Línea 1: import pygame, sys, random, time

- Importa varios módulos de Python:

  - `pygame`: Es la biblioteca principal para el desarrollo de juegos 2D, proporcionando funcionalidades para gráficos, sonido, entrada de usuario, etc.

  - `sys`: Proporciona acceso a algunas variables y funciones que interactúan fuertemente con el intérprete de Python, como `sys.exit()` para cerrar el programa.

  - `random`: Proporciona funciones para generar números aleatorios, utilizado aquí para la generación del laberinto y el movimiento de los enemigos.

  - `time`: Proporciona funciones relacionadas con el tiempo, utilizado aquí para el cronómetro del juego.

Línea 2: (En blanco)

- Línea en blanco: Separa la sección de importaciones de las constantes de configuración, mejorando la legibilidad.

Línea 3: # Configuración general

- Comentario que indica que las siguientes líneas definen parámetros globales para el juego.

Línea 4: WIDTH, HEIGHT = 640, 480

- Define dos variables globales, `WIDTH` y `HEIGHT`, y les asigna los valores 640 y 480 respectivamente. Estas representan las dimensiones en píxeles de la ventana del juego.

Línea 5: TILE_SIZE = 32

- Define una variable global `TILE_SIZE` y le asigna el valor 32. Esto representa el tamaño en píxeles de cada celda o "baldosa" del laberinto.

Línea 6: ROWS, COLS = HEIGHT // TILE_SIZE, WIDTH // TILE_SIZE

- Define dos variables globales, `ROWS` y `COLS`.

  - `HEIGHT // TILE_SIZE`: Calcula el número de filas en la cuadrícula del laberinto dividiendo la altura total de la pantalla por el tamaño de una celda, usando división entera (`//`).

  - `WIDTH // TILE_SIZE`: Calcula el número de columnas en la cuadrícula del laberinto dividiendo el ancho total de la pantalla por el tamaño de una celda, usando división entera.

- Estas variables determinarán las dimensiones de la matriz del laberinto.

Línea 7: (En blanco)

- Línea en blanco: Separa las constantes de dimensiones de las constantes de color.

Línea 8: # Colores

- Comentario que indica que las siguientes líneas definen las tuplas de colores RGB que se usarán en el juego.

Línea 9: BG_COLOR = (30, 30, 30)

- Define la variable global `BG_COLOR` como una tupla RGB `(30, 30, 30)`, que representa un color gris oscuro para el fondo de la pantalla.

Línea 10: WALL_COLOR = (50, 50, 50)

- Define la variable global `WALL_COLOR` como una tupla RGB `(50, 50, 50)`, un gris ligeramente más claro que el fondo, para las paredes del laberinto.

Línea 11: GOAL_COLOR = (0, 255, 0)

- Define la variable global `GOAL_COLOR` como una tupla RGB `(0, 255, 0)`, que representa el color verde puro, para la celda objetivo o salida del laberinto.

Línea 12: ENEMY_COLOR = (255, 80, 80)

- Define la variable global `ENEMY_COLOR` como una tupla RGB `(255, 80, 80)`, un color rojo, para los enemigos.

Línea 13: (En blanco)

- Línea en blanco: Separa las constantes de color de la sección de inicialización de Pygame.

Línea 14: # Inicializar

- Comentario que indica que las siguientes líneas se encargan de poner en marcha los módulos de Pygame.

Línea 15: pygame.init()

- Llama a la función `pygame.init()`. Esta función inicializa todos los módulos de Pygame que son necesarios para que la biblioteca funcione correctamente (como el módulo de video, fuente, sonido, etc.).

Línea 16: screen = pygame.display.set_mode((WIDTH, HEIGHT))

- Llama a `pygame.display.set_mode()` para crear la ventana principal del juego.

  - `(WIDTH, HEIGHT)`: Es una tupla que especifica las dimensiones de la ventana, utilizando las constantes definidas previamente.

