**CSE 1321L: Programming and Problem Solving I Lab**

**Lab 12**

**Collisions and Events**

# What students will learn:

- Detect collisions between two or more Rects.
- Events and the Event Queue.
- State-based key handling

# Content



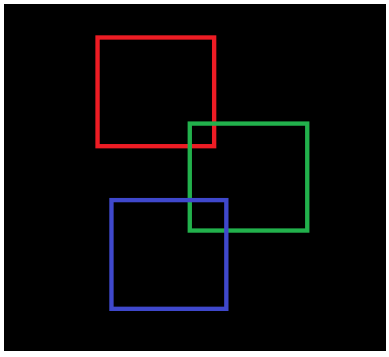
# Overview

In this lab we will leverage collisions, events, and player input control. Let's explain some of these concepts and how to implement them with Pygame.

## Collision

As we know, Rects are used to store spatial information. We create Rects to keep track of specific rectangular areas in our game. In some situations, it may be necessary to determine if two or more Rects collide, i.e.: If the area of one Rect **intersects** with the area of another Rects.

In the picture below, we can see 3 rectangles, where part of the red rectangle is inside of the green rectangle, and part of the blue rectangle is inside the green rectangle.



In this situation, we can say that the red and green rectangles are colliding. Similarly, we can say that the blue and green rectangles are also colliding. However, the red and blue rectangles are not colliding.

We could write and implement code that checks if two Rects are colliding. Fortunately, we do not have to, **Rect objects have several methods that handle collisions.** This is one of the reasons why we should prefer using Rects for storing area information.

To check if a surface collides with another surface, we can use one of the following surface object's functions (remember to use the dot operator to access these):

- **collidepoint(x, y)**
  - This method takes in two integers for the x and y coordinate and returns a Boolean and checks if a point is currently inside a Rect.
- **colliderect(Rect)**
  - This method takes in a rect type object and checks if any portion of the Rect in the parameter is inside the area of the rects calling this method. It returns a Boolean value.
- **collidelist(list)**
  - This method takes in a list containing rects. It returns the index of the first Rect in the list to collide with the rect calling this method. It returns -1 if there are no collisions.
- **collidelistall(list)**
  - This method takes in a list containing rects. It returns a list of all the indices that contain rects that collide with the rect calling this method. If there are no collisions, the returning list will be empty.

This is an example of how we can check rects colliding:

```
one = pygame.Rect(0, 0, 50, 50)
two = pygame.Rect(49, 49, 50, 50)
three = pygame.Rect(51, 51, 50, 50)
…
print(one.colliderect(two)) # returns True
print(one.colliderect(three)) # returns False
print(two.colliderect(three)) # returns True
```

Collisions can be used for many things. For example, If a character is falling down the screen and hits the ground, we can detect that the character has hit the ground by asking pygame if the character's Rect has collided with the ground's rect.

As another example, suppose a character shoots an enemy with a bow. The arrow will likely have its own rect, so pygame knows where to draw it on the screen. To determine if the enemy has been hit with the arrow, simply call **arrow_rect.colliderect(enemy)**, if the method call returns True then we can perform the corresponding actions like decreasing the enemy's health points.

## Events

Every time something happens in or to the Pygame window, that thing generates an Event. Moving the mouse around, clicking with the mouse, pressing keys on your keyboard, and minimizing the window are just a few examples. These events are then stored in the Event Queue, which keeps track not only of what events have happened, but the order in which they happened.

The Event Queue is crucial for communication with the outside world: While we tell the user what's going on inside the game through drawing Surfaces to the Display, the user tells us how they would like to interact with the game through Events.

