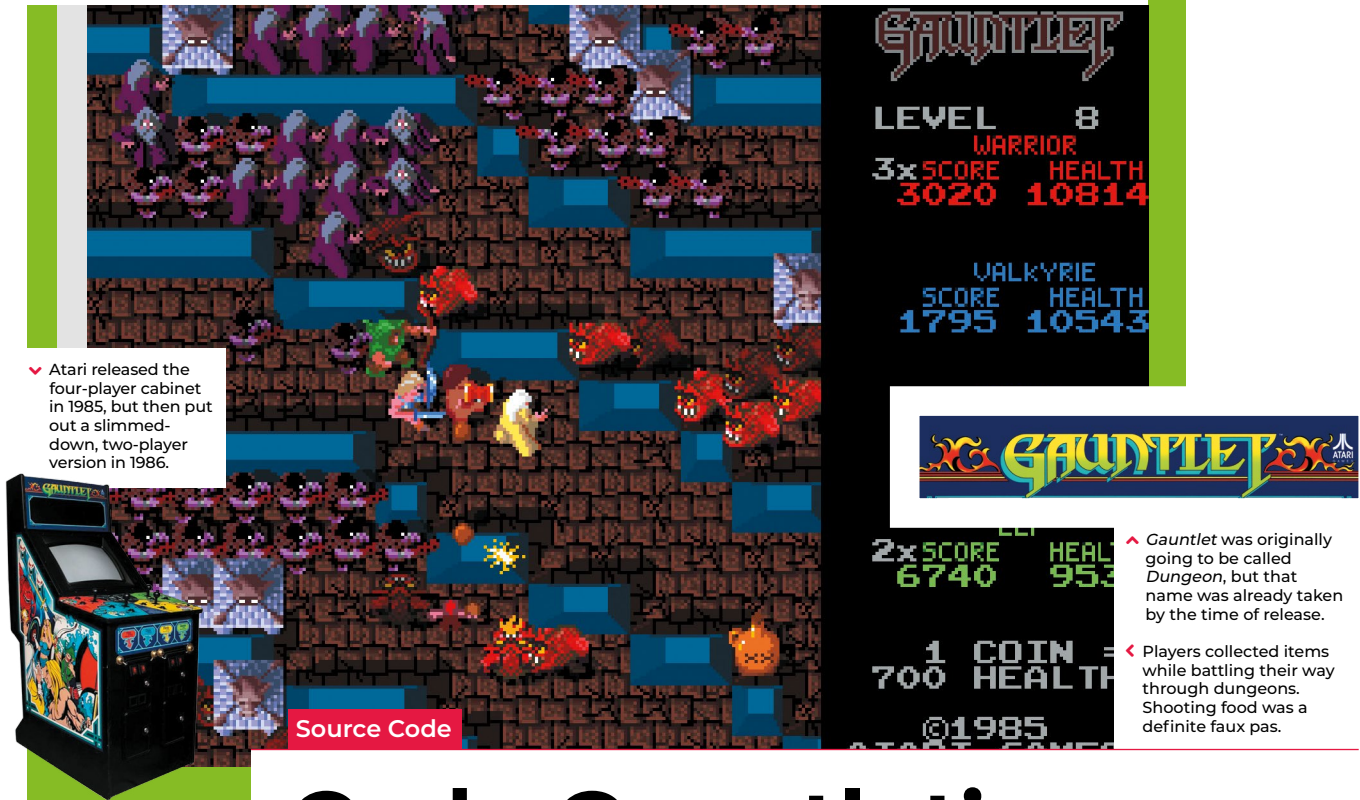
# start with 2 large-sized enemies
l1 = LargeEnemy(pos=(300, 150))
l2 = LargeEnemy(pos=(150, 300))

# destroy the first enemy in the enemies list
def on_key_down():
    if len(Enemy.enemies) > 0:
        Enemy.enemies[0].destroy()

# draw all enemies in static enemies list
def draw():
    screen.clear()
    for e in Enemy.enemies:
        e.draw()
```







## Code Gauntlet's four-player co-op

AUTHOR  
MARK VANSTONE

Four players dungeon crawling at once? Mark shows you how

**A**tari's *Gauntlet* was an eye-catching game, not least because it allowed four people to explore its dungeons together. Each player could choose one of four characters, each with its own abilities – there was a warrior, a Valkyrie, a wizard, and an elf – and surviving each dungeon required slaughtering enemies and the constant gathering of food, potions, and keys that unlocked doors and exits.

Designed by Ed Logg, and loosely based on the tabletop RPG *Dungeons & Dragons*, as well as John Palevich's 1983 dungeon crawler, *Dandy*, *Gauntlet* was a big success. It was ported to most of the popular home systems at the time, and Atari released a sequel arcade machine, *Gauntlet II*, in 1986.

Atari's original arcade machine featured four joysticks, but our example will mix keyboard controls and gamepad inputs. Before we deal with the movement, we'll need some characters and dungeon graphics. For this example, we can make our dungeon

from a large bitmap image and use a collision map to prevent our characters from clipping through walls. We'll also need graphics for the characters moving in eight different directions. Each direction has three frames of walking animation, which makes a total of 24 frames per character. We can use a Pygame Zero Actor object for each character and add a few extra properties to keep track of direction and the current animation frame. If we put the character Actors in a list, we can loop through the list to check for collisions, move the player, or draw them to the screen.

We now test input devices for movement controls using the built-in Pygame keyboard object to test if keys are pressed. For example, `keyboard.left` will return `True` if the left arrow key is being held down. We can use the arrow keys for one player and the **WASD** keys for the other keyboard player. If we register x and y movements separately, then if two keys are pressed – for example, up and left – we can read that as a diagonal movement. In this way, we can get all eight directions of movement from just four keys.

For joystick or gamepad movement, we need to import the joystick module from Pygame. This provides us with methods to count the number of joystick or gamepad devices that are attached to the computer, and then initialise them for input. When we check for input from these devices, we just need to get the x-axis value and the y-axis value and then make it into an integer. Joysticks and gamepads should return a number between -1 and 1 on each axis, so if we round that number, we will get the movement value we need.

We can work out the direction (and the image we need to use) of the character with a small lookup table of x and y values and translate that to a frame number cycling through those three frames of animation as the character walks. Then all we need to do before we move the character is check they aren't going to collide with a wall or another character. And that's it – we now have a four-player control system. As for adding enemy spawners, loot, and keys – well, that's a subject for another time. ☺



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag39](https://wfmag.cc/wfmag39)

# Four-player movement in Python

Here's Mark's code for a *Gauntlet*-style four-player mechanic. To get it running on your system, you'll need to install Pygame Zero – full instructions are available at [wfmag.cc/pgzero](https://wfmag.cc/pgzero).