- El objeto `Surface` devuelto, que representa la pantalla donde se dibujará todo, se asigna a la variable `screen`.

Línea 17: pygame.display.set_caption("Laberinto con Muñeco Animado, Enemigos y cronómetro")

- Llama a `pygame.display.set_caption()` para establecer el título que aparecerá en la barra de título de la ventana del juego.

Línea 18: clock = pygame.time.Clock()

- Crea un objeto `Clock` de Pygame y lo asigna a la variable `clock`. Este objeto se utilizará más tarde para controlar la velocidad del juego (fotogramas por segundo).

Línea 19: font = pygame.font.SysFont(None, 32)

- Crea un objeto `Font` de Pygame y lo asigna a la variable `font`.
  - `None`: Indica que se utilice la fuente predeterminada del sistema.
  - `32`: Es el tamaño de la fuente en puntos.
- Este objeto `font` se usará para renderizar texto en la pantalla (como el cronómetro y mensajes de victoria/derrota).

Línea 20: (En blanco)

- Línea en blanco: Separa la inicialización de Pygame de la definición de la función del generador de laberintos.

Línea 21: # ------- GENERADOR DE LABERINTO -------

- Comentario que actúa como un encabezado para la sección de código que genera el laberinto.

Línea 22: def generate_maze(rows, cols):

- Define una función llamada `generate_maze` que toma dos argumentos: `rows` (número de filas) y `cols` (número de columnas) para el laberinto.

Línea 23:  maze = [[1 for _ in range(cols)] for _ in range(rows)]

- Dentro de `generate_maze`: Inicializa una variable `maze` como una lista de listas (una matriz 2D).
  - `[[1 for _ in range(cols)] for _ in range(rows)]`: Utiliza una lista por comprensión anidada.
    - `for _ in range(rows)`: Crea `rows` número de filas.

- `[1 for _ in range(cols)]`: Para cada fila, crea una lista de `cols` número de elementos, todos inicializados a `1`.

- En este contexto, `1` representa una pared. Así, el laberinto comienza como una rejilla sólida de paredes.

Línea 24: visited = set()

- Dentro de `generate_maze`: Inicializa una variable `visited` como un conjunto vacío. Este conjunto se usará para llevar un registro de las celdas que ya han sido "visitadas" por el algoritmo generador para tallar caminos.

Línea 25: (En blanco)

- Dentro de `generate_maze`: Línea en blanco para separar la inicialización de variables de la definición de la función anidada `visit`.

Línea 26: def visit(r, c):

- Dentro de `generate_maze`: Define una función anidada llamada `visit` que toma dos argumentos: `r` (fila actual) y `c` (columna actual). Esta función implementa el algoritmo recursivo para tallar los caminos del laberinto (similar a una búsqueda en profundidad o DFS).

Línea 27: if 0 <= r < rows and 0 <= c < cols:

- Dentro de `visit`: Comprueba si la celda actual `(r, c)` está dentro de los límites del laberinto.

Línea 28: visited.add((r, c))

- Dentro de `visit` (y dentro del `if`): Si la celda es válida, la añade al conjunto `visited`.

Línea 29: maze[r][c] = 0

- Dentro de `visit`: Marca la celda actual `(r, c)` en la matriz `maze` con el valor `0`, lo que significa que ahora es un camino (no una pared).

Línea 30: dirs = [(0, 2), (2, 0), (0, -2), (-2, 0)]

- Dentro de `visit`: Define una lista `dirs` que contiene tuplas representando las cuatro posibles direcciones para moverse, saltando una celda (para dejar espacio para la pared que se derribará).
  - `(0, 2)`: Moverse 2 celdas a la derecha.
  - `(2, 0)`: Moverse 2 celdas hacia abajo.
  - `(0, -2)`: Moverse 2 celdas a la izquierda.
  - `(-2, 0)`: Moverse 2 celdas hacia arriba.

