Trabajo Práctico Final — Programación Computacional

Flappy Fish: Juego basado en Pygame

Julieta Zanoni, Mariia Osipova, Santino Scofano y Morena Roldan

Universidad de San Andrés

jzanoni@udesa.edu.ar
mosipova@udesa.edu.ar
sscofano@udesa.edu.ar
mroldan@udesa.edu.ar

12 de diciembre 2025

Resumen de la presentacion (I)

Resumen de la presentacion (II)

Introducción

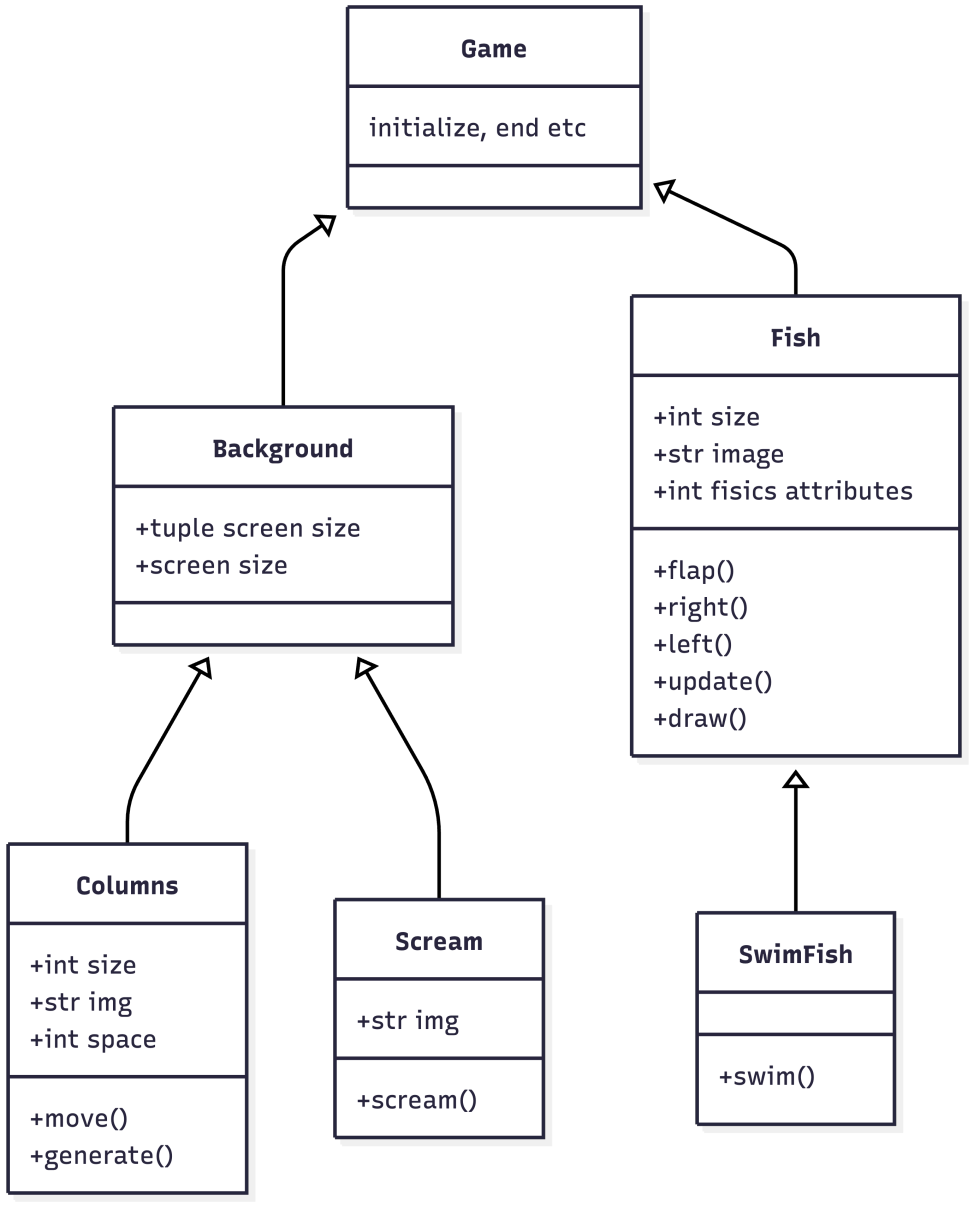
Nuestro trabajo práctico está dividido en dos partes: la primera es soble el desarrollo de un videojuego manual inspirado en Flappy Bird, llamado Flappy Fish, implementado en Python con la librería Pygame.

Introducción

La segunda parte se enfoca en la implementación de un Algoritmo Genético (AG) para entrenar a una población de “peces”, se juega de forma autónoma al videojuego manual.