```
import math
from pygame import image, Color, joystick

myChars = []
myDirs = [(0,1),(-1,1),(-1,0),(-1,-1),(0,-1),(1,-1),(1,0),(1,1)]
collisionmap = image.load('images/collisionmap.png')
joystick.init()
joyin0 = joyin1 = False
if(joystick.get_count() > 0):
    joyin0 = joystick.Joystick(0)
    joyin0.init()
if(joystick.get_count() > 1):
    joyin1 = joystick.Joystick(1)
    joyin1.init()

def makeChar(name,x,y):
    c = len(myChars)
    myChars.append(Actor(name+"_1", (x, y)))
    myChars[c].name = name
    myChars[c].frame = myChars[c].movex = myChars[c].movey =
    myChars[c].dir = 0

def draw():
    screen.blit("colourmap", (0,0))
    drawChars()

def drawChars():
    for c in range(len(myChars)):
        myChars[c].image = myChars[c].
        name+"_"+str(((myChars[c].dir*3)+1)+math.floor(myChars[c].
        frame/10))
        myChars[c].draw()

def update():
    checkInput()
    moveChars()

def checkInput():
    if keyboard.left: myChars[0].movex = -1
    if keyboard.right: myChars[0].movex = 1
    if keyboard.up: myChars[0].movey = -1
    if keyboard.down: myChars[0].movey = 1
    if keyboard.a: myChars[1].movex = -1
    if keyboard.d: myChars[1].movex = 1
    if keyboard.w: myChars[1].movey = -1
    if keyboard.s: myChars[1].movey = 1
    if joyin0:
        myChars[2].movex = round(joyin0.get_axis(0))
        myChars[2].movey = round(joyin0.get_axis(1))
    if joyin1:
        myChars[3].movex = round(joyin1.get_axis(0))
```

```
myChars[3].movey = round(joyin1.get_axis(1))
```

```
def moveChars():
    for c in range(len(myChars)):
        getCharDir(myChars[c])
        if myChars[c].movex or myChars[c].movey:
            myChars[c].frame += 1
            if myChars[c].frame >= 30: myChars[c].frame = 0
            testmove = (int(myChars[c].x + (myChars[c].movex
            *20)),int(myChars[c].y + (myChars[c].movey *20)))
            if collisionmap.get_at(testmove) == Color('black')
            and collideChars(c,testmove) == False:
                myChars[c].x += myChars[c].movex
                myChars[c].y += myChars[c].movey
                myChars[c].movex = 0
                myChars[c].movey = 0

def getCharDir(ch):
    for d in range(len(myDirs)):
        if myDirs[d] == (ch.movex,ch.movey):
            ch.dir = d

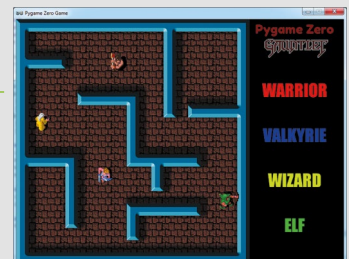
def collideChars(c,xy):
    for ch in range(len(myChars)):
        if myChars[ch].collidepoint(xy) and ch != c:
            return True
    return False

makeChar("warrior",60,60)
makeChar("valkyrie",500,450)
makeChar("wizard",460,180)
makeChar("elf",100,400)
```

## Finding sprites

If you want to reconstruct a retro game like *Gauntlet*, you can often find sprite sheets online – these are bitmaps with all the frames of animation for a character on one sheet. A good source of these files is [spriters-resource.com](https://spriters-resource.com), where you'll find sprite sheets for a wide range of retro games. Some coding systems can use the sprite-sheets as they are, but for this example, we have cut them up into separate frames. You can do this with Sprite Sheet Slicer, available at [wfmag.cc/slicer](https://wfmag.cc/slicer).

► Our four-player homage to the classic *Gauntlet* arcade game.



Source Code

# Create a turn-based combat system



AUTHOR  
RIK CROSS

Learn how to create the turn-based combat system found in games like Pokémon, Final Fantasy, and Undertale

In the late 1970s, high school student Richard Garriott made a little game called *Akalabeth*. Programmed in Applesoft BASIC, it helped set the template for the role-playing genre on computers. Even today, turn-based combat is still a common sight in games, with this autumn's *Pokémon Sword and Shield* revolving around a battle system which sees opponents take turns to plan and execute attacks or defensive moves.

The turn-based combat system in this article is text-only, and works by allowing players to choose to defend against or attack their opponent in turn. The battle ends when only one player has some health remaining.

Each **Player** taking part in the battle is added to the static **players** list as it's created. Players have a **name**, a **health** value (initially set to **100**) and a Boolean **defending** value (initially set to **False**) to indicate

whether a player is using their shield. Players also have an **inputmethod** attribute, which is the function used for getting player input for making various choices in the game. This function is passed to the object when created, and means that we can have human players that give their input through the keyboard, as well as computer players that make choices (in our case simply by making a random choice between the available options).

A base **Action** class specifies an action **owner** and an **opponent**, as well as an **execute()** method which has no effect on the game. Subclasses of the base class override this **execute()** method to specify the effect the action has on the **owner** and/or the **opponent** of the action. As a basic example, two actions have been created: **Defend**, which sets the owner's **defending** attribute to **True**, and **Attack**, which sets the owner's **defending** attribute to **False**, and lowers the opponent's **health** by a random amount depending on whether or not they are **defending**.

Players take turns to choose a single action to perform in the battle, starting with the human 'Hero' player. The **choose\_action()** method is used to decide what to do next (in this case either attack or defend), as well as an opponent if the player has chosen to attack. A player can only be selected as an opponent if they have a **health** value greater than 0, and are therefore still in the game. This **choose\_action()** method returns an **Action**, which is then executed using



It may look crude, but Richard Garriott's *Akalabeth* laid the groundwork for *Ultima*, and was one of the earliest CRPGs.

its **execute()** method. A few **time.sleep()** commands have also been thrown in here to ramp up the suspense!

After each player has had their turn, a check is done to make sure that at least two players still have a **health** value greater than 0, and therefore that the battle can continue. If so, the static **get\_next\_player()** method finds the next player still in the game to take their turn in the battle, otherwise, the game ends and the winner is announced.

Our example

battle can be easily extended in lots of interesting ways. The AI for choosing an action could also be made more sophisticated, by looking at opponents' **health** or **defending** attributes before choosing an action. You could also give each action a 'cost', and give players a number of action 'points' per turn. Chosen actions would be added to a list, until all of the points have been used. These actions would then be executed one after the other, before moving on to the next player's turn. ☺



With their emphasis on trading and collecting as well as turn-based combat, the *Pokémon* games helped bring RPG concepts to the masses.



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag28](https://wfmag.cc/wfmag28)

# Turn-based combat in Python

Here's Rik's code snippet, which creates a simple turn-based combat sequence in Python. To get it running on your system, you'll first need to install Pygame Zero – you can find full instructions at [wfmag.cc/pgzero](https://wfmag.cc/pgzero)