Línea 31: random.shuffle(dirs)

- Dentro de `visit`: Llama a `random.shuffle(dirs)` para mezclar aleatoriamente el orden de las direcciones en la lista `dirs`. Esto asegura que la generación del laberinto sea aleatoria y no siga siempre el mismo patrón.

Línea 32: for dr, dc in dirs:

- Dentro de `visit`: Inicia un bucle `for` que itera sobre cada tupla `(dr, dc)` (delta fila, delta columna) en la lista mezclada `dirs`.

Línea 33: nr, nc = r + dr, c + dc

- Dentro de `visit` (y dentro del `for`): Calcula las coordenadas de la "nueva celda" `(nr, nc)` a la que se intentará mover, que está dos pasos en la dirección `(dr, dc)` desde la celda actual `(r, c)`.

Línea 34: if 0 <= nr < rows and 0 <= nc < cols and (nr, nc) not in visited:

- Dentro de `visit` (y dentro del `for`): Comprueba tres condiciones para la "nueva celda" `(nr, nc)`:

- `0 <= nr < rows`: Que la fila `nr` esté dentro de los límites del laberinto.
- `0 <= nc < cols`: Que la columna `nc` esté dentro de los límites del laberinto.
- `(nr, nc) not in visited`: Que la celda `(nr, nc)` no haya sido visitada previamente.

Línea 35: wall_r, wall_c = r + dr // 2, c + dc // 2

- Dentro de `visit` (y dentro del `if` anidado): Si todas las condiciones anteriores son verdaderas, calcula las coordenadas de la pared `(wall_r, wall_c)` que se encuentra entre la celda actual `(r, c)` y la nueva celda `(nr, nc)`. Se usa división entera `// 2` porque `dr` y `dc` son `±2`.

Línea 36: maze[wall_r][wall_c] = 0

- Dentro de `visit`: "Derriba" la pared intermedia marcándola como un camino (`0`) en la matriz `maze`.

Línea 37: visit(nr, nc)

- Dentro de `visit`: Realiza una llamada recursiva a la función `visit` para la nueva celda `(nr, nc)`, continuando el proceso de tallado desde esa celda.

Línea 38: (En blanco)

- Dentro de `generate_maze`: Línea en blanco para separar la definición de `visit` de su primera llamada.

Línea 39: visit(1, 1)

- Dentro de `generate_maze`: Realiza la primera llamada a la función `visit` para iniciar la generación del laberinto desde la celda `(1, 1)`. Se usa (1,1) en lugar de (0,0) para asegurar que haya un borde de paredes alrededor del laberinto si la generación no alcanza todos los bordes.

Línea 40: maze[1][1] = 0

- Dentro de `generate_maze`: Asegura que la celda de inicio `(1, 1)` sea un camino. Esto es redundante si `visit(1,1)` ya lo hace, pero no causa daño.

Línea 41:  maze[rows - 2][cols - 2] = 2  # salida

- Dentro de `generate_maze`: Establece la celda en la esquina inferior derecha (pero una celda hacia adentro desde el borde, `rows-2`, `cols-2`) como la salida del laberinto, marcándola con el valor `2`.

- El comentario `# salida` aclara el propósito de esta línea.

Línea 42:  return maze

- Dentro de `generate_maze`: Devuelve la matriz `maze` completa, que ahora contiene el laberinto generado.

Línea 43: (En blanco)

- Línea en blanco: Separa la definición de la función `generate_maze` de su llamada.

Línea 44: maze = generate_maze(ROWS, COLS)

- Llama a la función `generate_maze` pasándole las constantes globales `ROWS` y `COLS` (calculadas previamente) y asigna el laberinto resultante a la variable global `maze`.

Línea 45: (En blanco)

- Línea en blanco: Separa la generación del laberinto de la sección del jugador.

Línea 46: # ------- JUGADOR -------

- Comentario que actúa como un encabezado para la sección de código relacionada con el jugador.