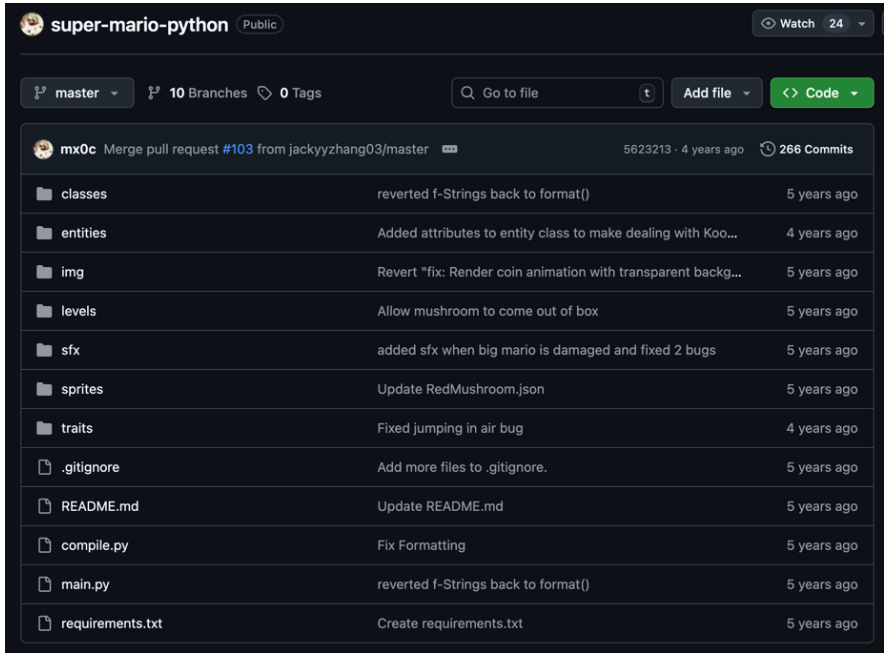
Arquitectura del Juego

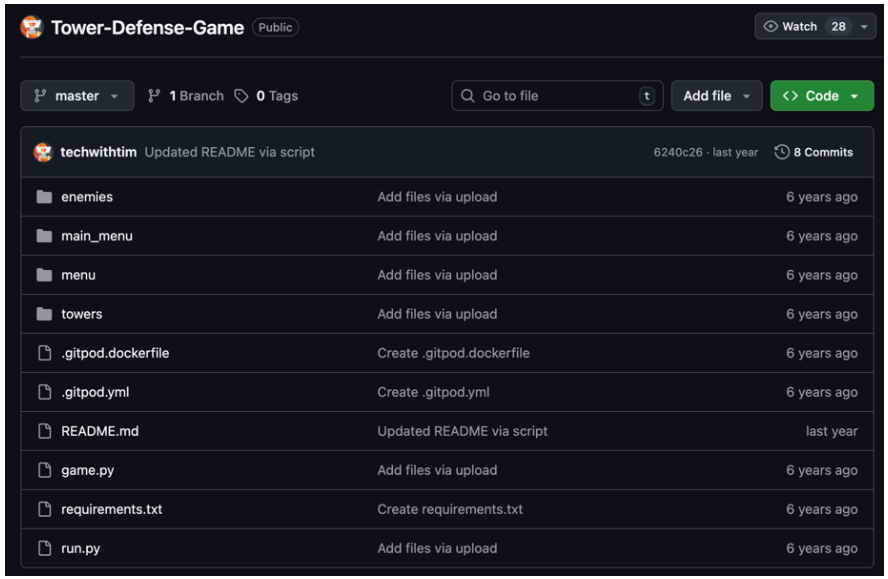
Pensando en la arquitectura del juego, nos enfrentamos al primer desafío: ¿cómo debíamos estructurar y organizar el proyecto? Comenzamos trabajando a partir de este borrador inicial.



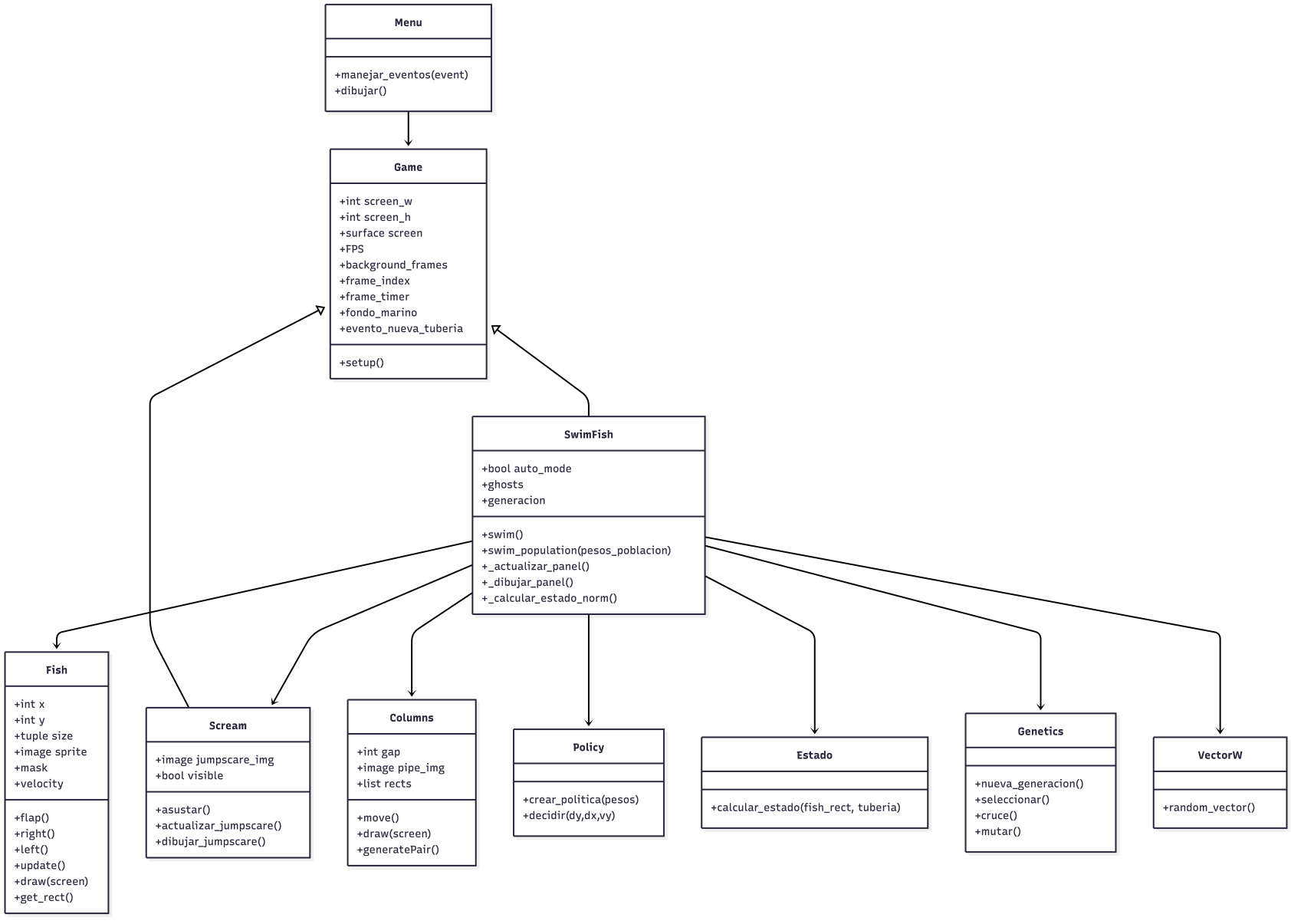
Arquitectura del Juego

Analizamos varios juegos desarrollados con Pygame y publicados de forma abierta. Estas referencias nos permitieron observar enfoques comunes de arquitectura y organización del código. Entre ellos, miramos proyectos como Super Mario Python y Tower Defence Game, que utilizamos como guía conceptual.





