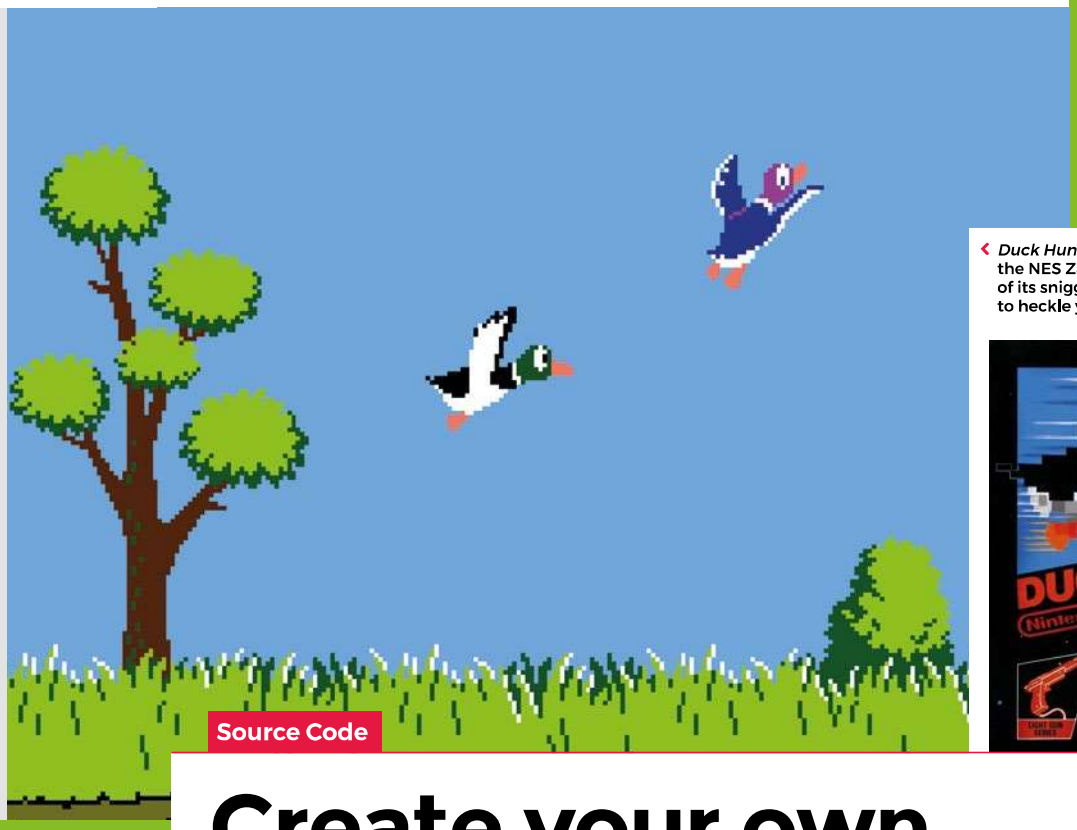


Duck Hunt made effective use of the NES Zapper, and made a star of its sniggering dog, who'd pop up to heckle you between stages.

Source Code

Create your own 2D shooting gallery

Rik shows you how to hit enemies with your mouse pointer as they move around the screen

AUTHOR
RIK CROSS

Shooting galleries have always been a part of gaming, from the *Seeburg Ray-O-Lite* in the 1930s to the light gun video games of the past 40 years. Nintendo's *Duck Hunt* – played with the NES Zapper – was a popular console shooting game in the early eighties, while titles such as *Time Crisis* and *The House of the Dead* kept the genre alive in the nineties and 2000s.

Here, I'll show you how to use a mouse to fire bullets at moving targets. Code written to instead make use of a light gun and a CRT TV (as with *Duck Hunt*) would look very different. In these games, pressing the light gun's trigger would cause the entire screen to go black and an enemy sprite to become bright white. A light sensor at the end of the gun would then check whether the gun is pointed at the white sprite, and if so would register a hit. If more than one enemy was on the screen when the trigger was pressed, each enemy would flash white for one frame in turn, so that the gun would know which enemy had been hit.

I've used two Pygame Zero event hooks for dealing with mouse input. Firstly, the `on_mouse_move()` function updates the position of the crosshair sprite whenever the mouse is moved. The `on_mouse_down()` function reacts to mouse button presses, with the left button being pressed to fire a bullet (if `numberofbullets > 0`) and the right button to reload (setting `numberofbullets` to `MAXBULLETS`).

