



# VJ - Videojuegos [Material]

---

Experiencia: ¡Videojuegos! 🤖🎮

---

## Tutorial

► Tabla de contenidos

### 1. Setup

Lo primero que necesitamos es instalar `pygame`, librería con la que desarrollaremos nuestro videojuego. Esta librería nos permite crear videojuegos en dos dimensiones, accediendo a componentes de hardware de nuestro sistema tal como el audio, video, teclado, mouse, entre otros.

Para instalarlo deberás ejecutar el siguiente comando en la terminal (si tienes algún problema con `pip`, puedes probar con `pip3`):

```
pip install pygame
```

pip es el package manager de Python, y viene incluido con la instalación de este.

- Si usas Anaconda no es necesario que instales la librería 😊

Para asegurarnos que la instalación fue exitosa correremos alguno de los ejemplos de la librería. Si una ventana de juego se abre, ¡vamos por excelente camino!

En la terminal ejecuta el siguiente comando:

```
python -m pygame.examples.aliens
```

Si tienes algún problema con usar el comando python, puedes reemplazarlo por python3

¡La documentación es nuestra mejor amiga! Ante cualquier duda no dudes en consultarla, esta práctica es lejos lo mejor que puedes hacer para entender lo que estás haciendo.

→ [Documentación](#) ←

## 2. Introducción a **pygame**

Porque para volvernos expertos necesitamos comprender un poco cómo funciona por detrás el desarrollo de videojuegos, veremos algunos conceptos claves y cómo son implementados por **pygame**.

### ¿Cómo se genera la interacción humano - computador?

1. **GUI**: *Graphical User Interface*. En 🇪🇸, interfaz gráfica de usuario. Es necesaria sin importar el lenguaje o librería que utilices, ya que permite interactuar con el computador mediante ventanas, íconos, menús, entre otros. La GUI opera mediante un mecanismo llamado **polling** para escuchar activamente las interacciones de, por ejemplo, teclado y mouse, con algún elemento de la interfaz.


OK OK, ¿Y la lógica del juego?

1. **Game Loop** : Aquí se encuentra la lógica principal, **todo lo que suele repetirse**. Sus tareas principales son procesar el input del usuario, actualizar el estado de elementos de la GUI y de los objetos del juego, y definir cómo este empieza, avanza y termina.
2. **Eventos** : La interfaz gráfica es una aplicación con una arquitectura basada en manejo de eventos. La interfaz se encarga de detectar eventos generados por el usuario como los mencionados en el punto anterior, y los procesa. Tenemos la **fuentes del evento** (qué objeto generó el cambio de estado), **el objeto evento** (encapsula la información sobre dicho cambio de estado) y **el objeto destino** (objeto al que se le notificará el cambio de estado).
3. **Señales** : Los cambios de estado son comunicados mediante señales. El flujo es que la fuente del evento envía una señal que contiene información sobre el objeto evento al objeto destino.

## 2.1 Inicialización y módulos


Porque **nada** es suficientemente básico, lo primero que debemos hacer para empezar a trabajar con **pygame** es importarlo e inicializarlo:

```
import pygame
pygame.init()
```

La línea 2 inicializa todos los módulos incluidos en la librería. Cada módulo ofrece acceso abstracto a partes del hardware de tu computador y métodos para trabajar con ellas. Sin esto, no podríamos detectar un click del mouse , por ejemplo.

## 2.3 Displays y surfaces

Ya hablamos del hardware, pero ¿qué pasa con el modelamiento de software? Para esto tenemos disponibles los siguientes módulos:

1. **Surface** : Es el elemento básico, el área donde podemos pintar en la interfaz. **Disclaimer:** Usaremos el término pintar o dibujar  para referirnos a agregar elementos gráficos a nuestra GUI. Podemos embeber imágenes, dibujar figuras geométricas, etcétera.
2. **Display** : ¡Aquí la magia ocurre! Es la ventana principal del juego, la cual se crea con **.set\_mode** (paciencia, lo veremos), retornando el **Surface** principal donde dibujaremos.

