# Pygame - Dev Notes - Control - Move - Coordinate Plane

A few notes on using controls to move shapes with Pygame.

## Contents


- Intro
- Interaction and events
    - keyboard
- 4-point
    - left and right
    - up and down
- 8-point
- Jump
- Jump and fall
- Move and jump


## Intro

We can listen for controller events, for example a keyboard, and then move our shapes around the game window.

These shapes may be moved anywhere on the screen using a standard coordinate plane.

So, we end up with a couple of patterns. These include the standard 4-point,

- left, right, up, and down

which equates to the standard cardinal points we find on a compass. The standard 4-point coordinate plane.

We may also use a broader 8-point pattern, which extends to

- up and right, up and left
- down and right, down and left

The 8-point coordinate plane may be achieved in a Pygame window either explicitly or implicitly, depending on how we've configured the listeners for interaction events.

## Interaction and events

After importing `pygame.events`, we may create required listeners for interaction control, e.g. for a key pressed down on a player's keyboard.

```
...
# check keyboard events - keydown
if event.type == pygame.KEYDOWN:
...
```

or a key released up

```
# check keyboard events - keyup
if event.type == pygame.KEYUP:
```

### 4-point

After creating a standard listener for an interaction event, for example a keyboard event, we may initially move our shape using one of 4-points on a coordinate plane. e.g.

- left, right, up, and down

#### left and right

We may then check a specific key event relative to keydown, perhaps a player request to move a shape left or right.

On the `KEYDOWN` event, we update the boolean value for the requested key,

```
# check keyboard events - keydown
if event.type == pygame.KEYDOWN:
    if event.key == pygame.K_LEFT:
        leftDown = True
    if event.key == pygame.K_RIGHT:
        rightDown = True
```

and then reset it to `FALSE` on the `KEYUP` event,

```
# check keyboard events - keyup
if event.type == pygame.KEYUP:
    if event.key == pygame.K_LEFT:
        leftDown = False
    if event.key == pygame.K_RIGHT:
        rightDown = False
```

We can use the set *boolean* value to modify the animation of a shape, e.g.

```
...
# event variables - keyboard
leftDown = False
rightDown = False
# some rect variables
rectSpeed = 4.0
...
# move left
if leftDown:
    # check shape doesn't exit window to left
    if rectX > 0.0:
        rectX -= rectSpeed
# move right
if rightDown:
    # check shape doesn't exit window to right
    if rectX + rectSize < winWidth:
        rectX += rectSpeed
...
```

In this example, we're checking the boolean value for left or right key down. If set to true, i.e. the player has pressed the key down, we can then check the shape's `x` coordinate position.

In effect, we check either the left or right side of the game window relative to the key pressed. Then, we either increment or decrement the shape's `x` coordinate by the set speed for our animation.

#### up and down

We may also use such interaction events to animate our shape up or down the screen. Again, we set a boolean value to `TRUE` or `FALSE` relative to the `KEYUP` or `KEYDOWN` event.

Then, we can animate our shape up and down the game window, e.g.

```
...
# event variables - keyboard
upDown = False
downDown = False
# some rect variables
rectSpeed = 4.0
...
# move up
if upDown:
    # check shape doesn't exit window at top
    if rectY > 0.0:
        rectY -= rectSpeed
# move down
if downDown:
    # check shape doesn't exit window at bottom
    if rectY + rectSize < winHeight:
        rectY += rectSpeed
```

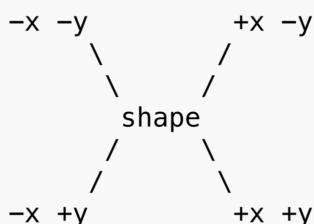
### 8-point

In addition to the standard left, right, up, and down directions, we may also combine these events to allow a user to move a shape in a diagonal direction.

For example, a player may simultaneously press `KEYDOWN` on both up and right. This will allow a player to move a shape at a 45 degree angle.

```
...
# move up
if upDown:
    # check shape doesn't exit window at top
    if rectY > 0.0:
        rectY -= rectSpeed
# move right
if rightDown:
    # check shape doesn't exit window to right
    if rectX + rectSize < winWidth:
        rectX += rectSpeed
```

Of course, a player may also use other available combinations to move the shape at one of 4 available angles of 45 degrees.



### Jump

To make a shape *jump*, we can start by defining a useful boolean variable `shapeJump`. We can then simply update this value to define whether the character is jumping or not.

We can also define a default pixel height for the jump itself. In effect, this is simply defining how far to move the shape up the game window.

```
jumpHeight = 30.0
```

Then, we can add a listener for the defined key. For example, we might simply use the obvious *UP* directional arrow on our keyboard,

```
...
# check keyboard events - keydown
if event.type == pygame.KEYDOWN:
    # check for directional UP key
    if event.key == pygame.K_UP:
        if not shapeJump:
            shapeJump = True
            shapeJY += jumpHeight
...
```

So, in this example, we're listening for the standard player `KEYDOWN` event, and then the actual directional *UP* key event. We check the boolean value of the variable `shapeJump`, update to `True` if the shape is not already jumping. Then, we incrementally update the value of the shape's requested jump Y value, `shapeJY`.

