Bouncing



The character moves around and bounces back at the borders.

We create a ball that moves on the screen at a speed of 150 pixels per second. When it reaches the border, it bounces back.

```
HEIGHT = 480
3
   ball = Actor('ball', anchor=('left', 'top'))
   velocity = {'x': 150, 'y': 150}
6
7
   def update(dt):
8
       if ball.x + ball.width >= WIDTH:
9
       velocity['x'] = -velocity['x']
10
       if ball.x < 0:
11
       velocity['x'] = -velocity['x']
12
       if ball.y + ball.height >= HEIGHT:
13
       velocity['y'] = -velocity['y']
       if ball.y < 0:
15
      velocity['y'] = -velocity['y']
16
       ball.x += velocity['x'] * dt
17
    ball.y += velocity['y'] * dt
18
19
   def draw():
       screen.fill((255, 255, 153))
20
21
       ball.draw()
```

The ball has a velocity with two components: one for each axis. when the ball touches the border, one of both components is "inverted".

Bouncing



The character moves around and bounces back at the borders.

We create a ball that moves on the screen at a speed of 150 pixels per second. When it reaches the border, it bounces back.

```
WIDTH = 640
   HEIGHT = 480
   ball = Actor('ball', anchor=('left', 'top'))
   velocity = {'x': 150, 'y': 150}
6
7
   def update(dt):
8
       if ball.x + ball.width >= WIDTH:
9
          velocity['x'] = -velocity['x']
10
       if ball.x < 0:
11
          velocity['x'] = -velocity['x']
12
       if ball.y + ball.height >= HEIGHT:
          velocity['y'] = -velocity['y']
13
14
       if ball.y < 0:
15
         velocity['y'] = -velocity['y']
16
       ball.x += velocity['x'] * dt
17
       ball.y += velocity['y'] * dt
18
19
   def draw():
20
       screen.fill((255, 255, 153))
       ball.draw()
```

The ball has a velocity with two components: one for each axis. when the ball touches the border, one of both components is "inverted".

Bouncing



The character moves around and bounces back at the borders.

We create a ball that moves on the screen at a speed of 150 pixels per second. When it reaches the border, it bounces back.

```
WIDTH = 640
   HEIGHT = 480
3
   ball = Actor('ball', anchor=('left', 'top'))
   velocity = {'x': 150, 'y': 150}
7
   def update(dt):
8
       if ball.x + ball.width >= WIDTH:
         velocity['x'] = -velocity['x']
9
10
       if ball.x < 0:
11
         velocity['x'] = -velocity['x']
12
       if ball.y + ball.height >= HEIGHT:
13
       velocity['y'] = -velocity['y']
14
       if ball.y < 0:
    velocity['y'] = -velocity['y']
15
16
       ball.x += velocity['x'] * dt
17
    ball.y += velocity['y'] * dt
18
19 def draw():
20
       screen.fill((255, 255, 153))
21
       ball.draw()
```

The ball has a velocity with two components: one for each axis. when the ball touches the border, one of both components is "inverted".

Bouncing



The character moves around and bounces back at the borders.

We create a ball that moves on the screen at a speed of 150 pixels per second. When it reaches the border, it bounces back.

```
WIDTH = 640
   HEIGHT = 480
3
   ball = Actor('ball', anchor=('left', 'top'))
   velocity = {'x': 150, 'y': 150}
7
  def update(dt):
8
      if ball.x + ball.width >= WIDTH:
9
          velocity['x'] = -velocity['x']
10
       if ball.x < 0:
11
          velocity['x'] = -velocity['x']
12
       if ball.y + ball.height >= HEIGHT:
          velocity['y'] = -velocity['y']
13
14
       if ball.y < 0:
15
         velocity['y'] = -velocity['y']
16
       ball.x += velocity['x'] * dt
17
    ball.y += velocity['y'] * dt
18
19 def draw():
20
       screen.fill((255, 255, 153))
       ball.draw()
```

The ball has a velocity with two components: one for each axis. when the ball touches the border, one of both components is "inverted".

In the ${\tt update}$ function, we check the current position and compare it to the windows size.

When creating the ball, we define the Actor to be anchored to the top left corner (the default being the center). This makes the comparison with the border simpler: we can diretly compare the top and left borders and can add the actor's width or height for the right and bottom borders.

The $\,_{\rm x}$ and $_{\rm y}$ parts of the new <code>ball</code>'s position are calculated by adding the corresponding velocity component, multiplied by <code>dt</code>. dt is the time elapsed since the last time the <code>update</code> function has been called: if we did not multiply the velocity by <code>dt</code>, the ball would move by 150 pixels on each call of <code>update</code> (very, very often!) and not every second.

When you push the "Play" button, you should see a bull bouncing around the window.

In the ${\tt update}$ function, we check the current position and compare it to the windows size.

When creating the ball, we define the Actor to be anchored to the top left corner (the default being the center). This makes the comparison with the border simpler: we can diretly compare the top and left borders and can add the actor's width or height for the right and bottom borders.

The $\,_{\rm x}$ and $_{\rm y}$ parts of the new <code>ball's</code> position are calculated by adding the corresponding velocity component, multiplied by <code>dt</code>. dt is the time elapsed since the last time the <code>update</code> function has been called: if we did not multiply the velocity by <code>dt</code>, the ball would move by 150 pixels on each call of <code>update</code> (very, very often!) and not every second.

When you push the "Play" button, you should see a bull bouncing around the window.

In the update function, we check the current position and compare it to the windows size.

When creating the ball, we define the Actor to be anchored to the top left corner (the default being the center). This makes the comparison with the border simpler: we can diretly compare the top and left borders and can add the actor's width or height for the right and bottom borders.

The $\,_{\rm x}$ and $_{\rm y}$ parts of the new ball's position are calculated by adding the corresponding velocity component, multiplied by dt. dt is the time elapsed since the last time the update function has been called: if we did not multiply the velocity by dt, the ball would move by 150 pixels on each call of update (very, very often!) and not every second.

When you push the "Play" button, you should see a bull bouncing around the window.

In the ${\tt update}$ function, we check the current position and compare it to the windows size.

When creating the ball, we define the Actor to be anchored to the top left corner (the default being the center). This makes the comparison with the border simpler: we can diretly compare the top and left borders and can add the actor's width or height for the right and bottom borders.

The $\,_{\rm x}$ and $_{\rm y}$ parts of the new <code>ball</code>'s position are calculated by adding the corresponding velocity component, multiplied by <code>dt</code>. dt is the time elapsed since the last time the <code>update</code> function has been called: if we did not multiply the velocity by <code>dt</code>, the ball would move by 150 pixels on each call of <code>update</code> (very, very often!) and not every second.

When you push the "Play" button, you should see a bull bouncing around the window.