

Sprites y Física

Mario se convierte en nuestro Sprite



De forma similar a cómo hemos hecho con Pygame, vamos a mostrar el uso de sprites y de la física que incorpora Pilas

En efecto, hay determinados detalles que incluso allí implementamos por nuestra cuenta y que Pilas ya incorpora de una manera más simple. Sólo hay que aprenderla. No temas; el trabajo que has empleado hasta aquí te habrá ayudado a crecer como programador/programadora.

En particular, podríamos introducir los movimientos de Mario de la misma forma que en el tutorial equivalente de Pygame. Esa va a ser la primera parte de este tutorial: vamos a poner frente a frente los pasos dados por Pygame con los propios de Pilas. Allí veremos las ventajas de usar Pilas, al ser el código más compacto y elegante.

Pero vamos a hacerlo también utilizando las técnicas de física que incorpora Pilas, a través de Box2D, en la segunda parte del tutorial. En ese momento, veremos los bonus que adquirimos... y la responsabilidad que conllevan.

A por la primera parte...

Pygame vs Pilas

¡Vamos allá! En el código, en negrita y fondo gris, indicaremos los cambios.

mario01p.py

En este primer paso sólo vamos a mostrar la imagen de Mario como resultado, pero internamente habremos creado un **sprite** que luego nos servirá para continuar y ampliarlo en los siguientes casos. Éste es el código:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario01.py
# Implementación de sprites
#-----

import sys
import pygame
from pygame.locals import *

# Clase MiSprite
class MiSprite(pygame.sprite.Sprite):
    # Inicializador de la clase
    def __init__(self, dibujo):
        # Importante: Primero hay que inicializar la clase Sprite original
        pygame.sprite.Sprite.__init__(self)

        # Almacenar en el sprite la imagen deseada
        self.image = pygame.image.load(dibujo)
        self.image = self.image.convert()

        # Definir el rect del sprite
        self.rect = self.image.get_rect()
        self.rect.topleft = (0, 150)

# Inicializar PyGame y crear la Surface del juego
pygame.init()
visor = pygame.display.set_mode((640, 480))

# Inicializar el sprite
```

Pygame

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario01p.py
# Implementación de sprites
#-----

import pilas

# Inicializar Pilas y crear la ventana por defecto
pilas.iniciar()

# Clase MiSprite
class MiSprite(pilas.actores.Actor):
    # Inicializador de la clase
    def __init__(self, imagen):
        # Importante: Primero hay que inicializar la clase original
        pilas.actores.Actor.__init__(self, imagen)

        # Situar el sprite en la posición deseada
        self.arriba = 90
        self.izquierda = -320

# Inicializar el sprite
sprite = MiSprite("mario.png")

# Inicializar el fondo
pilas.fondos.Color(pilas.colores.blanco)

pilas.ejecutar()
```

Pilas

```
sprite = MiSprite("mario.bmp")
grupo = pygame.sprite.RenderUpdates(sprite)

# Crear un reloj para controlar la animación
reloj = pygame.time.Clock()

# El bucle de la animación
while 1:
    #Fijar la animación a 60 fps
    reloj.tick(60)

    # Gestionar los eventos
    for evento in pygame.event.get():
        if evento.type == QUIT:
            pygame.quit()
            sys.exit()

    # Actualizar el sprite
    grupo.update()

    # Dibujar la escena
    visor.fill((255, 255, 255))
    grupo.draw(visor)

    # Mostrar la animación
    pygame.display.update()
```

Pygame

Pilas

Ya vemos una primera diferencia, ¿no? El código de Pilas es mucho más breve... Veámoslo:

Tras la importación de la librería y su inicialización, lo primero que hacemos en el código anterior es definir nuestra nueva **clase de sprite** que **deriva** de la clase **pilas.actores.Actor** de Pilas. Siguiendo la terminología de Pilas, deberíamos usar la palabra **actor** en lugar de la de sprite, pero la vamos a mantener a efectos de comparación:

```
class MiSprite(pilas.actores.Actor):
```

pilas.actores.Actor es la clase de sprite que implementa Pilas de forma nativa. Ya sabes que hay otros predefinidos (como Mono, o Torreta), pero ésta es la genérica. Al ponerla entre paréntesis le decimos a Pilas que cree la nuestra a partir de aquella.

Recuerda que toda clase debería tener definida la función especial `__init__()` en la que pueden ponerse todas aquellas tareas que queremos que se realicen al crearse. ¡Ojo, algo muy importante! Cuando definamos una clase basada en otra, es muy conveniente que llamemos a su vez a la **función que inicializa la clase original**. Eso se consigue escribiéndolo en primer lugar:

```
def __init__(self, imagen):  
    pilas.actores.Actor.__init__(self, imagen) ←
```

Un secreto con respecto al significado de **self**. Cuando creamos un objeto del tipo de la clase que estamos definiendo, **self representa al objeto mismo**. Es un convenio muy útil pero que al programador primerizo le causa algún que otro dolor de cabeza. Python está construido de forma que en toda función que se **defina** dentro de una clase, **el primer argumento debe ser siempre self**. Sin embargo, cuando se **invoca** a estas funciones, **nunca se pone el susodicho self**. Python se encarga por sí mismo de pasárselo a la función y no tienes que ocuparte tú de ello. Acuérdate siempre de esto; **en la definición sí, en el uso no**.

A su vez, si quieres **definir variables** que pertenezcan al **objeto** para poder utilizarlas posteriormente, siempre debes definir las con el como propiedades del objeto **self**, es decir, **debes definir las con el self**. por delante. En nuestro ejemplo, definimos las variables **self.arriba** y **self.izquierda** para situar nuestro sprite en la posición deseada. Estos **atributos** están ya predefinidos en Pilas e indican las posiciones de, respectivamente, los bordes superior e izquierdo del sprite. Recuerda que, a diferencia de Pygame, las coordenadas para los actores de Pilas se indican respecto a unos ejes cartesianos con origen en el centro de la ventana. Como, por defecto, la ventana de Pilas es de 640x480 pixeles, las coordenadas x (horizontales) están en un rango de -320 a +320 y las coordenadas y de -240 a +240. ¿Entiendes ahora por qué en Pygame se sitúa horizontalmente en la coordenada 0 y en Pilas en la coordenada -320? ¿Te fijas, también, que en Pygame la coordenada vertical aumenta hacia abajo, mientras que en Pilas el aumento es hacia arriba? Esto tendrá sus consecuencias más adelante.

Otra diferencia entre Pilas y Pygame es que no nos tenemos que preocupar de definir a mano un bucle que dibuje los fotogramas uno a uno para crear la animación. Tampoco nos tenemos que preocupar por controlar el tiempo para que se reproduzcan los fotogramas a la velocidad deseada. Por defecto, la velocidad es de 60 fotogramas por segundo. Si queremos una velocidad distinta, basta con indicarlo en la inicialización. Por ejemplo, para obtener 100 fotogramas por segundo (si tu ordenador lo soporta)

```
pilas.iniciar(rendimiento=100)
```

Dicho lo anterior, queda claro que simplemente nos tenemos que preocupar por situar los sprites/actores en el escenario y lanzar el motor con **pilas.ejecutar()**. ¿Qué nos queda, entonces?

Ha llegado el momento importante; vamos a crear a Mario. ¿Cómo lo hemos hecho? Hemos invocado el nombre de la clase del sprite pasándole como argumento el nombre del archivo que contiene la imagen que vamos a usar. Fíjate en la definición de la clase; allí verás que la función `__init__()` la hemos puesto con dos parámetros, **self** (que recuerda que se ignora cuando se llama a la función) y otra más que luego usamos en la definición para indicar la imagen del sprite:

```
sprite = MiSprite("mario.png")
```

(por supuesto, el archivo `"mario.png"` tiene que estar en la misma carpeta que el programa para que lo encuentre).

A diferencia, nuevamente, de Pygame, en Pilas no estamos obligados a definir un grupo de sprites. Solo tenemos que hacerlo cuando queramos definir comportamiento a todo un grupo como tal, lo que no es el caso. De todas formas, la forma de creación es muy similar.