To make the shape jump, or effectively move up the screen per iteration of the game loop, we can define a function to handle this jump, `jump()`. For example,

```
def jump():
    global shapeY, shapeJY, shapeJump

    # check if shape in air - use gravity to descend
    if shapeJump == True:
        shapeY -= shapeJY
        print("in the air %8.2f" % (shapeJY))
        shapeJump = False
```

We can check the output of the jump up the screen by simply printing the formatted float to the terminal. If you run this example, you'll notice that the shape will keep jumping as the player presses the *UP* directional key, well beyond the bounds of the top of the game window.

### Jump and fall

Now, we could make the shape move down the window by listening for an explicit player key press on the *DOWN* directional key.

However, it's more natural, and expected behaviour, to allow our shape to fall after the player has pressed the *UP* arrow. In effect, we're allowing our shape to jump, and then fall with a real-world behaviour of **gravity**.

For example, we've already seen how to update a shape's position to make it jump. To make it fall, we need to check that the shape is in the air, so to speak, and then gradually modify gravity to lower the shape to the original starting position in the Pygame window.

```
def jump():
    global shapeY, shapeJY, shapeJump, gravity
    # check upward speed > 1.0
    if shapeJY > 1.0:
        # gradually decrease upward speed to less than 1.0
        shapeJY = shapeJY * 0.9
    else:
        # less than 1.0, reset to 0.0 to allow shape to fall
        shapeJY = 0.0
        # stop jump
        shapeJump = False

    # check if shape in air - use gravity to descend
    if shapeY < winHeight - shapeSize:
        shapeY += gravity
        gravity = gravity * 1.1
    else:
        shapeY = winHeight - shapeSize
        gravity = 1.0

    shapeY -= shapeJY
```

In the above example, we start by checking whether the shape is still moving up the screen, effectively if the jump is still in progress. Whilst the upward speed of the shape is still above `1.0`, we gradually start to decrease the speed so it will eventually reach a limit for the jump.

The faster we decrease this upward motion, the shorter the shape will appear to jump. This will also negate the overall effect of the value of the variable `jumpHeight`, which now has less iterations of the *game loop* to move the shape up the screen.

Then, we need to check if the shape is actually moving up the screen, or effectively in the **air** for the jump. If not, then the shape will simply come to a halt as it rises up the screen due to the decrease in upward speed and motion.

This is where we need to add the perception of **gravity** to the shape's motion. So, whilst the shape appears to be in the **air**, or jumping up the screen, we start to add the number of pixels we define for the variable **gravity** to our shape's upward movement.

As the shape starts to fall down the game window, we slowly increase the value of the `gravity` variable to suggest a realistic downward fall. If not, then the jump and fall will not be timed correctly, and a player will perceive the shape's fall as very slow. It will simply seem unrealistic, as though the gravity is too low.

### Move and jump

We can now combine moving a shape horizontally, vertically, and jumping to create a shape that a player can move and control freely in the Pygame window. For example, our code is now as follows,

```
def move():
    global shapeX, shapeY, shapeRX, shapeJY, shapeJump, gravity

    # move left
    if leftDown:
        # check shape not exit window to left
        if shapeX > 0.0:
            shapeX -= shapeSpeed
    # move right
    if rightDown:
        # check shape not exit window to right
        if shapeX + shapeSize < winWidth:
            shapeX += shapeSpeed

    # check upward speed > 1.0
```

```
    if shapeJY > 1.0:
        # gradually decrease upward speed to less than 1.0
        shapeJY = shapeJY * 0.9
    else:
        # less than 1.0, reset to 0.0 to allow shape to fall
        shapeJY = 0.0
        # stop jump
        shapeJump = False

    # check if shape in air - use gravity to descend
    if shapeY < winHeight - shapeSize:
        shapeY += gravity
        gravity = gravity * 1.1
    else:
        shapeY = winHeight - shapeSize
        gravity = 1.0

    shapeY -= shapeJY
```

With the above `move` function, we've now combined horizontal movement with a vertical jump. So, our player can now make the shape move from left to right, and jump at the same time. Expected behaviour for many well-known platform genre games.

We can then update the **game loop** to include the required listeners and handlers for horizontal movement,

```
# create game loop
while True:
    # set clock
    #msElapsed = clock.tick(max_fps)
    #print(msElapsed)
    # 'processing' inputs (events)
    for event in EVENTS.get():
        # check keyboard events - keydown
        if event.type == pygame.KEYDOWN:
            # check for directional - LEFT and RIGHT
            if event.key == pygame.K_LEFT:
                leftDown = True
            if event.key == pygame.K_RIGHT:
                rightDown = True
            # check for directional - UP
            if event.key == pygame.K_UP:
                if not shapeJump:
                    shapeJump = True
                    shapeJY += jumpHeight
            # check for ESCAPE key
            if event.key == pygame.K_ESCAPE:
                gameExit()

        # check keyboard events - keyup
        if event.type == pygame.KEYUP:
            if event.key == pygame.K_LEFT:
                leftDown = False
            if event.key == pygame.K_RIGHT:
                rightDown = False
```

We'll also add the required listener for `KEYUP` so we can stop our shape from continuously moving right or left.

Our shape can now walk and jump across the game window.