```
import random, time

class Action():
    def __init__(self, owner, opponent):
        self.owner = owner
        self.opponent = opponent

    def execute(self):
        pass

class Attack(Action):
    def __init__(self, owner, opponent):
        super().__init__(owner, opponent)

    def execute(self):
        self.owner.defending = False
        if self.opponent.defending:
            hit = random.randrange(10,20)
        else:
            hit = random.randrange(20,40)
        self.opponent.health -= hit
        print('{} is hit! (-{})'.format(self.opponent.name, hit))

class Defend(Action):
    def __init__(self, owner, opponent):
        super().__init__(owner, opponent)

    def execute(self):
        self.owner.defending = True
        print(self.owner.name, 'is defending!')

class Player():
    players = []

    def __init__(self, name, inputmethod):
        self.name = name
        self.inputmethod = inputmethod
        self.health = 100
        self.defending = False
        self.players.append(self)

    def __str__(self):
        description = "Player: {}\n{}\nHealth = {}\nDefending = {}".format(
            self.name,
            '-' * (8 + len(self.name)),
            self.health,
            self.defending
        )
        return(description)

    @classmethod
    def get_next_player(cls, p):
        # get the next player still in the game
        current_index = cls.players.index(p)
        current_index = (current_index + 1) % len(cls.players)
        while cls.players[current_index].health < 1:
            current_index = (current_index + 1) % len(cls.players)
        return cls.players[current_index]

    def choose_action(self):
        print(self.name, ': [a]ttack or [d]efend?')

        action_choice = self.inputmethod(['a', 'd'])
        if action_choice == 'a':
            print('Choose an opponent')
            # build up a list of possible opponents
            opponent_list = []
            for p in self.players:
                if p != self and p.health > 0:
                    print('{} {}'.format(self.players.
index(p), p.name))
                    opponent_list.append(str(self.players.
index(p)))
            # use input to get the opponent of player's action
            opponent = self.players[int(self.inputmethod(opponent_
list))]
            return Attack(self, opponent)
        else:
            return Defend(self, None)

def human_input(choices):
    choice = input()
    while choice not in choices:
        print('Try again!')
        choice = input()
    return choice

def computer_input(choices):
    time.sleep(2)
    choice = random.choice(choices)
    print(choice)
    return choice

# add 2 players to the battle, with their own input method
hero = Player('The Hero', human_input)
enemy = Player('The Enemy', computer_input)

# the hero has the first turn
current_player = Player.players[0]
playing = True

# game loop
while playing:

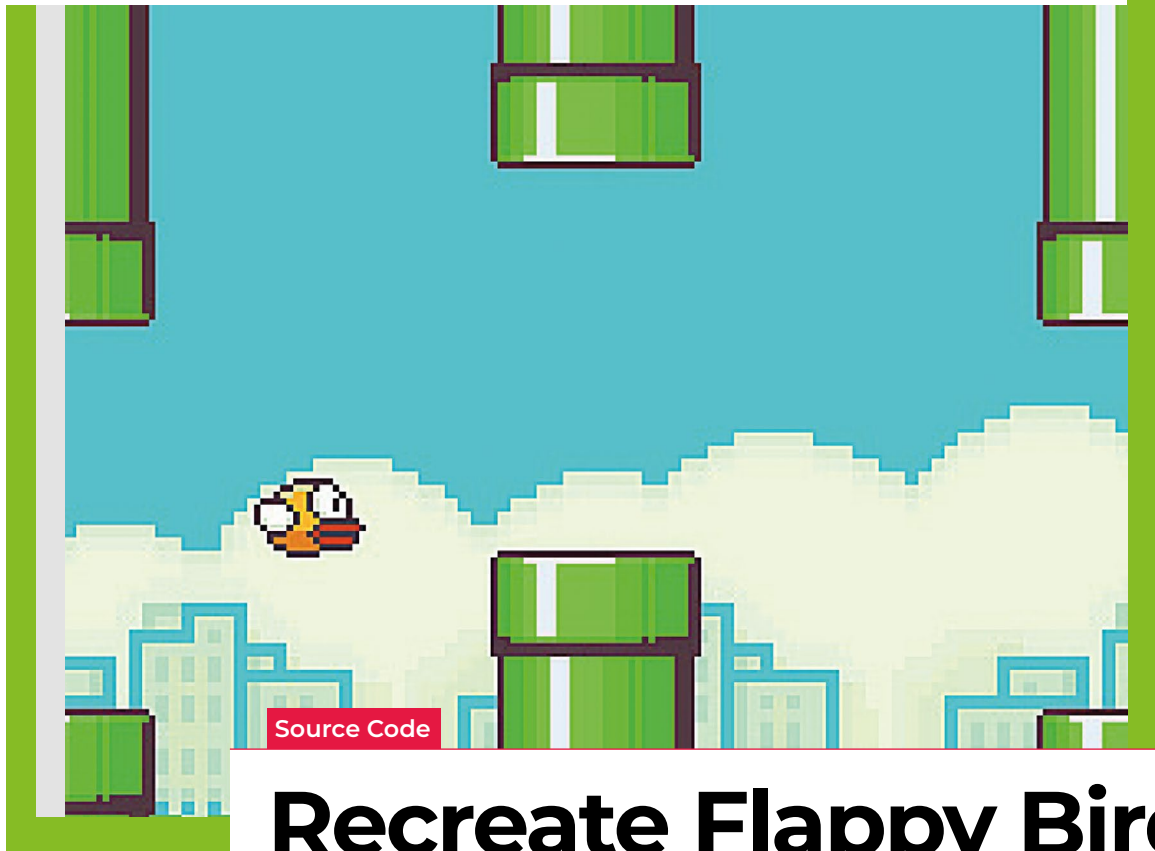
    # print all players with health remaining
    for p in Player.players:
        if p.health > 0:
            print(p, end='\n\n')

