Game development with pygame()
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/pygame/pygame.pdf

---

1 The pygame library

We're taking a big detour to consider the creation of video games in Python using the `pygame()` library. We will encounter some issues that seem far from our mathematical and computational concerns. However, a computer game is an example of a real computational issue with its own peculiar problems to be solved. It is worth learning how the same language we have been studying can be used for this purpose as well.

We must begin with a very simple outline of how a pygame video game program works. The most obvious issues are that a video game must present a display on the screen, allow interactions from a user, and adjust the display in response to user actions. This happens in real time, and so there must be no noticeable response delay.

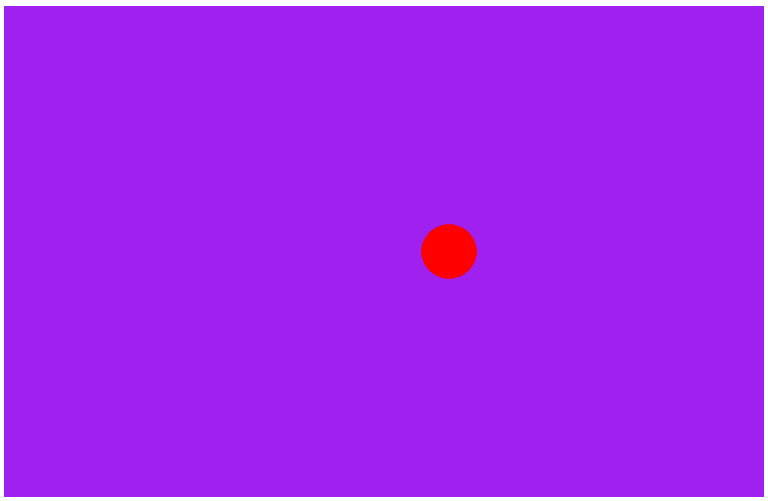
The game is essentially creating an animation one frame at a time. Typically, the frames are refreshed perhaps 60 times a second. Space is also discretize, with a fixed width and height, often 480 pixels high and 640 wide. And many of the items that will appear on the screen are themselves handled in an unusual way. Often an item such as a tree, a text legend, a character, is defined as a "rect", that is, a rectangle containing information. In this way, the entire rectangle can easily be moved left or right, up or down, or even rotated, as part of the game play. The rect usually contains an image, such as a `jpg` or `png` graphics file. These images may include a surrounding area of transparency, so that even though the object is stored in a rectangle, the transparent parts allow the background to show through, making a smooth blend.

If each change to the scene were displayed instantly, the player would see some bizarre transitions, and even some cases where an object appeared both where it was, and where it was about to be moved. To avoid such discontinuities, a double buffering system is used. There is a current screen, which the player sees, and a next screen, which is being prepared. Once the next screen has been completely defined, the two screens are swapped, using the `flip()` function.

The execution of the game begins with some initializations, including the screen size, and the initial background. There follows an infinite game loop, which checks for events, updates the screen in response to events or to the natural speed of objects on the screen, and then flips the old and new screens. Game play continues until the `quit` event is detected - that is, the player clicks on the **X** in the upper right corner of the screen.

2 Move the Circle

We start with a game small enough to see as a single block of code. This means it really can't be very elaborate, but it gives us a taste of what is to come. This game begins with a red circle in the middle of the screen. The user can press any of the `a`, `s`, `d` or `w` keys to make the circle jump left, down, right, or up. That's it. The Python code is simple enough to work out how this is done.