3. `image` : Módulo que nos permite cargar y guardar imágenes en distintos formatos para embeberlas en un objeto `Surface` .
4. `Rect` : En `pygame` los objetos `Surface` se representan por rectángulos al igual que las imágenes y ventanas. Como esta figura geométrica es tan usada existe la clase `Rect` para manejarlas. Con ella podremos dibujar jugadores, enemigos, colisiones, ¡y todo lo que queramos!

### 3. Diseño básico del juego

Para saber qué haremos, debemos modelar en papel la idea del juego. Este consiste en:

- **Nombre:** ¡Deberás ponerle un nombre a tu juego! Este quedará a tu criterio 😊
- **Objetivo:** Evitar obstáculos:
  - El jugador parte en el lado izquierdo de la pantalla.
  - Los obstáculos entran de manera aleatoria por el lado derecho y avanzan hacia la izquierda (←) en línea recta.
- El jugador puede moverse en **todas** las direcciones para evitar obstáculos.
- El jugador **no puede salirse de la ventana**.
- 🌟**GAME OVER** 🌟 cuando el jugador choca con un obstáculo o cuando cierra la ventana.

Algunas cosas que no aplicaremos pero verás más adelante en el **major** 😊:

- No hay múltiples vidas.
- No hay puntaje.
- El jugador no tiene poderes.
- No se avanza a otros niveles.



¡Te invitamos a desarrollar estas funcionalidades por tu cuenta! `pygame` es una librería con mucha documentación y tutoriales 😊

¡Ahora manos a la obra!

#### 3.1 Importar e inicializar librería

Como primer paso volveremos a importar e inicializar la librería, pero ahora agregando también la importación de `pygame.locals` para poder tener acceso fácil a las coordenadas de las teclas:

```
import pygame from pygame.locals import( K_UP, K_DOWN, K_LEFT,
K_RIGHT, K_ESCAPE, KEYDOWN, QUIT) pygame.init()
```

Los próximos códigos que se muestren, se añadirán en el mismo archivo.

Estas coordenadas son constantes y nos permiten acceder a los eventos nombrados anteriormente, como `click` del mouse, presión de teclas, etcétera. Luego lo veremos con más detalle pero básicamente se acceden de la forma `pygame.<CONSTANT>` donde `<CONSTANT>` es alguno de los importes de `pygame.locals`.

## 3.2 Implementar display

¡Ahora entonces nos ponemos a dibujar! Primero, debemos crear una pantalla que será nuestro `canvas`:

```
# definir el tamaño de la pantalla SCREEN_WIDTH = 800 SCREEN_HEIGHT =
600 # crear el objeto pantalla screen =
pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))
```

→ Acabamos de crear el `Surface` principal como explicamos anteriormente con el método `set_mode()`, el que recibe una tupla o lista con el tamaño que seteamos.

**!!** ¡Recuerda! Todo este tutorial es código secuencial

**Ya puedes correr el código y verás que una ventana aparece por un instante y desaparece 🤔**

## 3.3 Implementar ciclo del juego

Cada ciclo o iteración del loop en el juego es llamado `frame`, los que ocurren hasta que alguna condición de salida se cumple. Como explicamos en la modelación, estas condiciones de salida serán dos:

- El jugador choca con un obstáculo.

- El jugador cierra la ventana.

Por otro lado, la tarea constante del loop será procesar el `input` del usuario para poder mover al jugador por la pantalla, y esto hay que capturarlo y procesarlo de alguna forma (Recuerda que estos son eventos!).

### 3.4 Procesamiento de eventos

¿Cuáles son los eventos que pueden ocurrir? → Presionar una tecla, click del mouse, etcétera. Cada evento que ocurra es puesto en una cola **FIFO** (First In First Out), es decir, **el primer evento que llega es el primero en ejecutarse**, y así en orden hacia el final de la cola.