Línea 47: player_pos = [1, 1]

- Define una variable global `player_pos` como una lista `[1, 1]`. Esto representa la posición inicial del jugador en la cuadrícula del laberinto (columna 1, fila 1), que coincide con el punto de inicio de la generación del laberinto.

Línea 48: player_frame = 0

- Define una variable global `player_frame` y la inicializa a `0`. Se usará para controlar qué frame de la animación del jugador se muestra.

Línea 49: player_frames = []

- Define una variable global `player_frames` como una lista vacía. Esta lista almacenará las diferentes imágenes (superficies de Pygame) para la animación del jugador.

Línea 50: (En blanco)

- Línea en blanco: Separa las variables del jugador de la función que crea sus frames de animación.

Línea 51: def make_player_frames():

- Define una función llamada `make_player_frames` que no toma argumentos. Su propósito es crear las imágenes (frames) para la animación del jugador.

Línea 52:  for i in range(1):

- Dentro de `make_player_frames`: Inicia un bucle `for` que itera sobre `range(1)`. Esto significa que el bucle se ejecutará solo una vez, con `i` tomando el valor `0`. Por lo tanto, solo se creará un frame de animación para el jugador.

Línea 53:  surf = pygame.Surface((TILE_SIZE, TILE_SIZE), pygame.SRCALPHA)

- Dentro de `make_player_frames` (y dentro del `for`): Crea un nuevo objeto `Surface` de Pygame y lo asigna a `surf`.

  - `(TILE_SIZE, TILE_SIZE)`: Especifica que la superficie tendrá el mismo tamaño que una celda del laberinto.

  - `pygame.SRCALPHA`: Es una bandera que indica que la superficie debe admitir transparencia por píxel (canal alfa). Esto es útil si se quieren dibujar formas no rectangulares sin un fondo sólido.

Línea 54: pygame.draw.rect(surf, (100 + i*30, 200, 255), (6, 6, 20, 24), border_radius=6)

- Dentro de `make_player_frames`: Dibuja un rectángulo sobre la superficie `surf`.

  - `surf`: La superficie destino donde se dibujará.

  - `(100 + i*30, 200, 255)`: El color del rectángulo. Como `i` es `0`, el color será `(100, 200, 255)`, un azul claro.

  - `(6, 6, 20, 24)`: Una tupla `(x, y, ancho, alto)` que define la posición y dimensiones del rectángulo dentro de la superficie `surf`.

  - `border_radius=6`: Dibuja el rectángulo con esquinas redondeadas, con un radio de 6 píxeles.

Línea 55: pygame.draw.circle(surf, (255, 255, 255), (16, 10), 6)

- Dentro de `make_player_frames`: Dibuja un círculo sobre la superficie `surf`, encima del rectángulo.

  - `surf`: La superficie destino.

  - `(255, 255, 255)`: El color del círculo (blanco).

  - `(16, 10)`: Las coordenadas `(x, y)` del centro del círculo dentro de la superficie `surf`.

  - `6`: El radio del círculo en píxeles.

- Estas dos formas (rectángulo redondeado y círculo) componen la apariencia del jugador.

Línea 56:  player_frames.append(surf)

- Dentro de `make_player_frames`: Añade la superficie `surf` (con el dibujo del jugador) a la lista `player_frames`.

Línea 57: (En blanco)

- Línea en blanco: Separa la definición de `make_player_frames` de su llamada.

Línea 58: make_player_frames()

- Llama a la función `make_player_frames()` para poblar la lista `player_frames` con la(s) imagen(es) del jugador.

Línea 59: (En blanco)

- Línea en blanco: Separa la sección del jugador de la sección de los enemigos.

Línea 60: # -------- ENEMIGOS -------

- Comentario que actúa como un encabezado para la sección de código relacionada con los enemigos.

Línea 61: class Enemy:

- Define una clase llamada `Enemy`. Las instancias de esta clase representarán a los enemigos individuales en el juego.

