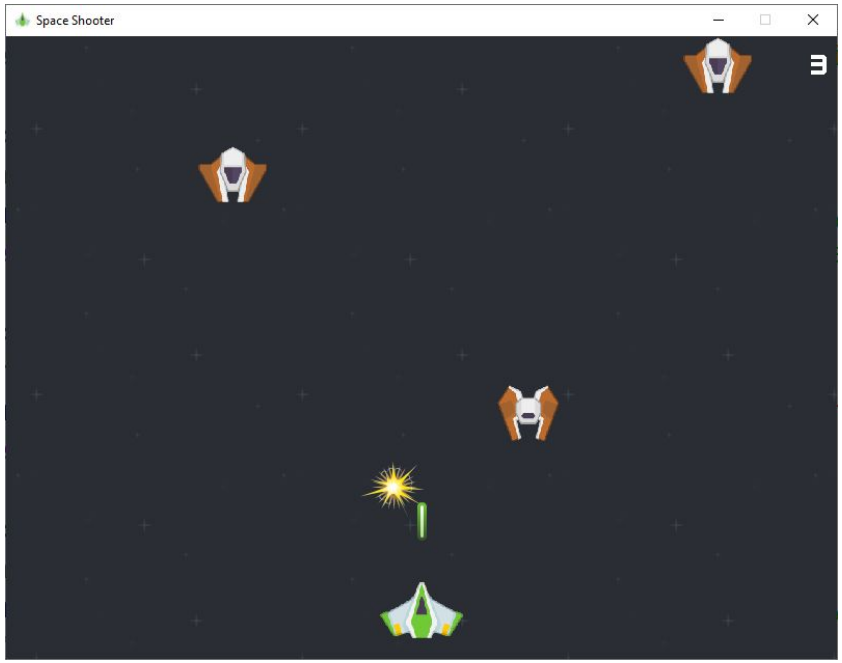
Programación de videojuegos 2D con Pygame

Jorge Guilló
<jguillo@gmail.com>

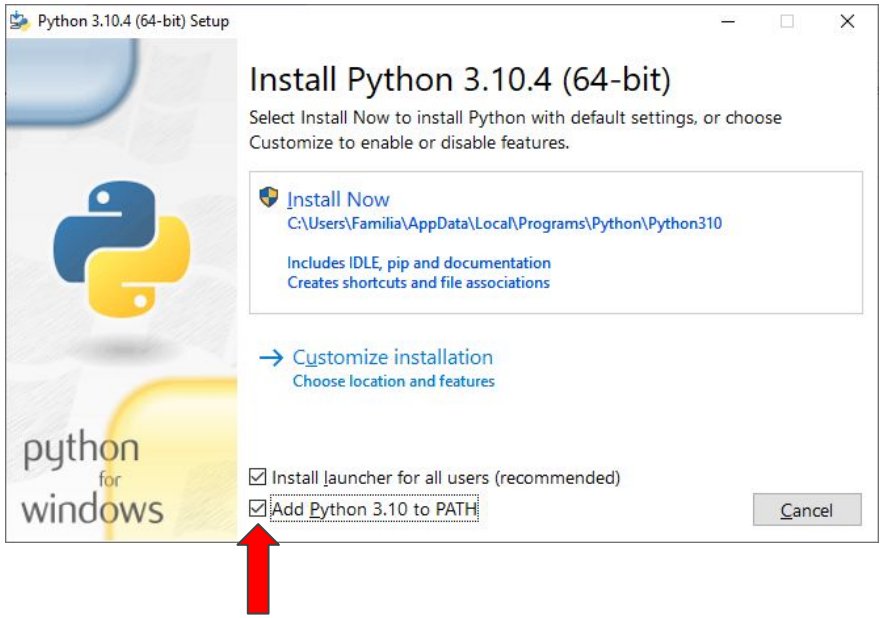
Introducción

- Vamos a crear un videojuego 2D de estilo "retro" utilizando el lenguaje de programación Python y su librería **pygame**
- Necesitaremos:
  - Python https://www.python.org/
  - Módulo pygame https://www.pygame.org/
  - Visual Studio Code (https://code.visualstudio.com/) o cualquier otro editor de texto



Instalación

- Instalar Python
  - Descargar el instalador de https://www.python.org/downloads/
  - **IMPORTANTE:** Marcar la opción de añadir al PATH
- Instalar el módulo pygame
  - Abrir una ventana de comandos (**cmd**)
  - Ejecutar **pip install pygame**
- Instalar Visual Studio Code
  - Descargar el instalador de https://code.visualstudio.com/
  - Instalar la extensión para Python



Descargar archivos del proyecto

En GitHub están disponibles los recursos (imágenes, sonidos, etc..) que vamos a usar en el curso así como todo el código que vamos a desarrollar, separado en varias fases paso a paso.

https://github.com/jguillo/curso-pygame

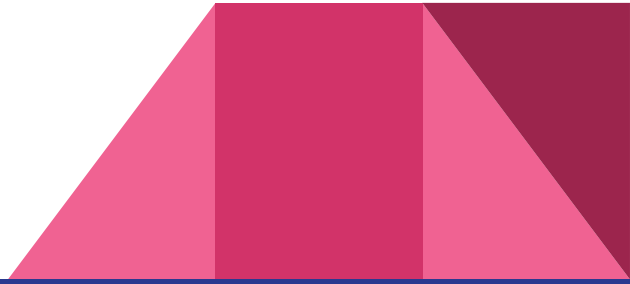
Crearemos un directorio en el equipo (por ejemplo, en el escritorio) donde guardaremos, como mínimo, la carpeta **assets**, que contiene los recursos del proyecto.

CRÉDITOS:

- **Gráficos:** Space Shooter Redux by Kenney Vleugels
  https://kenney.nl/assets/space-shooter-redux
- **Efectos de sonido:** Sci-Fi Sounds by Kenney Vleugels
  https://kenney.nl/assets/sci-fi-sounds
- **Música:** Space Ranger by Moire
  https://uppbeat.io/track/moire/space-ranger

Pygame

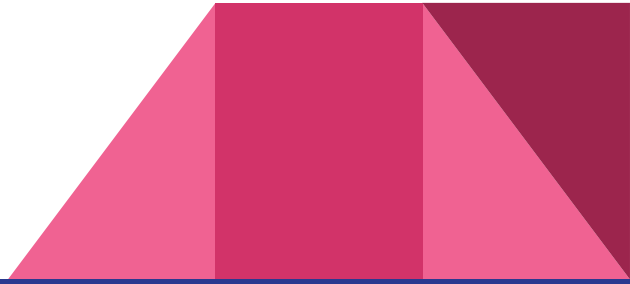
- Pygame es un conjunto de librerías diseñadas para programar videojuegos
- Construida sobre SDL
- Tiene módulos para cubrir todos los aspectos del videojuego:
  - Imágenes 2D (copiar, escalar, rotar,...)
  - Dibujar formas simples (círculos, rectángulos, líneas,...)
  - Uso de sprites (elementos 2D movibles)
  - Detectar colisiones
  - Entradas del teclado, ratón, joysticks y gamepads
  - Reproducir sonidos y música



Crear una ventana de pygame

Como primer paso, vamos a crear una ventana vacía, en la que posteriormente mostraremos nuestro videojuego.