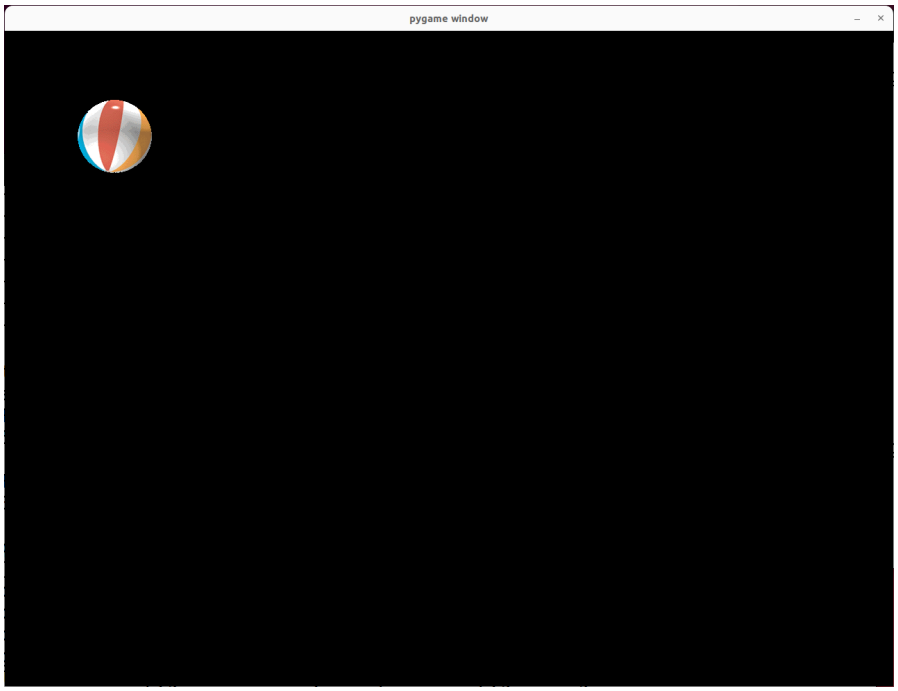
Certainly the least interesting game in history!

```
import pygame

pygame.init()
screen = pygame.display.set_mode ( ( 1280, 720 ) )
clock = pygame.time.Clock()
running = True
dt = 0

player_pos = pygame.Vector2(screen.get_width() / 2, screen.get_height() / 2)

while running:

  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      running = False

  screen.fill("purple")
```

```
    pygame.draw.circle ( screen , "red", player_pos , 40 )

    keys = pygame.key.get_pressed()
    if keys[pygame.K_w]:
      player_pos.y -= 300 * dt
    if keys[pygame.K_s]:
      player_pos.y += 300 * dt
    if keys[pygame.K_a]:
      player_pos.x -= 300 * dt
    if keys[pygame.K_d]:
      player_pos.x += 300 * dt

    pygame.display.flip()

    fps = 60
    dt = clock.tick ( fps ) / 1000

pygame.quit()
```

3 The Bouncing Ball

The bouncing ball game doesn't actually let the player do anything. It displays a colorful ball as it bounces around on the screen. However, it does suggest how animation is handled in a game. The ball is associated with a `rect` object named `ballrect`. It then is given animation using a `move()` function that includes x and y speed components.



The round ball is actually a rectangular picture with transparency.

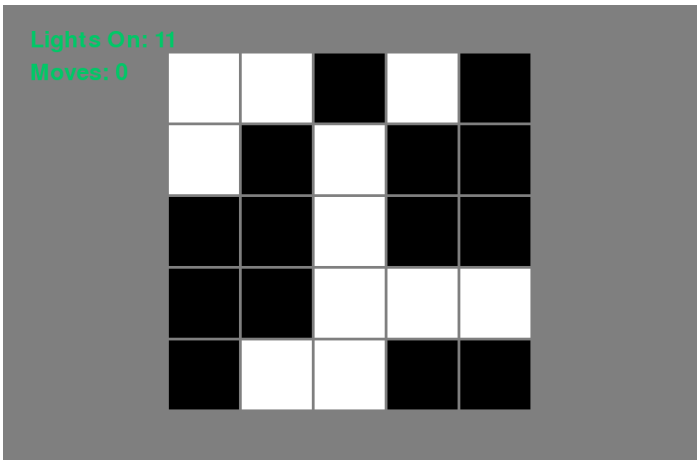
```
import pygame

pygame.init()
```

```
width = 1280
height = 960
screen = pygame.display.set_mode ( [ width, height ] )
speed = [ 1, 1 ]
black = 0, 0, 0
running = True

ball = pygame.image.load ( "bouncing_ball_data/intro_ball.gif" )
ballrect = ball.get_rect()

while running:

  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      running = False

  ballrect = ballrect.move ( speed )

  if ballrect.left < 0 or width < ballrect.right:
    speed[0] = - speed[0]
  if ballrect.top < 0 or height < ballrect.bottom:
    speed[1] = - speed[1]

  screen.fill ( black )
  screen.blit ( ball, ballrect )
  pygame.display.flip ( )

pygame.quit ( )

return
```