Línea 62:  def __init__(self, x, y):

- Dentro de la clase `Enemy`: Define el método constructor `__init__`. Se llama cuando se crea una nueva instancia de `Enemy`.

  - `self`: Referencia a la instancia que se está creando.

- `x`, `y`: Coordenadas iniciales (en términos de celdas de la cuadrícula, no píxeles) para el enemigo.

Línea 63: self.pos = [x, y]

- Dentro de `__init__`: Inicializa un atributo de instancia `self.pos` como una lista `[x, y]`, almacenando la posición actual del enemigo en la cuadrícula.

Línea 64: self.path = random.choice([(-1, 0), (1, 0), (0, -1), (0, 1)])

- Dentro de `__init__`: Inicializa un atributo de instancia `self.path`.

- `random.choice([(-1, 0), (1, 0), (0, -1), (0, 1)])`: Elige aleatoriamente una tupla de una lista de cuatro posibles direcciones de movimiento.

  - `(-1, 0)`: Izquierda (disminuir columna).

  - `(1, 0)`: Derecha (aumentar columna).

  - `(0, -1)`: Arriba (disminuir fila).

  - `(0, 1)`: Abajo (aumentar fila).

- Esta será la dirección inicial en la que el enemigo intentará moverse.

Línea 65: (En blanco)

- Dentro de la clase `Enemy`: Línea en blanco para separar métodos.

Línea 66: def move(self):

- Dentro de la clase `Enemy`: Define un método llamado `move` que no toma argumentos además de `self`. Este método se encargará de actualizar la posición del enemigo.

Línea 67: nx, ny = self.pos[0] + self.path[0], self.pos[1] + self.path[1]

- Dentro de `move`: Calcula la "nueva posición" `(nx, ny)` tentativa del enemigo sumando su dirección de movimiento actual (`self.path`) a su posición actual (`self.pos`).

  - `self.pos[0]` es la columna actual, `self.path[0]` es el cambio en columna.

  - `self.pos[1]` es la fila actual, `self.path[1]` es el cambio en fila.

Línea 68: if 0 <= nx < COLS and 0 <= ny < ROWS and maze[ny][nx] != 1:

- Dentro de `move`: Comprueba si la nueva posición `(nx, ny)` es válida:

  - `0 <= nx < COLS`: Que la nueva columna esté dentro de los límites del laberinto.

  - `0 <= ny < ROWS`: Que la nueva fila esté dentro de los límites del laberinto.

  - `maze[ny][nx] != 1`: Que la celda en la nueva posición en la matriz `maze` no sea una pared (valor `1`). Los enemigos pueden moverse por caminos (`0`) o la celda de meta (`2`).

Línea 69: self.pos = [nx, ny]

- Dentro de `move` (y dentro del `if`): Si la nueva posición es válida, actualiza la posición del enemigo `self.pos` a `[nx, ny]`.

Línea 70: else:

- Dentro de `move`: Si la condición del `if` no se cumple (es decir, el movimiento no es válido porque choca con un límite o una pared).

Línea 71: self.path = random.choice([(0, 1), (1, 0), (0, -1), (-1, 0)])

- Dentro de `move` (y dentro del `else`): Elige una nueva dirección de movimiento aleatoria para `self.path`. Las direcciones aquí están ordenadas de manera diferente a la inicialización, pero son las mismas cuatro opciones. Esto hace que el enemigo cambie de dirección cuando no puede avanzar.

Línea 72: (En blanco)

- Dentro de la clase `Enemy`: Línea en blanco para separar métodos.

Línea 73: def draw(self):

- Dentro de la clase `Enemy`: Define un método llamado `draw` que no toma argumentos además de `self`. Este método se encargará de dibujar el enemigo en la pantalla.

Línea 74: x, y = self.pos[0]*TILE_SIZE, self.pos[1]*TILE_SIZE

- Dentro de `draw`: Calcula las coordenadas en píxeles `(x, y)` de la esquina superior izquierda de la celda del enemigo.

