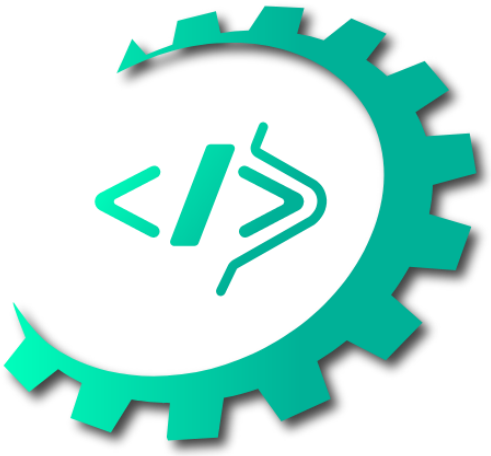

COMPUTACIÓN

Instituto Industrial Luis. A. Huergo

Alumno: Santino Rodríguez Fuchs

Profesor: Ignacio García

Bitácora de Trabajo (TP 1)

Para el desarrollo del trabajo práctico opté por realizar la propuesta número 1, basada en la recreación del clásico videojuego Space Invaders. Dado que se trata de un título ampliamente reconocido, con más de cuarenta versiones oficiales a lo largo de la historia y múltiples reinterpretaciones por parte de la comunidad, consideré pertinente realizar una búsqueda exhaustiva de información en internet que me permitiera comprender mejor su estructura y, al mismo tiempo, optimizar el proceso de programación mediante el aprovechamiento de recursos existentes.

El primer paso consistió en definir qué versión del juego deseaba recrear. Me incliné por realizar una adaptación de la versión original, incorporando colores en lugar del esquema tradicional en blanco y negro. Con este objetivo en mente, comencé por buscar los sprites de los personajes en formato PNG.

Para obtener dichos recursos, recurrí a un repositorio en GitHub en el cual encontré no solo las imágenes necesarias, sino también otros elementos complementarios como efectos de sonido y una fuente tipográfica similar a la utilizada en el juego original. El repositorio en cuestión es el siguiente: https://github.com/leerob/space-invaders.

A partir de allí comencé con la programación del juego. Dentro del repositorio se encuentra un archivo denominado spaceinvaders.py, del cual tomé como referencia gran parte del código relacionado con el modo de un jugador y la estructura del menú, adaptándolo según las necesidades y características de mi proyecto.

Modo de un Jugador:

Durante el desarrollo del modo de un jugador no surgieron mayores inconvenientes, aunque los pocos que se presentaron requirieron tiempo y dedicación para ser resueltos.

El principal desafío fue comprender un código fuente de aproximadamente 600 líneas, que contaba con escasos comentarios explicativos y utilizaba estructuras y recursos que, en algunos casos, no conocía en profundidad o directamente desconocía. Para superar esta dificultad, realicé múltiples búsquedas y lecturas complementarias que me permitieron interpretar el funcionamiento del código. A medida que lo comprendía, fui incorporando comentarios propios con el fin de facilitar futuras revisiones y evitar olvidar detalles importantes del funcionamiento interno.

Un segundo obstáculo, aunque de menor relevancia, fue que la totalidad del código se encontraba en inglés. Si bien esto no representó un impedimento significativo, requirió una atención adicional para interpretar correctamente nombres de funciones, clases y variables.

Modo de dos Jugadores:

Esta sección del desarrollo resultó considerablemente más compleja en comparación con el modo de un jugador. La incorporación de una segunda nave en el juego implicó no solo su programación y control independiente, sino también la modificación del tamaño de la pantalla, lo que conllevó la necesidad de reescribir manualmente gran parte de los valores posicionales utilizados previamente.

Además, a diferencia del modo de un jugador, esta funcionalidad no se encontraba contemplada en el código original del repositorio, por lo que me vi obligado a diseñar e implementar esta sección desde cero. Esto exigió un mayor grado de creatividad e innovación para lograr una integración funcional y coherente con el resto del proyecto.

La principal dificultad radicó en la implementación de la segunda nave sin comprometer la estructura del código. Para ello, fue necesario duplicar numerosas variables y modificar las clases correspondientes, procurando mantener una organización clara y evitar redundancias que afectaran la hegemonía del sistema.

Integración de Ambos Modos:

La integración de ambos modos (un jugador y multijugador) representó una de las etapas más desafiantes del desarrollo, dado que cualquier avance en otras áreas del proyecto requería, en muchos casos, ajustes en esta sección.