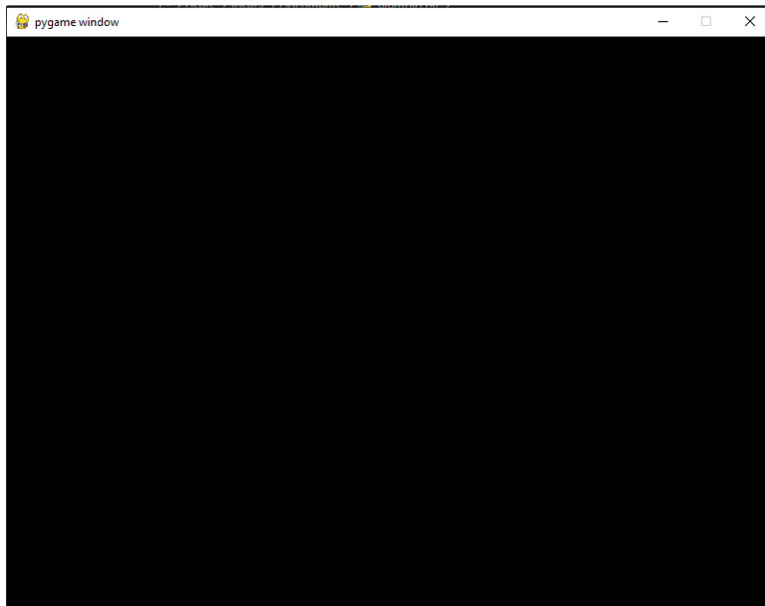
Cada evento nombrado tiene un tipo, los que nos interesarán son los `keypresses` y el cierre de la ventana. Los primeros son del tipo `KEYDOWN` y el segundo del tipo `QUIT`. Además, **cada tipo de evento tiene información asociada a él**, por ejemplo, `KEYDOWN` también nos dice cuál fue la tecla presionada.

Se puede acceder a todos los eventos de la cola con `pygame.event.get()`. Iremos iterando sobre esta, inspeccionando cada elemento, y procesándolo de la manera que corresponde:

```
# variable booleana para manejar el loop running = True # loop principal del juego while running: # iteramos sobre cada evento en la cola for event in pygame.event.get(): # se presiono una tecla? if event.type == KEYDOWN: # era la tecla de escape? -> entonces terminamos if event.key == K_ESCAPE: running = False # fue un click al cierre de la ventana? -> entonces terminamos elif event.type == QUIT: running = False
```

**Ya puedes correr nuevamente el código y verás que ahora la ventana no desaparece hasta que presionas la tecla `ESC` o la cierras.**





### 3.5 Dibujar la interfaz

Existen tres formas de dibujar 🖋️:

1. `screen.fill()` : Rellena el fondo.
2. `pygame.draw.circle()` : Esto dibuja, por ejemplo, un círculo en el `Surface` principal.
3. `Surface` que es la que utilizaremos 😎. Recordemos que este es un objeto rectangular, como una hoja de papel en la que podemos pintar.

Como mencionamos anteriormente, el `screen` es un `Surface`, pero tú puedes crear más aparte de este:

```
# rellena la screen con un color, en este caso blanco
screen.fill((255, 255, 255)) # crea un objeto Surface con la tupla con
sus medidas surf = pygame.Surface((50, 50)) # dale a tu nuevo Surface
un color distinto para diferenciarlo del principal surf.fill((0, 0,
0)) rect = surf.get_rect()
```

\* Este código va dentro del ciclo, en la misma indentación que `for event in pygame.event.get():`

Si quieres buscar colores RGB puedes hacerlo [aquí](#).

`rect = surf.get_rect()` es para acceder al `Rect` del objeto `Surface` que creamos, el que ocuparemos más adelante.

### 3.6 Métodos `blit()` y `flip()`

Crear un objeto `Surface` no es suficiente para poder verlo en nuestra pantalla, necesitamos asignárselo al `Surface` principal. Para esto, ocuparemos `blit()`: *Block Transfer*. Lo que hace es copiar los contenidos de un `Surface` a otro.

Luego de lo añadido anteriormente, añadiremos:

```
# dibújame surf en el display principal llamado screen (que,
reiteramos, es otro Surface) screen.blit(surf, (SCREEN_WIDTH / 2,
SCREEN_HEIGHT / 2)) pygame.display.flip()
```

`blit()` toma dos argumentos:

- El `Surface` que se dibujará.
- La ubicación en el `Surface` sobre el que irá el contenido de dicho `Surface`.