  - `self.pos[0]*TILE_SIZE`: Multiplica la columna del enemigo por `TILE_SIZE`.

  - `self.pos[1]*TILE_SIZE`: Multiplica la fila del enemigo por `TILE_SIZE`.

Línea 75: pygame.draw.rect(screen, ENEMY_COLOR, (x+4, y+4, TILE_SIZE-8, TILE_SIZE-8), border_radius=5)

- Dentro de `draw`: Dibuja un rectángulo en la pantalla (`screen`) para representar al enemigo.

  - `screen`: La superficie principal del juego.

  - `ENEMY_COLOR`: El color definido para los enemigos.

  - `(x+4, y+4, TILE_SIZE-8, TILE_SIZE-8)`: Define el rectángulo a dibujar.

    - `x+4, y+4`: Coordenadas de la esquina superior izquierda del rectángulo, con un pequeño desplazamiento de 4 píxeles desde el borde de la celda para hacerlo parecer más pequeño que la celda.

    - `TILE_SIZE-8, TILE_SIZE-8`: Ancho y alto del rectángulo, 8 píxeles más pequeño que `TILE_SIZE` (4 píxeles de margen por cada lado).

  - `border_radius=5`: Dibuja el rectángulo con esquinas redondeadas, con un radio de 5 píxeles.

Línea 76: (En blanco)

- Línea en blanco: Separa la definición de la clase `Enemy` de la creación de instancias.

Línea 77: enemies = [Enemy(COLS - 3, ROWS - 3), Enemy(3, ROWS - 4)]

- Crea una lista global llamada `enemies`.

- Contiene dos instancias de la clase `Enemy`:

  - `Enemy(COLS - 3, ROWS - 3)`: Un enemigo colocado cerca de la esquina inferior derecha.

  - `Enemy(3, ROWS - 4)`: Un enemigo colocado cerca de la esquina inferior izquierda.

Línea 78: (En blanco)

- Línea en blanco: Separa la creación de enemigos de la sección de funciones auxiliares.

Línea 79: # FUNCIONES

- Comentario que indica que las siguientes definiciones son funciones auxiliares para el juego.

Línea 80: def draw_maze():

- Define una función llamada `draw_maze` que no toma argumentos. Su propósito es dibujar el laberinto en la pantalla.

Línea 81:     for y in range(ROWS):

- Dentro de `draw_maze`: Inicia un bucle `for` que itera sobre cada fila `y` del laberinto, desde `0` hasta `ROWS-1`.

Línea 82:         for x in range(COLS):

- Dentro de `draw_maze` (y dentro del bucle `y`): Inicia un bucle `for` anidado que itera sobre cada columna `x` del laberinto, desde `0` hasta `COLS-1`.

Línea 83: tile = maze[y][x]

- Dentro de `draw_maze` (y dentro del bucle `x`): Obtiene el valor de la celda actual `(y, x)` de la matriz global `maze` y lo asigna a la variable `tile`.

Línea 84: rect = pygame.Rect(x*TILE_SIZE, y*TILE_SIZE, TILE_SIZE, TILE_SIZE)

- Dentro de `draw_maze`: Crea un objeto `pygame.Rect` que representa el rectángulo de la celda actual en coordenadas de píxeles.

  - `x*TILE_SIZE, y*TILE_SIZE`: Coordenadas de la esquina superior izquierda de la celda.

  - `TILE_SIZE, TILE_SIZE`: Ancho y alto de la celda.

- El objeto `Rect` se asigna a la variable `rect`.

Línea 85: if tile == 1:

- Dentro de `draw_maze`: Comprueba si el valor de `tile` es `1`. Si es así, la celda es una pared.

Línea 86: pygame.draw.rect(screen, WALL_COLOR, rect)

- Dentro de `draw_maze` (y dentro del `if tile == 1`): Dibuja un rectángulo en la `screen` con el color `WALL_COLOR` usando el `rect` definido para la celda actual.