- En primer lugar, debemos importar el módulo pygame:
  ```
  import pygame
  ```
- E inicializarlo con:
  ```
  pygame.init()
  ```
- Crearemos la ventana con la función **pygame.display.set_mode**, indicando el tamaño de la ventana:
  ```
  VENTANA = pygame.display.set_mode((800, 600))
  ```
- Podemos darle un título con **set_caption**:
  ```
  pygame.display.set_caption("Mi Juego")
  ```

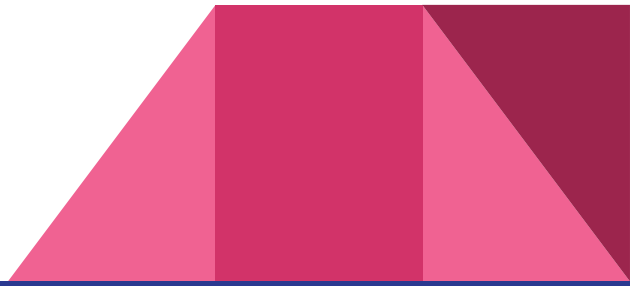


Bucle de eventos

Los videojuegos se basan principalmente en un bucle continuo en el que se realizan las operaciones del juego:

- Atender eventos (p.ej. teclas pulsadas, cierre de la ventana, etc..)
- Modificar los elementos del juego, según las acciones indicadas por el usuario y el propio comportamiento del juego.
- Detectar situaciones especiales (p.ej. muerte de los enemigos, etc.)
- Dibujar los elementos del juego en la pantalla

En este primer ejemplo sólo vamos a atender el evento de salida (QUIT) que se lanza cuando el usuario cierra la ventana del juego.



1-ventana.py

```
import pygame                                   # Importa el módulo pygame
pygame.init()                                   # Inicializa el módulo

WIDTH = 800                                     # Anchura de la ventana
HEIGHT = 600                                    # Altura de la ventana
WIN = pygame.display.set_mode((WIDTH, HEIGHT))# Crea la ventana
pygame.display.set_caption("Space Shooter")    # Establece el título

def main():                                     # Función principal
    fin = False                                 # Condición de salida del bucle
    while not fin:                              # Bucle del juego

        for event in pygame.event.get():  # Obtiene los eventos y los recorre
            if event.type == pygame.QUIT: # Evento QUIT
                fin = True                  # Salimos del bucle

    pygame.quit()                           # Cerramos pygame

# Si han ejecutado directamente este archivo, lanzamos main()
if __name__ == "__main__":
    main()
```

Surface

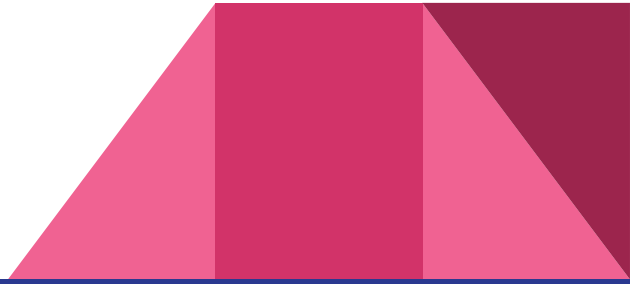
Las imágenes en pygame se representan con objetos **Surface**.

- Se pueden cargar de un fichero o crear nuevas con un tamaño determinado
- Se puede dibujar en ellas (líneas, rectángulos, imágenes, texto, etc..)
- El objeto que devuelve *pygame.display.set_mode* es también una Surface que representa la pantalla completa.
  - Lo que dibujemos en esta Surface se mostrará en la ventana.
- Para mostrar una imagen en la ventana:
  - Cargaremos la imagen como una nueva Surface con la función: **pygame.image.load(fichero)**
  - Dibujaremos la imagen en la ventana con la función **blit**
    - **ventana.blit(img, (x,y))**
    - Recibe el Surface con la imagen y la posición donde queremos mostrarla.
  - Para que los cambios en la Surface de pantalla se reflejen en la ventana real, actualizaremos la ventana con la función: **pygame.display.update()**

Sprite jugador

Vamos a crear un sprite para el jugador

- Crearemos la clase Jugador
- Cargaremos la imagen playerShip.png y la escalaremos a un tamaño adecuado
- Calcularemos la posición inicial del jugador centrado en la parte inferior de la pantalla.
- Añadiremos un método draw() para dibujar el sprite en la pantalla



Sistema de coordenadas

En pygame las coordenadas comienzan en la esquina superior izquierda.

- Las coordenadas se escriben como tuplas (x,y)
- La esquina superior izquierda tiene coordenadas (0,0).
- La coordenada horizontal **x** crece hacia la derecha.
- La coordenada vertical **y** crece hacia abajo.

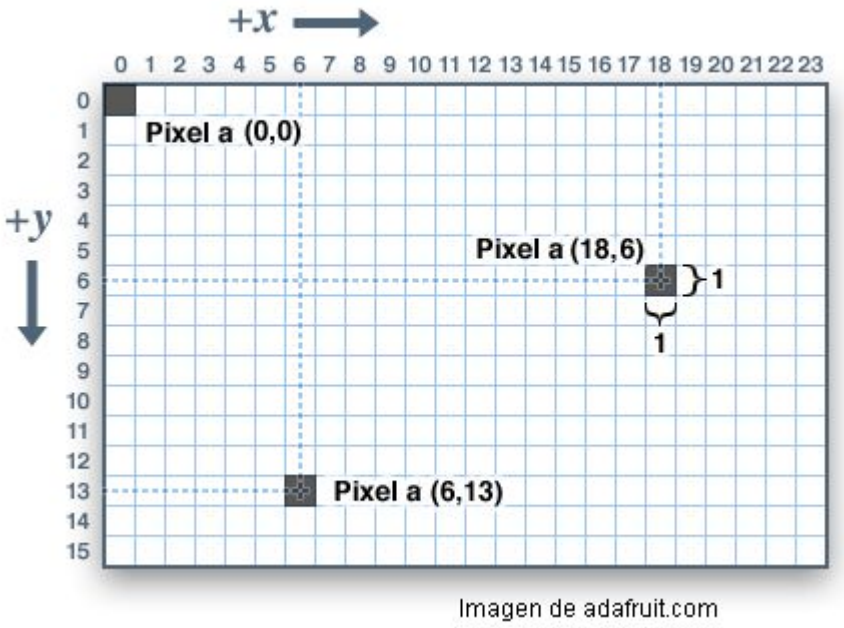


Imagen de adafruit.com

Dibujar imágenes

Vamos a empezar a dibujar imágenes en la pantalla

- Crearemos una función **cargarImagen(imagen)**
  - Recibe el nombre del archivo que queremos cargar
  - Construye la ruta completa (con el directorio assets)
  - Carga la imagen como un Surface y la devuelve
- Crearemos una función **crearFondo()**
  - Crea una imagen de fondo de estrellas construyendo un mosaico con la imagen stars.png
  - Va dibujando la imagen tantas veces como sea necesario, avanzando primero en horizontal y luego en vertical
- Crearemos una función **dibuja()**
  - Dibuja la pantalla completa
  - De momento dibuja el fondo y actualiza la pantalla para que se vean los cambios.
  - Le iremos añadiendo elementos según los vayamos creando
- Al inicio, cargaremos una imagen como icono de la ventana
- En el bucle de juego, llamaremos a la función **dibuja()**