En resumen, el código anterior dice que el contenido de `surf` irá al centro del contenido de `screen`.

Si corres el código ahora verás un cuadrado al medio, sin embargo, no está completamente centrado. Necesitaremos un poco de matemática para lograr esto → Esto es porque `blit()` pone la esquina izquierda superior del contenido a pintar (`surf` en este caso) en la ubicación dada y no el centro.

Reemplacemos el código justamente anterior por lo siguiente:

```
# calculamos el centro surf_center = ( (SCREEN_WIDTH -
surf.get_width()) / 2, (SCREEN_HEIGHT - surf.get_height()) / 2 ) #
dibujamos con las nuevas coordenadas screen.blit(surf, surf_center)
pygame.display.flip()
```

Como verás, después de usar `blit()`, ejecutamos `pygame.display.flip()`. El comando `.flip()` siempre es necesario luego de llamar a `blit()`, ya que actualiza la pantalla con todo lo que ha sido dibujado sobre ella desde la última vez que llamamos a este método. Sin `flip()` no se mostraría nada en nuestro juego.

## 4. Sprites



Como no podemos dibujar una persona con figuras geométricas tan fácilmente (¿o sí?) existen los **sprites**. Un sprite es una representación 2D de algo en pantalla: una imagen. `pygame` ofrece una clase `Sprite` que está diseñada para mostrar una o más representaciones gráficas de cualquier objeto del juego que quieras mostrar. Para usarla, debes heredar de ella.

## 4.1 Jugador

Cada elemento gráfico debe ser un Sprite, y el jugador no es una excepción. Agrega las siguientes líneas justo antes de haber inicializado `pygame` (`pygame.init()`):

```
# ahora el objeto Surface que habíamos dibujado en la pantalla principal es un atributo de Player # heredamos añadiendolo como parámetro a la clase Player
class Player(pygame.sprite.Sprite):
    def __init__(self): # nos permite invocar métodos o atributos de Sprite
        super(Player, self).__init__()
        self.surf = pygame.Surface((75, 25))
        self.surf.fill((255, 255, 255))
        self.rect = self.surf.get_rect()
```

No te preocupes si no entiendes completamente el código anterior, básicamente hicimos el constructor de la clase Player. Lo demás lo comprenderás más adelante, por ejemplo en Programación Avanzada.

Ahora, debemos definir e inicializar `.surf` para que sostenga la imagen a mostrar, que actualmente es una cajita negra. Además, debemos inicializar `.rect`, que utilizaremos para dibujar al jugador.

Para usar la nueva clase que definimos, necesitaremos un nuevo objeto y cambiar el código de dibujo también. Con los cambios debería verse así:

```
import pygame from pygame.locals import( K_UP, K_DOWN, K_LEFT,
K_RIGHT, K_ESCAPE, KEYDOWN, QUIT) # ahora el objeto Surface que
habíamos dibujado en la pantalla principal es un atributo de Player #
heredamos añadiendolo como parámetro a la clase Player class
Player(pygame.sprite.Sprite): def __init__(self): # nos permite
invocar métodos o atributos de Sprite super(Player, self).__init__()
self.surf = pygame.Surface((75, 25)) self.surf.fill((0, 0, 0))
self.rect = self.surf.get_rect() pygame.init() # definir el tamaño de
la pantalla SCREEN_WIDTH = 800 SCREEN_HEIGHT = 600 # crear el objeto
pantalla screen = pygame.display.set_mode((SCREEN_WIDTH,
SCREEN_HEIGHT)) # instanciamos al jugador player = Player() # variable
booleana para manejar el loop running = True # loop principal del
juego while running: # iteramos sobre cada evento en la cola for event
in pygame.event.get(): # se presiono una tecla? if event.type ==
KEYDOWN: # era la tecla de escape? -> entonces terminamos if event.key
== K_ESCAPE: running = False # fue un click al cierre de la ventana? -
> entonces terminamos elif event.type == QUIT: running = False #
rellena la screen con un color, en este caso blanco screen.fill((255,
255, 255)) # dibujamos al jugador en screen screen.blit(player.surf,
(SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2)) pygame.display.flip()
```