De forma genérica, un grupo vacío puede definirse con la instrucción

```
grupo = pilas.grupo.Grupo()
```

y puede añadirse un sprite al grupo con la instrucción

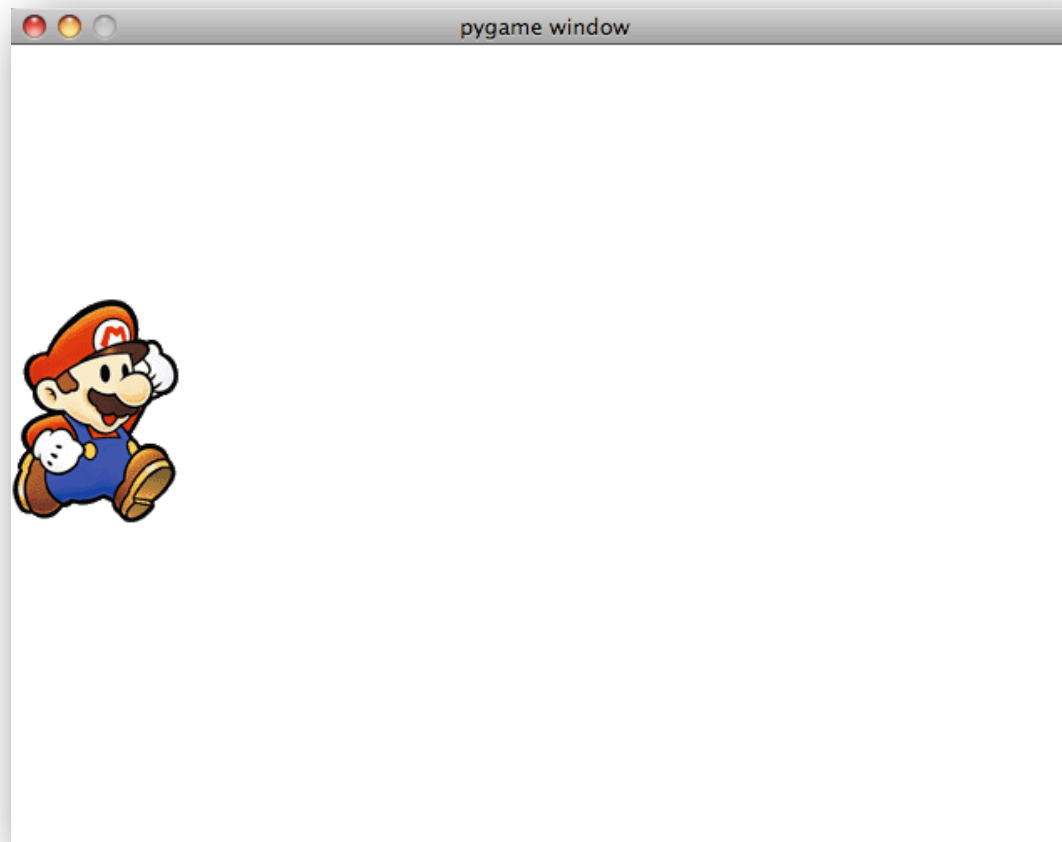
```
grupo.append(sprite)
```

Si te fijas, es algo similar, aunque más simple, a lo que hay que hacer con Pygame.

¡Solo queda añadir el fondo! Por defecto, el fondo tiene un color grisáceo. Para dejarlo en blanco, emulando la práctica de Pygame, nada más sencillo; usamos el actor predefinido de Pilas para tal efecto y uno de los colores que incorpora:

```
pilas.fondos.Color(pilas.colores.blanco)
```

¡Ya tenemos todos nuestros actores en el escenario! Como hemos dicho, solo queda lanzar el motor y he aquí el resultado:



mario02p.py

La ventaja de utilizar la **programación dirigida a objetos** que incorpora Pilas es que modificar el comportamiento de los sprites es muy sencillo. Vamos a darle movimiento a Mario. Nada más simple:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario02.py
# Movimiento sencillo
#-----

import sys
import pygame
from pygame.locals import *

# Clase MiSprite
class MiSprite(pygame.sprite.Sprite):
    # Inicializador de la clase
    def __init__(self, dibujo):
        # Importante: Primero hay que inicializar la clase Sprite original
        pygame.sprite.Sprite.__init__(self)

        # Almacenar en el sprite la imagen deseada
        self.image = pygame.image.load(dibujo)
        self.image = self.image.convert()

        # Definir el rect del sprite
        self.rect = self.image.get_rect()
        self.rect.topleft = (0, 150)

def update(self):
    # Modificar la posición del sprite
    self.rect.move_ip(1, 0)
```

Pygame

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario02p.py
# Movimiento sencillo
#-----

import pilas

# Inicializar Pilas y crear la ventana por defecto
pilas.iniciar()

# Clase MiSprite
class MiSprite(pilas.actores.Actor):
    # Inicializador de la clase
    def __init__(self, imagen):
        # Importante: Primero hay que inicializar la clase original
        pilas.actores.Actor.__init__(self, imagen)

        # Situar el sprite en la posición deseada
        self.arriba = 90
        self.izquierda = -320

def actualizar(self):
    # Modificar la posición del sprite
    self.x += 1

# Inicializar el sprite
sprite = MiSprite("mario.png")

# Inicializar el fondo
```

Pilas

```
# Inicializar PyGame y crear la Surface del juego
pygame.init()
visor = pygame.display.set_mode((640, 480))
```

```
# Inicializar el sprite
sprite = MiSprite("mario.bmp")
grupo = pygame.sprite.RenderUpdates(sprite)
```

```
# Crear un reloj para controlar la animación
reloj = pygame.time.Clock()
```

```
# El bucle de la animación
while 1:
    #Fijar la animación a 60 fps
    reloj.tick(60)
```

```
# Gestionar los eventos
for evento in pygame.event.get():
    if evento.type == QUIT:
        pygame.quit()
        sys.exit()
```

```
# Actualizar el sprite
grupo.update()
```

```
# Dibujar la escena
visor.fill((255, 255, 255))
grupo.draw(visor)
```

```
# Mostrar la animación
pygame.display.update()
```

Pygame

```
pilas.fondos.Color(pilas.colores.blanco)

pilas.ejecutar()
```

Pilas

¡Extremadamente sencillo! En cada fotograma, Pilas llama a la función **actualizar()** de cada **sprite** para saber dónde debe dibujarlo, así que simplemente debemos definir esa función dentro de nuestra **clase** para tener la tarea hecha. Esto es lo único que hemos añadido al código:

```
def actualizar(self):  
    # Modificar la posición del sprite  
    self.x += 1
```

¿Qué es lo que hace? En cada fotograma (es decir, 60 veces por segundo) aumenta en 1 unidad la coordenada x del sprite, lo que genera el movimiento hacia la derecha de Mario.



mario03p.py

Implementar el rebote es igual de sencillo:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario03.py
# Rebote
#-----

import sys
import pygame
from pygame.locals import *

# Clase MiSprite
class MiSprite(pygame.sprite.Sprite):
    # Inicializador de la clase
    def __init__(self, dibujo):
        # Importante: Primero hay que inicializar la clase Sprite original
        pygame.sprite.Sprite.__init__(self)

        # Almacenar en el sprite la imagen deseada
        self.image = pygame.image.load(dibujo)
        self.image = self.image.convert()

        # Definir el rect del sprite
        self.rect = self.image.get_rect()
        self.rect.topleft = (0, 150)

# Definir las velocidad
self.dx = 1

def update(self):
    # Modificar la posición del sprite
```

Pygame

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario03p.py
# Rebote
#-----

import pilas

# Inicializar Pilas y crear la ventana por defecto
pilas.iniciar()

# Clase MiSprite
class MiSprite(pilas.actores.Actor):
    # Inicializador de la clase
    def __init__(self, imagen):
        # Importante: Primero hay que inicializar la clase original
        pilas.actores.Actor.__init__(self, imagen)

        # Situar el sprite en la posición deseada
        self.arriba = 90
        self.izquierda = -320

# Definir la velocidad
self.dx = 1

def actualizar(self):
    # Modificar la posición del sprite
    self.x += self.dx
    # Comprobar si hay que cambiar el movimiento
    if self.izquierda < -320 or self.derecha > 320:
```

Pilas

```
self.rect.move_ip(self.dx, 0)
# Comprobar si hay que cambiar el movimiento
if self.rect.left < 0 or self.rect.right > 640:
    self.dx = -self.dx
```

```
# Inicializar PyGame y crear la Surface del juego
pygame.init()
visor = pygame.display.set_mode((640, 480))
```

```
# Inicializar el sprite
sprite = MiSprite("mario.bmp")
grupo = pygame.sprite.RenderUpdates(sprite)
```

```
# Crear un reloj para controlar la animación
reloj = pygame.time.Clock()
```

```
# El bucle de la animación
while 1:
    #Fijar la animación a 60 fps
    reloj.tick(60)
```

```
# Gestionar los eventos
for evento in pygame.event.get():
    if evento.type == QUIT:
        pygame.quit()
        sys.exit()
```

```
# Actualizar el sprite
grupo.update()
```

```
# Dibujar la escena
visor.fill((255, 255, 255))
grupo.draw(visor)
```

```
# Mostrar la animación
pygame.display.update()
```

Pygame

```
self.dx = -self.dx
```

```
# Inicializar el sprite
sprite = MiSprite("mario.png")
```

```
# Inicializar el fondo
pilas.fondos.Color(pilas.colores.blanco)
```

```
pilas.ejecutar()
```

Pilas

La forma de conseguir el rebote ya la conocemos; ver cuando se llega al borde de la ventana y cambiar el sentido del movimiento cambiando de signo la cantidad que sumas a la posición. Lógicamente, para poder cambiar ese signo, necesitamos almacenar la cantidad que sumamos a la posición en una variable. Como es un movimiento en el eje horizontal, llamemos a esa variable **dx**. Como es una variable que ha de **pertenecer al sprite**, vamos a poner su definición en la función `__init__()` de la **clase** y, por lo tanto, deberemos añadirle al nombre el ya conocido **self**:

```
self.dx = 1
```

Bien. Una vez hecho esto, implementar el rebote de Mario requiere modificar la definición de la función **actualizar()** del **sprite**:

```
def actualizar(self):  
    # Modificar la posición del sprite  
    self.x += self.dx  
    # Comprobar si hay que cambiar el movimiento  
    if self.izquierda < -320 or self.derecha > 320:  
        self.dx = -self.dx
```

En efecto, primero movemos el sprite la cantidad deseada (ahora es **self.dx**) y luego miramos si se ha llegado a uno de los extremos de la pantalla, en cuyo caso cambiamos el movimiento cambiando el signo de **self.dx**.