2-fondo.py

```
import os                                   # Importa el módulo os

# Función para cargar imágenes
def cargarImagen(imagen):
    ruta = os.path.join("assets", imagen)   # Construye la ruta completa
    return pygame.image.load(ruta)          # Carga la imagen y la devuelve

# Crea una imagen de fondo con un mosaico de estrellas
def crearFondo():
    img = pygame.surface.Surface((WIDTH, HEIGHT)) # Crea la imagen de tamaño completo
    pieza = cargarImagen("stars.png")             # Carga la imagen de mosaico
    y = 0                                   # Inicia recorrido vertical
    while (y < HEIGHT):                     # Recorre en vertical
        x = 0                               # Inicia recorrido horizontal
        while (x < WIDTH):                  # Recorre en horizontal
            img.blit(pieza, (x,y))          # Pinta el fondo en la posición (x,y)
            x += pieza.get_width()          # Avanza el ancho del fondo en horizontal
        y += pieza.get_height()             # Avanza el alto del fondo en vertical
    return pieza
```

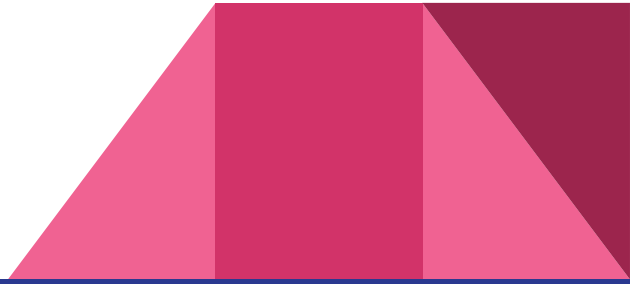
2-fondo.py

```
[...]
ICONO = cargarImagen("icon.png")    # Carga la imagen de icono
pygame.display.set_icon(ICONO)      # Establece el icono de la ventana

FONDO = crearFondo()   # Crea la imagen de fondo
[...]

# Función que dibuja la pantalla completa en cada iteración del juego
def dibuja():
   WIN.blit(FONDO, (0,0))   # Dibuja el fondo
   pygame.display.update()  # Actualiza la pantalla

[...]

def main():                              # Función principal
   jugando = True                       # Condición del bucle
   while jugando:                       # Bucle del juego
      [...]
      dibuja()                          # dibuja la pantalla
```

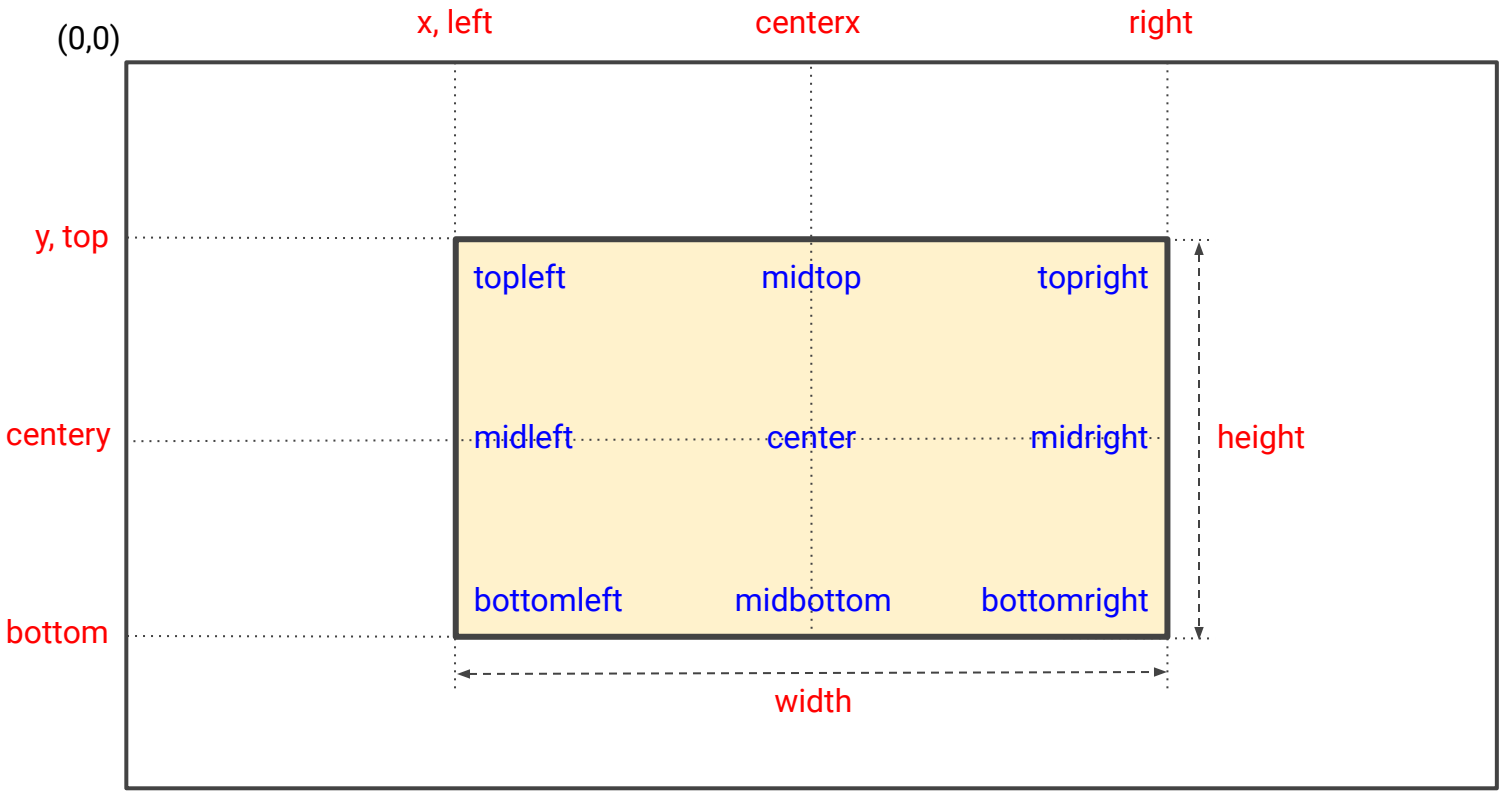
Rect

En pygame usaremos objetos **Rect** para representar rectángulos en la pantalla.

- Nos permite especificar la posición y el tamaño de los elementos de la pantalla.
- Se construyen con las coordenadas de su esquina superior izquierda y el tamaño (anchura, altura)
- Se pueden sustituir por tuplas (x, y, ancho, alto)
- Cuentan con propiedades adicionales para distintos puntos y coordenadas calculadas
  - Se pueden modificar y el rectángulo se mueve acorde con la modificación
  - Mantiene el tamaño salvo que se modifique width, height o size.