Si corres este código verás la cajita negra en el fondo blanco.

pygame window

— □ ×



Ahora, intenta cambiar `screen.blit(player.surf, (SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2))` por:

```
screen.blit(player.surf, player.rect)
```

Cuando le pasas un `Rect` a `blit()` ocupa las coordenadas de la esquina superior izquierda para dibujar en el `Surface`. Utilizaremos esto para poder hacer que el jugador se mueva.

## 4.2 Input de usuario

¡Ya sabemos dibujar! Ahora, empezaremos a controlar el juego con el teclado 🌨.

Dijimos que `pygame.event.get()` retorna la cola de eventos, los que vamos inspeccionando. Otra forma de realizar lo mismo es a través de `pygame.event.get_pressed()`, que retorna un diccionario que contiene todos los eventos `KEYDOWN` actuales en la cola.

Pondremos las siguientes líneas dentro de nuestro **loop** de juego, es decir, con una indentación:

```
pressed_keys = pygame.key.get_pressed()
```

Luego, **agregaremos el método `update` a la clase `Player` que acepte este diccionario**. Esto definirá el comportamiento del sprite en términos de las teclas que se presiona:

```
def update(self, pressed_keys): if pressed_keys[K_UP]:  
    self.rect.move_ip(0, -5) if pressed_keys[K_DOWN]: self.rect.move_ip(0,  
    5) if pressed_keys[K_LEFT]: self.rect.move_ip(-5, 0) if  
    pressed_keys[K_RIGHT]: self.rect.move_ip(5, 0)
```

→ `K_UP`, `K_DOWN` y las demás se refieren a las flechas del teclado. Si el valor del diccionario para esa llave es verdadero, entonces esa tecla está presionada y se moverá al `rect` del jugador en esa dirección utilizando `move_ip` (*move in place*). Finalmente, tenemos que llamar a `update()` cada `frame` o iteración para mover al jugador en respuesta a los eventos de tecla. Agregaremos esto justo después de la variable `pressed_keys` que agregamos antes. Nos quedaría entonces:

```
# loop principal del juego while running: # iteramos sobre cada evento
en la cola for event in pygame.event.get(): # se presiono una tecla?
if event.type == KEYDOWN: # era la tecla de escape? -> entonces
terminamos if event.key == K_ESCAPE: running = False # fue un click al
cierre de la ventana? -> entonces terminamos elif event.type == QUIT:
running = False # rellena la screen con un color, en este caso blanco
screen.fill((255, 255, 255)) # dibujamos al jugador en screen
screen.blit(player.surf, player.rect) pygame.display.flip() #
obtenemos todas las teclas presionadas actualmente pressed_keys =
pygame.key.get_pressed() # actualizamos el sprite del jugador basado
en las teclas presionadas player.update(pressed_keys)
```

¡Ahora puedes correr el programa y tu cajita negra se moverá!



👁👁 Notarás que tenemos un par de problemas con nuestro código:

- El rectángulo se mueve muy rápido se mantiene apretada la tecla.
- El rectángulo se sale de la pantalla.

Para resolver esto agregaremos condiciones dentro del método `update` de la clase `Player`:

```
def update(self, pressed_keys): if pressed_keys[K_UP]:
self.rect.move_ip(0, -5) if pressed_keys[K_DOWN]: self.rect.move_ip(0,
5) if pressed_keys[K_LEFT]: self.rect.move_ip(-5, 0) if
pressed_keys[K_RIGHT]: self.rect.move_ip(5, 0) # mantener al jugador
en pantalla if self.rect.left < 0: self.rect.left = 0 if
self.rect.right > SCREEN_WIDTH: self.rect.right = SCREEN_WIDTH if
self.rect.top < 0: self.rect.top = 0 if self.rect.bottom >
SCREEN_HEIGHT: self.rect.bottom = SCREEN_HEIGHT
```

¡Ahora agreguemos a los enemigos!