¿Ves qué sencillo? En este caso, el proceso, tanto en Pygame como en Pilas, es muy similar.

mario04p.py

Un nuevo paso. Esta vez, al igual que hacíamos con Guy, vamos a invertir la imagen del sprite cuando éste rebote. Éste es el código:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario04.py
# Rebote invirtiendo el sprite
#-----

import sys
import pygame
from pygame.locals import *

# Clase MiSprite
class MiSprite(pygame.sprite.Sprite):
    # Inicializador de la clase
    def __init__(self, dibujo):
        # Importante: Primero hay que inicializar la clase Sprite original
        pygame.sprite.Sprite.__init__(self)

        # Almacenar en el sprite la imagen deseada
        self.image = pygame.image.load(dibujo)
        self.image = self.image.convert()

        # Definir el rect del sprite
        self.rect = self.image.get_rect()
        self.rect.topleft = (0, 150)

        # Definir la velocidad
        self.dx = 1

    def update(self):
        # Modificar la posición del sprite
```

Pygame

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario04p.py
# Rebote invirtiendo el sprite
#-----

import pilas

# Inicializar Pilas y crear la ventana por defecto
pilas.iniciar()

# Clase MiSprite
class MiSprite(pilas.actores.Actor):
    # Inicializador de la clase
    def __init__(self, imagen):
        # Importante: Primero hay que inicializar la clase original
        pilas.actores.Actor.__init__(self, imagen)

        # Situar el sprite en la posición deseada
        self.arriba = 90
        self.izquierda = -320

        # Definir la velocidad
        self.dx = 1

    def actualizar(self):
        # Modificar la posición del sprite
        self.x += self.dx
        # Comprobar si hay que cambiar el movimiento
        if self.izquierda < -320 or self.derecha > 320:
```

Pilas

```

self.rect.move_ip(self.dx, 0)
# Comprobar si hay que cambiar el movimiento
if self.rect.left < 0 or self.rect.right > 640:
    self.dx = -self.dx
self.image = pygame.transform.flip(self.image, True, False)

# Inicializar PyGame y crear la Surface del juego
pygame.init()
visor = pygame.display.set_mode((640, 480))

# Inicializar el sprite
sprite = MiSprite("mario.bmp")
grupo = pygame.sprite.RenderUpdates(sprite)

# Crear un reloj para controlar la animación
reloj = pygame.time.Clock()

# El bucle de la animación
while 1:
    #Fijar la animación a 60 fps
    reloj.tick(60)

    # Gestionar los eventos
    for evento in pygame.event.get():
        if evento.type == QUIT:
            pygame.quit()
            sys.exit()

    # Actualizar el sprite
    grupo.update()

    # Dibujar la escena
    visor.fill((255, 255, 255))
    grupo.draw(visor)

pygame.display.update()

```

Pygame

```

self.dx = -self.dx
self.espejado = not self.espejado

# Inicializar el sprite
sprite = MiSprite("mario.png")

# Inicializar el fondo
pilas.fondos.Color(pilas.colores.blanco)

pilas.ejecutar()

```

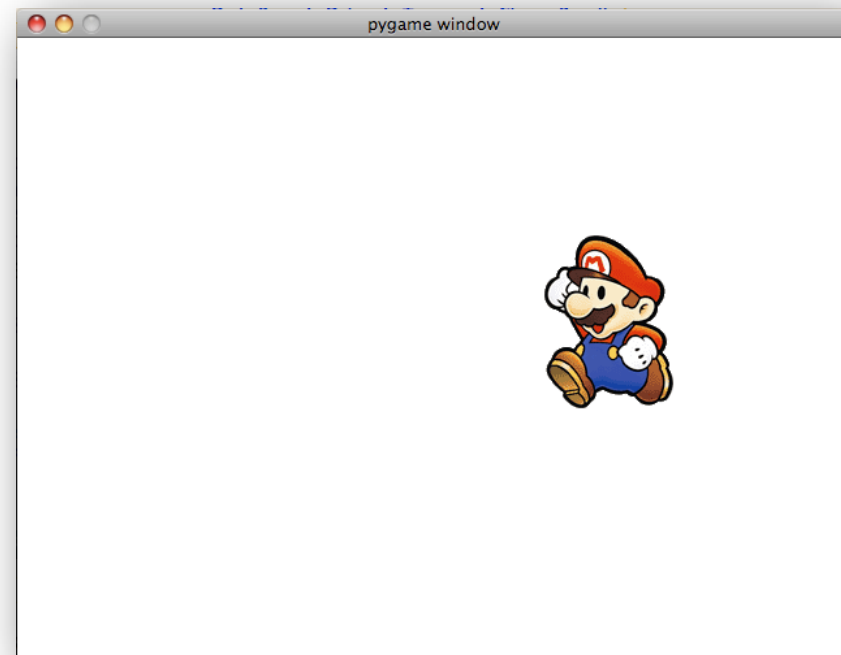
Pilas

¿Puedes creer que sea algo tan simple? Pues sí: en el mismo lugar del código donde miramos si el sprite ha llegado al borde (y cambiamos la dirección del movimiento en caso afirmativo), lo único que hacemos es transformar la imagen de Mario por su reflejo horizontal

```
self.espejado = not self.espejado
```

Invertir la imagen de un actor en Pylas es muy fácil; el atributo **self.espejado** nos permite indicar si la imagen lo está (**True**) o no (**False**). Con el **operador not**, conmutamos su estado entre uno y otro, con lo que a partir de entonces nuestro Mario mirará hacia el lado contrario cada vez que llegue a un borde de la ventana. ¡Perfecto!

Espero que, en este punto, comprendas las ventajas de la **programación dirigida a objetos** y del concepto de **Sprite** o **Actor** de Pylas; todo su comportamiento lo incluimos en la definición de la clase y cuando usemos un objeto de ese tipo, automáticamente se comportará como tal. Ello hace que los programas sean mucho más fácilmente modificables y ampliables, como estamos viendo.



mario05p.py

Implementar el movimiento en la dirección vertical, incluido el rebote correspondiente, te debería resultar ahora bastante fácil:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario05.py
# Implementando el movimiento vertical
#-----

import sys
import pygame
from pygame.locals import *

# Clase MiSprite
class MiSprite(pygame.sprite.Sprite):
    # Inicializador de la clase
    def __init__(self, dibujo):
        # Importante: Primero hay que inicializar la clase Sprite original
        pygame.sprite.Sprite.__init__(self)

        # Almacenar en el sprite la imagen deseada
        self.image = pygame.image.load(dibujo)
        self.image = self.image.convert()

        # Definir el rect del sprite
        self.rect = self.image.get_rect()
        self.rect.topleft = (0, 150)

        # Definir las velocidades
        self.dx = 1
        self.dy = 1

    def update(self):
```

Pygame

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario05p.py
# Implementando el movimiento vertical
#-----

import pilas

# Inicializar Pilas y crear la ventana por defecto
pilas.iniciar()

# Clase MiSprite
class MiSprite(pilas.actores.Actor):
    # Inicializador de la clase
    def __init__(self, imagen):
        # Importante: Primero hay que inicializar la clase original
        pilas.actores.Actor.__init__(self, imagen)

        # Situar el sprite en la posición deseada
        self.arriba = 90
        self.izquierda = -320

        # Definir la velocidad
        self.dx = 1
        self.dy = -1

    def actualizar(self):
        # Modificar la posición del sprite
        self.x += self.dx
        self.y += self.dy
```

Pilas


```

# Modificar la posición del sprite
self.rect.move_ip(self.dx, self.dy)
# Comprobar si hay que cambiar el movimiento
if self.rect.left < 0 or self.rect.right > 640:
    self.dx = -self.dx
    self.image = pygame.transform.flip(self.image, True, False)
if self.rect.top < 0 or self.rect.bottom > 480:
    self.dy = -self.dy

# Inicializar PyGame y crear la Surface del juego
pygame.init()
visor = pygame.display.set_mode((640, 480))

# Inicializar el sprite
sprite = MiSprite("mario.bmp")
grupo = pygame.sprite.RenderUpdates(sprite)

# Crear un reloj para controlar la animación
reloj = pygame.time.Clock()

# El bucle de la animación
while 1:
    #Fijar la animación a 60 fps
    reloj.tick(60)

    # Gestionar los eventos
    for evento in pygame.event.get():
        if evento.type == QUIT:
            pygame.quit()
            sys.exit()

    # Actualizar el sprite
    grupo.update()

    # Dibujar la escena
    visor.fill((255, 255, 255))

```

Pygame

```

# Comprobar si hay que cambiar el movimiento
if self.izquierda < -320 or self.derecha > 320:
    self.dx = -self.dx
    self.espejado = not self.espejado
if self.arriba > 240 or self.abajo < -240:
    self.dy = -self.dy

# Inicializar el sprite
sprite = MiSprite("mario.png")

# Inicializar el fondo
pilas.fondos.Color(pilas.colores.blanco)

pilas.ejecutar()

