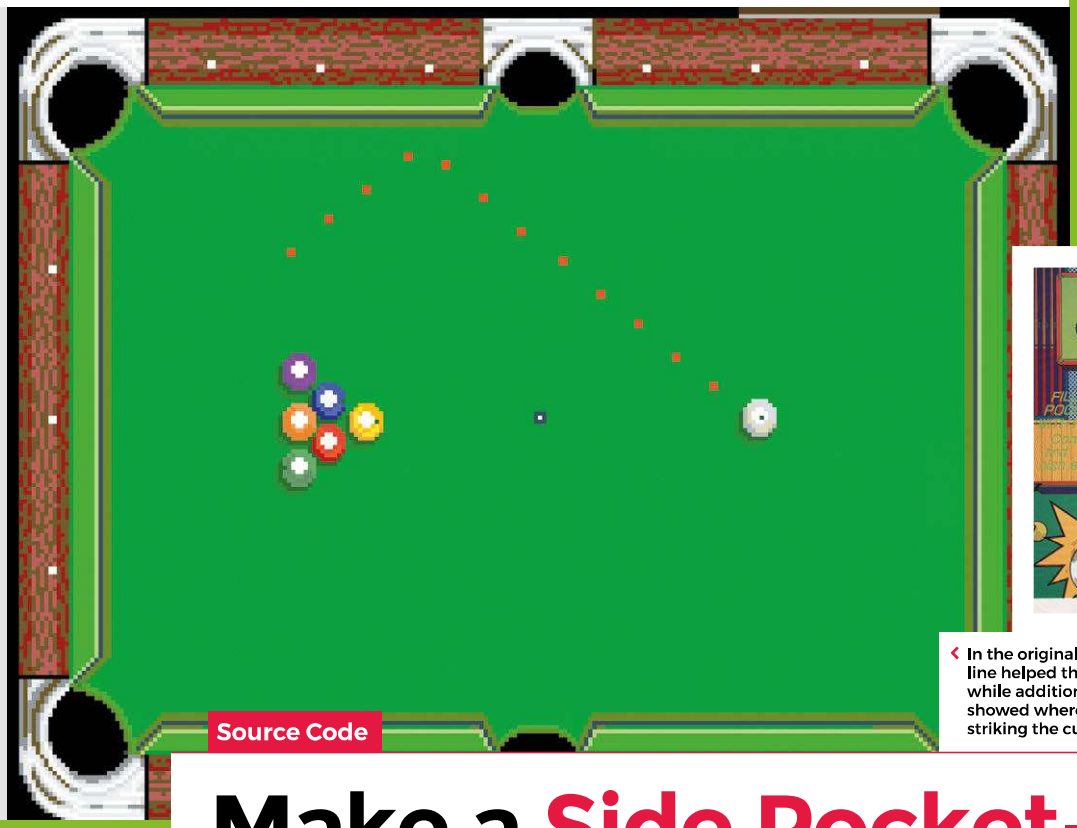
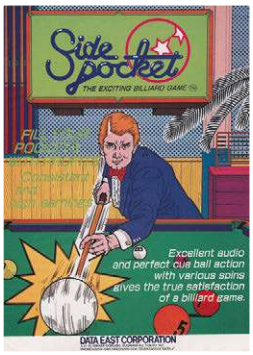




In the original *Side Pocket*, the dotted line helped the player line-up shots, while additional functions on the UI showed where and how hard you were striking the cue ball.

Source Code

Make a Side Pocket-esque pool game

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

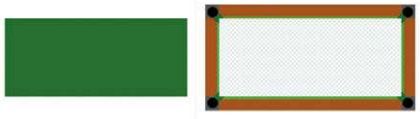
AUTHOR
MAC BOWLEY

Created 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**; originally, this was one 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.

"Before I think about aiming and moving balls, I need a table to play on"

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.



```
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]
```

```
                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.

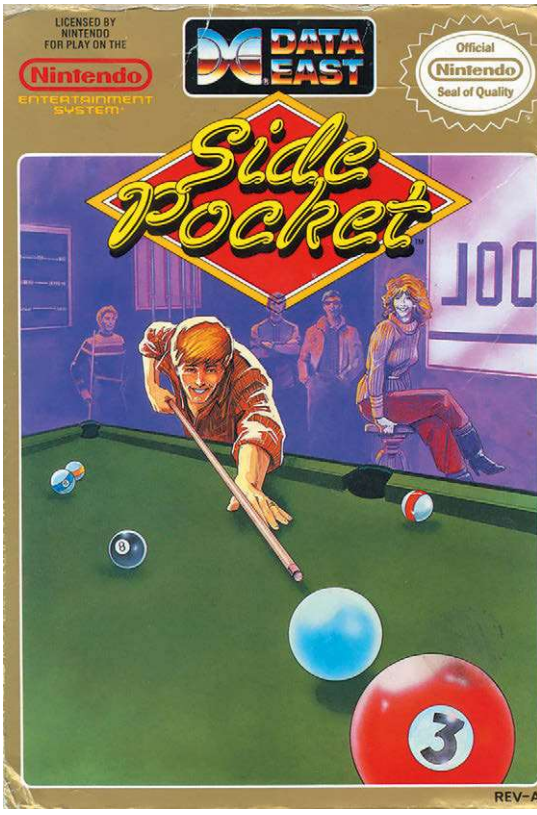
```
# The in-built functions need radian
rot_radians = shot_rotation * (math.
pi/180)

