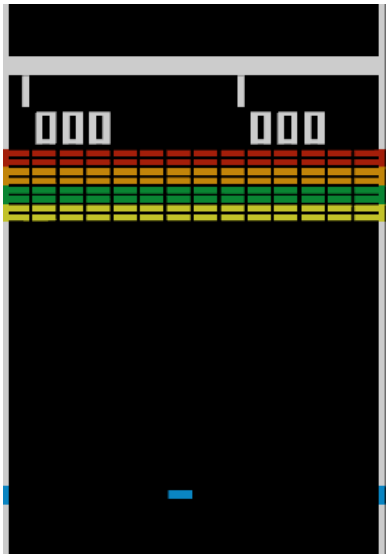
AE1205 Assignment 4: Breakout

Your job in Assignment 4 is to recreate the 1976 arcade classic *Breakout*. In case you don't know it (it was, after all, quite a while ago), in Breakout, the goal is to clear several layers of bricks that fill the top of the screen, by repeatedly bouncing a ball off a paddle into them (for an interactive example, go here: https://breakout-game.com). The game was received very well, and spawned dozens of imitation games and variants. The image below shows a screenshot of the original arcade game.



For the assignment you will use PyGame to generate the graphics. Have another look at lecture 6, chapter 12 of the reader, and the online documentation (https://www.pygame.org/docs/) to help you get started with the basics.

In the assignment, we'll follow these steps:

1. Initialise PyGame, and set up the game window.
2. Initialise the data that defines the 'state' of the game.
3. Drawing a static picture: blocks, paddle, and ball.
4. Making the game dynamic: the game loop, and processing keyboard inputs.
5. Updating the state of the game: numerical integration of ball position, collision detection, and updating the block states.
6. More control! Making the paddle non-flat.
7. (Optional) A game isn't a game if you can't lose: counting the number of lives.

Step 1: Setting up your window

As discussed in the lecture, setting up PyGame and creating a window is pretty easy: The PyGame module itself has an `init()` function that you need to call, and creating a window requires you to specify a resolution (number of pixels width and height):

```
import pygame as pg

# Initialise PyGame
pg.init()

# Setting up the screen
screenwidth = 800
screenheight = 800
reso = (screenwidth, screenheight)
scr = pg.display.set_mode(reso)
```

Step 2: The state of the game

As a programmer, in any game you typically have to keep track of the state of all non-stationary (or otherwise non-constant) objects. We'll start out with three objects that you need to keep track of:

1. The paddle position
2. The ball position and velocity
3. The grid of bricks: which ones still exist, which ones have already been hit.

The game is two-dimensional, so paddle position, ball position, and ball velocity each consist of both an x and a y coordinate. Initialise these three vectors as Python lists `paddle_pos`, `ball_pos`, and `ball_spd`. Initialise the paddle position horizontally in the middle, vertically 20 pixels from the bottom of the window, the ball position 30 pixels to the right of the middle of the window, and give it an initial speed of `[600, 600]`.

The grid of bricks we can set up as a list of lists. Because the state of each individual brick has just two possible values (existing or cleared), the initial grid can contain a value of `1` for each brick. These can then be changed to `0` for each brick that is hit during the game.

Be aware of the following caveat when initialising this grid: while it may be tempting to set up the grid like this: `grid = nrows * [ ncols * [1]]`, this will get you into trouble. Just try the following example:

```
>>> grid = 3 * [3 * [1]]
>>> grid[0][0] = 0
>>> print(grid)
[[0, 1, 1], [0, 1, 1], [0, 1, 1]]
```

As you can see, changing the first element in the first row actually changes the first element in all rows! The reasons for this will be explained in lecture 7. To solve this problem in this assignment, create your grid instead with a **`for`**-loop. To set up the size of the grid, use `nrows = 8`, `ncols = 20`.

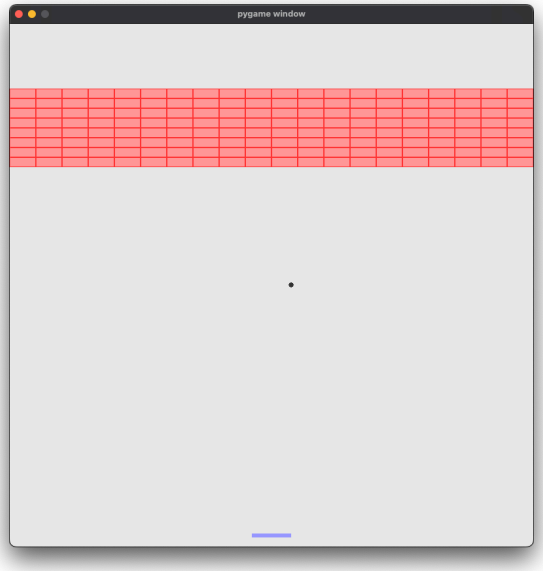
Step 3: Static drawing

Drawing the window begins by setting the background colour, using `scr.fill()`. Set this up with any colour you like :)

