CSE 1321L: Programming and Problem Solving I Lab

Lab 8

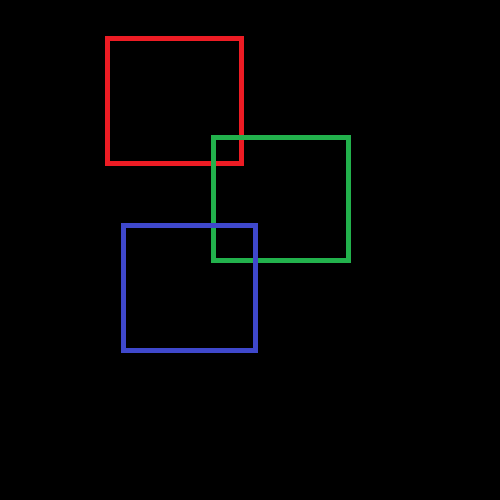
Collisions and Events

What students will learn:

* Detect collisions between two or more Rects
* Events and the event queue

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 Rect.

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 rectangle are colliding. Similarly, we can say that the blue and green rectangle are also colliding. However, the red and the blue rectangle are not colliding.

We could write code which checks if two Rects are colliding. Fortunately, we don’t have to, and it’s one of the reasons why we should prefer using Rects for storing area information: Rect objects have methods which can check collisions for us.

collidepoint(x,y) -> bool: Checks if a point is currently inside a Rect  
colliderect(Rect) -> bool: Checks if any portion of the Rect in the parameter is inside the area of the Rect calling this method

collidelist(list) -> index: Returns the index of the first Rect in the parameter list to collide with the Rect calling this method. Returns -1 if there are no collisions.  
collidelistall(list) -> list: 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.

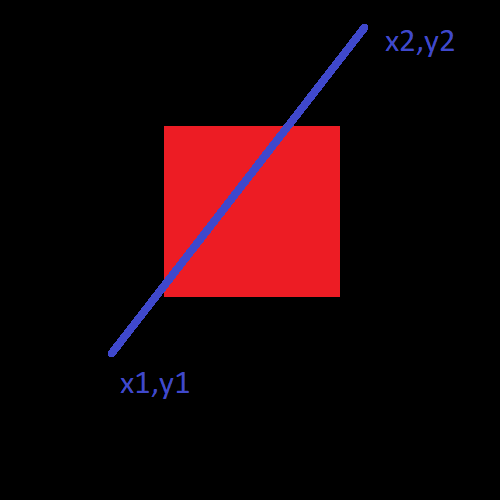
As an example, supposed the code below:

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

Finally, we can also check if a portion of a line is inside a Rect by using clipline():

clipline(x1,y1,x2,y2) -> ((cx1,cy1),(cx2,cy2)) or ()

The method above, called from a Rect, takes in 4 parameters: the coordinates of one end of a line and the coordinates of the other end of a line. It returns the coordinates of the line cropped by the Rect’s area. You can see that in the example below:

 A red square with green line

Description automatically generated

The first picture contains a Rect drawn to the screen through a Surface. A line is drawn through the Rect using pygame.draw.line(screen, (0,0,255), (x1,y1), (x2,y2)). As we can see, part of the blue line goes over the red rectangle. As such, if we were to call that Rect’s clipline() and pass the coordinates of the blue line, the result would be the green line: the part of the blue line which is inside the red rectangle.

If no parts of a line are inside the Rect, then clipline() will return an empty tuple.

Collision can be used for many things: 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 another 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 second character has been hit with the arrow, simply call arrow\_rect.colliderect(enemy), taking action as appropriate.