The image of the ball is stored in a local directory. A copy of this image is retrieved, and associated with a rect called `ballrect` which is then referenced when we want to know where the ball is or to change its speed. Notice how the ball's speed is reflected if it reaches the boundary.

4 Lights Out

Lights Out! was originally an electronic children's toy. It displayed a 5×5 grid of squares, each of which could light up. On starting the game, a seemingly random set of squares were illuminated. The player's task was to get all the lights switched off. Pressing any light reversed its status...but it also reversed the status of the (usually) four immediate neighbors. What seemed a simple task could perplex a child. It also intrigued mathematicians, because this puzzle has deep connections with linear algebra modulo 2, and because it is possible to determine that there are some initial conditions which are not solvable.



The player clicks on any square to reverse its status, and that of its (usually) four neighbors.

While the game screen looks terribly simple, the code is complex. For one reason, it offers both a $4x4$ and $5x5$ grid. It also allows a variation in which the boundaries of the grid wrap around, the so-called "toroidal" version of the game. And the game offers a startup menu page, and each choice on that menu requires a separate image file from which the full menu is constructed.

Computational people will be interested to see that, in response to the user choosing a square to press, the so called "von Neumann neighborhood" of four neighbors, must be calculated, after which any neighbors lying outside the grid limits are discarded.

The program also handles a lot of text messages, which are awkward to deal with, since the programmer wants these messages to be carefully centered. To do so requires knowing where the center of the screen is, and the font size being used.

So while this is a carefully crafted programming, the many details can make it difficult to see the main, simple action of the game loop:

- the user selects a square;
- the indices of the valid neighbors are determined;
- the state (ON/OFF) of each affected square is reversed;
- the image of each affected square is updated.
- the move and light counters are updated.
- the new screen is displayed.

Perhaps the main point of interest for this game is that its structure is similar to many other mathematical puzzle games involving a grid and user interactions that require the selection of a grid square on each step. So this program could be a starting point for a new program that does Life, Sudoku, Tetris, Tic-Tac-Toe, or other grid-based games and puzzles.

5 Chimp

Now let's consider a totally non-mathematical non-grid based game, which instead has some silly animations and sound effects. The goal is for the player, who controls the image of a fist, to punch a chimp as it moves left and right on the screen.

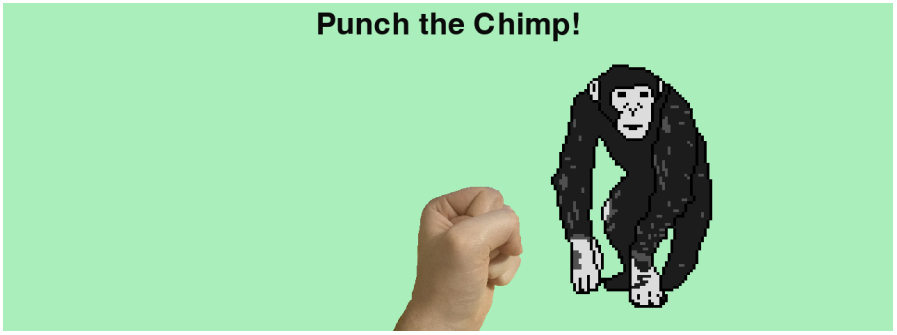
An unusual feature of this game is the way that the user's punch can hit or miss. The game examines the

current positions of the fist and the chimp and decides if they have collided. If so, the chimp spins around briefly in reaction, and a punching sound is heard - otherwise a "whiff" is heard instead.