    # current player's action executed
    action = current_player.choose_action()
    time.sleep(2)
    action.execute()

    # continue only if more than 1 player with health remaining
    if len([p for p in Player.players if p.health > 0]) > 1:
        current_player = Player.get_next_player(current_player)
        time.sleep(2)
    else:
        playing = False

for p in Player.players:
    if p.health > 0:
        print('**', p.name, 'wins!')
```





◀ *Flappy Bird*: ridiculously big in 2014, at least for a while.

Source Code

# Recreate Flappy Bird's flight mechanic



AUTHOR  
RIK CROSS

Learn how to create your own version of the simple-yet-addictive side-scroller

**F**lappy Bird was released by programmer Dong Nguyen in 2013, and made use of a straightforward game mechanic to create an addictive hit. Tapping the screen provided 'lift' to the main character, which is used strategically to navigate through a series of moving pipes. A point is scored for each pipe successfully passed. The idea proved so addictive that Nguyen eventually regretted his creation and removed it from the Google and Apple app stores. In this article, I'll show you how to recreate this simple yet time-consuming game, using Python and Pygame Zero.

The player's motion is very similar to that employed in a standard platformer: falling down towards the bottom of the screen under gravity. See the article, *Super Mario-style jumping physics* in Wireframe #7 for more on creating this type of movement. Pressing a button (in our case, the **SPACE** bar) gives the player some upward thrust

by setting its velocity to a negative value (i.e. upwards) larger than the value of gravity acting downwards. I've adapted and used two different images for the sprite (made by Imaginary Perception and available on [opengameart.org](http://opengameart.org)), so that it looks like it's flapping its wings to generate lift and move upwards.

Sets of pipes are set equally spaced apart horizontally, and move towards the player slowly each frame of the game. These pipes are stored as two lists of rectangles, **top\_pipes** and **bottom\_pipes**, so that the player can attempt to fly through gaps between the top and bottom pipes. Once a pipe in the **top\_pipes** list reaches the left side of the screen past the player's position, a **score** is incremented and the top and corresponding bottom pipes are removed from their respective lists. A new set of pipes is created at the right edge of the screen, creating a continuous challenge for the player. The y-position of

the gap between each newly created pair of pipes is decided randomly (between minimum and maximum limits), which is used to calculate the position and height of the new pipes.

The game stops and a Game Over message appears if the player collides with either a pipe or the ground. The collision detection in the game uses the **player.collidirect()** method, which checks whether two rectangles overlap. As the player sprite isn't exactly rectangular, it means that the collision detection isn't pixel-perfect, and improvements could be made by using a different approach. Changing the values for **GRAVITY**, **PIPE\_GAP**, **PIPE\_SPEED**, and **player.flap\_velocity** through a process of trial and error will result in a game that has just the right amount of frustration! You could even change these values as the player's score increases, to add another layer of challenge. 🐦



# A flapping bird in Python

Here's Rik's code, which recreates *Flappy Bird*'s avian mayhem in Python. To get it running on your system, you'll need to install Pygame Zero – you can find instructions at [wfmag.cc/pgzero](http://wfmag.cc/pgzero)

```
from random import randint

WIDTH = 1000
HEIGHT = 600

# pipes are dark green, move 2 pixels per frame and
# have a gap of 150 pixels between top and bottom pipes
PIPE_COLOUR = (38,155,29)
PIPE_SPEED = 2
PIPE_GAP = 150

GRAVITY = 0.2

# create top and bottom pipes, with a gap in between
top_pipes = [
    Rect((500,0),(50,200)),
    Rect((1000,0),(50,300))
]

bottom_pipes = [
    Rect((500,200 + PIPE_GAP), (50,HEIGHT - 200 - PIPE_GAP)),
    Rect((1000,300 + PIPE_GAP), (50,HEIGHT - 300 - PIPE_GAP))
]

player = Actor('player-down',(100,400))
# define initial and flap velocities
player.y_velocity = 0
player.flap_velocity = -5
player.score = 0

playing = True

def update():

    global playing
    if playing:

        # space key to flap
        if keyboard.space and player.y_velocity > 0:
            player.y_velocity = player.flap_velocity

        # acceleration is rate of change of velocity
        player.y_velocity += GRAVITY
        # velocity is rate of change of position
        player.y += player.y_velocity

        # player image depends on velocity
        if player.y_velocity > 0:
            player.image = 'player-down'
        else:
            player.image = 'player-up'
        for pipe_list in top_pipes, bottom_pipes:
```

```
        for pipe in pipe_list:
            pipe.x -= PIPE_SPEED
            if pipe.x < -50:
                pipe_list.remove(pipe)

        # create new pipes
        if len(top_pipes) < 2:
            player.score += 1
            h = randint(150,350)
            top_pipes.append(Rect((1000,0),(50,h)))
            bottom_pipes.append(Rect((1000,h + PIPE_GAP),(50,
HEIGHT - h - PIPE_GAP)))

        # game over if player collides with a pipe...
        for p in top_pipes + bottom_pipes:
            if player.colliderect(p):
                playing = False

        # ...or touches the ground
        if player.y > (HEIGHT - 20):
            playing = False

def draw():

    if playing:

        screen.clear()

        screen.blit('background', (0,0))

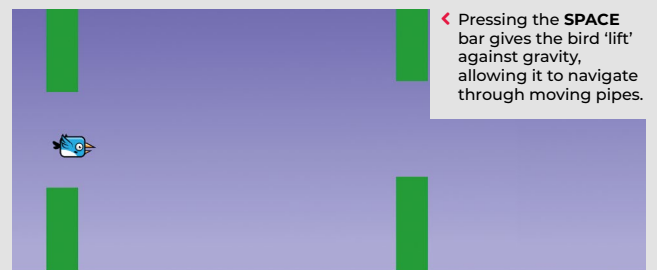
        for pipe in top_pipes + bottom_pipes:
            screen.draw.filled_rect(pipe, PIPE_COLOUR)