Inicialmente, planteé la idea de crear un módulo que se encargara de invocar uno u otro modo dependiendo del botón presionado en el menú principal. Esta solución resultó funcional en una primera instancia. Sin embargo, detecté que al finalizar cualquiera de los modos, el programa se cerraba por completo, lo que imposibilitaba regresar al menú principal o cambiar de modo sin reiniciar la aplicación.

Ante esta limitación, opté por desarrollar un nuevo módulo encargado de gestionar la lógica general del flujo del juego. En él se importan los tres archivos principales del proyecto (menú, modo un jugador y modo multijugador), lo cual permitió evitar la generación de bucles complejos y errores imprevistos, asegurando así una mayor estabilidad en la ejecución del programa.

Explicación del Código:

El archivo Space_Invaders.py constituye el núcleo del juego en su modalidad de un solo jugador. En su sección inicial se encuentran las instrucciones de importación de los módulos y bibliotecas necesarias para el funcionamiento general del programa.

```
from pygame import *

import sys

from os.path import abspath, dirname

from random import choice

import config  # Importar el estado de muteo
```

En primer lugar, se importa el módulo `pygame` utilizando una instrucción global (`from pygame import *`), lo que permite acceder a todas las funciones y clases de esta biblioteca sin necesidad de prefijar su nombre. Esta decisión, aunque común en proyectos pequeños o didácticos, implica ciertas consideraciones de legibilidad y manejo del espacio de nombres, pero en este contexto resulta útil para facilitar el acceso directo a los métodos de renderizado, eventos, sonido y gráficos.

A continuación, se incluye la biblioteca `sys`, indispensable para la correcta finalización del programa a través de `sys.exit()`, especialmente cuando se requiere cerrar el juego de forma controlada tras eventos como el cierre de ventana o errores fatales.

Luego, se importan dos funciones del módulo `os.path`: `abspath` y `dirname`. Estas se utilizan para obtener rutas absolutas en relación al directorio actual del archivo. Esto es especialmente útil cuando se manejan recursos externos (como imágenes y sonidos), permitiendo garantizar su localización correcta independientemente del entorno en el que se ejecute el juego.

También se importa la función `choice` desde el módulo `random`, la cual será empleada más adelante para introducir elementos de aleatoriedad en el comportamiento del juego, como el movimiento o la aparición de enemigos.

Finalmente, se importa el módulo `config`, un archivo propio del proyecto que contiene la variable encargada de gestionar el estado de muteo del juego. Esta variable permite habilitar o deshabilitar el sonido de manera global, contribuyendo a una experiencia de usuario más flexible y personalizada.

A continuación del bloque de importaciones, se definen una serie de constantes que especifican las rutas de los recursos utilizados por el juego, tales como fuentes tipográficas, imágenes y sonidos:

```
BASE_PATH = abspath(dirname(__file__))

FONT_PATH = BASE_PATH + '/Letras/'

IMAGE_PATH = BASE_PATH + '/Images/'

SOUND_PATH = BASE_PATH + '/Sounds/'
```

La constante `BASE_PATH` se obtiene mediante la combinación de `abspath()` y `dirname(__file__)`, lo cual permite identificar de forma segura y absoluta el directorio en

el que se encuentra el archivo actual. Esta estrategia es fundamental para garantizar que los recursos sean accedidos correctamente, independientemente de desde dónde se ejecute el script.

A partir de esta ruta base, se construyen las demás rutas específicas para los distintos tipos de archivos:

- `FONT_PATH` señala la carpeta donde se almacenan las fuentes tipográficas personalizadas utilizadas en el juego, en este caso ubicada en un subdirectorio llamado Letras.
- `IMAGE_PATH` hace referencia al directorio Images, que contiene los sprites, fondos y otros elementos gráficos que componen la interfaz visual del juego.
- `SOUND_PATH` apunta al directorio Sounds, donde se encuentran los archivos de audio utilizados para los efectos sonoros y la música de fondo.

Estas constantes permiten una gestión más ordenada y mantenible de los recursos, evitando el uso de rutas absolutas dispersas a lo largo del código y facilitando la modificación de la estructura del proyecto en caso de ser necesario.

En esta sección del código se definen los colores que serán utilizados a lo largo del juego, la configuración inicial de la ventana principal, así como la carga automatizada de los recursos gráficos esenciales para su funcionamiento.

```
# Colores (R, G, B)

WHITE = (255, 255, 255)

GREEN = (78, 255, 87)

YELLOW = (241, 255, 0)

BLUE = (80, 255, 239)

PURPLE = (203, 0, 255)

RED = (237, 28, 36)
```

Los colores están definidos mediante tuplas RGB (Red, Green, Blue), lo que permite su utilización directa en los métodos de renderizado de Pygame. Estos valores personalizados buscan una estética visual coherente con la temática retro del juego original *Space Invaders*, permitiendo una fácil identificación de los elementos en pantalla.