```

Pilas

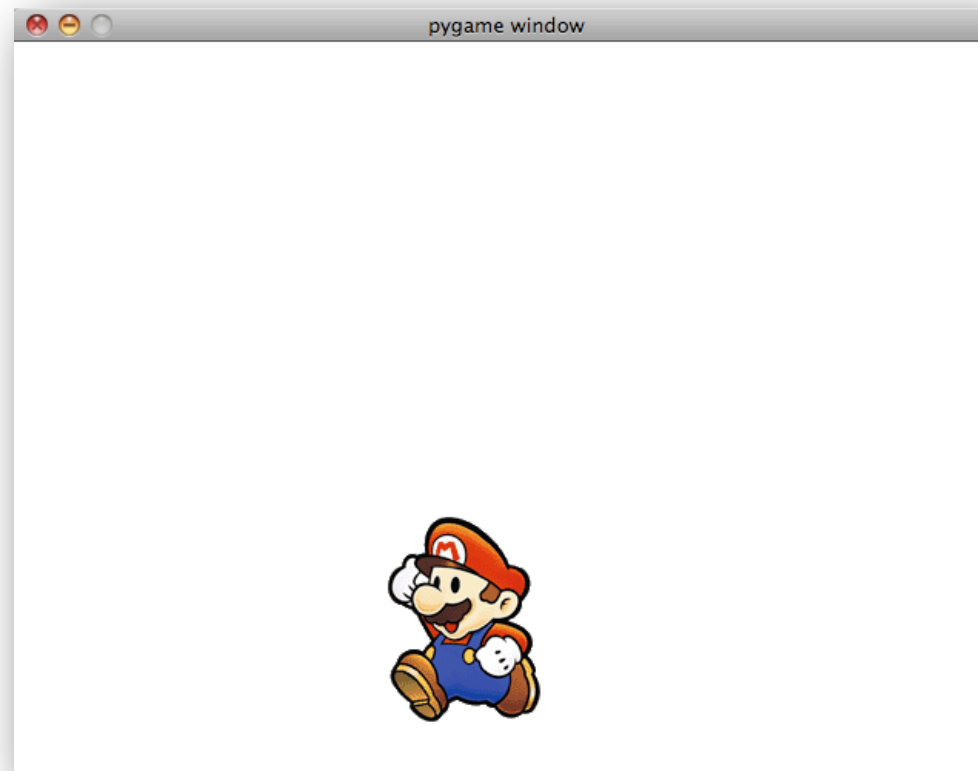
```
grupo.draw(visor)
```

```
# Mostrar la animación  
pygame.display.update()
```

Pygame

Pilas

¿Hace falta explicar algo? Sólo hemos añadido en `__init__()` la nueva velocidad vertical `self.dy` y en `actualizar()` la hemos añadido al movimiento y a la comprobación del rebote. Bueno, quizás sí... Observa cómo en Pilas, al aumentar la coordenada vertical hacia arriba (y no hacia abajo como en Pygame), el valor inicial de `self.dy` es `-1`; sólo así se consigue que Mario descienda.



mario06p.py

Hasta ahora sólo hemos hecho movimientos rectilíneos y uniformes. ¿Qué tal simular algo más real, como una caída con gravedad?

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario06.py
# Simular la gravedad
#-----

import sys
import pygame
from pygame.locals import *

# Clase MiSprite
class MiSprite(pygame.sprite.Sprite):
    # Inicializador de la clase
    def __init__(self, dibujo):
        # Importante: Primero hay que inicializar la clase Sprite original
        pygame.sprite.Sprite.__init__(self)

        # Almacenar en el sprite la imagen deseada
        self.image = pygame.image.load(dibujo)
        self.image = self.image.convert()

        # Definir el rect del sprite
        self.rect = self.image.get_rect()
        self.rect.topleft = (0, 150)

        # Definir las velocidades
        self.dx = 1
        self.dy = 1

    def update(self):
```

Pygame

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario06p.py
# Simular la gravedad
#-----

import pilas

# Inicializar Pilas y crear la ventana por defecto
pilas.iniciar()

# Clase MiSprite
class MiSprite(pilas.actores.Actor):
    # Inicializador de la clase
    def __init__(self, imagen):
        # Importante: Primero hay que inicializar la clase original
        pilas.actores.Actor.__init__(self, imagen)

        # Situar el sprite en la posición deseada
        self.arriba = 90
        self.izquierda = -320

        # Definir la velocidad
        self.dx = 1
        self.dy = -1

    def actualizar(self):
        # Modificar la posición del sprite
        self.x += self.dx
        self.y += self.dy
```

Pilas

```
# Modificar la posición del sprite
self.rect.move_ip(self.dx, self.dy)
# Comprobar si hay que cambiar el movimiento
if self.rect.left < 0 or self.rect.right > 640:
    self.dx = -self.dx
    self.image = pygame.transform.flip(self.image, True, False)
if self.rect.top < 0 or self.rect.bottom > 480:
    self.dy = -self.dy
# Simular la gravedad sumando una cantidad a la velocidad vertical
self.dy = self.dy + 0.5
```

Pygame

```
# Inicializar PyGame y crear la Surface del juego
pygame.init()
visor = pygame.display.set_mode((640, 480))

# Inicializar el sprite
sprite = MiSprite("mario.bmp")
grupo = pygame.sprite.RenderUpdates(sprite)

# Crear un reloj para controlar la animación
reloj = pygame.time.Clock()

# El bucle de la animación
while 1:
    #Fijar la animación a 60 fps
    reloj.tick(60)

    # Gestionar los eventos
    for evento in pygame.event.get():
        if evento.type == QUIT:
            pygame.quit()
            sys.exit()

    # Actualizar el sprite
    grupo.update()
```

```
# Comprobar si hay que cambiar el movimiento
if self.izquierda < -320 or self.derecha > 320:
    self.dx = -self.dx
    self.espejado = not self.espejado
if self.arriba > 240 or self.abajo < -240:
    self.dy = -self.dy
# Simular la gravedad sumando una cantidad a la velocidad vertical
self.dy -= 0.5
```

Pilas

```
# Inicializar el sprite
sprite = MiSprite("mario.png")

# Inicializar el fondo
pilas.fondos.Color(pilas.colores.blanco)

pilas.ejecutar()
```

```
# Dibujar la escena
visor.fill((255, 255, 255))
grupo.draw(visor)
```

```
# Mostrar la animación
pygame.display.update()
```

Pygame

Pilas

¿Te podías imaginar que era tan sencillo? Lo único que hemos hecho ha sido añadir en la función **actualizar()** la siguiente línea de código:

```
self.dy -= 0.5
```

La explicación es muy simple. La gravedad lo único que hace es empujarnos hacia abajo de manera constante. Y la manera de hacerlo en el método **actualizar()** es, por lo tanto, **restar** una cantidad constante a la velocidad en el eje vertical. Fíjate que tiene precisamente el efecto deseado; cuando el **sprite** va hacia arriba (**self.dy** es entonces positiva) al restarle **0.5** lo que hace es frenarse y cuando va hacia abajo (**self.dy** negativa) acelerarse. Fíjate, de nuevo, en la diferencia de signos con Pygame.

Por supuesto, sin más que variar ese **0.5** conseguimos una gravedad más o menos intensa.

¡Ejecuta el programa! Veras como el rebote es ahora más divertido...

No obstante, hay dos detalles que debemos cuidar. El primero es que en mucho juegos tipo plataforma, no queremos que el protagonista se frene y que rebote siempre a la misma altura, no que se vaya frenando. Lo segundo es que tenemos un pequeño **bug**; si esperamos lo suficiente, veremos como Mario... ¡termina por atravesar los bordes de la ventana!

En el próximo paso veremos cómo solucionar esto.

mario07p.py

El que el rebote sea siempre a la misma altura y el que Mario termine atravesando el suelo de la ventana pueden solucionarse con una sola línea:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario07.py
# Movimiento de plataforma
#-----

import sys
import pygame
from pygame.locals import *

# Clase MiSprite
class MiSprite(pygame.sprite.Sprite):
    # Inicializador de la clase
    def __init__(self, dibujo):
        # Importante: Primero hay que inicializar la clase Sprite original
        pygame.sprite.Sprite.__init__(self)

        # Almacenar en el sprite la imagen deseada
        self.image = pygame.image.load(dibujo)
        self.image = self.image.convert()

        # Definir el rect del sprite
        self.rect = self.image.get_rect()
        self.rect.topleft = (0, 150)

        # Definir las velocidades
        self.dx = 5.0
        self.dy = 1.0

    def update(self):
```

Pygame

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario07p.py
# Movimiento de Plataforma
#-----

import pilas

# Inicializar Pilas y crear la ventana por defecto
pilas.iniciar()

# Clase MiSprite
class MiSprite(pilas.actores.Actor):
    # Inicializador de la clase
    def __init__(self, imagen):
        # Importante: Primero hay que inicializar la clase original
        pilas.actores.Actor.__init__(self, imagen)

        # Situar el sprite en la posición deseada
        self.arriba = 90
        self.izquierda = -320

        # Definir la velocidad
        self.dx = 5.0
        self.dy = -1.0

    def actualizar(self):
        # Modificar la posición del sprite
        self.x += self.dx
        self.y += self.dy
```

Pilas

```

# Modificar la posición del sprite
self.rect.move_ip(self.dx, self.dy)
# Comprobar si hay que cambiar el movimiento
if self.rect.left < 0 or self.rect.right > 640:
    self.dx = -self.dx
    self.image = pygame.transform.flip(self.image, True, False)
self.rect.move_ip(self.dx, self.dy)
if self.rect.top < 0 or self.rect.bottom > 480:
    self.dy = -self.dy
self.rect.move_ip(self.dx, self.dy)
# Simular la gravedad sumando una cantidad a la velocidad vertical
self.dy = self.dy + 0.5