Propiedades de Rect



- En rojo, valores enteros
- En azul, tuplas (x,y)
- También existe la tupla size (width, height)

Sprites

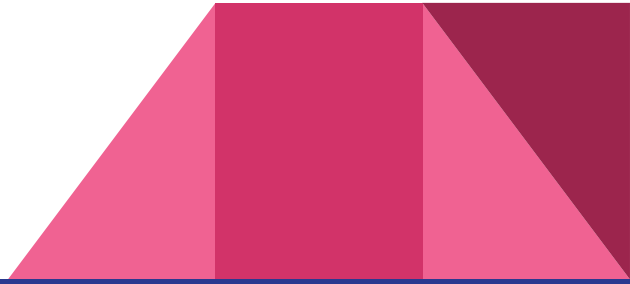
Para representar elementos del juego, usaremos clases **Sprite**

- image: Surface con la imagen que representa el elemento
- rect: Rect con la posición y tamaño del elemento
- update(): Método que controla el comportamiento del elemento

Para implementar un elemento del juego, crearemos una clase que derive de Sprite

- Cargaremos la imagen e inicializaremos la posición del elemento
- Implementaremos el método update() para actualizar la posición del elemento según las reglas del juego.

Podemos crear distintas clases para distintos elementos (jugadores, enemigos, balas, elementos estáticos,...)



3-jugador.py

```
SHIP_WIDTH = 80       # Anchura de la nave
SHIP_HEIGHT = 54      # Altura de la nave
[...]
# Sprite Jugador
class Jugador(pygame.sprite.Sprite): # Deriva de Sprite
   # CONSTRUCTOR
   def __init__(self):
       super().__init__()                                # Llama al constructor de Sprite
       ship = cargarImagen("playerShip.png")             # Carga la imagen
       self.image = pygame.transform.scale(ship, (SHIP_WIDTH,SHIP_HEIGHT))  # Reduce el tamaño
       self.rect = self.image.get_bounding_rect()        # Crea el Rect con el tamaño de la imagen
       self.rect.midbottom = (WIDTH // 2, HEIGHT - 20) # Ajusta la posición

   # Dibuja el sprite en pantalla
   def draw(self):
       WIN.blit(self.image, self.rect)     # Dibuja la imagen en la ventana

nave = Jugador()

# Función que dibuja la pantalla completa en cada iteración del juego
def dibuja():
   WIN.blit(FONDO, (0,0))      # Dibuja el fondo
   nave.draw()                 # Dibuja la nave
   pygame.display.update()     # Actualiza la pantalla
```

Entrada de teclado

Vamos a mover nuestro jugador usando las teclas.

Para atender las pulsaciones del teclado tenemos dos opciones:

- **Eventos**: Entre los eventos que se procesan en pygame, tenemos KEYDOWN y KEYUP, para cuando se pulsa y se suelta una tecla.
  - Útil para acciones inmediatas (p.ej. disparos)
  - En **event.key** tenemos la tecla que se ha pulsado/soltado
  - En **event.mod** nos indica si se estaba pulsando además alguna tecla modificadora (Ctrl, Alt,...)
- **Polling**: En cada iteración del juego, se comprueba qué teclas están pulsadas
  - Útil para acciones sostenidas en el tiempo (p.ej. movimiento)
  - Leemos las teclas pulsadas con **pygame.key.get_pressed()**
  - Nos devuelve un array de booleanos indicándonos si cada tecla está pulsada o no
- Los códigos de las teclas los tenemos en constantes **pygame.K_XXX**
  - **K_a, K_b, … K_0, K_9, … K_LEFT, K_RIGHT, … K_SPACE, K_LCTRL, …**
  - Puedes consultarlas en la documentación: https://www.pygame.org/docs/ref/key.html

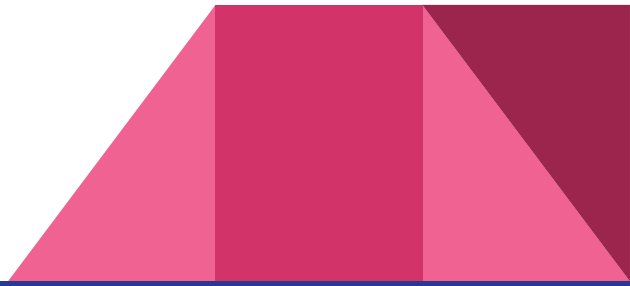
Control de tiempo

Cuando empezamos a mover elementos en el juego, debemos controlar el tiempo en el bucle de juego.

- Si no, el juego irá más deprisa o más despacio según la potencia del ordenador.

Lo que haremos será forzar un número de iteraciones (frames) por segundo

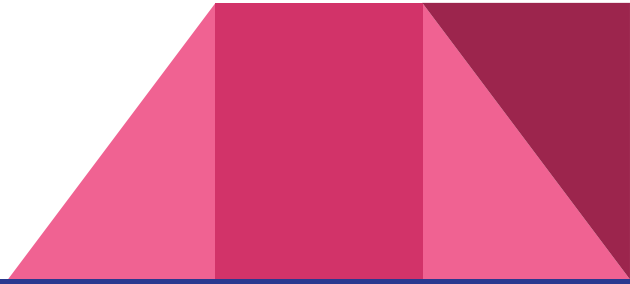
- Crearemos un objeto **pygame.time.Clock**
- Una vez en cada iteración del bucle, llamaremos al método **tick(frames)** indicando los frames por segundo (FPS) que queremos que tenga el juego
- Se suele usar un valor entre 20 y 100, normalmente 60



Movimiento del jugador

Vamos a mover el sprite del jugador con las teclas de las flechas

- Usaremos polling
- En el método update() del sprite Jugador:
  - Leeremos las teclas que hay pulsadas
  - Modificaremos la posición horizontal y vertical del sprite según las teclas pulsadas
  - Controlaremos que no nos salgamos de la pantalla.
- En el bucle principal:
  - Llamaremos al método update() del sprite
  - Forzaremos la velocidad a 60 FPS

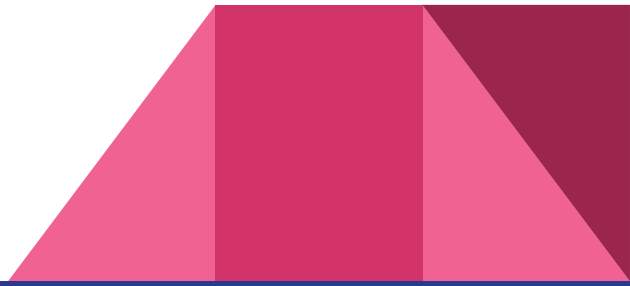


4-movimiento.py

```
VEL_JUGADOR = 10      # Velocidad de la nave
[...]
class Jugador(pygame.sprite.Sprite): # Deriva de Sprite
   [...]
   # Movimiento del jugador
   def update(self):
        keys = pygame.key.get_pressed() # Lee las teclas pulsadas
        if (keys[pygame.K_UP]):         # Arriba
            self.rect.y -= VEL_JUGADOR
        if (keys[pygame.K_DOWN]):       # Abajo
            self.rect.y += VEL_JUGADOR
        if (keys[pygame.K_LEFT]):       # Izquierda
            self.rect.x -= VEL_JUGADOR
        if (keys[pygame.K_RIGHT]):      # Derecha
            self.rect.x += VEL_JUGADOR

        if self.rect.left < 0:          # Sobrepasa el borde izquierdo
            self.rect.left = 0
        if self.rect.right > WIDTH:     # Sobrepasa el borde derecho
            self.rect.right = WIDTH
        if self.rect.top < 0:           # Sobrepasa el borde superior
            self.rect.top = 0
        if self.rect.bottom > HEIGHT:   # Sobrepasa el borde inferior
            self.rect.bottom = HEIGHT
```