Arquitectura del Juego



Arquitectura del Juego

El proyecto se estructura en los siguientes módulos:

- **game.py** — clase base del juego (ventana, fondo, sonido y tuberías).
- **fish.py** — física, movimiento y máscara de colisión del pez.
- **screamer.py** — módulo del screamer para el juego.
- **generacion_de_tuberias.py** — creación, posición y movimiento de las tuberías.
- **menu.py** — interfaz del menú principal y manejo de opciones.
- **swim_fish.py** — lógica del juego (manual/modo con Algoritmo Genético).
- **ml/** — política del agente, cálculo del estado, generación de pesos y genética.

Módulo game.py (I)

La clase `Game` actúa como el marco general del que hereda `SwimFish`. Su constructor realiza la configuración inicial del entorno:

- Inicializa Pygame y el objeto `Clock` (FPS = 120).
- Crea la ventana principal del juego (1000×600 píxeles).
- Carga los fotogramas del fondo animado.

```
class Game:
    def __init__(self):
        pygame.init()
        self.clock = pygame.time.Clock()
        self.FPS = 120
        self.screen = pygame.display.set_mode((1000, 600))

        # Animación de fondo
        self.animation_folder = "../data/img/fondo_animado"
        self.background_frames = self._load_background_frames()
        self.frame_index = 0
        self.frame_rate = 30
```

Módulo game.py (II)

- Define la música de fondo y sonido del salto.
- Sprite de tubería y máscara pixel-perfect.

```
self.music_path = "../data/audios/linkin park fondo.ogg"
pygame.mixer.music.load(self.music_path)

self.sonido_salto = pygame.mixer.Sound(
    "../data/audios/efecto bubble.ogg")

# Fondo marino
self.fondo_marino = pygame.image.load(
    "../data/img/pixil-frame-0.png").convert()
self.fondo_marino = pygame.transform.scale(
    self.fondo_marino, (self.screen_w, self.screen_h))
```

Módulo game.py (III)

- Define el hueco vertical entre tuberías = `300`.
- Define el evento `evento_nueva_tuberia` cada 1500 ms.

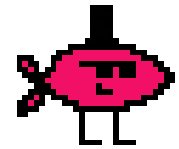
```
# Tuberías
self.imagen_tuberia = pygame.image.load(
    "../data/img/alga2.png").convert_alpha()
self.imagen_tuberia = pygame.transform.scale(
    self.imagen_tuberia, (70, 400))
self.tuberia_mask = pygame.mask.from_surface(
    self.imagen_tuberia)
self.hueco_entre_tuberias = 300

self.evento_nueva_tuberia = pygame.USEREVENT
pygame.time.set_timer(self.evento_nueva_tuberia, 1500)
```

Entonces, Game sabe todo sobre la ventana, el fondo y las tuberías, pero no sabe nada sobre quién está volando alrededor de este mundo o cómo se desarrolla exactamente el juego; esa es responsabilidad de SwimFish.

Módulo fish.py: Fish y su fisica (I)

El Fish en fish.py una clase independiente que luego se utiliza dentro de otras clases (principalmente dentro de SwimFish) como un objeto compuesto.



- En el constructor se carga la imagen del pez, se escala al tamaño indicado y se crean su `rect` y la máscara de colisión.

```
class Fish:
    def __init__(self, x, y, size, image):

        self._start_pos = (x, y)

        self.size = size
        self.original_image = pygame.image.load(
            image).convert_alpha()
        self.original_image = pygame.transform.scale(
            self.original_image, (size[0], size[1]))
        self.image = self.original_image
        self.rect = self.image.get_rect(center = (x, y))
        self.mask = pygame.mask.from_surface(self.image)
```

Módulo fish.py: Fish y su fisica (II)

- El pez posee una velocidad vertical, una gravedad constante, una fuerza de salto, (donde un valor negativo indica movimiento ascendente) y una velocidad máxima de caída.
- El método `flap()` simplemente reinicia la velocidad al valor de `jump_strength`, produciendo un impulso instantáneo hacia arriba.

```
        self.velocity = 0
        self.gravity = 0.3
        self.jump_strength = -10
        self.max_fall_speed = 100
        self.air_resistance = 0.9

    def flap(self):
        self.velocity = self.jump_strength
```

Módulo fish.py: Fish y su fisica (III)

- El método `update()` incrementa la velocidad con la gravedad, la limita según `max_fall_speed`, actualiza la posición vertical del `rect` y recalcula la rotación del sprite: el ángulo es proporcional a la velocidad, pero está acotado aproximadamente entre $-30°$ durante el ascenso y $+90°$ durante la caída. Tras la rotación, se recalculan el `rect` y la máscara.

```
def update(self):
    self.velocity += self.gravity

    if self.velocity > self.max_fall_speed:
        self.velocity = self.max_fall_speed

    self.rect.y += self.velocity
    self._rotar_pez()
```

Módulo fish.py: Fish y su fisica (IV)

- El método `_rotar_pez()` controla la orientación visual del sprite según su velocidad vertical. Calcula un ángulo proporcional a la velocidad: cuando el pez asciende se limita a $-30°$, y cuando cae a $+90°$. Rota la imagen original con `pygame.transform.rotate()` y actualiza el centro del `rect` para mantener la posición. Finalmente, se recalcula la máscara de colisión a partir de la imagen rotada.

```
def _rotar_pez(self):
    angulo = self.velocity * 3
    if self.velocity > 0:
        angulo = min(angulo, 90)
    else:
        angulo = max(angulo, -30)
    self.image = pygame.transform.rotate(
        self.original_image, -angulo)
    old_center = self.rect.center
    self.rect = self.image.get_rect(center=old_center)
    self.mask = pygame.mask.from_surface(self.image)
```

Módulo fish.py: Fish y su fisica (V)

- El método `reset()` restablece el pez a su posición inicial, reinicia su velocidad y reconstruye el `rect` y la máscara originales. Se usa en el caso de una colision para reiniciar el juego.

```
def reset(self):
    self.rect.center = self._start_pos
    self.velocity = 0
    self.image = self.original_image
    self.rect = self.image.get_rect(center=self._start_pos)
    self.mask = pygame.mask.from_surface(self.image)
```

Módulo generacion_de_tuberias.py (I)

La clase `tuberias` en `generacion_de_tuberias.py` define un par de obstáculos: el tubo de algas superior e inferior.