```

Pygame

```

# Inicializar PyGame y crear la Surface del juego
pygame.init()
visor = pygame.display.set_mode((640, 480))

```

```

# Inicializar el sprite
sprite = MiSprite("mario.bmp")
grupo = pygame.sprite.RenderUpdates(sprite)

```

```

# Crear un reloj para controlar la animación
reloj = pygame.time.Clock()

```

```

# El bucle de la animación
while 1:
    #Fijar la animación a 60 fps
    reloj.tick(60)

```

```

# Gestionar los eventos
for evento in pygame.event.get():
    if evento.type == QUIT:
        pygame.quit()
        sys.exit()

```

```

# Comprobar si hay que cambiar el movimiento
if self.izquierda < -320 or self.derecha > 320:
    self.dx = -self.dx
    self.espejado = not self.espejado
self.x += self.dx
if self.arriba > 240 or self.abajo < -240:
    self.dy = -self.dy
self.y += self.dy
# Simular la gravedad sumando una cantidad a la velocidad vertical
self.dy -= 0.5

```

Pilas

```

# Inicializar el sprite
sprite = MiSprite("mario.png")

```

```

# Inicializar el fondo
pilas.fondos.Color(pilas.colores.blanco)

```

```

pilas.ejecutar()

```

```
# Actualizar el sprite
grupo.update()

# Dibujar la escena
visor.fill((255, 255, 255))
grupo.draw(visor)


# Mostrar la animación
pygame.display.update()
```

Pygame

Pilas

En efecto, basta darle al sprite un pequeño empujoncito extra en el momento en el que llega al borde. Eso nos deja el movimiento con el mismo valor que justo antes de producirse el rebote, con lo que llegará hasta la misma altura al subir (observa que la altura a la que llega Mario es como un quesito y ese **0.5** que restamos a la velocidad como un ratón; va mordiéndole y quitando un trozo tras otro hasta que lo termina, de ahí que el rebote vaya, de partida, disminuyendo poco a poco) y no se producirán efectos extraños en las paredes.

```
if self.arriba > 240 or self.abajo < -240:
    self.dy = -self.dy
    self.y += self.dy
```



(De paso, silenciosamente, hemos cambiado el valor inicial de **self.dx** a **5.0**, para que el movimiento sea más ágil y más plataformero.)

Por cierto, si necesitáramos solucionar el tema de atravesar el suelo por su cuenta, tampoco sería complicado. ¿Te imaginas cómo? Fácil; antes de cambiar la posición del sprite con **self.y += self.dy** deberías comprobar si ya está en el suelo, en cuyo caso deberías poner el valor de **self.dy** a cero para que dejara de bajar. (Pregunta: ¿Cómo saber si los pies de Mario están en el borde de la ventana? Respuesta: con el atributo **abajo** que tiene todo objeto de tipo **Actor**)

mario08p.py

Ya que estamos trabajando con sprites. ¿Por qué limitarnos a un sólo Mario? Traigamos a su hermano gemelo a la fiesta...

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario08.py
# Varios sprites
#-----
```

```
import sys
import pygame
from pygame.locals import *
```

```
# Clase MiSprite
class MiSprite(pygame.sprite.Sprite):
```

```
# Inicializador de la clase
```

```
def __init__(self, dibujo, posX, posY):
```

```
# Importante: Primero hay que inicializar la clase Sprite original
pygame.sprite.Sprite.__init__(self)
```

```
# Almacenar en el sprite la imagen deseada
```

```
self.image = pygame.image.load(dibujo)
```

```
self.image = self.image.convert_alpha()
```

```
# Definir el rect del sprite
```

```
self.rect = self.image.get_rect()
```

```
self.rect.topleft = (posX, posY)
```

```
# Definir las velocidades
```

```
self.dx = 5.0
```

```
self.dy = 1.0
```

```
def update(self):
```

Pygame

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario08p.py
# Varios sprites
#-----
```

```
import pilas
```

```
# Inicializar Pilas y crear la ventana por defecto
pilas.iniciar()
```

```
# Clase MiSprite
```

```
class MiSprite(pilas.actores.Actor):
```

```
# Inicializador de la clase
```

```
def __init__(self, imagen, posX, posY):
```

```
# Importante: Primero hay que inicializar la clase original
pilas.actores.Actor.__init__(self, imagen)
```

```
# Situar el sprite en la posición deseada
```

```
self.izquierda = posX
```

```
self.arriba = posY
```

```
# Definir la velocidad
```

```
self.dx = 5.0
```

```
self.dy = -1.0
```

```
def actualizar(self):
```

```
# Modificar la posición del sprite
```

```
self.x += self.dx
```

```
self.y += self.dy
```

Pilas

Pygame

```

# Modificar la posición del sprite
self.rect.move_ip(self.dx, self.dy)
# Comprobar si hay que cambiar el movimiento
if self.rect.left < 0 or self.rect.right > 640:
    self.dx = -self.dx
    self.image = pygame.transform.flip(self.image, True, False)
    self.rect.move_ip(self.dx, self.dy)
if self.rect.top < 0 or self.rect.bottom > 480:
    self.dy = -self.dy
    self.rect.move_ip(self.dx, self.dy)
# Simular la gravedad sumando una cantidad a la velocidad vertical
self.dy = self.dy + 0.5

# Inicializar PyGame y crear la Surface del juego
pygame.init()
visor = pygame.display.set_mode((640, 480))

# Inicializar los sprites
sprite = MiSprite("mario.png", 0, 150)
grupo = pygame.sprite.RenderUpdates(sprite)
sprite2 = MiSprite("mario.png", 210, 50)
grupo.add(sprite2)

# Crear un reloj para controlar la animación
reloj = pygame.time.Clock()

# El bucle de la animación
while 1:
    #Fijar la animación a 60 fps
    reloj.tick(60)

# Gestionar los eventos
for evento in pygame.event.get():
    if evento.type == QUIT:
        pygame.quit()

```

Pilas

```

# Comprobar si hay que cambiar el movimiento
if self.izquierda < -320 or self.derecha > 320:
    self.dx = -self.dx
    self.espejado = not self.espejado
    self.x += self.dx
if self.arriba > 240 or self.abajo < -240:
    self.dy = -self.dy
    self.y += self.dy
# Simular la gravedad sumando una cantidad a la velocidad vertical
self.dy -= 0.5

# Inicializar los sprites
sprite = MiSprite("mario.png", -320, 90)
sprite2 = MiSprite("mario.png", -110, 190)

# Inicializar el fondo
pilas.fondos.Color(pilas.colores.blanco)

pilas.ejecutar()

```

```
sys.exit()

# Actualizar los sprites
grupo.update()

# Dibujar la escena
visor.fill((255, 255, 255))
grupo.draw(visor)

# Mostrar la animación
pygame.display.update()
```

Pygame**Pilas**

Para empezar, hemos modificado ligeramente la definición de nuestro tipo de sprite ya que si no todos los sprites que creamos de este tipo aparecerán en el mismo sitio (y por tanto sólo veríamos uno). Esto es fácil de solucionar; ponemos dos parámetros más en el método `__init__()` que inicializa la clase.

```
def __init__(self, imagen, posX, posY):
```

Los parámetros **posX** y **posY** se encargarán de situar en su posición inicial al sprite. Ello quiere decir que hay que modificar unas líneas más de esta función:

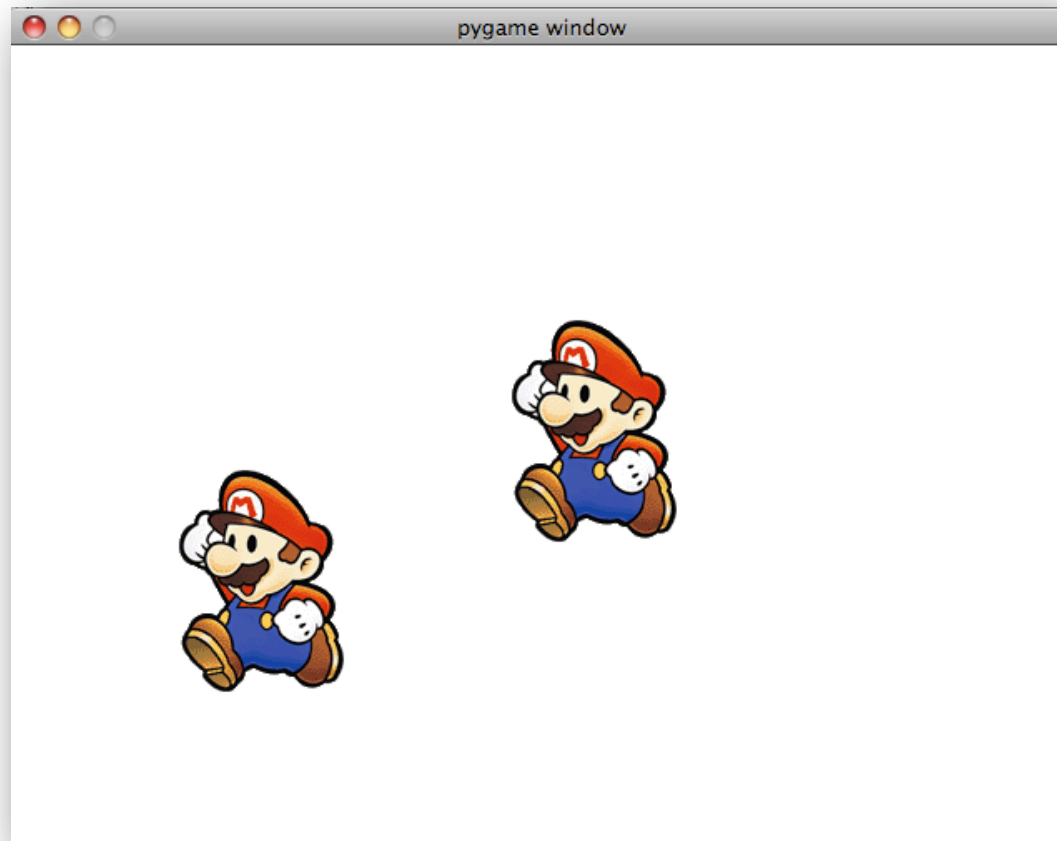
```
self.izquierda = posX
self.arriba = posY
```

para que así, en efecto, el sprite se posicione allí al principio.