Línea 87: elif tile == 2:

- Dentro de `draw_maze`: Si la celda no es una pared (`tile != 1`), comprueba si `tile` es `2`. Si es así, la celda es la meta.

Línea 88: pygame.draw.rect(screen, GOAL_COLOR, rect)

- Dentro de `draw_maze` (y dentro del `elif tile == 2`): Dibuja un rectángulo en la `screen` con el color `GOAL_COLOR` usando el `rect` definido para la celda actual.

- Las celdas de camino (donde `tile == 0`) no se dibujan explícitamente aquí; simplemente se verán del color de fondo `BG_COLOR` que se usa para rellenar la pantalla al inicio de cada fotograma.

Línea 89: (En blanco)

- Línea en blanco: Separa la función `draw_maze` de `draw_player`.

Línea 90: def draw_player():

- Define una función llamada `draw_player` que no toma argumentos. Su propósito es dibujar al jugador en la pantalla.

Línea 91:  global player_frame

- Dentro de `draw_player`: Declara que se va a utilizar (y potencialmente modificar) la variable global `player_frame`.

Línea 92:  frame = player_frames[player_frame // 8 % len(player_frames)]

- Dentro de `draw_player`: Selecciona el frame de animación actual para el jugador.

  - `len(player_frames)`: Obtiene la cantidad de frames de animación disponibles para el jugador (actualmente 1).

  - `player_frame // 8`: Realiza una división entera de `player_frame` por 8. Esto hace que la animación cambie de frame cada 8 fotogramas del juego.

  - `% len(player_frames)`: Usa el operador módulo para asegurarse de que el índice resultante esté dentro del rango válido de la lista `player_frames`. Si `len(player_frames)` es 1, el resultado del módulo siempre será 0.

  - `player_frames[...]`: Accede al frame (superficie de Pygame) correspondiente de la lista `player_frames`.

- El frame seleccionado se asigna a la variable `frame`.

Línea 93: px = player_pos[0]*TILE_SIZE

- Dentro de `draw_player`: Calcula la coordenada x en píxeles (`px`) de la esquina superior izquierda de la celda del jugador, multiplicando la columna del jugador (`player_pos[0]`) por `TILE_SIZE`.

Línea 94: py = player_pos[1]*TILE_SIZE

- Dentro de `draw_player`: Calcula la coordenada y en píxeles (`py`) de la esquina superior izquierda de la celda del jugador, multiplicando la fila del jugador (`player_pos[1]`) por `TILE_SIZE`.

Línea 95: screen.blit(frame, (px, py))

- Dentro de `draw_player`: Dibuja (copia) la superficie del frame del jugador (`frame`) en la pantalla principal (`screen`) en la posición de píxeles `(px, py)`. `blit` significa Block Image Transfer.

Línea 96: (En blanco)

- Línea en blanco: Separa la función `draw_player` de `move_player`.

Línea 97: def move_player(dx, dy):

- Define una función llamada `move_player` que toma dos argumentos: `dx` (cambio en la coordenada x/columna) y `dy` (cambio en la coordenada y/fila).

Línea 98: new_x = player_pos[0] + dx

- Dentro de `move_player`: Calcula la nueva columna potencial (`new_x`) del jugador sumando `dx` a la columna actual del jugador (`player_pos[0]`).

Línea 99: new_y = player_pos[1] + dy

- Dentro de `move_player`: Calcula la nueva fila potencial (`new_y`) del jugador sumando `dy` a la fila actual del jugador (`player_pos[1]`).

Línea 100:  if 0 <= new_x < COLS and 0 <= new_y < ROWS:

- Dentro de `move_player`: Comprueba si la nueva posición (`new_x`, `new_y`) está dentro de los límites del laberinto (columnas de 0 a `COLS-1` y filas de 0 a `ROWS-1`).