- Se selecciona una `altura_referencia` en el rango [150, 450];
- A partir de esa posición se construyen los rectángulos de dos tuberias, tal que entre ambos se mantenga un hueco vertical de altura `hueco` (parámetro recibido desde `Game`).

```
class tuberias:
    def __init__(self, x, hueco, imagen):
        self.imagen_tuberia = imagen
        self.altura_referencia = random.randint(150, 450)
        self.x = x
        self.hueco = hueco
        self.tubo_arriba = self.imagen_tuberia.get_rect(
            midbottom = (x, self.altura_referencia - hueco // 2))
        self.tubo_abajo = self.imagen_tuberia.get_rect(
            midtop = (x, self.altura_referencia + hueco // 2))
```

Módulo generacion_de_tuberias.py (II)

- El método `mover_tuberias()` simplemente desplaza la coordenada `x` y sincroniza ese movimiento con los rectángulos de ambas tuberías.
- El método `dibujar_tuberias()` dibuja la tubería superior invertida y la inferior en su orientación normal.

```
def mover_tuberias(self):
    self.x -= self.velocidad
    self.tubo_arriba.x = self.x
    self.tubo_abajo.x = self.x

def dibujar_tuberias(self,screen):
    screen.blit(pygame.transform.flip(self.imagen_tuberia, False,
    screen.blit(self.imagen_tuberia, self.tubo_abajo)
```

Módulo generacion_de_tuberias.py (III)

- Ambas tuberías comparten una misma coordenada horizontal `x`, que posteriormente se reduce a una velocidad constante `velocidad = 4`, pidiendo que las tuberias se desplacen de derecha a izquierda.
- Se almacena `gap_y = altura_referencia`, es decir, la coordenada vertical del centro del pasaje, utilizada para calcular la característica `dy` del agente.

```
self.velocidad = 4
self.gap_y = self.altura_referencia
```

Módulo menu.py (I)

El menú en `menu.py` recibe la pantalla y sus dimensiones, y crea el título `FLAPPY FISH!` junto con dos opciones: o el juego manual o la simulacion (AG).

Módulo menu.py (II)

El método `manejar_eventos()` devuelve el modo de juego cuando el usuario hace clik con el mouse sobre la opción correspondiente o al presionar las teclas 1/2.

```
def manejar_eventos(self, event):
            if event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 1:
                    mouse_pos = event.pos
                    if self.rect_single.collidepoint(mouse_pos):
                        self.seleccion = 'SINGLE'
                        return self.seleccion
                    if self.rect_evolutivo.collidepoint(mouse_pos):
                        self.seleccion = 'EVOLUTIVO'
                        return self.seleccion

            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_1:
                    self.seleccion = 'SINGLE'
                    return self.seleccion
                elif event.key == pygame.K_2:
                    self.seleccion = 'EVOLUTIVO'
                    return self.seleccion
```

Dos modos de juego: Manual vs. Simulación

En Flappy Fish tenemos dos modos de juego:

Single Player (Juego Manual)

- El usuario controla un único pez con el teclado.
- Cada salto se produce cuando el jugador presiona `ESPACIO`.
- El bucle principal del juego lee los eventos de teclado y decide si llamar a `fish.flap()`.

Simulación (Algoritmo Evolutivo)

- No se usan teclas para controlar el pez.
- Una población de agentes decide cuándo saltar.
- Cada agente tiene su propia política de salto, parametrizada por un vector de pesos.

Intuición del modo evolutivo

La idea intuitiva es sustituir al humano por una colección de “jugadores artificiales”, donde cada uno aprende a jugar debido las generaciones pasadas.

Tras varias generaciones, el juego “descubre” políticas que controlan el pez mejor que el azar.

Arquitectura de la parte AG

El proyecto se estructura en los siguientes módulos:

- **ml/vector_w.py** — generación de vectores de pesos iniciales.
- **ml/policy.py** — definición de la política `decidir(dy,dx,vy)`.
- **ml/calcular_estado.py** — construcción del estado (dy, dx) a partir de la próxima tubería.
- **ml/genetics.py** — selección proporcional, cruza, mutación y nueva generación.

El archivo `swim_fish.py` conecta estos módulos con la lógica del juego.

Módulo ml/calcular_estado.py (I)

La función `calcular_estado` resume la geometría del juego en dos números: `dy` y `dx`.

- `rect_jugador`: es el `pygame.Rect` del pez (su posición en pantalla).
- `proxima_pipe`: es el objeto tubería que el pez tiene inmediatamente por delante.

```
def calcular_estado(rect_jugador, proxima_pipe):
    ...
return dy, dx
```

Módulo ml/calcular_estado.py (II)

- `dy`:
  - `rect_jugador.centery` = coordenada vertical del centro del pez.
  - `proxima_pipe.gap_y` = coordenada vertical del centro del hueco.
  - `dy = centery - gap_y`: si `dy > 0`, el hueco está por encima del pez (en el sistema de coordenadas de la pantalla, menor `y` implica más arriba), es decir, “hay que subir”.

```
dy = rect_jugador.centery - proxima_pipe.gap_y
```

Módulo ml/calcular_estado.py (III)

- `dx`:
  - `proxima_pipe.top_rect().left` = borde izquierdo de la tubería superior.
  - `rect_jugador.right` = borde derecho del pez.
  - `dx = left_tubo - right_pez`: distancia horizontal hasta la tubería.
- `vy` no se calcula aquí: se toma directamente de `fish.velocity`.

```
dx = proxima_pipe.top_rect().left - rect_jugador.right
```

¿Por qué calcular_estado() no usa directamente el objeto Fish?