A continuación, se configura la ventana del juego y se establecen los recursos visuales necesarios:

```
# Cargado de Imagenes y Pantalla

SCREEN = display.set_mode((800, 600))

FONT = FONT_PATH + 'space_invaders.ttf'
```

- `SCREEN` representa la superficie principal donde se dibujarán todos los elementos del juego. Se establece un tamaño de 800 píxeles de ancho por 600 de alto, proporción estándar para muchos juegos en 2D.

- `FONT` define la fuente tipográfica utilizada, especificando la ruta hacia el archivo `.ttf` correspondiente. Este recurso se utilizará principalmente para renderizar texto en pantalla, como puntuaciones, mensajes o indicadores de estado.

Luego, se establece una lista con los nombres de las imágenes necesarias:

```
IMG_NAMES = ['ship', 'mystery',
             'enemy1_1', 'enemy1_2',
             'enemy2_1', 'enemy2_2',
             'enemy3_1', 'enemy3_2',
             'explosionblue','explosiongreen','explosionpurple',
             'laser', 'enemylaser']
```

A partir de esta lista, se genera un diccionario llamado `IMAGES` que carga de manera automática todos los archivos `.png` correspondientes desde el directorio de imágenes:

```
IMAGES = {name: image.load(IMAGE_PATH +
'{}.png'.format(name)).convert_alpha() for name in IMG_NAMES}
```

Este mecanismo permite acceder a cada sprite por su nombre de forma organizada y centralizada. Además, el `método .convert_alpha()` garantiza que las imágenes mantengan su transparencia, optimizando al mismo tiempo su rendimiento durante el renderizado.

Cabe destacar que esta sección en particular representó una dificultad significativa durante las primeras etapas del desarrollo, ya que no contaba con conocimientos previos sobre la estructura y funcionamiento de los diccionarios por comprensión en Python, ni sobre el método adecuado para cargar múltiples imágenes de forma eficiente. Por tal motivo, investigué su funcionamiento y realicé varias pruebas de implementación hasta lograr una versión funcional y comprensible. Este proceso me permitió adquirir nuevas herramientas y consolidar conceptos fundamentales para el manejo automatizado de recursos gráficos en proyectos de videojuegos.

A continuación, se define la clase `Ship`, que representa la nave controlada por el jugador. Esta clase hereda de `pygame.sprite.Sprite`, lo cual permite aprovechar las funcionalidades integradas del sistema de *sprites* que ofrece la biblioteca Pygame:

```
class Ship(sprite.Sprite):
```

La herencia de `sprite.Sprite` permite que la nave funcione como un objeto gráfico autónomo, con propiedades y métodos específicos, entre ellos la asignación de una imagen, una hitbox (`rect`) y la posibilidad de actualizar su estado o detectar colisiones con otros objetos del juego. Este enfoque es esencial para mantener una arquitectura orientada a objetos, modular y escalable. Para más detalles técnicos sobre esta clase, puede consultarse la documentación oficial de Pygame en el siguiente enlace:

https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Sprite

En el constructor de la clase (`__init__`), se inicializan las principales propiedades de la nave:

```
def __init__(self):
    sprite.Sprite.__init__(self)
```

```
    self.image = IMAGES['ship']
    self.rect = self.image.get_rect(topleft=(375, 540))
    self.invulnerable = True
    self.tiempoDeCreacion = time.get_ticks()
    self.speed = 5
```

El método `update` se encarga de actualizar el estado y la posición de la nave según las teclas presionadas y el tiempo actual:

```
def update(self, keys, currentTime, pantalla):
    if self.invulnerable and (currentTime - self.tiempoDeCreacion
> 1000):
        self.invulnerable = False
    if keys[K_LEFT] and self.rect.x > 10:
        self.rect.x -= self.speed
    if keys[K_RIGHT] and self.rect.x < 740:
        self.rect.x += self.speed
    pantalla.blit(self.image, self.rect)
```

La clase `Bullet` representa las balas del juego, hereda de `pygame.sprite.Sprite` para gestionar la imagen y el comportamiento de las balas. En su constructor, se carga la imagen correspondiente, se define la posición inicial, la velocidad, la dirección y el lado de quien la disparó.

```
class Bullet(sprite.Sprite):
    def __init__(self, xpos, ypos, direction, speed, filename,
side):
        sprite.Sprite.__init__(self)
        self.image = IMAGES[filename]
        self.rect = self.image.get_rect(topleft=(xpos, ypos))
        self.speed = speed
        self.direction = direction
        self.side = side
        self.filename = filename
```