4-movimiento.py

```
def main():                                 # Función principal
    reloj = pygame.time.Clock()             # Reloj para FPS
    jugando = True                          # Condición del bucle
    while jugando:                          # Bucle del juego
        for event in pygame.event.get():    # Obtiene los eventos y los recorre
            if event.type == pygame.QUIT:   # Evento QUIT
                jugando = False             # Salimos del bucle
        nave.update()                       # mueve la nave
        dibuja()                            # dibuja la pantalla
        reloj.tick(60)                      # Fuerza FPS

    pygame.quit()                           # Cerramos pygame
```



Groups

Las clases Group permiten agrupar Sprites para trabajar con ellos más fácilmente.

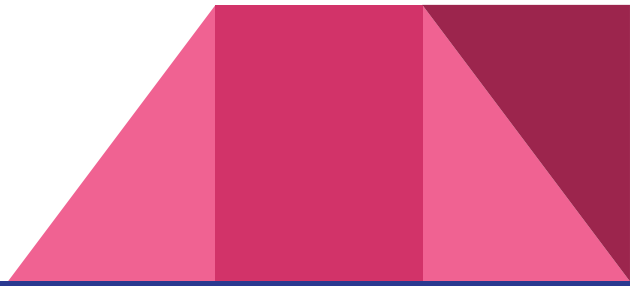
- Un Sprite puede pertenecer a muchos grupos.
- Tendremos siempre un grupo principal con todos los sprites del juego.
  - Así podemos mandarlos a dibujar todos a la vez.
- Además, tendremos distintos grupos para cada tipo de sprite (enemigos, balas, etc.)

La clase Group cuenta con los métodos:

- **add(sprite)**: Añade el sprite al grupo
- **remove(sprite)**: Quita el sprite del grupo
- **sprites()**: Devuelve la lista de sprites que pertenecen al grupo
- **empty()**: Vacía el grupo quitando todos los sprites
- **update()**: Llama al método update() de todos los sprites del grupo.
- **clear(ventana, fondo)**: Borra de la ventana los sprites del grupo, dibujando una imagen de fondo sobre ellos.
- **draw(ventana)**: Dibuja todos los sprites en la ventana

A su vez, la clase Sprite cuenta con los métodos:

- **add(grupo)**: Añade el sprite al grupo
- **remove(grupo)**: Quita el sprite del grupo
- **groups()**: Devuelve una lista con los grupos a los que pertenece el sprite
- **kill()**: Quita el sprite de todos los grupos a los que pertenece



Enemigos

Vamos a crear un nuevo tipo de sprite para representar a los enemigos del juego.

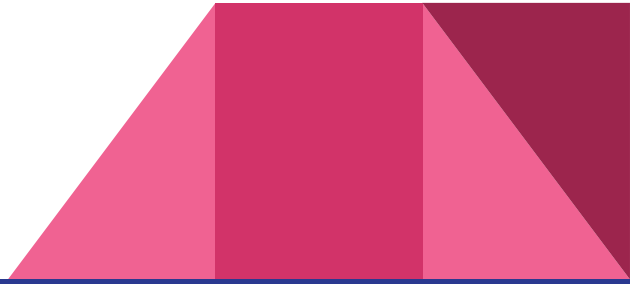
Los enemigos aparecen por la parte superior de la pantalla y se mueven siempre hacia abajo, aunque pueden ir directos hacia abajo o en diagonal.

- En el constructor:
  - Elegiremos un modelo de nave aleatoriamente
  - Lo escalaremos al tamaño aproximado de la nave del jugador
  - Posición vertical por encima del borde de la ventana
  - Posición horizontal aleatoria entre 0 y el ancho de la ventana
  - Velocidad horizontal aleatoria entre izquierda (negativa), derecha (positiva) o 0 (solo abajo)
- En el método update controlaremos el movimiento de la nave enemiga.
  - Irán siempre hacia abajo y si chocan con un borde lateral cambian la dirección horizontal.
- Crearemos un grupo para gestionar los enemigos
- Crearemos otro grupo para tener todos los sprites (enemigos y jugador).
- En el constructor de cada sprite, lo añadiremos a los grupos que corresponda
- En el método dibuja()
  - Llamaremos al método draw() del grupo general
  - Ya no necesitaremos implementar el método draw() en los sprites.
- En el bucle:
  - Crearemos enemigos de manera aleatoria
    - Estableceremos un máximo de enemigos en pantalla, una probabilidad de que aparezcan y un tiempo de espera mínimo entre enemigos sucesivos.
  - En lugar de llamar a update de cada sprite, usaremos el método update del grupo general.

Elementos aleatorios

Para generar sucesos aleatorios usaremos el módulo **random**

- **random.random()**: Devuelve un decimal aleatorio entre 0 y 1 (excluyendo el 1)
- **random.randint(min,max)**: Genera un entero aleatorio entre min y max (ambos incluidos)
- **random.uniform(a,b)**: Genera un número decimal aleatorio entre min y max (ambos incluidos)
- **random.choice(lista)**: Devuelve uno de los elementos de la lista aleatoriamente



5-enemigos.py

```
import random           # Importa el módulo random

VEL_ENEMIGO = 5         # Velocidad de los enemigos
MAX_ENEMIGOS = 6        # Número máximo de enemigos simultáneos
PROB_ENEMIGO = 40       # Probabilidad de que aparezca un nuevo enemigo
ESPERA_ENEMIGOS = 20    # Espera mínima entre enemigos

IMAGENES_ENEMIGOS = [  # Modelos de nave enemiga
    cargarImagen("enemy1.png"), cargarImagen("enemy2.png"), cargarImagen("enemy3.png")
]

# Grupos
enemigos = pygame.sprite.Group()
todo = pygame.sprite.Group()

# Sprite Jugador
class Jugador(pygame.sprite.Sprite):
    # CONSTRUCTOR
    def __init__(self):
        [...]
        self.add(todo)  # Añade el sprite al grupo general
    # Quitamos el método draw()
```