### 4.3 Enemigos

Como los enemigos aparecerán de manera aleatoria, utilizaremos la librería `random` por lo que la importaremos en las primeras líneas:

```
import random
```

Luego, crearemos una nueva clase llamada `Enemy` que herede de `Sprite`, por supuesto:

```
class Enemy(pygame.sprite.Sprite): def __init__(self): super(Enemy,
self).__init__() self.surf = pygame.Surface((20, 10))
self.surf.fill((0, 0, 0)) self.rect = self.surf.get_rect( center = (
random.randint(SCREEN_WIDTH + 20, SCREEN_WIDTH + 100),
random.randint(0, SCREEN_HEIGHT), ) ) self.speed = random.randint(5,
10) # el sprite tendra velocidad # cuando traspase el lado izq de la
pantalla lo eliminamos def update(self): self.rect.move_ip(-
self.speed, 0) if self.rect.right < 0: self.kill()
```

Te recomendamos añadirla después de Player :)

Si te fijas la estructura es muy similar, solo que ahora tenemos un método `kill()` que veremos en la siguiente sección.

## 5. Sprite Group

Otra clase muy útil que ofrece `pygame` es `Sprite Group`. Este es un objeto que agrupa sprites (*¡duh mr. obvious!*). Esta clase expone métodos que nos ayudan a detectar interacciones entre distintos sprites, ayudándonos en este caso con las colisiones.

Crearemos dos grupos, esto lo añadiremos después de que creamos el objeto `player`:

```
# instanciamos al jugador. Por ahora esto solo es una cajita negra
player = Player() # creamos grupos para tener a los enemigos y todos
los sprites # enemies es para detectar colisiones y actualizar
posiciones # all_sprites es para renderizar enemies =
pygame.sprite.Group() all_sprites = pygame.sprite.Group()
all_sprites.add(player) # variable booleana para manejar el loop
running = True # resto del código #. #.
```

Ahora, cuando llamamos `kill()` en la clase `Enemy`, el sprite es removido de cada grupo que pertenece, por lo que se eliminarán todas las referencias de dicho sprite también, lo que nos ayudará aprovechar la memoria.

Como ahora tenemos un grupo `all_sprites`, debemos cambiar la forma en que dibujamos los objetos. En vez de llamar a `blit()` solo en el jugador, iteramos sobre todo lo que se encuentra en `all_sprites`:

```
# rellena la screen con un color, en este caso blanco
screen.fill((255, 255, 255)) # dibujamos todos los sprites
for entity in all_sprites: screen.blit(entity.surf, entity.rect) # actualizamos
la interfaz pygame.display.flip()
```

Desde ahora, todo lo que se agregue a `all_sprites` será dibujado en cada frame, sea un enemigo o un jugador.

¡A crear los enemigos que debemos evitar!

## 6. Eventos personalizados



Una opción sería crear una cantidad específica de enemigos de una vez, pero el juego terminaría muuuuy rápido, por lo que vamos a definir intervalos en los que se irán creando y agregando a `all_sprites` y `enemies`.

Como nuestra cola de eventos está diseñada para ver todos los eventos aleatorios que ocurren en cada iteración, podemos manejar lo anterior de la siguiente forma:

```
# crear el objeto pantalla screen =  
pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT)) # evento para  
crear enemigos cada cierto intervalo de tiempo ADDENEMY =  
pygame.USEREVENT + 1 pygame.time.set_timer(ADDENEMY, 350) #  
instanciamos al jugador. Por ahora esto solo es una cajita negra  
player = Player() # resto del código #. #.
```

`ADDENEMY = pygame.USEREVENT + 1` es porque `pygame` define los eventos como enteros internamente, y cada uno de estos debe ser único, a excepción de `USEREVENT`, y como este es el único evento reservado, le sumamos 1 para asegurarnos de que sea único.