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:
```

Download the code from GitHub: **wfmag.cc/wfmag36**

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.



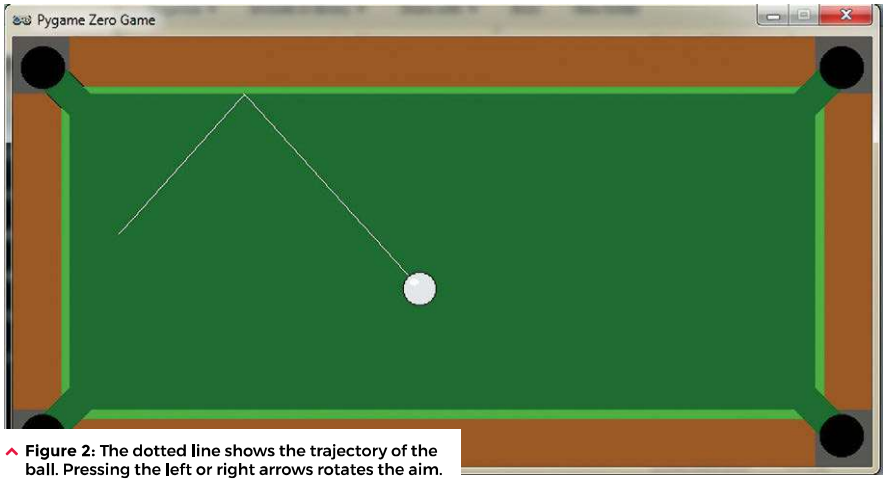


Figure 2: The dotted line shows the trajectory of the ball. Pressing the left or right arrows rotates the aim.

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

"Now for the final problem: getting the balls to collide with each other and the pockets"

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.
collideract(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 `collideract()` 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)
```

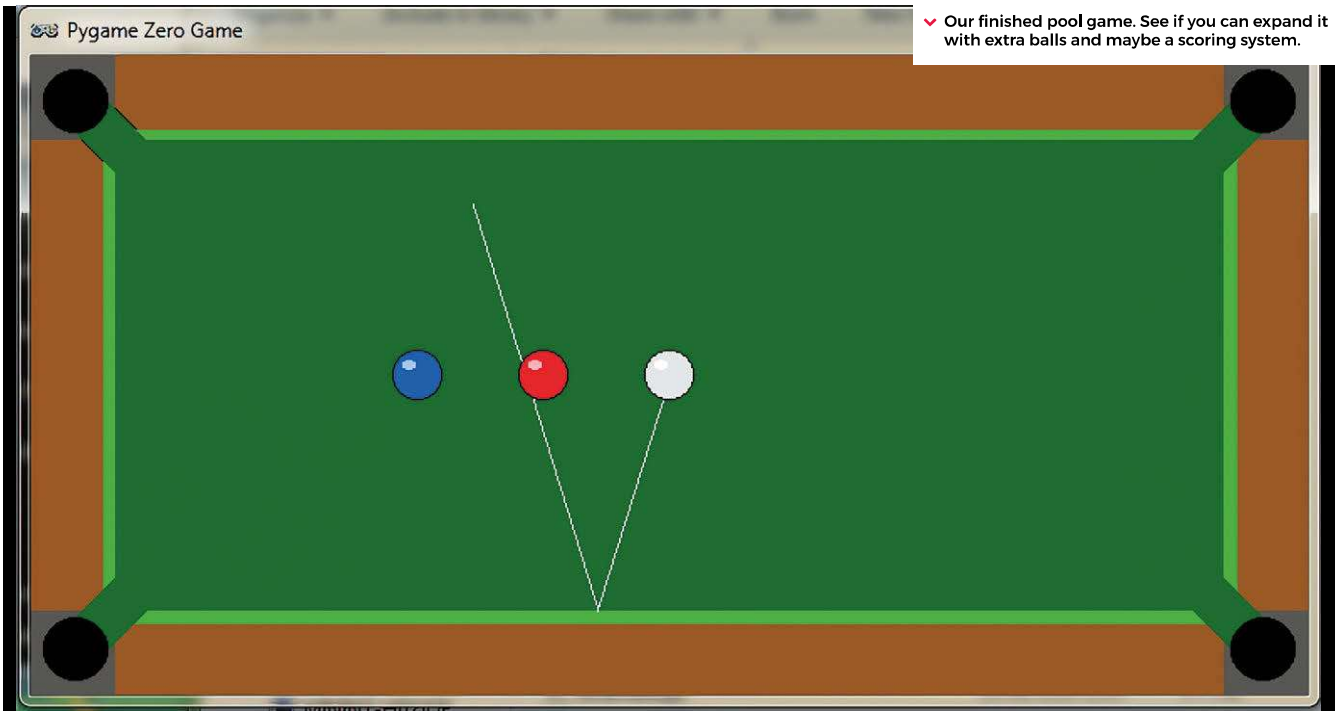
And there you have it: the beginnings of an arcade pool game in the *Side Pocket* tradition. You can get the full code and assets from **wfmag.cc/wfmag36**, and you can find some suggestions for improving and expanding the game in the box on the right.

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!



Our finished pool game. See if you can expand it with extra balls and maybe a scoring system.