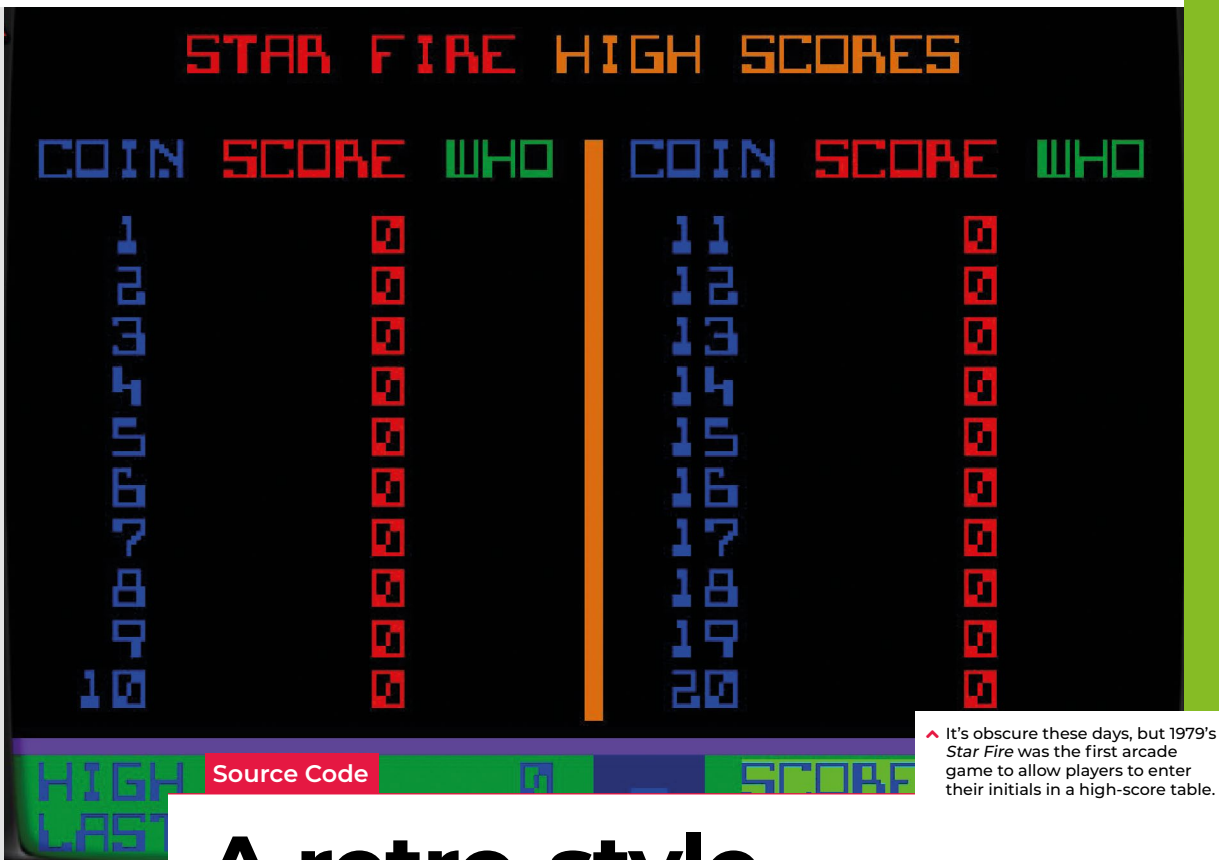
        screen.draw.text(str(player.score), (20, 20),
            fontsize=40, color="white")

        player.draw()

    else:

        screen.draw.text('Game Over!', (420, 200),
            fontsize=40, color="white")
```





## A retro-style high-score table

Here's how to create a high-score table that you can easily add to your own games



AUTHOR  
RIK CROSS

First appearing in arcade games back in the late seventies, high-score tables add an additional challenge, and keep players coming back for more. In this article, I'll show you how to create a high-score table that you can add to your games.

A simple way to track a top score is to create a **highscore** variable. The **highscore** is initially set to 0, and at the end of each game a sufficiently high **score** becomes the new **highscore**.

The principle for creating a high-score table is the same, except now you need to store a list of scores and names, and at the end of each game you need to add the new **score** and **name** to the table if the score is high enough. The high-score table is stored as a list, containing (**score**, **name**) tuples. (In Python, a tuple can be thought of as a list whose elements cannot be changed.)

So a **highscores** list containing three (**score**, **name**) tuples might look like this:

```
highscores =  
[(100, 'Rik'), (86, 'Becci'), (45, 'Steve')]
```

Square brackets **[]** are used to access a particular element of a list, so **highscores[0]** is (100, 'Rik'), and **highscores[1]** is (86, 'Becci'), etc (remember we count lists starting at position 0). **score** is the first item (item 0) of each tuple and **name** is second item (item 1), and we use the same square bracket notation to access elements of a tuple. Therefore, you can access the first score in the list using **highscores[0][0]**, and the first name by using **highscores[0][1]**. It's best to have some scores to aspire to, so when creating the **highscores** list I'll populate it with ten low scores:

```
highscores = [(0, 'Player') for i in  
range(10)]
```

The code for checking a score and (optionally) adding it to the table will be put in a function called **addscore(score)**. This function will take a **score** as a parameter, and add to the **highscores** list if the score is higher than the lowest score in the list.

Firstly, there's no need to do anything if the score isn't high enough to make it into the table. If the **score** isn't higher than the tenth score in the list (the first tuple element of list element nine – **highscores[9][0]**) then the function returns without updating the table:

```
if score < highscores[9][0]:  
    return
```

If the function hasn't returned, then the score is to be added to the table, and next we need a name to attach to the score:

```
# get the player's name  
name = input('High score! What is your  
name?')
```

Next, the **score** and **name** need to be inserted into the **highscores** list at the correct position. To do this, we start with a variable called **pos** which is set to 0 (the first item in the list). This variable is then incremented until either we reach the end of the list, or until the player's score is higher than the next score in the list. Here, the value of **pos** is where we place the new score.