El método `update` se encarga de mover la bala y de renderizarla en pantalla. Si la bala se sale de los límites de la ventana, se elimina automáticamente con `self.kill()`.

```
  def update(self, keys, *args):
    if args and hasattr(args[-1], "blit"):
      pantalla = args[-1]
```

```
        pantalla.blit(self.image, self.rect)
    else:
        print("Advertencia: No se pasó pantalla correctamente a update()")

    self.rect.y += self.speed * self.direction
    if self.rect.y < 15 or self.rect.y > 600:
        self.kill()
```

La clase `Enemy` representa a los enemigos del juego. Hereda de `pygame.sprite.Sprite` para gestionar las imágenes y el comportamiento. En su constructor, se asigna la fila y columna del enemigo para posicionarlo en la matriz del juego, se cargan sus imágenes y se define la hitbox.

```
class Enemy(sprite.Sprite):
    def __init__(self, row, column):
        sprite.Sprite.__init__(self)
        self.row = row
        self.column = column
        self.images = []
        self.load_images()
        self.index = 0
        self.image = self.images[self.index]
        self.rect = self.image.get_rect()
```

El método `toggle_image` alterna entre las imágenes del enemigo, creando una animación sencilla de "parpadeo" entre dos imágenes. Si se alcanza el final de la lista de imágenes, comienza de nuevo desde el inicio.

```
    def toggle_image(self):
        self.index += 1
        if self.index >= len(self.images):
            self.index = 0
        self.image = self.images[self.index]
```

El método `update` se encarga de actualizar la posición del enemigo en pantalla, renderizando su imagen actual.

```
    def update(self, *args):
        if args:
            pantalla = args[-1]
```

```
            pantalla.blit(self.image, self.rect)
```

El método `load_images` carga las imágenes de los enemigos según su fila, asignando un conjunto específico de imágenes para cada tipo de enemigo. Además, las redimensiona para mantener una consistencia visual en el juego.

```
    def load_images(self):
        images = {0: ['1_2', '1_1'],
                  1: ['2_2', '2_1'],
                  2: ['2_2', '2_1'],
                  3: ['3_1', '3_2'],
                  4: ['3_1', '3_2'],
                  }
        img1, img2 = (IMAGES['enemy{}'.format(img_num)] for
img_num in
                      images[self.row])
        self.images.append(transform.scale(img1, (40, 35)))
        self.images.append(transform.scale(img2, (40, 35)))
```

La clase `EnemiesGroup` hereda de `pygame.sprite.Group` y maneja un conjunto de enemigos organizados en una matriz. Controla su movimiento, animación y comportamiento colectivo.

```
class EnemiesGroup(sprite.Group):
    def __init__(self, columns, rows, enemyPosition):
        sprite.Group.__init__(self)
        self.enemies = [[None] * columns for _ in range(rows)]
        self.columns = columns
        self.rows = rows
        self.leftAddMove = 0
        self.rightAddMove = 0
        self.moveTime = 600
        self.direction = 1
        self.rightMoves = 30
        self.leftMoves = 30
        self.moveNumber = 15
        self.timer = time.get_ticks()
        self.bottom = enemyPosition + ((rows - 1) * 45) + 35
```

```
        self._aliveColumns = list(range(columns))
        self._leftAliveColumn = 0
        self._rightAliveColumn = columns - 1
```

El método `update` controla el movimiento del grupo de enemigos. Si ha pasado el tiempo suficiente, los enemigos se mueven en la dirección indicada (derecha o izquierda) y, al llegar al límite, bajan una fila. La animación de las imágenes también se alterna.

```
    def update(self, current_time):
        if current_time - self.timer > self.moveTime:
            if self.direction == 1:
                max_move = self.rightMoves + self.rightAddMove
            else:
                max_move = self.leftMoves + self.leftAddMove

            if self.moveNumber >= max_move:
                self.leftMoves = 30 + self.rightAddMove
                self.rightMoves = 30 + self.leftAddMove
                self.direction *= -1
                self.moveNumber = 0
                self.bottom = 0
                for enemy in self:
                    enemy.rect.y += ENEMY_MOVE_DOWN
                    enemy.toggle_image()
                    if self.bottom < enemy.rect.y + 35:
                        self.bottom = enemy.rect.y + 35
            else:
                velocity = 10 if self.direction == 1 else -10
                for enemy in self:
                    enemy.rect.x += velocity
                    enemy.toggle_image()
                self.moveNumber += 1
            self.timer += self.moveTime
```

