Game Development 4 – Collisions

Last week we added obstacles to Birdie's world. However, touching an obstacle in any way did not cause Birdie any trouble. This week, we will change this by handling collisions between the bird and the obstacles.

Task 1: Introduce Function to Check for End of Game

The first step to handle collisions is to detect collisions in the first place. In our game we want a collision between birdie and an obstacle to cause the game to be over. We therefore define a function to check whether the game is over:

```
def is_game_over():
    return False
```

When called, this function checks if the game is over. To answer the question of whether the game is over this function can return (answer) "True" (yes) or "False" (no). A good way to start using such a function is to first return "False". This way, the game is never over (the same situation as before) but we can start using the function.

It is always good practice to use (call) a function that you defined as soon as possible to avoid mistakes. In this case we can now use is_game_over() in our main event loop after we handle the events:

```
while True:
    …
    if is_game_over():
        print ("Game over!")
    …
```

Note that this is the existing game loop – You only need to add two lines here! Now, run the game and observe the output in the shell. Is this what you expected to see? If not, go through the code again and ensure that you understand what is happening.

Task 2: Check for Collisions

Now that we are using the function to check whether a game is over we can use more sophisticated logic to find out whether the game should be over. In particular, in this case the game should be over, i.e. the function should return "True" if the bird collides with any of the obstacles. Otherwise it should return "False".

To check whether the bird collides with any obstacle we look at every obstacle in turn and check whether it and the bird collide as this is an easier task for the computer. To look at each obstacle in turn we use a so-called "for loop". Adapt your existing function to use a for-loop:

```
def is_game_over():
    for obstacle in obstacles:
        if obstacle.rect.colliderect(bird.rect):
            return True
    return False
```

Can you guess what the highlighted lines of code are responsible for and how they work? Run the game and observe the shell. What do you see now when you collide with an obstacle?

Task 3: Handling Game Over State

We can now detect when the bird collides with any obstacle. This should mean that the game is over. The second step in handling collisions is to decide what happens if a collision occurs. For simplicity we will simply halt the game and not allow the player to continue playing.

There are several ways to do this. One of the simplest is to call a function in the game event loo that never returns back to the user, i.e. never finishes executing.

```
def game_over():
    while True:
        clock.tick(30)
        pygame.event.pump()
```

Here, the last two lines ensure that the computer does not run out of breath and only runs the loop 30 times per second and not any more times and that any events that are sent are ignored. Please ask if you would like to know more about this.

Instead of printing “Game over!” to the shell, we can now call this function directly when we detect any collision. Replace the print statement with a call to this new function.

Can you explain why this function never finishes executing?

Task 4: Allowing the User to Exit the Game

Once a game is over, we can allow the user to exit the game by pressing the space key. Use **`pygame.key.get_pressed()`** and the **`K_SPACE`** to exit the game. You can exit the game by calling **`sys.exit()`.**

Add your code into the game_over() function. Why do we place the code here and not in handle_events()?