Lo último que queda es, simplemente, crear los sprites:

```
sprite = MiSprite("mario.png", -320, 90)
sprite2 = MiSprite("mario.png", -110, 190)
```

Respecto del primer Mario, poco que decir. Lo único que hemos modificado es que, ahora, en la creación del sprite hay que indicar la posición en la que lo queremos. Con el segundo sprite hacemos lo mismo (con otra posición distinta). Observa, como indicamos antes, que en Pilas no tenemos que añadir el nuevo sprite a un grupo, ya que éstos no son estrictamente necesarios.



Dos cosas podrás notar cuando ejecutes el programa. La primera es que, como la imagen de Mario tiene transparencia, cuando se superponen los dos gemelos no hay ningún problema; si no tuviera transparencia, se vería el cuadrado blanco que envuelve a la imagen y no quedaría bonito (en Pygame puede usarse un truco al respecto). La segunda es que cada sprite ignora al otro. ¿Cómo podríamos gestionar la colisión entre ambos de manera que, por ejemplo, rebotaran?... Tendremos que pasar a la última versión del programa.

mario09p.py

Hay varias formas de abordar el tema. La que viene a continuación es sólo una de las más sencillas:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario09.py
# Sprites con colisión
#-----

import sys
import pygame
from pygame.locals import *

# Clase MiSprite
class MiSprite(pygame.sprite.Sprite):
    # Inicializador de la clase
    def __init__(self, dibujo, posX, posY):
        # Importante: Primero hay que inicializar la clase Sprite original
        pygame.sprite.Sprite.__init__(self)

        # Almacenar en el sprite la imagen deseada
        self.image = pygame.image.load(dibujo)
        self.image = self.image.convert_alpha()

        # Definir el rect del sprite
        self.rect = self.image.get_rect()
        self.rect.topleft = (posX, posY)

        # Definir las velocidades
        self.dx = 5.0
        self.dy = 1.0

    def update(self):
```

Pygame

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario09p.py
# Sprites con colisión
#-----

import pilas

# Inicializar Pilas y crear la ventana por defecto
pilas.iniciar()

# Clase MiSprite
class MiSprite(pilas.actores.Actor):
    # Inicializador de la clase
    def __init__(self, imagen, posX, posY):
        # Importante: Primero hay que inicializar la clase original
        pilas.actores.Actor.__init__(self, imagen)

        # Situar el sprite en la posición deseada
        self.izquierda = posX
        self.arriba = posY

        # Definir la velocidad
        self.dx = 5.0
        self.dy = -1.0

    # Aumentar el radio de colisión
    self.radio_de_colision = 50

    def actualizar(self):
```

Pilas

```
# Modificar la posición del sprite
self.rect.move_ip(self.dx, self.dy)
# Comprobar si hay que cambiar el movimiento
if self.rect.left < 0 or self.rect.right > 640:
    self.dx = -self.dx
    self.image = pygame.transform.flip(self.image, True, False)
    self.rect.move_ip(self.dx, self.dy)
if self.rect.top < 0 or self.rect.bottom > 480:
    self.dy = -self.dy
    self.rect.move_ip(self.dx, self.dy)
# Simular la gravedad sumando una cantidad a la velocidad vertical
self.dy = self.dy + 0.5
```

Pygame

```
# Inicializar PyGame y crear la Surface del juego
pygame.init()
visor = pygame.display.set_mode((640, 480))
```

```
# Inicializar los sprites
sprite = MiSprite("mario.png", 0, 150)
grupo = pygame.sprite.RenderUpdates(sprite)
sprite2 = MiSprite("mario.png", 210, 50)
grupo2 = pygame.sprite.RenderUpdates(sprite2)
```

```
# Crear un reloj para controlar la animación
reloj = pygame.time.Clock()
```

```
# El bucle de la animación
while 1:
    #Fijar la animación a 60 fps
    reloj.tick(60)
```

```
# Gestionar los eventos
for evento in pygame.event.get():
    if evento.type == QUIT:
        pygame.quit()
```

```
# Modificar la posición del sprite
self.x += self.dx
self.y += self.dy
# Comprobar si hay que cambiar el movimiento
if self.izquierda < -320 or self.derecha > 320:
    self.dx = -self.dx
    self.espejado = not self.espejado
    self.x += self.dx
if self.arriba > 240 or self.abajo < -240:
    self.dy = -self.dy
    self.y += self.dy
# Simular la gravedad sumando una cantidad a la velocidad vertical
self.dy -= 0.5
```

Pilas

```
# Definir el rebote en la colisión
def rebotar(un_sprite, otro_sprite):
    un_sprite.dx = -un_sprite.dx
    un_sprite.dy = -un_sprite.dy
    otro_sprite.dx = -otro_sprite.dx
    otro_sprite.dy = -otro_sprite.dy
```

```
# Inicializar los sprites
sprite = MiSprite("mario.png", -320, 90)
sprite2 = MiSprite("mario.png", -110, 190)
```

```
# Inicializar el fondo
pilas.fondos.Color(pilas.colores.blanco)
```

```
# Añadir colisión entre sprites
pilas.escena_actual().colisiones.agregar(sprite, sprite2, rebotar)
```

```
pilas.ejecutar()
```

```
sys.exit()
```

```
# Mira si hay alguna colisión:
```

```
if pygame.sprite.spritecollideany(sprite, grupo2):  
    sprite.dx = -sprite.dx  
    sprite.dy = -sprite.dy  
    sprite2.dx = -sprite2.dx  
    sprite2.dy = -sprite2.dy
```

```
# Actualizar los sprites
```

```
grupo.update()
```

```
grupo2.update()
```

```
# Dibujar la escena
```

```
visor.fill((255, 255, 255))
```

```
grupo.draw(visor)
```

```
grupo2.draw(visor)
```

```
# Mostrar la animación
```

```
pygame.display.update()
```

Pygame

Pilas

Lo primero es una **advertencia**: como cualquier programador sabe, los documentos de **referencia** (en los que se incluyen las librerías, objetos e instrucciones disponibles en el lenguaje) son un **compañero inseparable** en la aventura de programar. ¿No tienes cerca el manual de Pilas? Cógelo ahora mismo. Piensa que te nombramos unas pocas funciones u objetos pero hay muchas más. Y muchas más opciones.

Bien, vamos al tema de la **detección de colisiones**. Pilas incorpora mecanismos que gestionan los posibles choques entre sprites (y con grupos también). ¡Mira el **Manual de Pilas**! Como notarás enseguida, sobre todo si has realizado antes el tutorial de **Disparar a Monos**, la manera más cómoda es **agregar la colisión a la escena actual** y asociarla, cuando tenga lugar, a la ejecución de una función. Fíjate cómo lo hemos escrito:

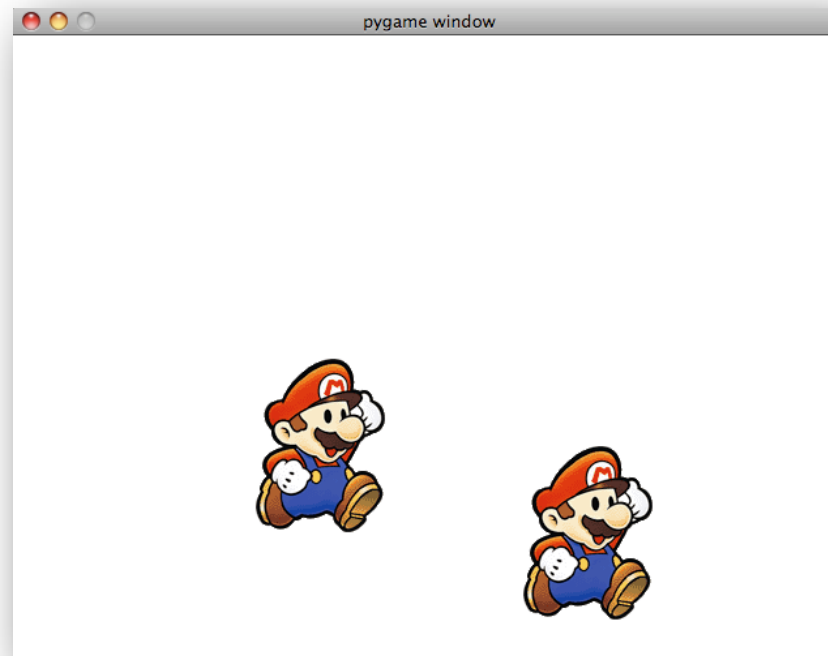
```
pilas.escena_actual().colisiones.agregar(sprite, sprite2, rebotar)
```