```
pos = 0
while pos < len(highscores) and score
<= highscores[pos][0]:
    pos += 1
```

To add the new **score** and **name** to the list, we next need to split the **highscores** list into two parts; all scores before the position to add, and all scores after the position to add. In Python, the shorthand syntax for all elements up to and including **pos** is **highscores[:pos]** and all items from **pos** to the end of the list is **highscores[pos:]**. We then add the **(score,name)** tuple to the list:

```
highscores = highscores[:pos] +
[(score,name)] + highscores[pos:]
```

We only need the top ten scores in the list, and the addition of another score means there are now eleven. To save just the top ten scores, we can use the same shorthand:

```
highscores = highscores[:10]
```

To print the scores, we first need to print the table headings, separated by a tab (**\t**). Then, we can loop through each tuple in the **highscores** list and print the **score** and the **name**, again separated by a tab. Using tabs rather than spaces keeps the data lined up.

```
def drawtabletext():
    # print the table headings
    print('Score\tName')
    # print score and name pairs in order
    for s in highscores:
        print("{0}\t{1}".format(s[0],s[1]))
```

As an additional challenge you could save the table to a file when the game is quit so it can be retrieved later. For a retro feel, you could also limit the player's name to three characters, using 'up' and 'down' to cycle through the alphabet! 🎮

## High-score table in Python

Here's the full high-score table Python code. To get it running on your system, you'll first need to install Pygame Zero – you can find full instructions at [wfmag.cc/XVzieD](http://wfmag.cc/XVzieD)

```
# highscore list is initially filled with low scores
highscores = [(0,'Player') for i in range(10)]

def addscore(score):
    global highscores
    # only add the score if it is greater than the
    # current lowest score in the highscores list
    if score < highscores[9][0]:
        return
    # get the player's name
    name = input('High score! What is your name?')
    # starting at 0, increment the 'pos' variable
    # until it's at the position to insert the score
    pos = 0
    while pos < len(highscores) and score <= highscores[pos][0]:
        pos += 1
    # add the (score, name) tuple
    # at the correct place in the list
    highscores = highscores[:pos] + [(score,name)] + highscores[pos:]
    # only store the top 10 scores in the list
    highscores = highscores[:10]

def drawtabletext():
    # print the table headings
    print('Score\tName')
    # print each score and name pair in order
    for s in highscores:
        print("{0}\t{1}".format(s[0],s[1]))

# prints the table in Pygame Zero
def drawtablepygame():
    # print the table headings
    screen.draw.text('Score', topleft=(50,50), fontsize=40)
    screen.draw.text('Name', topleft=(150,50), fontsize=40)
    # using 'enumerate()' gives the position of each tuple in the list
    # which is used to calculate the vertical draw position of the data
    for pos,data in enumerate(highscores):
        screen.draw.text(str(data[0]), topleft=(50,100+(pos*50)), fontsize=40)
        screen.draw.text(data[1], topleft=(150,100+(pos*50)), fontsize=40)

def draw():
    drawtablepygame()

# use the 'addscore()' function to add some scores
addscore(64)
addscore(30)
addscore(87)

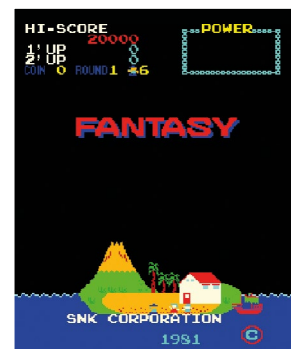
# print the populated table
drawtabletext()
```



A screenshot of the 'CONTINUE?' screen from the video game Ninja Gaiden. The word 'CONTINUE?' is written in large, red, stylized letters at the top. Below it is a large yellow number '2'. The background shows a character in a blue robe lying down, with a sword and other game elements visible.


Source Code

◀ *Ninja Gaiden's* dramatic continue screen. Who would be cruel enough to walk away?



▲ SNK's *Fantasy*, released in 1981, was the first arcade game to feature a continue screen.

# Create your own continue screen

Learn how to create game states, and rules for moving between them



AUTHOR  
RIK CROSS

The continue screen, while much less common now, was a staple feature of arcade games, providing an opportunity (for a small fee) to reanimate the game's hero and to pick up where they left off.

Games such as Tecmo's *Ninja Gaiden* coin-op (known in some regions as *Shadow Warriors*) added jeopardy to their continue screen, in an effort to convince us to part with our money.

Often, a continue screen is one of many screens that a player may find themselves on; other possibilities being a title screen or an instruction screen. I'll show you how you can add multiple screens to a game in a structured way, avoiding a tangle of `if...else` statements and variables.

A simple way of addressing this problem is to create separate update and draw functions for each of these screens, and then switch between these functions as

required. Functions are 'first-class citizens' of the Python language, which means that they can be stored and manipulated just like any other object, such as numbers, text, and class instances. They can be stored in variables and other data types such as lists and dictionaries, and passed as parameters to (or returned from) other functions.

**"The continue screen was a staple of arcade games"**

We can take advantage of the first-class nature of Python functions by storing the functions for the current screen in variables, and then calling them in the main `update()` and `draw()` functions. In the following example, notice the difference between storing a function in a variable (by using the function name without parentheses) and calling the function (by including parentheses).

```
currentupdatefunction =
updatecontinuescreen
currentdrawfunction = drawcontinuescreen

def update():
    currentupdatefunction()

def draw():
    currentdrawfunction()
```

The example code above calls `currentupdatefunction()` and `currentdrawfunction()`, which each store a reference to separate update and draw functions for the continue screen. These continue screen functions could then also include logic for changing which function is called, by updating the function reference stored in `currentupdatefunction` and `currentdrawfunction`.

This way of structuring code can be taken a step further by making use of state machines. In a state machine, a system can be in one of a (finite) number of predefined



# Game states in Python

You'll need to install Pygame Zero to get Rik's code running. You can find instructions at [wfmag.cc/pgzero](http://wfmag.cc/pgzero)



Download  
the code  
from GitHub:  
[wfmag.cc/  
wfmag19](http://wfmag.cc/wfmag19)

```
class State():
    def __init__(self):
        self.rules = {}
    def addrule(self, state, rule):
        self.rules[state] = rule
    def update(self):
        pass
    def draw(self):
        pass

class StateMachine():
    def __init__(self):
        self.current = None
        self.frame = 0
    def update(self):
        if self.current == None:
            return
        self.frame += 0.01
        for s, r in self.current.rules.items():
            if r():
                self.current = s
                self.frame = 0
        self.current.update()
    def draw(self):
        if self.current == None:
            return
        self.current.draw()

sm = StateMachine()

def drawtitle():
    screen.draw.text("Title screen", (50, 50), fontsize=40,
color="white")
    screen.draw.text("Press [space] to start", (50, 80),
fontsize=40, color="white")
```