To draw the grid, ball, and paddle, we need some more information than just the game state: we need to know the size of each element, and for the grid we still need to know at what vertical position of the screen it should reside. Use the following values to set these properties as constants at the beginning of your file:

- **Grid:** The top row of bricks starts 100 pixels from the top of the window, brick width and height are 40 and 15 pixels, and there should be no space between bricks. Use `pg.draw.rect()` to draw each brick. To prepare for the animated game, already use the grid state from the previous step to check if each brick should be drawn.
- **Ball:** Draw the ball at its initial position using a circle with radius 4.
- **Paddle:** Draw the paddle at its initial position using a rectangle, with a width of 60, and a height of 6 pixels.

The resulting window should look like this:



Step 4: The game loop

Let's start moving stuff around! Because the game loop ends only on a certain condition (such as when escape is pressed), we need a **`while`**- loop to implement it. Let's start with the example from the lecture and reader:

```
# Keep track of escape press
escape = False

while not escape:
    # Check for ESC key or quit event:
    keys = pg.key.get_pressed()
    escape = keys[pg.K_ESCAPE]

    # And check if the window close button is clicked
    pg.event.pump()
    for event in pg.event.get():
        if event.type == pg.QUIT:
            escape = True
```

To move the paddle, we can use the arrow keys: `keys[pg.K_LEFT]` and `keys[pg.K_RIGHT]`. Use these key registrations to increment or decrement the horizontal paddle position in the game loop, by `gain` pixels each integration step. Here `gain` is another constant, which you can set to 5. Don't forget to limit this horizontal position to the edges of the window! The left edge of the paddle shouldn't exceed the left side of the window, and the right edge the right side.

When running this you'll probably notice that paddle speed is pretty high. You can further control this (and as an added benefit give your CPU some rest) with a `pg.time.wait(timestep)` call in your game loop. Tune the constant `timestep` to obtain an appropriate paddle speed.

Step 5: Updating the state and detecting collisions

This is where the ball speed comes into play: use Euler integration to update the ball position with the ball speed, as described in lecture 5. Use the timestep from the previous step (which is in milliseconds) to perform the numerical integration: `pos = pos + spd * timestep / 1000`.

Now things get interesting, because if we leave the speed like this the ball will just fly out of the screen. To make the game work we therefore need to detect collisions between the ball and the paddle, the

walls of the screen, and the bricks. The logic for this is roughly the same for each:

1. After updating the position of the ball, check if this position update intersects with a wall, a brick, or the paddle. To do this correctly you'd need to take into account the radius of the ball!
2. If the ball has hit a side wall or a side edge of a brick, the horizontal speed of the ball should be reversed for the next iteration. If it has hit the top wall, the paddle, or the top or bottom of a brick, the vertical speed of the ball should be reversed.
3. Most likely the calculated position of the ball isn't right at the edge of a wall, the paddle, or a brick, but somewhat over it. Calculate the distance with which the ball has exceeded this edge, and use it to mirror the ball position around the x or y coordinate of the object the ball struck.
4. When checking if the ball has hit a brick, it can be useful to first check if the ball position has ended up in between the top and bottom of one of the rows of the brick area. If it did, find the edge(s) of the bricks it could have intersected with, and in which order. Go through these bricks one by one to find out if they still exist, using the `grid` array. If a hit is found, update the `grid` state, and the speed of the ball accordingly.

Step 6: Make a rounded paddle

If you got your code working this far you'll probably have noticed that the bounces of the ball on the paddle are pretty predictable, and don't give you any real control over the ball. To improve this you can make the angle with which the ball bounces back from the paddle dependent on the location along the top of the paddle where the ball hits it. Your code should rotate the exit speed of the ball by a maximum of `maxangle` degrees. When the ball hits the extreme edges of the paddle this should be the complete negative and positive `maxangle` degrees. This rotation should depend linearly on the collision point along the paddle, if it hits the paddle somewhere in between the edges.

If you try this out you'll see that you can make the ball go purely horizontally, which isn't very productive. You should therefore limit the calculated exit angle so that the ball will always bounce back up.

Step 7: Game over! (Optional)

Congratulations! You are done with the assignment. But if you want to get your game closer to the real thing you need to at least be able to lose! This requires you to treat the bottom edge of the screen differently (no bounces here), keep track of the number of lives, and reset the speed and position of the ball for each life. It'd also be nice if you print the remaining number of lives somewhere on the screen.

Optional 2: Powerups!

The many variants of the breakout game also added features to the game. The most prominent one is the addition of powerups: hitting specific bricks would release some powerup to fall down, which if you catch it would give your paddle additional features (make it smaller/wider, let it change the magnitude of ball speed, or even give it a cannon!). The possibilities are endless!