The most common way to interact with the Event Queue is by iterating through it using

```
pygame.event.get():
for event in pygame.event.get():
      # handle your events here.
      if event.type == TYPE:
            pass # do something
```

For every event in the Event Queue, we must first check its type. Depending on the event type, we will take appropriate action. For example, if we detect that the user pressed the right arrow, we can try to move the character on screen to the right. Here are some events you'll likely need to use, along with what fields it has available:

- **QUIT**:
  - User clicked the close button.
- **KEYDOWN**:
  - User pressed a keyboard key.
  - **key**:
    - what key was pressed. This doesn't translate to what the content of the key is, but to what Pygame constant it is associated with. For example, the ESCAPE key is associated with K_ESCAPE. The q key is associated with K_q. You can find a list of most of these constants in the next section.
  - **mod**:
    - if a modifier key was being pressed (SHIFT, CTRL, or ALT)
  - **unicode**:
    - the character corresponding to the key pressed. Pressing the "s" key on your keyboard and reading this field will give you a string containing the letter "s".
- **KEYUP**: The user released a keyboard key. Same fields as above.
- **MOUSEMOTION**: User moved the mouse inside the Display.
  - **pos**: the new mouse position as a tuple (x,y)
- **MOUSEBUTTONUP**: The user pressed a mouse button.
  - **pos**: position of the mouse as a tuple (x,y)
  - **button**: what button was being pressed. 1 for left button, 2 for middle button, 3 for right button.
- **MOUSEBUTTONDOWN**: The user released a mouse button. Same fields as above.
- **MOUSEWHEEL**:
  - The user scrolled with the mouse wheel.
  - **y**: an integer representing how much the user scrolled. Positive numbers mean scrolling up and negative numbers mean scrolling down.

And here's an example as to how these events would be read from the Event Queue and acted upon:

```
for event in pygame.event.get(): # for each event inside the queue…
	if event.type == QUIT:
		sys.exit(0) # if the user clicks the close button, close the game
	if event.type == KEYDOWN and event.key == K_ESCAPE:
		sys.exit(0) # if the user presses ESCAPE, close the game
	if event.type == MOUSEMOTION:
		# if the mouse is moved, record its new position
		mouse_position = event.pos
	if event.type == KEYDOWN:
		if event.key == K_LEFT:
			# if left arrow is pressed, move the character left
			player = player.move(-5,0)
		elif event.key == K_RIGHT:
			# if right arrow is pressed, move the character right
			player = player.move(5,0)
```

## State based key handling

The key module is used specifically for getting input from the keyboard. Using the key module is more appropriate when you want to check if a key is being pressed continuously as opposed to if a key was just pressed or released.

The syntax for it is as follows:

```
# main gameplay loop
while True:
	for event in pygame.events.get():
		pass # handle all events

	# outside the Event Queue loop

	# gets all the keys currently being pressed
	keys = pygame.key.get_pressed()
	if keys[K_w]: # checks if w is being pressed
		# moves the player up every time the main loop iterates
		player.move(0,-5)
	if keys[K_s]: # checks if s is being pressed
		# moves the player down every time the main loop iterates
		player.move(0,5)
```

Notice that using **get_pressed()** is only appropriate for checking if keys are **currently being pressed**. It's not appropriate for being used to let the user type in words into the program, as the order that the keys were pressed is not preserved. If you wish to preserve the orders in which the keys were pressed, use the Event Queue.

The key module documentation contains a list of all key constants available in Pygame. You can find all available constants at the Pygame key documentation, at: https://www.pygame.org/docs/ref/key.html.

As with previous weeks, all labs should have the appropriate file names:

- Lab12A.py
- Lab12B.py
- Lab12C.py

Lastly, make sure you review the sample output and make sure the output of your program follows the exact same format shown in the **Demo Videos**.

# Lab12A: Collision Color Toggle

Build a simple Pygame application containing a vertical bar centered in the x-axis that changes color every time an oscillating blue square passes through it.

## Requirements:

- The display size should be 400 x 400 pixels.
- The application framerate must be fixed to 60 Frames Per Second
- The background color should be black.
- The vertical rectangle should have a dimension of 50 x 400 pixels.
- The vertical rectangle should be positioned centered on the x-axis.
- The vertical rectangle should be colored Red.
- The oscillating square should have a dimension of 50 x 50 pixels.
- This square should be placed on the left edge of the display and centered on the y-axis.
- This square should be colored Blue.
- The square should move to the right edge of the display with a speed of 5 pixels.
- Once the square right edge touches the display right edge, it should move to the left with the same speed.
- Repeat this motion continuously.
- Using **colliderect()**, change the Red vertical rectangle to green whenever the two rects are colliding, changing it back to red when the two rects are not colliding anymore.
- **Both the red rectangle and the blue square should implement a Rect to handle positioning and handle movement.**
- The program should stop when the user presses ESCAPE.
- The program should stop when the user closes the window.