5-enemigos.py

```
# Sprite enemigo
class Enemigo(pygame.sprite.Sprite): # Clase que deriva de Sprite
    # CONSTRUCTOR
    def __init__(self):
        super().__init__()
        # Elige un tipo de nave
        ship = random.choice(IMAGENES_ENEMIGOS)
        # Calcula el tamaño del sprite ajustándolo al tamaño del jugador
        self.rect = ship.get_bounding_rect().fit((0,0,SHIP_WIDTH,SHIP_HEIGHT))
        # Reduce el tamaño de la imagen
        self.image = pygame.transform.scale(ship, self.rect.size)
        # Posición horizontal aleatoria en todo el ancho de la ventana
        self.rect.x = random.uniform(0, WIDTH - self.rect.width)
        # Posición vertical por encima del borde superior
        self.rect.y = -self.rect.height
        # Velocidad horizontal aleatoria (izquierda, abajo, derecha)
        self.velx = random.choice([-VEL_ENEMIGO, 0 , VEL_ENEMIGO])
        # Añade el sprite a sus grupos
        self.add(enemigos, todo)
```

5-enemigos.py

```
# class Enemigo
    # Movimiento del enemigo
    def update(self):
        self.rect.x += self.velx    # Movimiento horizontal
        self.rect.y += VEL_ENEMIGO  # Movimiento vertical
        # Ajustes de posición
        if self.rect.left < 0:      # Sobrepasa el borde izquierdo
            self.rect.left = 0      # Se queda en el borde
            self.velx = -self.velx  # y cambia de dirección

        if self.rect.right > WIDTH: # Sobrepasa el borde derecho
            self.rect.right = WIDTH # Se queda en el borde
            self.velx = -self.velx  # y cambia de dirección

        if self.rect.y > HEIGHT:   # Sobrepasa el borde inferior
            self.kill()            # Eliminamos el sprite

def dibuja():
    WIN.blit(FONDO, (0,0))      # Dibuja el fondo
    todo.draw(WIN)              # Dibuja los sprites
    pygame.display.update()     # Actualiza la pantalla
```

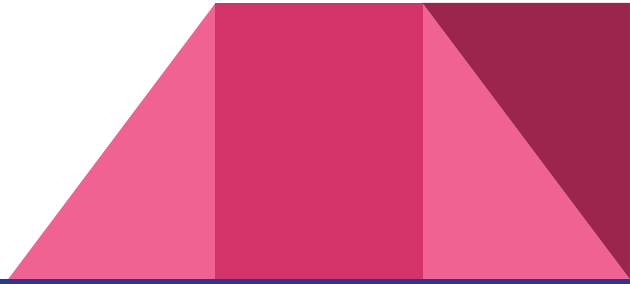
5-enemigos.py

```
def main():                             # Función principal
    esperaEnemigo = 0                   # Tiempo de espera entre enemigos
    reloj = pygame.time.Clock()         # Reloj para FPS
    jugando = True                      # Condición del bucle
    while jugando:                      # Bucle del juego

        for event in pygame.event.get():    # Obtiene los eventos y los recorre
            if event.type == pygame.QUIT:   # Evento QUIT
                jugando = False             # Salimos del bucle

        # Generar enemigos si no estamos en espera y no hemos alcanzado el máximo
        if esperaEnemigo == 0 and len(enemigos) < MAX_ENEMIGOS:
            # Aplica la probabilidad
            if (random.uniform(0,100) < PROB_ENEMIGO):
                Enemigo()                       # Crea el sprite enemigo
                esperaEnemigo = ESPERA_ENEMIGOS   # Inicia el tiempo de espera

        elif esperaEnemigo > 0:  # Estamos en espera
            esperaEnemigo -= 1   # Descuenta del tiempo de espera

        todo.update()                   # actualiza los sprites
        dibuja()                        # dibuja la pantalla
        reloj.tick(60)                  # Fuerza FPS

    pygame.quit()                       # Cerramos pygame
```

Disparos

El siguiente paso es poder disparar a los enemigos

- Crearemos un nuevo sprite BalaJugador
  - Lo iniciaremos centrado sobre la nave (recibe la nave en el constructor)
  - Crearemos un nuevo grupo para las balas del jugador
  - Las balas se mueven hacia arriba, hasta que se salen de la pantalla
- En el main, atenderemos el evento KEYDOWN
  - Si el usuario pulsa la tecla de espacio, crearemos una bala

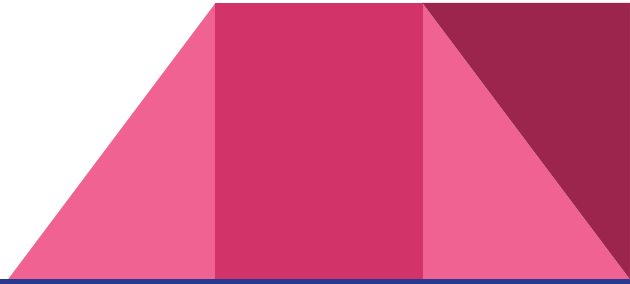


6-balas.py

```
VEL_BALAS = 20        # Velocidad de las balas
balasJugador = pygame.sprite.Group()

# Sprite bala del jugador
class BalaJugador(pygame.sprite.Sprite): # Clase que deriva de Sprite
    # CONSTRUCTOR
    def __init__(self, nave): # Recibe la nave como parámetro
        super().__init__()
        # Carga la imagen y su rectángulo
        self.image = cargarImagen("laserGreen.png")
        self.rect = self.image.get_bounding_rect()
        # Posición centrada encima del jugador
        self.rect.midbottom = nave.rect.midtop
        # Añade el sprite a sus grupos
        self.add(balasJugador, todo)

    # Movimiento de la bala
    def update(self):
        self.rect.y -= VEL_BALAS  # Se mueve hacia arriba
        # Ajustes
        if self.rect.bottom < 0:  # Sobrepasa el borde superior
            self.kill
```

6-balas.py

```
# Función main()
[...]
        for event in pygame.event.get():    # Obtiene los eventos y los recorre

            if event.type == pygame.QUIT:   # Evento QUIT
                jugando = False             # Salimos del bucle

            if event.type == pygame.KEYDOWN:    # Tecla pulsada
                if event.key == pygame.K_SPACE: # Espacio
                    BalaJugador(nave)           # Crea la bala
                if event.key == pygame.K_ESCAPE:# ESC
                    jugando = False             # Salimos del bucle
[...]
```



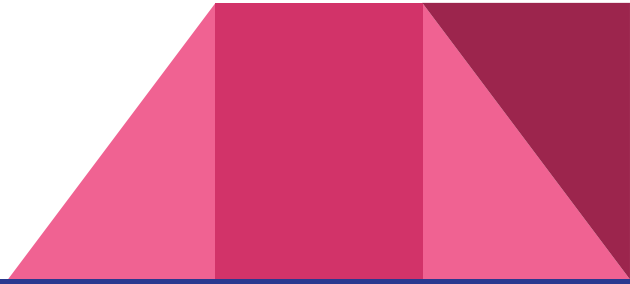
Colisiones

Para que el juego funcione, debemos detectar las colisiones entre los objetos del juego.