El estado del agente usa variables relativas:

$$dy = y_{\text{pez}} - y_{\text{hueco}}, \qquad dx = x_{\text{tubería}} - x_{\text{pez}}.$$

El objeto `Fish` sólo conoce su propia posición (`fish.rect`) y su velocidad (`fish.velocity`), pero no sabe nada sobre la próxima tubería ni sobre el hueco.
Por eso `calcular_estado(rect_jugador, proxima_pipe)` recibe:

- la geometría del pez (`rect_jugador`),
- la tubería más cercana (`proxima_pipe`),
- y a partir de ambos calcula `dy` y `dx`.

La velocidad `vy` se toma directamente de `fish.velocity` fuera de esta función, para que el módulo no dependa del tipo concreto `Fish` y se mantenga más modular.

Módulo policy.py (I)

En nuestro proyecto llamamos **política** a una función que definimos como

$$\gamma : \mathbb{R}^3 \to \{\text{True}, \text{False}\}, \qquad (dy, dx, vy) \mapsto \gamma(dy, dx, vy).$$

El valor booleano que devuelve la política determina si el pez debe saltar en ese instante o no.

Módulo politica.py (II): creación de la política

La función γ está implementada en `policy.py` mediante `crear_politica`, que construye una política parametrizada por un vector de 6 pesos:

$$w = [w_0, w_1, w_2, w_3, w_4, w_5].$$

Si `pesos` es `None`, se genera un vector aleatorio con `random_vector(low, high)`.
Si `pesos` contiene 6 números, se usan como pesos ya definidos (por ejemplo, provenientes de la generación anterior).

```
def crear_politica(pesos=None, low=-0.5, high=0.5):
    if pesos is None:
        pesos = random_vector(low, high)

    w = list(pesos)
```

Detalle de diseño: parámetros `low` y `high`

En la función `crear_politica` los parámetros `low` y `high` aparecen dos veces:

- La función `random_vector` ya define sus propios valores por defecto `low=-0.5` y `high=0.5`.
- `crear_politica` vuelve a declarar `low=-0.5` y `high=0.5`, pero en nuestro código nunca llamamos a `crear_politica` con otros valores.

Por lo tanto, en la próxima versión del juego se podría simplificar la función a:

- eliminar `low` y `high` de la firma de `crear_politica`,
- y llamar simplemente `random_vector()` sin argumentos.

Este sería un ejemplo de pequeña refactorización posible para reducir duplicación y mantener la interfaz más limpia.

Módulo policy.py (IV): función interna decidir

Dentro de `crear_politica` se define la función interna `decidir(dy, dx, vy)`, que implementa la política del agente:

- Normaliza el estado: *dy*, *dx*, *vy*.
- Calcula un valor usando los pesos *w*.
- Devuelve `True` si el pez debe saltar.

```
def decidir(dy, dx, vy):
    dy_n = dy / 400.0
    dx_n = dx / 400.0
    vy_n = vy / 300.0

    valor = (w[0] +
             w[1] * dy_n +
             w[2] * (dy_n * dy_n) +
             w[3] * dx_n +
             w[4] * (dx_n * dx_n) +
             w[5] * vy_n)
    return valor > 0
return decidir, pesos
```

Normalización del vector

En todo el juego trabajamos con números en píxeles.

$dy \in [-400, 400]$ diferencia vertical entre el pez y el centro del hueco.
$dx \in [0, 400]$ distancia horizontal hasta la próxima tubería.
$vy \in [-10, 10]$ velocidad vertical típica del pez.

Para la política es más cómodo usar números pequeños, del orden de -1 a 1. Por eso hacemos:

$$dy_n = \frac{dy}{400}, \quad dx_n = \frac{dx}{400}, \quad vy_n = \frac{vy}{300}.$$

Así todos los valores están en un rango parecido. Ninguna variable tiene números "gigantes" que tapen a las demás. Con números pequeños, los pesos `w0,...,w5` pueden ser simples (por ejemplo entre `-0.5` y `0.5`) y el algoritmo evolutivo puede buscar buenas combinaciones de pesos de manera más estable.

Módulo ml/genetics.py: selección proporcional (I)

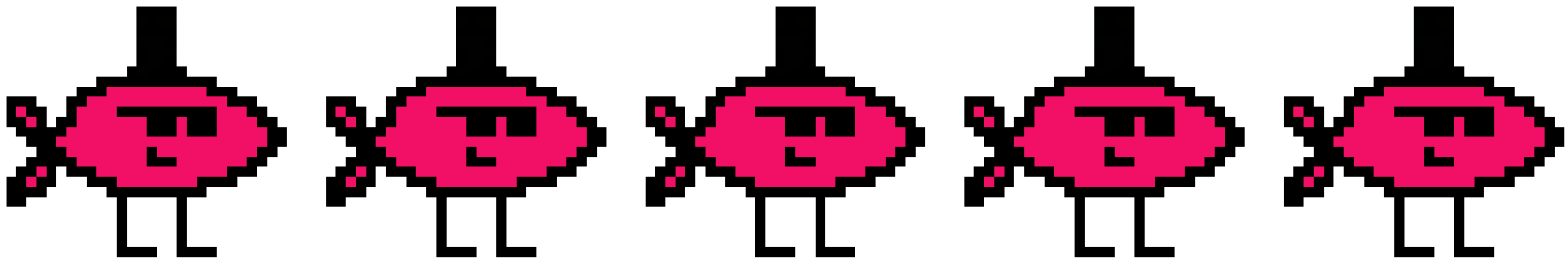
Queremos elegir padres de la población de forma que:

- Cada individuo tiene un `fitness` (un número que mide qué tan bien jugó).
- Cuanto más alto es el fitness, más probabilidad queremos darle de ser elegido.
- Pero los individuos con fitness bajo todavía deben tener una pequeña oportunidad.