Luego, insertamos este evento a la cola de eventos de forma regular en intervalos, para lo que utilizaremos `time` cada 250 milisegundos:

```
# loop principal del juego while running: # iteramos sobre cada evento
en la cola for event in pygame.event.get(): # se presiona una tecla?
if event.type == KEYDOWN: # era la tecla de escape? -> entonces
terminamos if event.key == K_ESCAPE: running = False # fue un click al
cierre de la ventana? -> entonces terminamos elif event.type == QUIT:
running = False # es un evento que agrega enemigos? elif event.type ==
ADDENEMY: new_enemy = Enemy() enemies.add(new_enemy)
all_sprites.add(new_enemy) # rellena la screen con u color, en este
caso blanco screen.fill((255, 255, 255)) # dibujamos todos los sprites
for entity in all_sprites: screen.blit(entity.surf, entity.rect) #
actualizamos la interfaz pygame.display.flip() # obtenemos todas las
teclas presionadas actualmente pressed_keys = pygame.key.get_pressed()
# actualizamos el sprite del jugador basado en las teclas presionadas
player.update(pressed_keys) # obtenemos todas las teclas presionadas
actualmente pressed_keys = pygame.key.get_pressed() # actualizamos el
sprite del jugador basado en las teclas presionadas
player.update(pressed_keys) # actualizamos los enemigos
enemies.update()
```

¡Ahora si corres el programa verás pequeñas cajitas negras pasando por la pantalla!



## 7. Detectar colisiones

Detectaremos cuándo el jugador choca con los enemigos, en el fondo nos interesa ver cuándo dos sprites se superponen. Lo bueno es que `pygame` tiene métodos para ahorrarnos la matemática de esto, que nos retorna `True` si hay una colisión, y `False` en caso contrario:

```
# dibujamos todos los sprites for entity in all_sprites:
screen.blit(entity.surf, entity.rect) # vemos si algun enemigo a
chocado con el jugador if pygame.sprite.spritecollideany(player,
enemies): # si pasa, removemos al jugador y detenemos el loop del
juego player.kill() running = False # actualizamos la interfaz
pygame.display.flip() # resto del código #. #.
```



Ya tenemos todos los eventos básicos del juego, ¡Wooooohoooo! 🤖

## 8. Sprites personalizados

Solo tenemos cajitas negras, por lo que ahora necesitamos que el jugador sea efectivamente una persona y los enemigos, pues enemigos. Estos archivos los encontrarás en **la carpeta del curso**, y si gustas, puedes usar tus propios sprites, los cuales puedes encontrar en lugares como [Kenney](#), [OpenGameArt](#), o [ShoeBox](#) si es que quieres crear tus propios sprites.

Para agregarlos, necesitaremos hacer algunos cambios a los constructores de las clases `Player` y `Enemy`:

```
from pygame.locals import( K_UP, K_DOWN, K_LEFT, K_RIGHT, K_ESCAPE,
KEYDOWN, QUIT, RLEACCEL) # definir el tamaño de la pantalla
SCREEN_WIDTH = 800 SCREEN_HEIGHT = 600 # ahora el objeto Surface que
habíamos dibujado en la pantalla principal es un atributo de Player #
heredamos añadiendolo como parámetro a la clase Player class
Player(pygame.sprite.Sprite): def __init__(self): # nos permite
invocar métodos o atributos de Sprite super(Player, self).__init__()
self.surf = pygame.image.load("profe_Jorge.png").convert()
self.surf.set_colorkey((255, 255, 255), RLEACCEL) self.rect =
self.surf.get_rect() # resto del código #. #.
```

```
class Enemy(pygame.sprite.Sprite):
    def __init__(self):
        super().__init__()
        self.surf = pygame.image.load("papitas.png").convert()
        self.surf.set_colorkey((255, 255, 255), RLEACCEL) # la posicion inicial es generada aleatoriamente, al igual que la velocidad
        self.rect = self.surf.get_rect( center = ( random.randint(SCREEN_WIDTH + 20, SCREEN_WIDTH + 100), random.randint(0, SCREEN_HEIGHT), ) )
        self.speed = random.randint(5, 10) # el sprite tendra velocidad # cuando traspase el lado izq de la pantalla lo eliminamos
    def update(self):
        self.rect.move_ip(-self.speed, 0)
        if self.rect.right < 0:
            self.kill() # resto del código #.
```