Los métodos `add_internal` y `remove_internal` gestionan la adición y eliminación de enemigos dentro de la matriz. La eliminación de un enemigo actualiza la velocidad de movimiento del grupo.

```
    def add_internal(self, *sprites):
        super(EnemiesGroup, self).add_internal(*sprites)
        for s in sprites:
            self.enemies[s.row][s.column] = s


    def remove_internal(self, *sprites):
        super(EnemiesGroup, self).remove_internal(*sprites)
        for s in sprites:
            self.kill(s)
        self.update_speed()
```

El método `random_bottom` elige aleatoriamente una columna viva y devuelve el enemigo más bajo de esa columna para disparar.

```
    def random_bottom(self):
        col = choice(self._aliveColumns)
        col_enemies = (self.enemies[row - 1][col]
                       for row in range(self.rows, 0, -1))
        return next((en for en in col_enemies if en is not None),
None)
```

El método `update_speed` ajusta la velocidad del grupo según el número de enemigos restantes.

```
    def update_speed(self):
        if len(self) == 1:
            self.moveTime = 200
        elif len(self) <= 10:
            self.moveTime = 400
```

Finalmente, el método `kill` elimina un enemigo de la matriz y ajusta la cantidad de movimiento según la columna en la que se encontraba.

```
    def kill(self, enemy):
        self.enemies[enemy.row][enemy.column] = None
        is_column_dead = self.is_column_dead(enemy.column)
        if is_column_dead:
            self._aliveColumns.remove(enemy.column)
        if enemy.column == self._rightAliveColumn:
            while self._rightAliveColumn > 0 and is_column_dead:
```

```
                self._rightAliveColumn -= 1
                self.rightAddMove += 5
                is_column_dead =
self.is_column_dead(self._rightAliveColumn)
        elif enemy.column == self._leftAliveColumn:
            while self._leftAliveColumn < self.columns and
is_column_dead:
                self._leftAliveColumn += 1
                self.leftAddMove += 5
                is_column_dead =
self.is_column_dead(self._leftAliveColumn)
```

La clase `Blocker` crea los bloques de protección del jugador. Cada bloque es un sprite cuadrado con color y posición definidos, que se dibuja en pantalla.

```
class Blocker(sprite.Sprite):  # Bloques de protección
    def __init__(self, size, color, row, column):
        sprite.Sprite.__init__(self)
        self.height = size
        self.width = size
        self.color = color
        self.image = Surface((self.width, self.height))
        self.image.fill(self.color)
        self.rect = self.image.get_rect()
        self.row = row
        self.column = column

    def update(self, pantalla):
        pantalla.blit(self.image, self.rect)
```

La clase `Mystery` representa la nave especial que aparece cada cierto tiempo y se desplaza horizontalmente por la pantalla. Su aparición está temporizada y su sonido característico depende del estado de muteo. Se alterna entre los extremos de la pantalla, reiniciando su temporizador cada vez que sale.

```
class Mystery(sprite.Sprite):
    def __init__(self):
        sprite.Sprite.__init__(self)
        self.image = transform.scale(IMAGES['mystery'], (75, 35))
```

```
        self.rect = self.image.get_rect(topleft=(-80, 45))
        self.row = 5
        self.moveTime = 25000
        self.direction = 1
        self.timer = time.get_ticks()
        self.mysteryEntered     =     mixer.Sound(SOUND_PATH     +
'mysteryentered.wav')
        self.mysteryEntered.set_volume(0.0 if config.MUTEADO else
0.3)
        self.playSound = True

    def update(self, keys, currentTime, *args):
        if not args: return
        pantalla = args[-1]
        resetTimer = False
        passed = currentTime - self.timer

        if passed > self.moveTime:
            if  (self.rect.x  <  0  or  self.rect.x  >  800)  and
self.playSound:
                self.mysteryEntered.play()
                self.playSound = False
            if self.rect.x < 840 and self.direction == 1:
                self.mysteryEntered.fadeout(4000)
                self.rect.x += 2
                pantalla.blit(self.image, self.rect)
            if self.rect.x > -100 and self.direction == -1:
                self.mysteryEntered.fadeout(4000)
                self.rect.x -= 2
                pantalla.blit(self.image, self.rect)

        if self.rect.x > 830:
            self.playSound = True
            self.direction = -1
```

```
            resetTimer = True
        if self.rect.x < -90:
            self.playSound = True
            self.direction = 1
            resetTimer = True
        if passed > self.moveTime and resetTimer:
            self.timer = currentTime
```