"Pressing the light gun's trigger would cause the entire screen to go black"

Each time a bullet is fired, a check is made to see whether any enemy sprites are colliding with the crosshair – a collision means that an enemy has been hit. Luckily, Pygame Zero has a `colliderect()` function to tell us whether the rectangular boundary around two sprites intersects. If this helper function wasn't available, we'd instead need to use sprite x and y coordinates, along with width and height data (w and h below) to check whether the two sprites intersect both horizontally and vertically. This is achieved by coding the following algorithm:

- Is the left-hand edge of sprite 1 further left than the right-hand edge of sprite 2 (`x1 <x2+w2`)?
- Is the right-hand edge of sprite 1 further right than the left-hand edge of sprite 2 (`x1+w1 > x2`)?
- Is the top edge of sprite 1 higher up than the bottom edge of sprite 2 (`y1 < y2+h2`)?
- Is the bottom edge of sprite 1 lower down than the top edge of sprite 2 (`y1+h1 > y2`)?

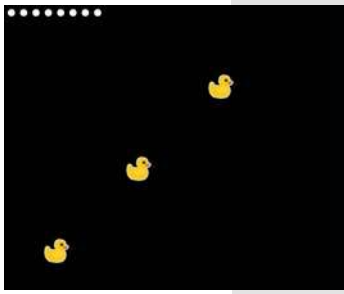
If the answer to the four questions above is `True`, then the two sprites intersect (see **Figure 1**). To give visual feedback, hit enemies briefly remain on the screen (in this case, 50 frames). This is achieved by setting a `hit` variable to `True`, and then decrementing a timer once this variable has been set. The enemy's deleted when the timer reaches 0.

A simple shooting gallery in Python

You'll need to install Pygame Zero to get Rik's code running. You can find instructions at **wfmag.cc/pgzero**

Download the code from GitHub: **wfmag.cc/wfmag20**

```
WIDTH = 800
HEIGHT = 800

crosshair = Actor('crosshair')

# creating a new enemy
def newEnemy(pos):
    e = Actor('enemy', pos=pos)
    e.hit = False
    e.timer = 50
    e.hits = []
    return e

# creating a bullet that has hit an enemy
def newHit(pos):
    h = Actor('bullet', pos=pos)
    return h

# create 3 enemies at various positions
enemies = []
for p in [(0,200),(-200,400),(-400,600)]:
    enemies.append(newEnemy(p))

numberofbullets = 8
MAXBULLETS = 8

def on_mouse_move(pos, rel, buttons):
    crosshair.pos = pos

def on_mouse_down(pos,button):
    global numberofbullets
    # left to fire
    if button == mouse.LEFT and numberofbullets > 0:
        # check whether an enemy has been hit
        for e in enemies:
            if crosshair.colliderect(e):
                # if hit, add position to 'hits' list
                e.hits.append(newHit(pos))
                e.hit = True
                break
        numberofbullets = max(0, numberofbullets -1)
    # right to reload
    if button == mouse.RIGHT:
        numberofbullets = MAXBULLETS

def update():
    for e in enemies:
        # hit enemies continue to display
        # until timer reaches 0
        if e.hit:
            e.timer -= 1
            if e.timer <= 0:
                enemies.remove(e)
        # move enemies if not hit
        else:
            e.x = min (e.x+2, WIDTH)

def draw():
    screen.clear()
    # draw enemies
    for e in enemies:
        e.draw()
        # draw enemy hits
        for h in e.hits:
            h.draw()
    crosshair.draw()
    # draw remaining bullets
    for n in range(numberofbullets):
        screen.blit('bullet',(10+(n*30),10))
```



Our simple shooting gallery in Python. You could try adding randomly spawning ducks, a scoreboard, and more.

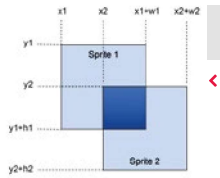


Figure 1: A visual representation of a collision algorithm, which checks whether two sprites intersect.

As well as showing an enemy for a short time after being hit, successful shots are also shown. A problem that needs to be overcome is how to modify an enemy sprite to show bullet holes. A **hits** list for each enemy stores bullet sprites, which are then drawn over enemy sprites.

Storing hits against an enemy allows us to easily stop drawing these hits once the enemy is removed. In the example code, an enemy stops moving once it has been hit. If you don't want this behaviour, then you'll also need to update the position of the bullets in an enemy's **hits** list to match the enemy movement pattern.

When decrementing the number of bullets, the **max()** function is used to ensure that the bullet count never falls below 0. The **max()** function returns the highest of the numbers passed to it, and as the maximum of 0 and any negative number is 0, the number of bullets always stays within range.

There are a couple of ways in which the example code could be improved. Currently, a hit is registered when the crosshair intersects with an enemy – even if they are barely touching. This means that often part of the bullet is drawn outside of the enemy sprite boundary. This could be solved by creating a clipping mask around an enemy before drawing a bullet. More visual feedback could also be given by drawing missed shots, stored in a separate list.