The images of the fist and the chimp, and the sounds of the punch and whiff, are resources that the program needs to access. In this version of the code, all these items are files in a subdirectory.

The code begins with an extensive set of definitions of the behaviors of the chimp and the fist. First time readers should move ahead to the game loop, and study that simple process first.

The next item that should be considered is how the program determines whether the punch has landed. This is done by briefly inflating the rectangle around the fist, and using the collision checker to determine whether the inflated fist rectangle intersects the chimp rectangle. This question is answered by a call to `colliderect()`.



The chimp moves left and right; the player controls the punching fist.

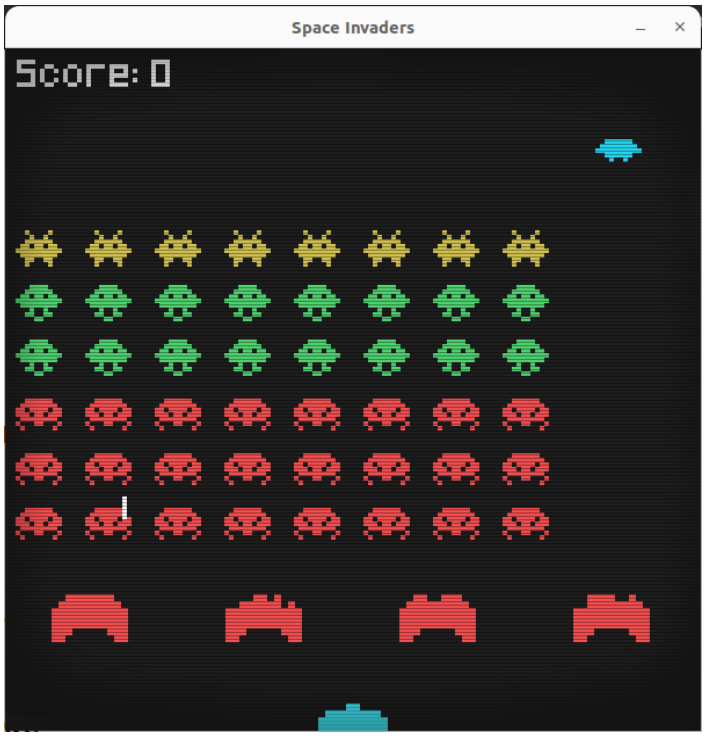
6 Space Invaders

Space Invaders was a classic arcade game in which an array of aliens gradually descends down the screen, dropping bombs, while the player slides back and forth at the bottom, hiding behind bunkers and then dashing out to fire a laser upwards.

I have an example of a simulation of this game using `pygame()`. It reproduces the more important features of the game. The code turns out to be quite complex, and is expressed using a lot of object-oriented statements. In the main function, however, we can still locate the game loop, within which there is the event handler, and the statements that update the screen. Notice, in particular, that the bunkers (called "obstacles" in the code) must gradually crumble under the alien attack. So displaying them is not simply a question of finding a fixed image. Each obstacle must be allowed to crumble in its own way, as the attack proceeds.

The player is only allowed a single shot at a time, and cannot fire again until the current shot has hit a target, or disappeared off the top of the screen. The aliens seem to fire at random, and not very accurately.

I am showing you this game, even though it is much more involved than the previous examples, because it is much more playable and interesting. If you find yourself attracted to the idea of developing your own games, this is a good starting point for you to work from!



A decent version of a famous video game, with action, collisions, and sound effects.