Esto se conoce como método de la ruleta:

- Imaginamos una ruleta donde cada individuo ocupa un sector.
- El tamaño de cada sector es proporcional a su fitness.
- Giramos la ruleta (elegimos un número aleatorio) y vemos en qué sector cae.
- El individuo de ese sector es el padre seleccionado.

Módulo ml/genetics.py: selección proporcional - preparación de los datos (II)



La función `seleccionar_proporcional` toma:

- `pesos_poblacion`: lista de vectores de pesos (individuos).
- `fitnesses`: lista de fitness (uno por individuo).

```
import random
import math
def seleccionar_proporcional(pesos_poblacion, fitnesses):
    #...
```

Módulo ml/genetics.py: selección proporcional - preparación de los datos (III)

- `fitness_pos`: Para cada fitness `f`, calculamos `max(f, 0.0)`: si algún fitness fuera negativo, lo reemplazamos por 0. Luego elevamos al cuadrado: `(max(f, 0.0))**2` para aumentar la diferencia entre individuos "buenos" y "regulares".
- `total = sum(fitness_pos)`: `total` es la suma de todas las "porciones" de la ruleta.
- `if total == 0`: Si todos los fitness son 0, la ruleta no tiene sectores, así que simplemente elegimos un individuo al azar.

```
fitness_pos = [max(f, 0.0) ** 2 for f in fitnesses]
total = sum(fitness_pos)
if total == 0:
    return random.choice(pesos_poblacion)
```

Módulo ml/genetics.py: selección proporcional - sorteo en la ruleta (IV)

- `r = random.uniform(0, total)`: Elegimos un número aleatorio entre 0 y `total`. Es tipo "parar la ruleta" en un punto al azar del círculo.
- `acum = 0.0`: será la "suma acumulada" de los sectores mientras recorremos la población.

```
r = random.uniform(0, total)
acum = 0.0
```

Módulo ml/genetics.py: selección proporcional - sorteo en la ruleta (V)

- `for w, fit in zip(...):`
  - Recorremos, en paralelo, cada vector de pesos `w` y su fitness positivo `fit`.
  - En cada paso sumamos: `acum += fit`.
  - Cuando `acum` supera o iguala a `r`, devolvemos `w`: ese individuo ha sido seleccionado.
- `return pesos_poblacion[-1]`:
  - Es un caso de seguridad: si por redondeos no hemos devuelto antes, elegimos el último individuo.

```
for w, fit in zip(pesos_poblacion, fitness_pos):
    acum += fit
    if acum >= r:
        return w
return pesos_poblacion[-1]
```

Probabilidad de ser elegido como padre

En la función `seleccionar_proporcional` primero transformamos los valores de fitness:

$$\texttt{fitness\_pos}[i] = \max(f_i, 0)^2,$$

$$\texttt{total} = \sum_j \texttt{fitness\_pos}[j].$$

La probabilidad de que el individuo i sea elegido como padre es:

$$P(\text{individuo } i) = \frac{\texttt{fitness\_pos}[i]}{\texttt{total}} = \frac{\max(f_i, 0)^2}{\sum_j \max(f_j, 0)^2}.$$

Módulo ml/genetics.py: selección proporcional - sorteo en la ruleta (IV)

Imaginemos una población de 3 individuos:

$$f_1 = 1,\ f_2 = 2,\ f_3 = 3.$$

Tras aplicar `max(f,0)`^{2}:

$$[1,\ 4,\ 9], \qquad total = 14.$$

Elegimos $r = 5.3$.

Acumulación:

- acum = 1 $\rightarrow$ sigue
- acum = 5 $\rightarrow$ sigue
- acum = 14 $\rightarrow$ $14 \geq 5.3$: seleccionado el 3º

Mayor fitness = mayor tramo de “ruleta”.

Cruce uniforme()

La función `cruce_uniforme()` mezcla los genes (pesos) de dos padres para producir un hijo.

- `p1`, `p2`: vectores de pesos.
- `largo`: longitud mínima entre ambos.
- Para cada índice:
  - con probabilidad 0.5 se toma `p1[i]`,
  - con probabilidad 0.5 se toma `p2[i]`.
- El hijo es una combinación gen a gen de ambos padres.

```
def cruce_uniforme(p1, p2):
    largo = min(len(p1), len(p2))
    hijo = []
    for i in range(largo):
        if random.random() < 0.5:
            hijo.append(p1[i])
        else:
            hijo.append(p2[i])
    return hijo
```

Mutación de pesos: mutar()

La función `mutar` introduce pequeñas perturbaciones aleatorias en los pesos de un individuo.

- `pesos`: vector original.
- `prob_gen`: probabilidad de mutar cada gen (0.1 por defecto).
- `sigma`: desviación estándar del ruido gaussiano.
- Para cada gen: con probabilidad `prob_gen` se añade ruido $\mathcal{N}(0, \sigma^2)$.
- El resultado es un vector cercano al original pero con variación suficiente para explorar nuevas soluciones.

```
def mutar(pesos, prob_gen=0.1, sigma=0.05):
    hijo = list(pesos)
    for i in range(len(hijo)):
        if random.random() < prob_gen:
            hijo[i] += random.gauss(0.0, sigma)
    return hijo
```

Generación de nueva_generacion() (I)

La función `nueva_generacion` crea la siguiente población de pesos usando los fitness de la generación actual.