- Enemigos - Jugador
- Enemigos - Balas del jugador

En el módulo sprite tenemos distintos métodos collide para detectar cuando los rectángulos de dos sprites se solapan:

- pygame.sprite.**collide_rect**(sprite1, sprite2)
  - Devuelve True si los sprites se solapan
- pygame.sprite.**spritecollide**(sprite, grupo,dokill)
  - Devuelve la lista de sprites del grupo que colisionan con el sprite
  - Si dokill=1, los sprites se eliminan del grupo
- pygame.sprite.**groupcollide**(g1, g2, kill1, kill2)
  - Devuelve un diccionario, indicando para cada sprite del grupo g1, la lista de sprites de g2 que colisionan con él.
  - Si kill1=1, los sprites de g1 con colisiones se borran
  - Si kill2=1, los sprites de g2 con colisiones se borran



Colisiones

- Además haremos efectos de explosiones
  - Crearemos un sprite Explosion
  - Recibirá un sprite como parámetro y se posicionará centrado sobre el sprite
  - En el update simplemente se descuenta un contador de 10 frames
  - Al finalizar, se autodestruye el sprite
- En la función main, detectaremos colisiones:
  - Entre las balas y los enemigos
    - Crearemos una explosión en la posición del jugador y otra en la del enemigo
    - Eliminaremos los sprites del enemigo y de la bala
  - Entre los enemigos y el jugador
    - Crearemos una explosión en la posición del jugador y otra en la del enemigo
    - Eliminaremos los sprites del enemigo y del jugador
    - Devolveremos si ha muerto el jugador
  - Si el jugador ha muerto, se reinicia el juego.

7-colisiones.py

```
# Sprite explosión
class Explosion(pygame.sprite.Sprite): # Clase que deriva de Sprite
    # CONSTRUCTOR
    def __init__(self, sprite): # Recibe un sprite como parámetro
        super().__init__()
        # Carga la imagen y su rectángulo
        self.image = cargarImagen("explosion.png")
        self.rect = self.image.get_bounding_rect()
        # Posición centrada encima del jugador
        self.rect.center = sprite.rect.center
        self.paso = 10
        # Añade el sprite al grupo general
        self.add(todo)

    # Pasos de la explosión
    def update(self):
        self.paso -= 1
        if self.paso == 0:
            self.kill()
```

7-colisiones.py

```
# Función que detecta colisiones
def detectarColisiones():
    # Busca colisiones entre enemigos y balas,
    # eliminando tanto el enemigo como la bala
    enemigos_tocados = pygame.sprite.groupcollide(enemigos, balasJugador, True, True)
    for enemigo, balas in enemigos_tocados.items():
        # Crea una explosión en la posición del enemigo
        Explosion(enemigo)

    muerte = False # En principio el jugador no ha muerto
    # Busca colisiones entre enemigos y jugador, eliminando el enemigo
    enemigos_chocan = pygame.sprite.spritecollide(nave, enemigos, True)
    for enemigo in enemigos_chocan:
        # Crea explosiones en el enemigo y en la nave
        Explosion(enemigo)
        Explosion(nave)
        nave.kill()   # Borra la nave
        muerte = True # El jugador ha muerto
    return muerte
```

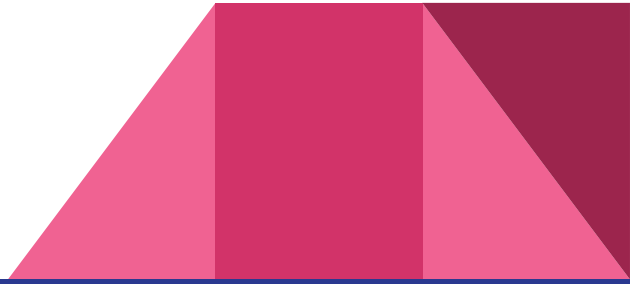
7-colisiones.py

```
# Función que reinicia el juego
def reinicio():
    global nave # Vamos a modificar una variable global
    # Para el juego dos segundos
    pygame.time.delay(2000)
    # Vacía los grupos y elimina todos los sprites
    todo.empty()
    enemigos.empty()
    balasJugador.empty()
    # Reinicia el sprite del jugador
    nave = Jugador()

# En el main()
        [...]
        muerte = detectarColisiones()   # detectar colisiones

        todo.update()                   # actualiza los sprites
        dibuja()                        # dibuja la pantalla

        if muerte:          # Si el jugador ha muerto
            reinicio()      # Reinciia el juego
        [...]
```

Mostrar textos

Para mostrar textos en la pantalla se usan los objetos **Font**

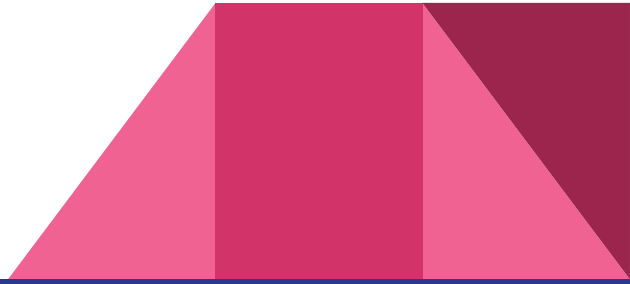
- Se crea un objeto Font con la ruta de una fuente TTF y un tamaño (altura en pixels)
- Para crear un texto se usa el método **font.render(texto, suavizado, color)**
  - Recibe el texto que se quiere mostrar (no acepta saltos de línea)
  - Un booleano indicando si se quieren bordes suaves (antialiasing) en las letras
  - El color del texto, como una tupla (rojo,verde,azul)
- Devuelve una Surface que se puede añadir a la pantalla con blit



Marcador y Game Over

Vamos a añadir un marcador con la puntuación del juego, una pantalla de título y un indicador de Game Over cuando el jugador muere.

- El marcador lo implementaremos como un sprite
  - En lugar de cargar una imagen, la crearemos en el método update con la puntuación
  - El propio sprite llevará la cuenta de los puntos, con un método aumenta()
- El título y el indicador de Game Over los mostraremos con funciones, que llamaremos cuando sea necesario
  - Además pausaremos el juego usando `pygame.time.delay(milisegundos)`



8-textos.py

```
pygame.init()  # Iniciamos pygame para poder crear fuentes antes de abrir la ventana

# Función para crear fuentes de un tamaño determinado
def crearFuente(size):
    # Construye la ruta completa a la fuente TTF
    ruta = os.path.join("assets", "kenvector_future_thin.ttf")
    # Crea la fuente y la devuelve
    return pygame.font.Font(ruta, size)

# Fuentes
FONT_MARCADOR = crearFuente(30)
FONT_GAMEOVER = crearFuente(100)
FONT_TITULO = crearFuente(80)

# Colores de texto
COLOR_MARCADOR = (255,255,255)  # Blanco
COLOR_GAMEOVER = (255, 0, 0)    # Rojo
COLOR_TITULO   = (0, 255, 0)    # Verde
```

8-textos.py

```
# Sprite marcador
class Marcador(pygame.sprite.Sprite):
    # CONSTRUCTOR
    def __init__(self):
        super().__init__()
        self.puntos = 0  # Inicia la puntuación
        self.add(todo)   # Se añade al grupo general

    # Actualiza la imagen del marcador
    def update(self):
        # Crea el texto del marcador
        self.image = FONT_MARCADOR.render(str(self.puntos), False, COLOR_MARCADOR)
        self.rect = self.image.get_bounding_rect()
        # Lo coloca en la esquina superior derecha
        self.rect.topright = (WIDTH-10, 10)

    # Aumenta la puntuación
    def aumenta(self):
        self.puntos += 1

marcador = Marcador() # Crea el sprite marcador
```