`EnemyExplosion` muestra brevemente una animación cuando un enemigo muere. Usa dos tamaños de imagen para generar un efecto de expansión y desaparece después de 400 milisegundos.

```
class EnemyExplosion(sprite.Sprite):
    def __init__(self, enemy, *groups):
        super(EnemyExplosion, self).__init__(*groups)
        self.image = transform.scale(self.get_image(enemy.row),
(40, 35))
        self.image2 = transform.scale(self.get_image(enemy.row),
(50, 45))
        self.rect = self.image.get_rect(topleft=(enemy.rect.x,
enemy.rect.y))
        self.timer = time.get_ticks()

    @staticmethod
    def get_image(row):
        img_colors = ['purple', 'blue', 'blue', 'green', 'green']
        return IMAGES['explosion{}'.format(img_colors[row])]

    def update(self, current_time, *args):
        if args:
            pantalla = args[-1]
            passed = current_time - self.timer
            if passed <= 100:
                pantalla.blit(self.image, self.rect)
            elif passed <= 200:
                pantalla.blit(self.image2, (self.rect.x - 6,
self.rect.y - 6))
```

```
        if current_time - self.timer > 400:
            self.kill()
```

`MysteryExplosion` muestra el puntaje ganado al destruir la nave misteriosa. El texto aparece dos veces intermitente antes de desaparecer.

```
class MysteryExplosion(sprite.Sprite):
    def __init__(self, mystery, score, *groups):
        super(MysteryExplosion, self).__init__(*groups)
        self.text = Text(FONT, 20, str(score), WHITE,
                         mystery.rect.x + 20, mystery.rect.y + 6)
        self.timer = time.get_ticks()

    def update(self, current_time, *args):
        if args:
            pantalla = args[-1]
            passed = current_time - self.timer
            if passed <= 200 or 400 < passed <= 600:
                self.text.draw(pantalla)
            elif passed > 600:
                self.kill()
```

`ShipExplosion` muestra la nave destruida por un breve período de tiempo después de recibir daño, y luego desaparece.

```
class ShipExplosion(sprite.Sprite):
    def __init__(self, ship, *groups):
        super(ShipExplosion, self).__init__(*groups)
        self.image = IMAGES['ship']
        self.rect  =  self.image.get_rect(topleft=(ship.rect.x,
ship.rect.y))
        self.timer = time.get_ticks()

    def update(self, current_time, *args):
        if args:
            pantalla = args[-1]
            passed = current_time - self.timer
```

```
            if 300 < passed <= 600:
                pantalla.blit(self.image, self.rect)
        if current_time - self.timer > 900:
            self.kill()
```

`Life` representa una vida extra visible en pantalla (nave pequeña).

```
class Life(sprite.Sprite):
    def __init__(self, xpos, ypos):
        sprite.Sprite.__init__(self)
        self.image = IMAGES['ship']
        self.image = transform.scale(self.image, (23, 23))
        self.rect = self.image.get_rect(topleft=(xpos, ypos))

    def update(self, *args):
        if args and hasattr(args[-1], "blit"):
            pantalla = args[-1]
            pantalla.blit(self.image, self.rect)
```

`Text` muestra un mensaje en pantalla en una posición específica.

```
class Text(object):
    def __init__(self, textFont, size, message, color, xpos,
ypos):
        self.font = font.Font(textFont, size)
        self.surface = self.font.render(message, True, color)
        self.rect = self.surface.get_rect(topleft=(xpos, ypos))

    def draw(self, surface):
        surface.blit(self.surface, self.rect)
```

Configuración recomendada para Linux: En la línea `mixer.pre_init(44100, -16, 1, 4096)`, se realiza una configuración especial del mixer de Pygame. Esta configuración se recomienda para los usuarios de Linux, ya que puede evitar problemas relacionados con la latencia o la reproducción incorrecta de sonidos en algunos sistemas operativos basados en Linux.
La documentación oficial de Pygame menciona que en plataformas como Linux, este ajuste puede mejorar la compatibilidad del sistema de audio. Si no estás utilizando Linux, puedes omitir esta configuración sin problema.