En la línea anterior, lo que le decimos a Pilas es “Hey, a partir de ahora, cuando tenga lugar en la escena actual la colisión entre el **sprite** y el **sprite2**, ejecuta la función **rebotar()**”. Claro está, debemos definir esa función:

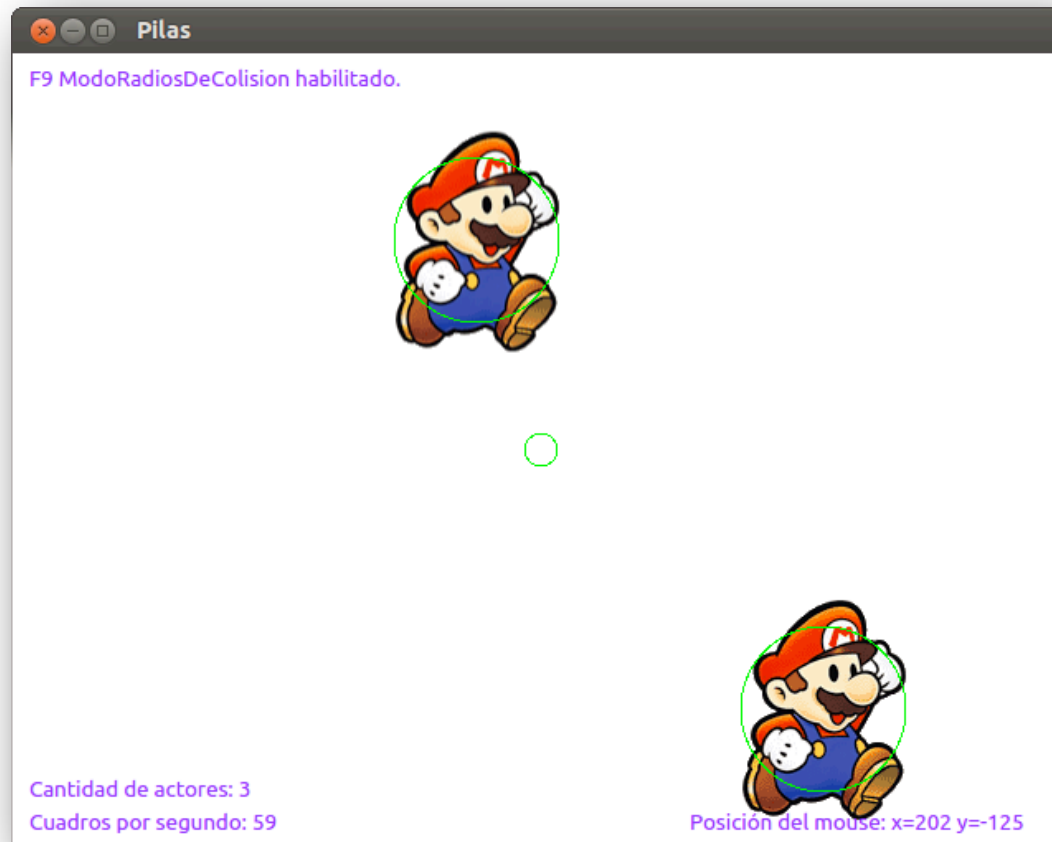
```
def rebotar(un_sprite, otro_sprite):  
    un_sprite.dx = -un_sprite.dx  
    un_sprite.dy = -un_sprite.dy  
    otro_sprite.dx = -otro_sprite.dx  
    otro_sprite.dy = -otro_sprite.dy
```

En las colisiones, **Pilas pasa a la función asociada**, automáticamente, **los sprites** (los actores...) **que han chocado**, de ahí que los incluyamos en la definición. Esto es así incluso cuando indicamos, al agregar la colisión, un **grupo entero de sprites**. Podíamos haber creado 500 Marios, añadirlos al mismo grupo, y agregar la definición de la colisión de uno de ellos con el grupo; la función se ejecutará en el momento en el que el Mario elegido choque con cualquier otro.

Bien. ¿Qué es lo que hacemos en la función **rebotar()**? Pues simplemente invertimos las velocidades de movimiento, es decir, los atributos **dx** y **dy** de cada sprite; de esta forma, y análogamente al rebote en los bordes de la ventana, cambiamos las direcciones de movimiento de los dos sprites. Ejecuta ahora el programa... ¡Genial!



Un detalle importante que observarás, es que la colisión tiene lugar cuando se han mezclado algo los dos Marios, no cuando se rozan estrictamente sus bordes. Esto tiene que ver con lo que se denomina **radio de colisión**. Para visualizarlo, mientras está el programa en ejecución, pulsa la tecla **F9**:



Has entrado en un **modo de depuración** de Pilas con el que puedes visualizar información interesante (¿recuerdas el Manual?...). En particular, verás un círculo verde que indica el **radio de colisión** del actor. Sólo cuando chocan los círculos correspondientes de los actores es cuando se lanza el evento de colisión. Por defecto, el radio de colisión de un actor genérico tiene un tamaño de **10** pixels. Para nuestro Mario es algo pequeño ¿no crees?. Por eso, en el `__init__()` de la definición del sprite hemos añadido la línea

```
self.radio_de_colision = 50
```

La diferencia entre no incluirla y hacerlo la puedes visualizar en las imágenes siguientes:



Esto sí. Con este cambio, el rebote es más realista (prueba a quitar la línea y lo comprobarás).

Por cierto; cuando rebotan los dos Marios, el uno contra el otro, no invierten su dibujo. ¿Sabrías solucionarlo? Ya puestos, ¿sabrías hacer que comiencen desde una posición aleatoria? ¿Y que salgan, no dos, si no muchos marios, por ejemplo cada vez que se haga click con el ratón?

Ante nosotros se extiende un campo de creatividad ilimitado...

... Pero una vez que hayas terminado el reto, ¡no te vayas todavía! Hay una segunda parte de este tutorial. Ya te avisamos...

La Física de Pilas

mario_pilas.py

Hemos estado escribiendo este tutorial sobre Pilas siguiendo la metodología que corresponde a la librería de Pygame. Pero Pilas tiene algo que no incorpora Pygame por defecto, y es, ni más ni menos, que **comportamiento físico real** de los objetos que empleemos. Y todo ello gracias a la librería **Box2D** que incluye. Así que ¿qué te parece que escribamos el programa completo de Mario, tal como quedaría, al modo exclusivo de Pilas al usar su Física incorporada?

¡Vamos a ello! Quedaría algo así:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#-----
# mario_pilas.py
# Acotres con colisión y Física
#-----

import pilas

pilas.iniciar()

# Clase MiActor
class MiActor(pilas.actores.Actor):
    # Inicializador de la clase
    def __init__(self, imagen, x=0, y=0, v=0):
        # Importante: Primero hay que inicializar la clase original
        pilas.actores.Actor.__init__(self, imagen)
        self.x = x
        self.y = y
        self.rectangulo = pilas.fisica.Rectangulo(x=self.x, y=self.y,
            ancho=self.ancho, alto=self.alto,
            restitution=0.95, friccion=0.001)
        self.rectangulo.definir_velocidad_lineal(v, 0)
        self.imitar(self.rectangulo)
```

```
def actualizar(self):
    # Si invierte velocidad, invertir imagen
    if self.rectangulo.obtener_velocidad_lineal()[0] < 0:
        self.espejado = True
    else:
        self.espejado = False

pilas.fondos.Color(pilas.colores.blanco)
mario1 = MiActor("mario.png", x=-260, y=0, v=10)
mario2 = MiActor("mario.png", x=250, y=50, v=-10)