### ¡Si corres el juego ahora, ya no habrán cajitas negras!

Además de lo anterior, podemos agregar imágenes de fondo. Para lograr esto, primero debemos añadir la imagen de fondo justo después de crear el objeto

`screen` :

```
# crear el objeto pantalla screen =
pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))
background_image = pygame.image.load("san_joaquin.jpg").convert()
```

Y dentro de nuestro ciclo, comentamos `screen.fill((255, 255, 255))` y lo reemplazamos por:

```
screen.blit(background_image, [0, 0])
```

Además, podemos pintar nubes, por ejemplo. Para esto:

1. Creamos la clase `Cloud`.
2. Agregamos una imagen de nube.
3. Agregamos el método `update()` que mueve la nube hacia la izquierda de la pantalla.
4. Creamos un evento que cree nuevas nubes cada cierto tiempo, al igual que lo hicimos con los enemigos.

5. Agregamos las nubes a un nuevo grupo `clouds`.
6. Actualizamos y dibujamos las nubes en el loop del juego.

```
class Cloud(pygame.sprite.Sprite):
    def __init__(self):
        super().__init__()
        self.surf = pygame.image.load("cloud.png").convert()
        self.surf.set_colorkey((255, 255, 255), RLEACCEL) # posicion inicial
        random self.rect = self.surf.get_rect( center = (
            random.randint(SCREEN_WIDTH + 20, SCREEN_WIDTH + 100),
            random.randint(0, SCREEN_HEIGHT), ) ) # igual que los enemigos
    def update(self):
        self.rect.move_ip(-5, 0)
        if self.rect.right < 0:
            self.kill()
```

Añadimos el evento personalizado:

```
# evento para crear enemigos cada cierto intervalo de tiempo
ADDENEMY = pygame.USEREVENT + 1
pygame.time.set_timer(ADDENEMY, 350)
ADD_CLOUD = pygame.USEREVENT + 2
pygame.time.set_timer(ADD_CLOUD, 1000)
```

Creamos el grupo:

```
# creamos grupos para tener a los enemigos y todos los sprites
# cloud para las nubes # enemies es para detectar colisiones y actualizar
posiciones # all_sprites es para renderizar
enemies = pygame.sprite.Group()
clouds = pygame.sprite.Group()
all_sprites = pygame.sprite.Group()
all_sprites.add(player)
```

Lo agregamos al loop:

```
# loop principal del juego while running: # iteramos sobre cada evento
en la cola for event in pygame.event.get(): # se presiona una tecla?
if event.type == KEYDOWN: # era la tecla de escape? -> entonces
terminamos if event.key == K_ESCAPE: running = False # fue un click al
cierre de la ventana? -> entonces terminamos elif event.type == QUIT:
running = False # es un evento que agrega enemigos? elif event.type ==
ADDENEMY: new_enemy = Enemy() enemies.add(new_enemy)
all_sprites.add(new_enemy) # es un evento que agrega nubes? elif
event.type == ADDCLOUD: new_cloud = Cloud() clouds.add(new_cloud)
all_sprites.add(new_cloud) # resto del código #. #.
```

Y nos preocupamos que se actualice:

```
# actualizamos los enemigos enemies.update() # actualizamos las nubes
clouds.update()
```

## 9. Velocidad del juego

Probablemente sientas que el juego se está moviendo muy rápido, para solucionarlo debemos manejar la tasa de frames, que va tan rápido como tu procesador lo permita. Para esto, utilizaremos `clock` del módulo `time`.

Definiremos `clock` antes de la variable `running`:

```
clock = pygame.time.Clock()
```

Llamamos a `tick` para avisar que se llegó al final del frame y actualizamos el display:

```
# actualizamos la interfaz pygame.display.flip() # tasa de 30 frames
por segundo clock.tick(30)
```

Y ¡voilà! Tenemos nuestro juego 😊.