https://www.pygame.org/docs/ref/mixer.html#pygame.mixer.pre_init

Código comentado con triple comillas: Algunas secciones del código están comentadas utilizando tres comillas dobles (""" . . . """). Esto indica que hay partes del código que fueron dejadas sin usar o que no se implementaron completamente en esta versión. Este código fue inicialmente pensado para realizar ciertas tareas (como mostrar texto relacionado con los enemigos y sus puntuaciones), pero luego decidí cambiar el formato del código y adaptarlo a nuevas necesidades del juego. Sin embargo, dado que esas secciones podrían ser útiles en el futuro, las dejé comentadas en lugar de eliminarlas por completo. Es posible que en el futuro vuelva a utilizarlas, por lo que por ahora solo las he comentado para evitar complicaciones adicionales al eliminar definitivamente esas líneas.

```
class SpaceInvaders(object): # Codigo del Juego

    def __init__(self):

        mixer.pre_init(44100, -16, 1, 4096)

        init()

        self.bullets = sprite.Group()

        self.allSprites = sprite.Group()

        self.shipAlive = True

        self.score = 0

        self.player = Ship()

        self.sounds = {

            'shoot': mixer.Sound('Sounds\shoot.wav'),

            'shoot2': mixer.Sound('Sounds\shoot2.wav')

        }

        self.allSprites.add(self.player)

        self.clock = time.Clock()

        self.caption = display.set_caption('Space Invaders')

        self.screen = SCREEN

        self.menu = image.load(IMAGE_PATH + 'image_second.webp')

        self.menu = transform.scale(self.menu, (800, 600))

        self.background = image.load(IMAGE_PATH + 
'background.jpg')

        self.startGame = False

        self.mainScreen = True

        self.gameOver = False

        self.enemyPosition = ENEMY_DEFAULT_POSITION

        self.titleText2 = Text(FONT, 25, 'Press any key to 
continue', WHITE,
```

```
                                    201, 540)
        self.gameOverText = Text(FONT, 50, 'Game Over', WHITE,
250, 270)
        self.nextRoundText = Text(FONT, 50, 'Next Round', WHITE,
240, 270)
        """self.enemy1Text = Text(FONT, 25, '    =    10 pts',
GREEN, 368, 400)
        self.enemy2Text = Text(FONT, 25, '    =  20 pts', BLUE,
368, 450)
        self.enemy3Text = Text(FONT, 25, '    =  30 pts', PURPLE,
368, 500)
        self.enemy4Text = Text(FONT, 25, '    =  ?????', RED, 368,
550)"""
        self.scoreText = Text(FONT, 20, 'Score', WHITE, 5, 5)
        self.livesText = Text(FONT, 20, 'Lives ', WHITE, 640, 5)


        self.life1 = Life(715, 3)
        self.life2 = Life(742, 3)
        self.life3 = Life(769, 3)
        self.livesGroup = sprite.Group(self.life1, self.life2,
self.life3)
```

La función `reset(self, score)` se encarga de reiniciar todos los grupos y variables importantes al iniciar una nueva partida o pasar de nivel. Lo hace creando nuevamente las entidades necesarias (jugador, enemigos, balas, etc.) y restableciendo las variables del juego. Se utiliza para preparar el juego desde cero después de que termina una ronda o el juego.

```
def reset(self, score):
    self.player = Ship()
    self.playerGroup = sprite.Group(self.player)
    self.explosionsGroup = sprite.Group()
    self.bullets = sprite.Group()
    self.mysteryShip = Mystery()
    self.mysteryGroup = sprite.Group(self.mysteryShip)
    self.enemyBullets = sprite.Group()
    self.make_enemies()
    self.allSprites = sprite.Group(self.player, self.enemies,
                                   self.livesGroup,self.mysteryShip)
```

```
    self.keys = key.get_pressed()


    self.timer = time.get_ticks()
    self.noteTimer = time.get_ticks()
    self.shipTimer = time.get_ticks()
    self.score = score
    self.create_audio()
    self.makeNewShip = False
    self.shipAlive = True
```

La función `make_blockers(self, number)` crea los bloques de defensa en la pantalla. Los bloques se distribuyen en una cuadrícula de 4 filas por 9 columnas, y se colocan en posiciones específicas en función de un número de bloque (que puede cambiar su posición). Se usa para crear la defensa que los jugadores pueden utilizar para cubrirse de las balas enemigas.

```
def make_blockers(self, number):
    blockerGroup = sprite.Group()
    for row in range(4):
        for column in range(9):
            blocker = Blocker(10, GREEN, row, column)
            blocker.rect.x = 50 + (200 * number) + (column * blocker.width)
            blocker.rect.y = BLOCKERS_POSITION + (row * blocker.height)
            blockerGroup.add(blocker)
    return blockerGroup
```