- `pesos_poblacion`: vectores de pesos actuales.
- `fitnesses`: fitness correspondiente.
- `elitismo`: número de mejores individuos copiados sin cambio.
- `prob_mut`: probabilidad de mutación por gen.
- Proceso inicial:
  - `zip` junta pesos y fitness,
  - `sort(reverse=True)` ordena por fitness descendente,
  - los mejores `elitismo` pasan directamente a la nueva población.

```
def nueva_generacion(pesos_poblacion, fitnesses,
                     prob_mut=0.1, elitismo=2):
    combinados = list(zip(pesos_poblacion, fitnesses))
    combinados.sort(key=lambda x: x[1], reverse=True)
    nuevos = [w for w, f in combinados[:elitismo]]
```

Generación de nueva_generacion() (II)

- `n`: tamaño de la población.
- Mientras `nuevos` tenga menos de `n` individuos:
- En cada iteración: seleccionar dos padres, generar un hijo mediante cruce, mutarlo si corresponde y añadirlo a `nuevos`.
- Al finalizar, `nuevos` es la nueva generación lista para la siguiente época.

```
n = len(pesos_poblacion)
while len(nuevos) < n:
    p1 = seleccionar_proporcional(pesos_poblacion, fitnesses)
    p2 = seleccionar_proporcional(pesos_poblacion, fitnesses)
    hijo = cruce_uniforme(p1, p2)
    hijo = mutar(hijo, prob_gen=prob_mut)
    nuevos.append(hijo)

return nuevos
```

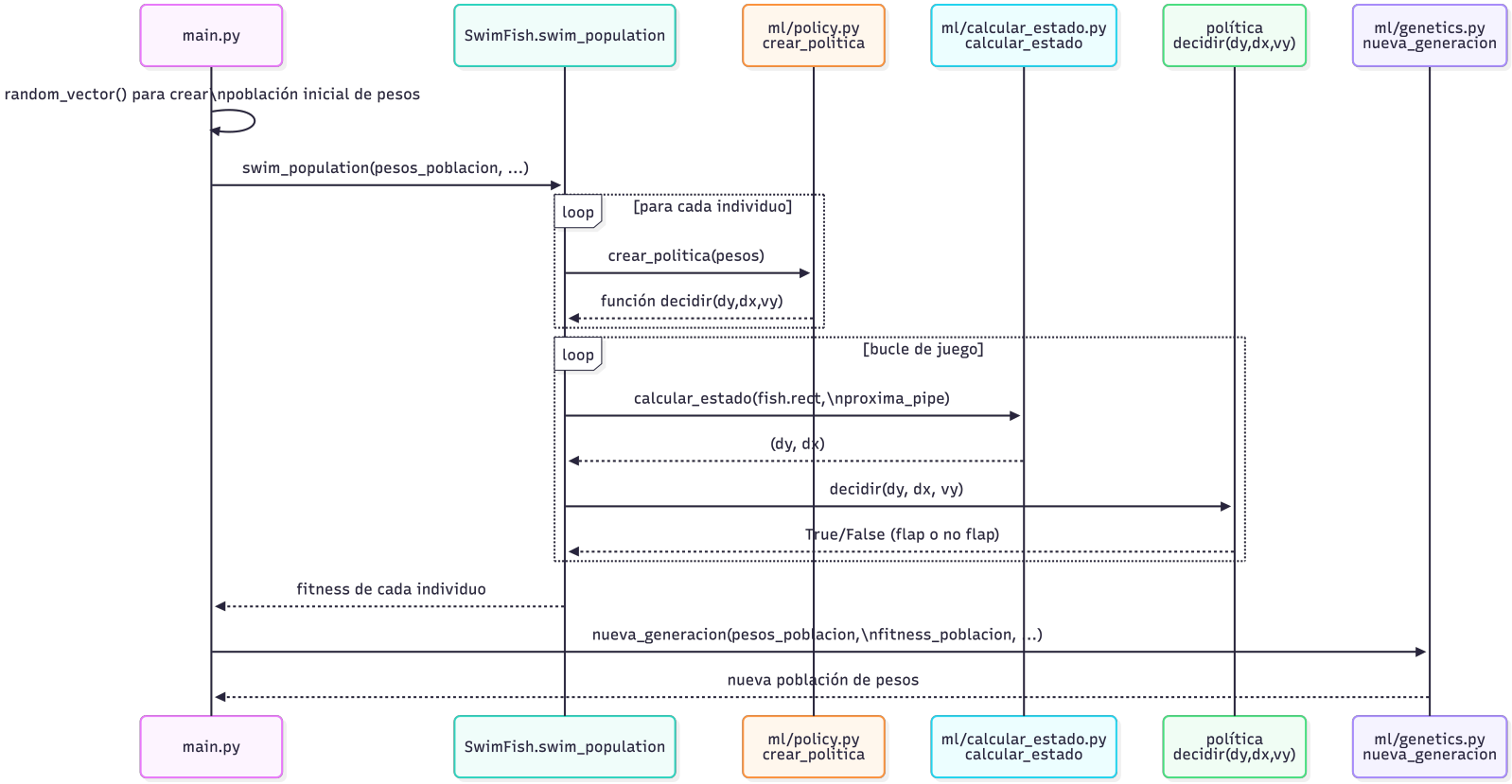
Quién llama a quién (visión general) (I)

El flujo de llamadas en modo evolutivo es:

1. `main.py`
   - crea la población inicial de pesos con `random_vector()`,
   - llama a `SwimFish.swim_population(...)`.
2. `SwimFish.swim_population`
   - para cada vector de pesos, llama a `crear_politica`,
   - dentro del bucle de juego llama a `calcular_estado` y luego a `decidir(dy,dx,vy)`.
3. Al final de la generación:
   - `swim_population` devuelve fitness por individuo,
   - `main.py` llama a `nueva_generacion` de `ml/genetics.py`.

Así se cierran las generaciones: juego $\Rightarrow$ fitness $\Rightarrow$ genética $\Rightarrow$ nuevos pesos.

Quién llama a quién (visión general) (II)



Parámetros clave del entrenamiento

Algunos parámetros importantes:

- `tam_poblacion`:
  - número de agentes por generación (por ejemplo, 100).
- `num_epocas`:
  - número máximo de generaciones (por ejemplo, 50).
- `tiempo_max`:
  - límite de pasos por generación en `swim_population`.
- `umbral_distancia`:
  - número de tuberías superadas que consideramos "suficiente" para un pez.
- `elitismo`:
  - cantidad de mejores individuos copiados sin cambios a la nueva generación.