pilas.ejecutar()
```

¿Breve, no? Veamos lo que hemos hecho, por que lo implicado es mucho.

Para empezar hemos creado una nueva clase de actor basada en el genérico **pilas.actores.Actor**, algo que ya hicimos en la primera parte de este tutorial. En el método `__init__()` de la clase, ya puestos, además de las coordenadas donde queremos que aparezca nuestro Mario, hemos incluido como argumento la velocidad (horizontal) con la que queremos que arranque. Les hemos puesto unos valores por defecto, por supuesto, de manera que podríamos crear los actores sin necesidad de indicar nada más, en cuyo caso aparecería en el centro y sin velocidad horizontal.

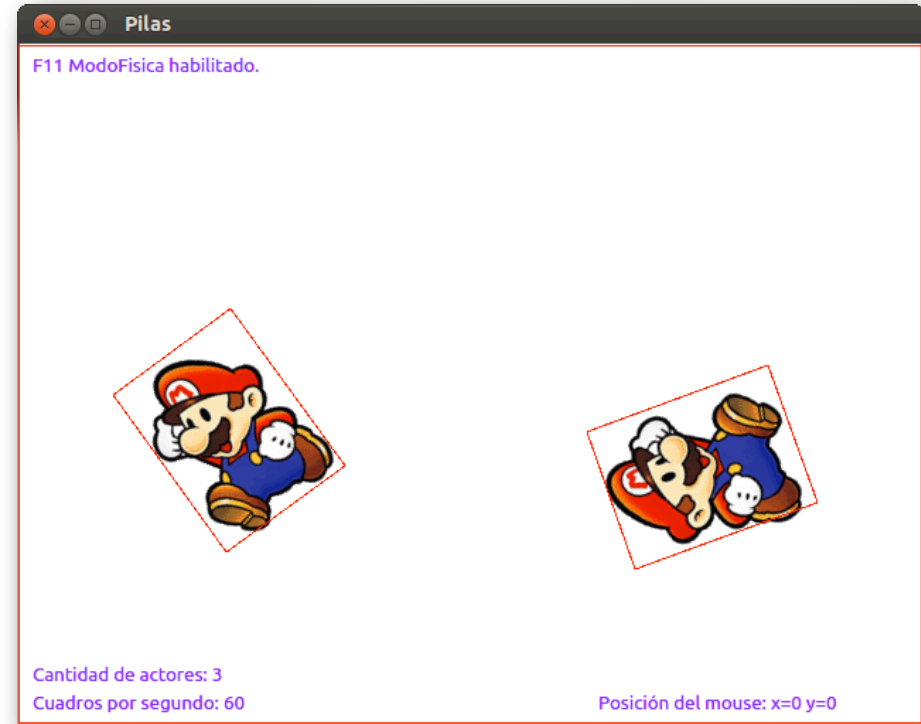
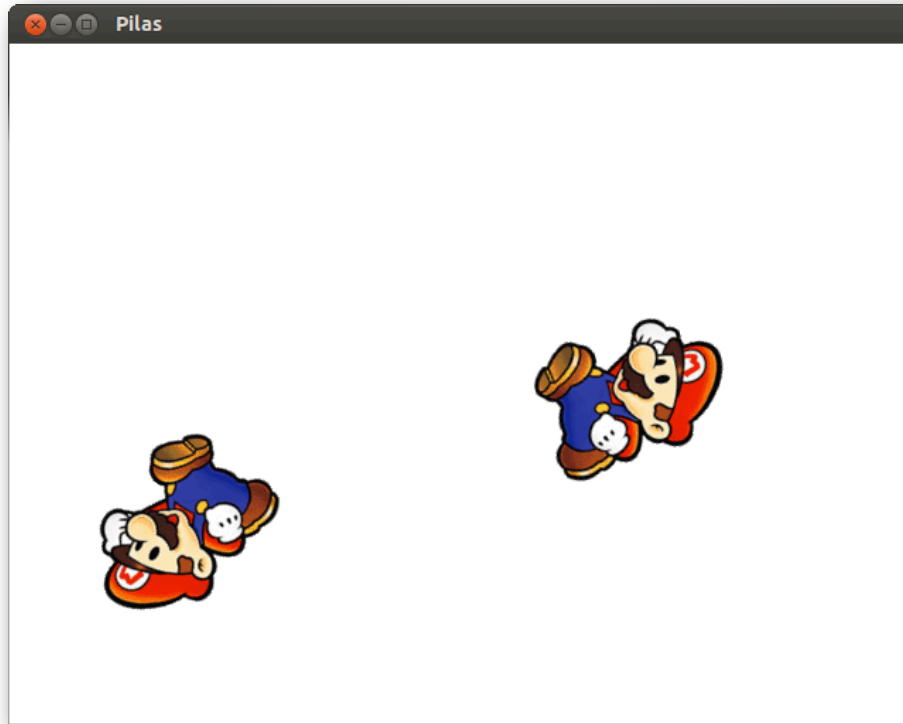
La diferencia fundamental la encontramos en esta línea de código:

```
self.rectangulo = pilas.fisica.Rectangulo(x=self.x, y=self.y,
                                         ancho=self.ancho, alto=self.alto,
                                         restitution=0.95, friccion=0.001)
```

El módulo **pilas.fisica** implementa todas las reglas físicas a las que hemos aludido y en él podemos encontrar objetos que representan comportamientos reales. En particular, y almacenándolo en la variable **self.rectangulo**, hemos creado un objeto de tipo **pilas.fisica.Rectangulo**. ¿Quieres verlo? Veamos...

Prueba a ejecutar el programa y observa. ¡Verás cómo los dos Marios chocan, rebotan y dan vueltas!

Ahora vuelve a ejecutar el programa y pulsa la tecla **F11**, para habilitar el modo física. ¿Ves los rectángulos?



En efecto, un objeto **pilas.fisica.Rectangulo** es un **objeto físico con todo el comportamiento esperable de una caja rectangular**. ¿Cómo le decimos a Pylas que nuestro actor se comporte como el rectángulo que hemos creado? En la siguiente línea de su `__init__()` está la clave:

```
self.imitar(self.rectangulo)
```

Usando el método **imitar()** de todo actor podemos conseguir que se imite el comportamiento de otro objeto, en este caso, el rectángulo físico (por supuesto, en ambos casos, ¡no te olvides del **self**!)

¿No es fantástico que se ocupe de todo lo demás el motor de Pylas?

Eso sí, para que la simulación física funcione bien, hay que definir las cosas con cabeza. Fíjate, más arriba, en los parámetros que le hemos pasado al Rectangulo para su creación:

- La posición y el tamaño (**x**, **y**, **ancho** y **alto**) son los de Mario (**self.x**, **self.y**, **self.ancho**, **self.alto**).
- Un valor de **restitucion** cercano a **1**. Este parámetro controla que los choques sean más o menos elásticos.
- Un valor de **friccion** cercano a **0** para que los actores no se vean en exceso frenados

...

Lo has adivinado. ¿Estabas pensando en ello? El **Manual de Pilas** y las funciones **help()**, **dir()** y **pilas.ver()** son tus amigas. Hay más opciones que ajustar, solo la experiencia y la práctica te darán tus elecciones óptimas.

pilas.fisica.Rectangulo

Como ejemplo, escribe `help(pilas.fisica.Rectangulo)` en el intérprete de Pilas. Esto es lo que obtendrás como resultado (mostramos solo un extracto):

Help on `class Rectangulo` in module `pilas.fisica`:

```
class Rectangulo(Figura)
| Representa un rectángulo que puede colisionar con otras figuras.
|
| Se puede crear un rectángulo independiente y luego asociarlo
| a un actor de la siguiente forma:
|
|     >>> rect = pilas.fisica.Rectangulo(50, 90, True)
|     >>> actor = pilas.actores.Pingu()
|     >>> actor.imitar(rect)
|
| Method resolution order:
|   Rectangulo
|   Figura
|   __builtin__.object
|
| Methods defined here:
```

Breve explicación de la clase



```
| __init__(self, x, y, ancho, alto, dinamica=True, densidad=1.0, restitution=0.5, friccion=0.2, amortiguacion=0.1,  
|         fisica=None, sin_rotacion=False)
```

| Methods inherited from Figura:

```
| definir_posicion(self, x, y)
```

```
| definir_rotacion(self, angulo)
```

```
| definir_velocidad_lineal(self, dx=None, dy=None)
```

```
| definir_x(self, x)
```

```
| definir_y(self, y)
```

```
| detener(self)
```

Hace que la figura regrese al reposo.

```
| eliminar(self)
```

Quita una figura de la simulación.

```
| empujar(self, dx=None, dy=None)
```

```
| impulsar(self, dx, dy)
```

```
| obtener_rotacion(self)
```

```
| obtener_velocidad_lineal(self)
```

```
| obtener_x(self)
```

```
| obtener_y(self)
```

Parámetros para su creación



Otros métodos y atributos



Hay un par de cosas más que hemos ajustado en la definición del actor con el objetivo de que veas la flexibilidad disponible. Para empezar, ¿recuerdas que dábamos como opción, en la creación de Mario, dar su velocidad horizontal inicial? ¿Cómo se la indicamos a la simulación física? He aquí la respuesta:

```
self.rectangulo.definir_velocidad_lineal(v, 0)
```

¿Te has fijado en el texto que ha devuelto la función `help()`? Allí encontrarás, precisamente, el método `definir_velocidad_lineal()`. Fíjate, también, que le pasamos los valores en el orden correcto; `v` como velocidad horizontal y `0` como vertical.

Finalmente, como complemento, también hemos usado el método `actualizar()` que ya conocemos para reflejar la imagen de Mario, cuando intercambia su velocidad horizontal, usando otro método del rectángulo; `obtener_velocidad_lineal()`.

Código Fuente de Pilas

Un buen ejercicio es ir mirando cómo está escrito Pilas, de esta manera puedes profundizar más allá del Manual. Escribe `pilas.ver(pilas.fisica.Rectangulo.obtener_velocidad_lineal)` en el intérprete de Pilas y obtendrás

```
def obtener_velocidad_lineal(self):  
    # TODO: convertir a pixels  
    velocidad = self._cuerpo.linearVelocity  
    return (velocidad.x, velocidad.y)
```

Aquí puedes ver cómo, lo que devuelve, es una tupla de 2 valores; la velocidad horizontal y la velocidad vertical. Es por eso por lo que en el programa miramos si `self.rectangulo.obtener_velocidad_lineal()[0]` es menor que `0`, para elegir, vía `self.espejado`, si ponemos a Mario mirando hacia un lado o hacia el otro (según sea su movimiento horizontal, que es el elemento 0 de la tupla).

¡Ya está todo hecho! ¿Te ves con ganas de hacer las mismas mejoras (u otras) que hemos comentado en la primera parte de este tutorial? Investiga, investiga, que se aprende haciendo, no leyendo...