Pygame 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)

Pygame Key: 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 events in pygame.events.get():

pass **# handle all events**

**# outside the Event Queue loop**

keys = pygame.key.get\_pressed() **# gets all the keys currently being pressed**

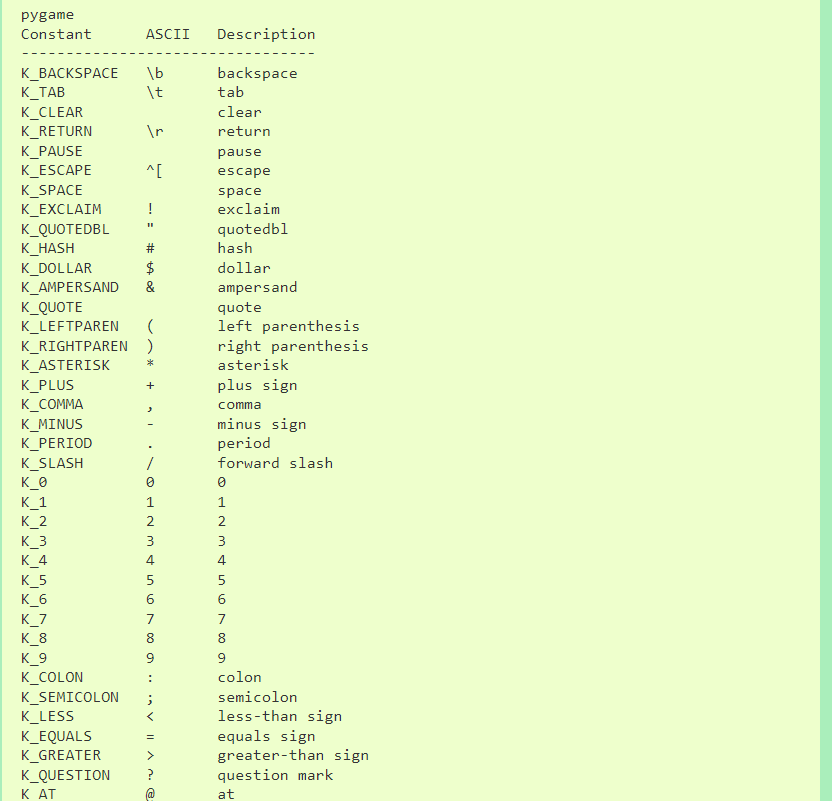
if keys[K\_w]: **# checks if w is being pressed**

player.move(0,-5) **# moves the player up every time the main loop iterates**

if keys[K\_s]: **# checks if s is being pressed**  
player.move(0,5) **# moves the player down every time the main loop iterates**

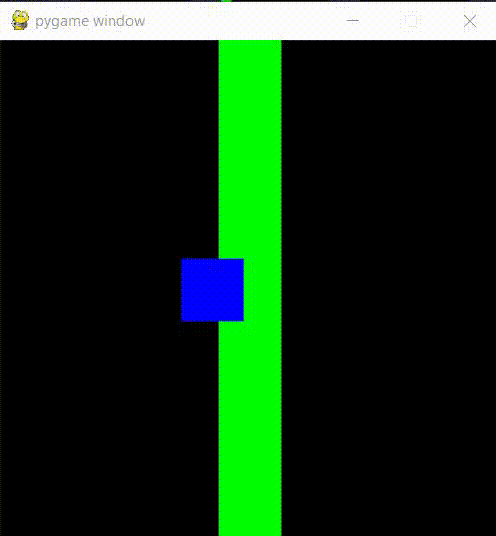
Notice that using get\_pressed() is only appropriate for checking if keys are currently being pressed. It’s **not appropriate for being used to let the user type in words into the program**, as the order that the keys were pressed is not preserved. If you wish to preserve the orders in which the keys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

The key module documentation contains a list of all key constants available in Pygame. Here are a few of the first ones:



You can find all available constants at the Pygame key documentation, at:

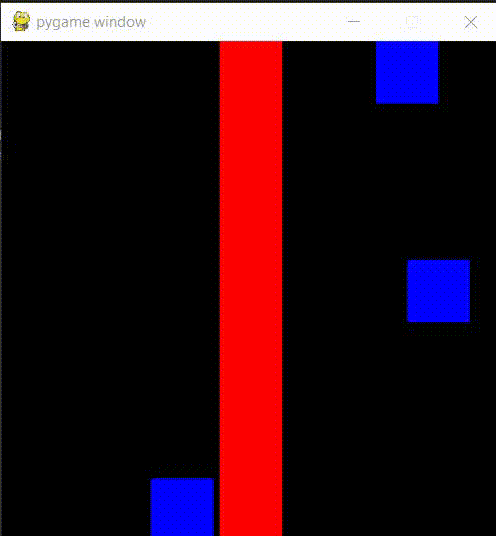
<https://www.pygame.org/docs/ref/key.html>.

**Lab8A**: Create a Pygame Display that is 400x400. Create a Rect that is 50x400 and place it along the center of the Display, blitting to the Display a red Surface at the Rect’s coordinates.

Create a Rect that is 50x50 and place its left border at the center of the left border of the Display. Blit to the Display a blue Surface at the Rect’s coordinates.

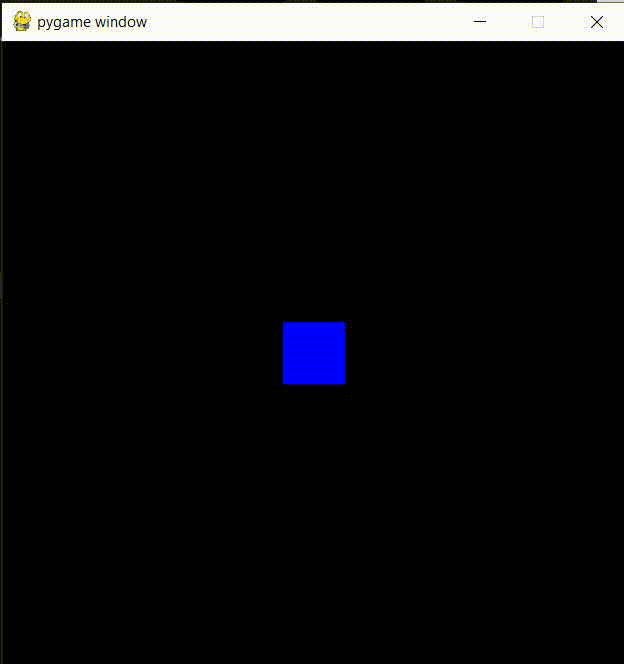
Move the blue Rect from the left to the right of the Display at a speed of 5, and then back again from the right to the left. Repeat this cycle until the user closes the program.

**Using colliderect(), change the red Surface to green whenever the two Rects are colliding, and change it back to red when the two Rects are not colliding.**



**Lab8B**:Expand on Lab8A by having two more Rects with associated blue Surfaces, one which bounces along the bottom border of the Display at a speed of 20, and one which bounces along the top border at a speed of 10.

**Using collidelist() or collideall(), change the red Surface’s color to green whenever ANY of the other three blue Surfaces’ Rects collide with the red Surface’s Rect. Change the Surface color back to red if no collision is occurring.**



**Lab8C**: Create a Display that is 500x500. Create a Rect that is 50x50, placing it at the center of the Display. Blit to the Display a blue Surface at the Rect’s coordinates and with the Rect’s dimensions.

**Read from the Event Queue, taking the following actions:**

* If the user presses w, move the Rect up 5.
* If the user presses s, move the Rect down 5.
* If the user presses d, move the Rect 5 to the right.
* If the user presses a, move the Rect 5 to the left.
* If the user presses r, place the Rect back at the center of the Display.

**Do not allow the user to go offscreen.** In the example above, you can see at the bottom right which key is being pressed. **You do not need to include what key is being pressed to your Display.** It is only present here for demonstration purposes.

**Submit your files to Gradescope as Lab8A.py, Lab8B.py, and Lab8C.py**.