Construcción de la población inicial en main.py

En `main.py` (modo evolutivo), la población inicial se genera así:

```
from ml.vector_w import random_vector

tam_poblacion = 100

# Generar lista de vectores de pesos aleatorios
pesos_poblacion = [random_vector() for _ in range(tam_poblacion)]
```

- Cada elemento de `pesos_poblacion` es un vector de pesos.
- Esta lista se pasa a `SwimFish.swim_population(...)` para evaluar el desempeño de cada individuo en el juego.

Bucle de entrenamiento en main.py (I)

El núcleo del entrenamiento en modo evolutivo es un bucle de generaciones:

- `num_epocas` fija el número máximo de generaciones que queremos simular (por ejemplo, 50).
- `juego_auto.generacion = generacion` guarda el número de generación actual, que se muestra en el panel de estadísticas.

```
num_epocas = 50   # número máximo de generaciones
for generacion in range(num_epocas):
    juego_auto.generacion = generacion
    #...
```

Bucle de entrenamiento en main.py (II)

- El bucle `for generacion in range(num_epocas)` recorre las generaciones 0, 1, 2, ... hasta que se cumpla un criterio de parada.

```
for generacion in range(num_epocas):
    juego_auto.generacion = generacion
    # Simular toda la población en el juego
    pesos_poblacion, fitness_poblacion, estado = \
        juego_auto.swim_population(pesos_poblacion,
                                   tiempo_max=400,
                                   umbral_distancia=50)
    if estado in ('MENU', 'QUIT'):
        break  # el usuario interrumpió la simulación
    # Crear nueva generación de pesos a partir de fitness
    pesos_poblacion = nueva_generacion(pesos_poblacion,
                                       fitness_poblacion,
                                       elitismo=5)
```

Bucle de entrenamiento en main.py (III)

Dentro de cada iteración del bucle de generaciones suceden dos pasos:

1. Simulación de la generación actual
   - `swim_population` hace volar a todos los peces de la población con sus políticas respectivas.
   - Calcula el `fitness` de cada individuo.
   - Devuelve: `pesos_poblacion`: la lista de pesos de la generación, `fitness_poblacion`: lista de fitness, `estado`: indica si el usuario volvió al menú o cerró el juego.

```
pesos_poblacion, fitness_poblacion, estado = \
    juego_auto.swim_population(pesos_poblacion,
                               tiempo_max=400,
                               umbral_distancia=50)
```

Bucle de entrenamiento en main.py (IV)

2. Creación de la nueva generación
   - `nueva_generacion` usa los `fitness` para: copiar los 5 mejores individuos (elitismo), generar el resto mediante selección, cruce y mutación.
   - La nueva `pesos_poblacion` será la entrada para la siguiente generación del bucle.

```
pesos_poblacion = nueva_generacion(pesos_poblacion,
                                   fitness_poblacion,
                                   elitismo=5)
```

Así se encadenan juego y genética generación tras generación:

$$\text{pesos} \Rightarrow \text{juego} \Rightarrow \text{fitness} \Rightarrow \text{nuevos pesos}.$$

swim_population(): inicialización (I)

Al entrar en `swim_population` se preparan las estructuras de la generación:

```
def swim_population(self, pesos_poblacion, tiempo_max, umbral_distanci
    num_agentes = len(pesos_poblacion)

    peces = []
    politicas = []
    vivos = [True] * num_agentes
    scores = [0] * num_agentes
    time_alive = [0] * num_agentes
```

- `num_agentes`: cantidad de peces en la población.
- `peces`, `politicas`: listas de objetos `Fish` y funciones `decidir`.
- `vivos`, `scores`, `time_alive`: estado, tuberías superadas y tiempo de vida de cada pez.

swim_population(): creación de peces y políticas (II)

Luego se instancia un pez y una política por cada vector de pesos:

```
for pesos in pesos_poblacion:
    fish = Fish(self.screen_w // 4,
                self.screen_h // 2,
                (50, 35),
                "../data/img/fish.png")
    decidir, _ = crear_politica(pesos)

    peces.append(fish)
    politicas.append(decidir)

# Inicializar tuberías y mundo común...
```

- Para cada `pesos` se crea un `Fish` en una posición inicial fija.
- `crear_politica(pesos)` devuelve la función `decidir(dy,dx,vy)` para ese agente.
- Todos los peces comparten el mismo “mundo” (tuberías, fondo, tiempo).

swim_population(): bucle principal (III)

Dentro de `swim_population` se ejecuta el bucle principal de simulación:

```
paso = 0
running = True
while running:
    paso += 1
    self._actualizar_fondo()
    self._actualizar_tuberias()
    for i, fish in enumerate(peces):
        if not vivos[i]:
            continue
        dy, dx = calcular_estado(fish.rect,
                                 self.proxima_tuberia())
        vy = fish.velocity
        if politicas[i](dy, dx, vy):
            fish.flap()
        fish.update()
        self._actualizar_estado_del_pez(i, fish, scores, vivos)
        time_alive[i] += 1
```

swim_population(): qué pasa en cada paso (IV)

En cada iteración del bucle:

1. Se incrementa `paso` y se actualiza el mundo (`_actualizar_fondo`, `_actualizar_tuberias`).
2. Para cada pez vivo:
   - se calcula el estado relativo (dy, dx, vy);
   - la política `politicas[i]` decide si hacer `flap()`;
   - `fish.update()` aplica física (gravedad, movimiento);
   - `_actualizar_estado_del_pez` detecta choques y suma puntos si pasa tuberías;
   - `time_alive[i]` aumenta en 1.

Condiciones de parada de la simulación

El bucle de `swim_population` termina cuando:

- Todos los peces han muerto: es decir, no queda ningún `vivo[i] == True`.
- O bien se alcanza el tiempo máximo: paso >= tiempo_max
- O bien una cantidad suficiente de peces ha alcanzado `umbral_distancia`:
  - por ejemplo, `scores[i] >= umbral_distancia` para un cierto número de individuos.
  - en ese caso puede considerarse que la generación ya "aprendió lo suficiente".

Al parar, se calcula el fitness de cada agente.