En las siguiente sección la función `create_audio()` gestiona los sonidos que el juego utiliza en diferentes situaciones. Se crea un diccionario `self.sounds` que contiene los sonidos clave, como disparos, muertes de enemigos, y explosiones. Para cada sonido en el diccionario, se asigna un archivo `.wav` correspondiente, y el volumen se ajusta dependiendo de si el juego está en modo muteado (`config.MUTEADO`). Además, se inicializa una lista de notas musicales (`self.musicNotes`) para efectos de música en el juego, donde cada nota también tiene su volumen ajustado. La función también establece un índice de notas para controlar la secuencia de las notas musicales.

```
def create_audio(self):
  self.sounds = {}


  for SoundName in ['shoot', 'shoot2', 'invaderkilled', 'mysterykilled', 'shipexplosion']:
```

```
        self.sounds[SoundName] = mixer.Sound(SOUND_PATH + '{}.wav'.format(SoundName))
        self.sounds[SoundName].set_volume(0.0 if config.MUTEADO else 0.2)

    self.musicNotes = [mixer.Sound(SOUND_PATH + '{}.wav'.format(i)) for i in range(4)]
    for Sound in self.musicNotes:
        Sound.set_volume(0.0 if config.MUTEADO else 0.5)

    self.noteIndex = 0
```

La función `play_main_music(currentTime)` se encarga de reproducir la música en intervalos sincronizados con el movimiento de los enemigos. Cada vez que pasa el tiempo adecuado, se reproduce una nota musical en secuencia. Si el índice de la nota excede el límite, se reinicia al primer valor de la secuencia.

```
def play_main_music(self, currentTime):
    if currentTime - self.noteTimer > self.enemies.moveTime:
        self.note = self.musicNotes[self.noteIndex]
        if self.noteIndex < 3:
            self.noteIndex += 1
        else:
            self.noteIndex = 0

        self.note.play()
        self.noteTimer += self.enemies.moveTime
```

La función `should_exit(evt)` es un método estático que determina si el juego debe cerrarse. Evalúa si el jugador ha hecho clic en el botón de cerrar ventana o ha presionado la tecla *Escape*. Si ocurre alguna de esas acciones, se retorna `True`.

```
@staticmethod
def should_exit(evt):
    return evt.type == QUIT or (evt.type == KEYUP and evt.key == K_ESCAPE)
```

La función `check_input()` gestiona toda la interacción del usuario con el teclado. Se encarga de:

- Detectar si el jugador quiere cerrar el juego (usando `should_exit`).
- Iniciar la partida desde el menú si se presiona cualquier tecla.

- Crear los bloques defensivos, agrupar las vidas y reiniciar todos los elementos del juego cuando se empieza desde el menú.
- Detectar si el jugador dispara presionando la barra espaciadora mientras el juego está en marcha. Si no hay balas activas y la nave sigue viva, se permite el disparo. Además, si el jugador ya tiene más de 1000 puntos, el disparo se vuelve doble (dos balas en lugar de una), aumentando así la potencia ofensiva.

Esto asegura que el juego responda correctamente tanto a los controles de inicio como a la mecánica básica de disparo.

```
def check_input(self):
    self.keys = key.get_pressed()


    for e in event.get():
        if self.should_exit(e):
            return True


        if e.type == KEYUP and self.mainScreen:
            self.allBlockers = sprite.Group(self.make_blockers(0),
                                            self.make_blockers(1),
                                            self.make_blockers(2),
                                            self.make_blockers(3))
            self.livesGroup.add(self.life1, self.life2, self.life3)
            self.reset(0)
            self.startGame = True
            self.mainScreen = False


        if e.type == KEYDOWN and self.startGame:
            if e.key == K_SPACE:
                if len(self.bullets) == 0 and self.shipAlive:
                    if self.score < 1000:
                        bullet = Bullet(self.player.rect.x + 23,
                                        self.player.rect.y + 5, -1,
                                        15, 'laser', 'center')
                        self.bullets.add(bullet)
```

```
            self.allSprites.add(self.bullets)
            self.sounds['shoot'].play()
        else:
            leftbullet = Bullet(self.player.rect.x + 8,
                                self.player.rect.y + 5, -1,
                                15, 'laser', 'left')
            rightbullet = Bullet(self.player.rect.x + 38,
                                 self.player.rect.y + 5, -1,
                                 15, 'laser', 'right')
            self.bullets.add(leftbullet)
            self.bullets.add(rightbullet)
            self.allSprites.add(self.bullets)
            self.sounds['shoot2'].play()

return False
```