```
titlescreen = State()
titlescreen.draw = drawtitle

def drawgame():
    screen.draw.text("Game screen", (50, 50), fontsize=40,
color="white")
    screen.draw.text("Press [e] to end game", (50, 80),
fontsize=40, color="white")
    gamescreen = State()
    gamescreen.draw = drawgame

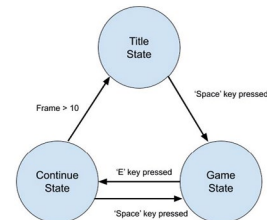
def drawcontinue():
    screen.draw.text("Continue screen", (50, 50), fontsize=40,
color="white")
    screen.draw.text("Press [space] to play again", (50, 80),
fontsize=40, color="white")
    screen.draw.text(str(int(10 - sm.frame)+1), (50, 110),
fontsize=40, color="white")
    continuescreen = State()
    continuescreen.draw = drawcontinue

titlescreen.addrule(gamescreen, lambda: keyboard.space)
gamescreen.addrule(continuescreen, lambda: keyboard.e)
continuescreen.addrule(titlescreen, lambda: sm.frame >= 10)
continuescreen.addrule(gamescreen, lambda: keyboard.space)

sm.current = titlescreen

def update():
    sm.update()

def draw():
    screen.clear()
    sm.draw()
```



Rules define conditions that need to be satisfied in order to move between states.

states, and rules determine the conditions under which a system can transition from one state into another.

A state machine (in this case a very simplified version) can be implemented by first creating a core `State()` class. Each game state has its own `update()` and `draw()` methods, and a rules dictionary containing state:rule pairs – references to other state objects linked to functions for testing game conditions. As an example, the `continuescreen` state has two rules:

- Transition to the `gamescreen` state if the **SPACE** key is pressed;
- Transition to the `titlescreen` state if the frame timer reaches 10.

This is pulled together with a `StateMachine()` class, which keeps track of the current state. The state machine calls the `update()` and `draw()` methods for the current state, and checks the rules for transitioning between states. Each rule in the current state's rules list is executed, with the state machine updating the reference to its current state if the rule function returns **True**. I've also added a frame counter that is incremented by the state machine's `update()` function each time it is run. While not a necessary part of the state machine, it does allow the continue screen to count down from 10, and could have a number of other uses, such as for animating sprites.

Something else to point out is the use of lambda functions when adding rules to states. Lambda functions are small, single-expression anonymous functions that return the result of evaluating its expression when called. Lambda functions have been used in this example simply to make the code a little more concise, as there's no benefit to naming the functions passed to `addrule()`.

State machines have lots of other potential uses, including the modelling of player states. It's also possible to extend the state machine in this example by adding `onenter()` and `onexit()` functions that can be called when transitioning between states. 🐍

# Wireframe magazine's GameDev.tv course bundle

Learn how to make games for less with Wireframe  
and GameDev.tv's exclusive reader offer



Ever wanted to make a 2D platformer in Unity? Fancy learning how to make your first 3D models in Blender? How about diving into some C++ programming in Unreal Engine? You can learn all this and more for a bargain price, thanks to GameDev.tv and Wireframe's exclusive reader offer. You'll get full lifetime access to three online course bundles for a single one-off fee!



## COMPLETE C# UNITY GAME DEVELOPER COURSE

Learn Unity in C# and code your first five 2D video games for web, Mac, and PC.

Learn how to create video games using Unity, with the world's most popular online game development courses.

We start super-simple, so you need no prior experience of Unity or coding!

You'll build multiple games, including:

- Snow Boarder:** a simple side-scrolling jumping game
- Laser Defender:** a top-down space shooter with enemies to shoot and dodge
- TileVania:** a fast-paced classic platformer