# Lab12B: Collision Color Toggle 2

This lab expands on what you have built in Lab12A.

## Requirements:

- The application should contain 2 more rects.
- The application framerate must be fixed to 60 Frames Per Second
- These rects will have similar specifications to the blue squares.
- The second (top) square should be placed in the top left corner of the display.
- The third (bottom) square should be placed in the bottom left corner of the display.
- All three rects will have the same motion as in Lab12A, they all move side to side.
- The top square will move at a speed of 10.
- The middle square will move at the same speed as in Lab12A, 5
- The bottom square will move at a speed of 20.
- Using **collidelist()** or **collidelistall()**, change the Red vertical rectangle to green whenever **ANY** of the other three blue surface's rect collide with the red surface rect. Change the surface color back to Red if no collision is occurring.
- **All surfaces, the red rectangle and the blue squares should implement a Rect to handle positioning and handle movement.**

- The program should stop when the user presses ESCAPE.
- The program should stop when the user closes the window.

# Lab12C: Movement Control

Build a Pygame application that displays a blue square in the center of the screen. The player can control the square's movement using the keyboard.

## Requirements:

- The display size should be 500 x 500 pixels.
- The application framerate must be fixed to 60 Frames Per Second
- The background color should be black.
- The application should display a blue square of dimensions 50 x 50 pixels.
- The initial position of this square should be in the center of the display.
- Using State-Based key handling:
    - If the user presses the “w” key:
        - Move the Rect UP 5 pixels.
        - Print to the terminal the following event log:
          `[EVENT] KEY STATE: W KEY IS BEING PRESSED -> MOVING PLAYER UP`
    - If the user presses the “a” key:
        - Move the Rect to the LEFT 5 pixels.
        - Print to the terminal the following event log:
          `[EVENT] KEY STATE: A KEY IS BEING PRESSED -> MOVING PLAYER TO THE LEFT`
    - If the user presses the “s” key:
        - Move the Rect DOWN 5 pixels.
        - Print to the terminal the following event log:
          `[EVENT] KEY STATE: S KEY IS BEING PRESSED -> MOVING PLAYER DOWN`
    - If the user presses the “d” key:
        - Move the Rect to the RIGHT 5 pixels.
        - Print to the terminal the following event log:
          `[EVENT] KEY STATE: D KEY IS BEING PRESSED -> MOVING PLAYER TO THE RIGHT`
- Using Event-Based key handling:
    - If the user presses the “r” key:
        - Place the square back at the center of the display.
        - Print to the terminal the following event log:
          `[EVENT] KEYDOWN: USER PRESSED BUTTON R -> RESETTING PLAYER POSITION`
- Do not allow the user to go offscreen.
    - You must also ensure the square does not move outside the display boundaries by checking before applying movement.
    - If the users try to go offscreen print the following event log:
      `[EVENT] KEY STATE: X KEY IS BEING PRESSED -> CANNOT MOVE PLAYER OUT OF BOUNDS`
    - **X** being the key being pressed, either W, A, S, or D
- **The blue square should implement a Rect to handle positioning and handle movement.**
- The program should stop when the user presses ESCAPE.
- The program should stop when the user closes the window.

## Submission Instructions:

- Programs must follow the output format provided in the Demo Videos.
- Programs must be working correctly.
- Programs must be written in Python.
- Programs must be implemented exclusively using Pygame
- Programs must be submitted with the correct **.py** format.
- Programs must be saved in files with the correct file name:
    - Lab12A.py
    - Lab12B.py
    - Lab12C.py
- Programs (source code files) must be uploaded to Gradescope by the due date.