8-textos.py

```
# Muestra el mensaje de GAME OVER
def gameover():
    # Crea el texto "GAME OVER"
    gameover = FONT_GAMEOVER.render("GAME OVER", False, COLOR_GAMEOVER)
    rect = gameover.get_bounding_rect()
    # Centrado en la pantalla
    rect.center = (WIDTH // 2, HEIGHT //2)
    # Lo dibuja en la pantalla
    WIN.blit(gameover, rect)
    # Actualiza la pantalla
    pygame.display.update()

# Función que reinicia el juego
def reinicio():
    global nave, marcador
    gameover()
    [...]
    # Reinicia los sprites de jugador y marcador
    nave = Jugador()
    marcador = Marcador()
```

8-textos.py

```
# Dibuja la pantalla de título
def mostrarTitulo():
    WIN.blit(FONDO, (0,0))          # Dibuja el fondo
    WIN.blit(nave.image, nave.rect) # Dibuja la nave
    # Crea el título
    titulo = FONT_TITULO.render("SPACE SHOOTER", False, COLOR_TITULO)
    rect = titulo.get_bounding_rect()
    # Centrado en la pantalla
    rect.center = (WIDTH // 2, HEIGHT //2)
    # Lo dibuja en la pantalla
    WIN.blit(titulo, rect)
    # Actualiza la pantalla
    pygame.display.update()
    # Espera 3 segundos
    pygame.time.delay(3000)

# Función principal del juego
def main():
    mostrarTitulo()                     # Muestra el título inicial
```

Sonido

Los sonidos en pygame se dividen en dos categorías:

- Efectos de sonido: sonidos de corta duración
  - Se crean como objetos **pygame.mixer.Sound**
  - Se pueden reproducir en cualquier momento
  - Hay 8 canales para que suenen múltiples efectos simultáneamente

```
sonido = pygame.mixer.Sound("sonido.wav")
sonido.play()
```

- Música: sonido de fondo de larga duración
  - Se gestionan con el módulo **pygame.mixer.music**
  - Sólo puede haber una música sonando

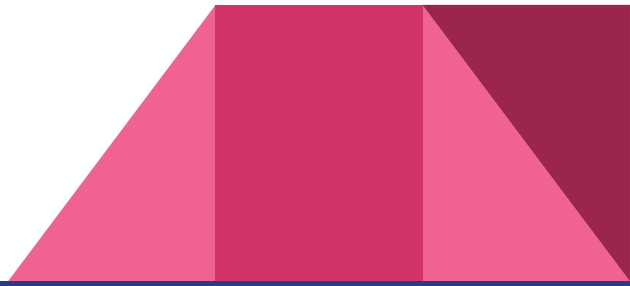
```
pygame.mixer.music.load("musica.mp3")
pygame.mixer.music.play(loops=-1) # Se repite indefinidamente
```

Se puede trabajar con archivos OGG, WAV o MP3

Efectos y música

Vamos a añadir efectos de sonido (disparos y explosiones) y música de fondo

- Crearemos los objetos Sound al inicio y los reproduciremos cuando sea necesario.
- Al iniciar el juego, cargaremos la música y la reproduciremos.



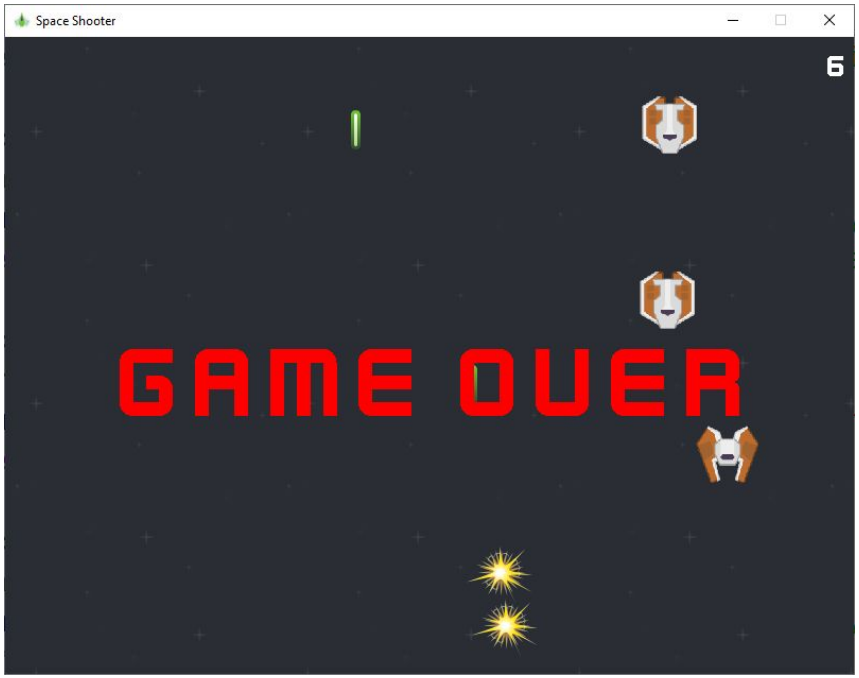
9-sonido.py

```
# Función para cargar sonidos
def cargarSonido(sonido):
    ruta = os.path.join("assets", sonido)   # Construye la ruta completa
    return pygame.mixer.Sound(ruta)         # Carga el sonido y lo devuelve

# Función para reproducir la música
def iniciarMusica():
    ruta = os.path.join("assets", "space-ranger.mp3")
    pygame.mixer.music.load(ruta)           # Carga la música,
    pygame.mixer.music.set_volume(0.7)      # ajusta el volumen
    pygame.mixer.music.play()               # y la reproduce


# Sonidos
sndExplosion = cargarSonido("explosion.ogg")
sndLaser = cargarSonido("laser.ogg")
```

9-sonido.py

```
# Sprite bala del jugador
class BalaJugador(pygame.sprite.Sprite): # Clase que deriva de Sprite
    # CONSTRUCTOR
    def __init__(self, nave): # Recibe la nave como parámetro
        [...]
        # Reproduce sonido
        sndLaser.play()

# Sprite explosión
class Explosion(pygame.sprite.Sprite): # Clase que deriva de Sprite
    # CONSTRUCTOR
    def __init__(self, sprite): # Recibe un sprite como parámetro
        [...]
        # Reproduce sonido
        sndExplosion.play()

# Función principal del juego
def main():
    iniciarMusica()                    # Lanza la música
    [...]
```

¡Hemos terminado!



Posibles mejoras:

- Enemigos que disparan
- Fondo con desplazamiento
- Distintos tipos de enemigos
- Enemigos o balas "inteligentes" que persiguen al jugador
- Health Points / Escudos
- Varias vidas
- Power-Ups
- Boss fights
- Dos jugadores
- Joystick o gamepad
- Guardar High-Scores
- ...