- Quiz Master:** a quiz game to learn setting up the user interface

By the end of the course, you'll be confident in the basics of coding and game development, and hungry to learn more.

Get over **£600** of  
online courses for  
around **£30!\*** Visit  
[wfmag.cc/gamedevsc](http://wfmag.cc/gamedevsc)



**UNREAL ENGINE**

### UNREAL C++ DEVELOPER COURSE

Ready to make games in the amazing world of Unreal Engine 5? This “critically acclaimed” and “insanely successful” Unreal Engine course was created in collaboration with Epic Games.

We start super-simple, so no prior experience of Unreal or coding is needed. We believe project-based learning is the best way to learn Unreal Engine, so you'll create five Unreal games (including a tank game and FPS).

You'll learn C++, the powerful industry-standard language from scratch. And by the end of the course, you'll be confident in the basics of coding and game development.

*“Any serious game programmer needs to know C++”*  
Jason Gregory, lead programmer at Naughty Dog, creators of *Uncharted* and *The Last of Us*.

Benefit from world-class support, both from other students, and instructors who are regularly on the forums.

Grab the bundle and start making UE5 games now!



### THE COMPLETE BLENDER CREATOR: LEARN 3D MODELLING FOR BEGINNERS

Learn how to create 3D models and assets for games using Blender, the free-to-use 3D production suite.

You'll be amazed at what you can achieve with our tutorials:

- Create assets for video games
- Make unique 3D-printed gifts
- Design your dream house, car, and more
- Express yourself through 3D artwork

The course is project-based, so you'll apply your new skills to real 3D models. All project files are included, as well as additional references and resources, so you'll never get stuck. If you're a complete beginner, we'll teach you all the modelling fundamentals you'll need.

If you're an artist, we'll teach you how to bring your assets to life. If you're a coder, we'll teach you modelling and design principles.

Dive in now – you won't be disappointed!

Go to [wfmag.cc/gamedevsc](http://wfmag.cc/gamedevsc) to get your exclusive discount

\*Bundle original price \$702 inc VAT. Discounted price \$35.10 inc. VAT. All prices are approximate and based on exchange rate at time of going to press.



# CUSTOMPC

THE BEST-SELLING MAG FOR PC HARDWARE, OVERCLOCKING, GAMING & MODDING

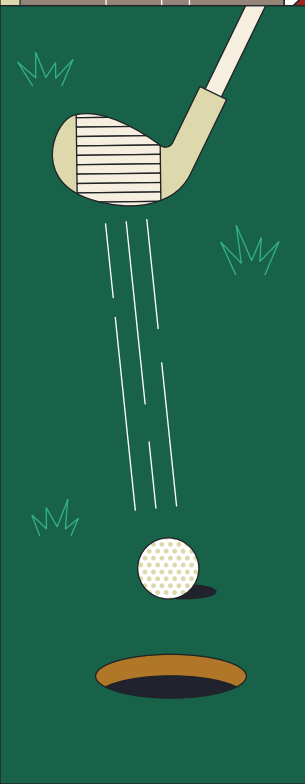
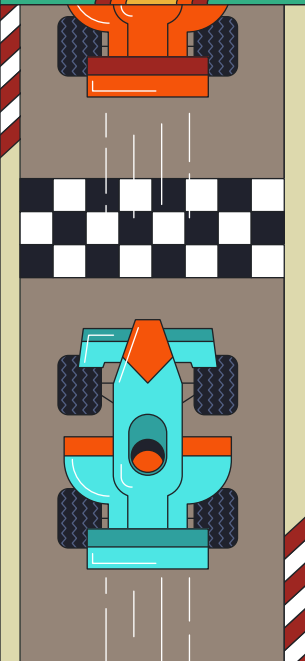
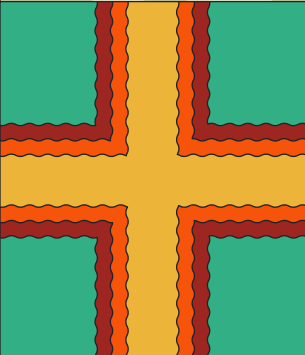
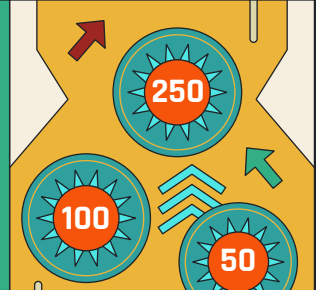
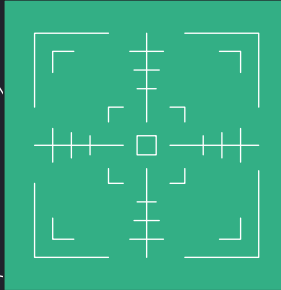
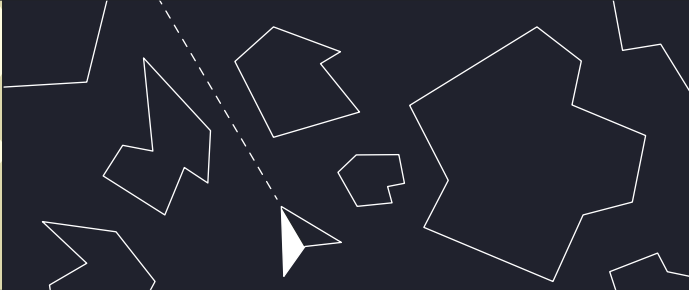
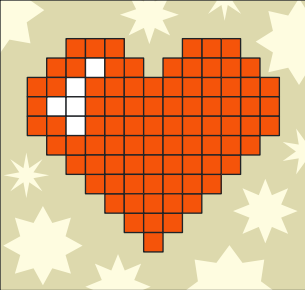
THE MAGAZINE FOR

## PC HARDWARE ENTHUSIASTS



# ISSUE 232 OUT NOW

VISIT [CUSTOMPC.CO.UK](https://www.custompc.co.uk) TO LEARN MORE






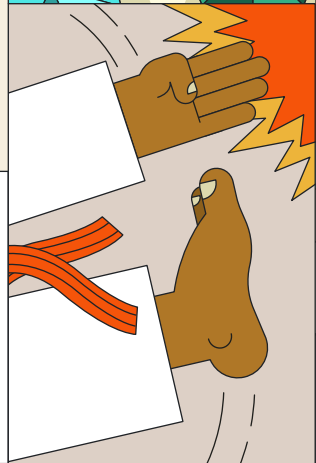
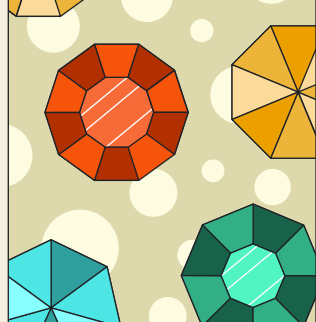
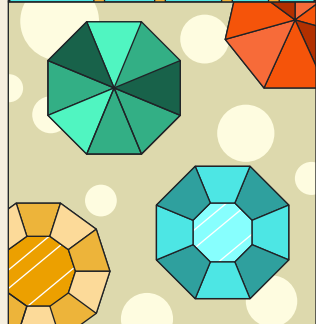
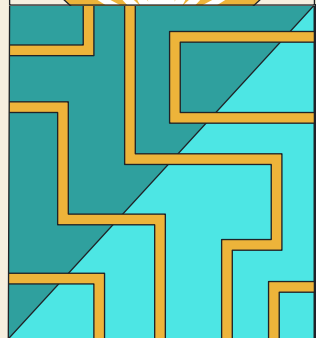
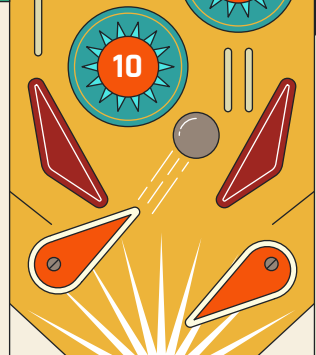
## Thank you for reading Source Code from Wireframe

Published every month, Wireframe is the unique magazine that lifts the lid on video games. Aimed at gamers who want to go behind the scenes of their favourite hobby, Wireframe looks at how games are made, who makes them, and - thanks to our unique Toolbox section - shows you how to make games of your own.

Every issue is packed with in-depth features, news, reviews, and previews, plus lively opinion pieces from developers and key industry figures. Want to know what it's like to make one of the most reviled games of all time? Keen to find out more about the untold stories behind classics like Monkey Island, Castlevania, or Dead Space? Ever wondered how to make your own racing game?

**You'll find all this and more every month in Wireframe magazine.**

-  Buy in paper format or download a free PDF at: **wfmag.cc**
-  To subscribe, visit **wfmag.cc/subscribe**
-  You can find Raspberry Pi Press's other books and magazines at: **store.rpipress.cc**



CONTINUE?  
▶ YES  
NO