Línea 101:  if maze[new_y][new_x] != 1:

- Dentro de `move_player` (y dentro del `if` de límites): Si la nueva posición está dentro de los límites, comprueba si la celda correspondiente en la matriz `maze` (`maze[new_y][new_x]`) no es una pared (valor `1`).

Línea 102:  player_pos[0] = new_x

- Dentro de `move_player` (y dentro del `if` de no pared): Si la nueva celda no es una pared, actualiza la columna del jugador (`player_pos[0]`) a `new_x`.

Línea 103:  player_pos[1] = new_y

- Dentro de `move_player`: Actualiza la fila del jugador (`player_pos[1]`) a `new_y`.

Línea 104: (En blanco)

- Línea en blanco: Separa la definición de `move_player` de la función principal del juego.

Línea 105: # LOOP PRINCIPAL

- Comentario que indica el inicio del bucle principal del juego.

Línea 106: def main():

- Define la función principal del juego llamada `main`.

Línea 107:  start_time = time.time()

- Dentro de `main`: Obtiene el tiempo actual en segundos desde la Época (un punto fijo en el pasado, usualmente 1 de enero de 1970) usando `time.time()` y lo almacena en `start_time`. Esto marca el inicio del juego para el cronómetro.

Línea 108:  win = False

- Dentro de `main`: Inicializa una variable booleana `win` a `False`. Esta bandera se usará para rastrear si el jugador ha ganado.

Línea 109:  game_over = False

- Dentro de `main`: Inicializa una variable booleana `game_over` a `False`. Esta bandera se usará para rastrear si el juego ha terminado (ya sea por ganar o perder).

Línea 110: (En blanco)

- Dentro de `main`: Línea en blanco para separar la inicialización de variables del bucle principal.

Línea 111:  while True:

- Dentro de `main`: Inicia un bucle `while True` que constituye el bucle principal del juego. Este bucle se ejecutará continuamente hasta que se rompa explícitamente (por ejemplo, cuando el juego termina).

Línea 112:  screen.fill(BG_COLOR)

- Dentro del bucle `while`: Rellena toda la `screen` con el color `BG_COLOR`. Esto limpia la pantalla en cada fotograma antes de redibujar los elementos del juego.

Línea 113:  draw_maze()

- Dentro del bucle `while`: Llama a la función `draw_maze()` para dibujar el laberinto en la pantalla.

Línea 114: draw_player()

- Dentro del bucle `while`: Llama a la función `draw_player()` para dibujar al jugador en su posición actual.

Línea 115: (En blanco)

- Dentro del bucle `while`: Línea en blanco para separar el dibujo del jugador del dibujo de los enemigos.

Línea 116: for enemy in enemies:

- Dentro del bucle `while`: Inicia un bucle `for` que itera sobre cada objeto `enemy` en la lista global `enemies`.

Línea 117: enemy.draw()

- Dentro del bucle `while` (y dentro del `for enemy`): Llama al método `draw()` de cada objeto `enemy` para dibujarlo en la pantalla.

Línea 118: (En blanco)

- Dentro del bucle `while`: Línea en blanco para separar el dibujo de los enemigos de la lógica del cronómetro.

Línea 119: # Cronómetro

- Dentro del bucle `while`: Comentario que indica la sección del código para el cronómetro.

Línea 120: elapsed = time.time() - start_time

- Dentro del bucle `while`: Calcula el tiempo transcurrido (`elapsed`) desde el inicio del juego restando `start_time` del tiempo actual `time.time()`.

Línea 121: timer_text = font.render(f"Tiempo: {elapsed:.1f}s", True, (255, 255, 255))

- Dentro del bucle `while`: Renderiza el texto del cronómetro.

  - `font.render()`: Método del objeto `font` para crear una nueva superficie con texto dibujado en ella.

  - `f"Tiempo: {elapsed:.1f}s"`: Una f-string que formatea el texto a mostrar. `elapsed:.1f` formatea el tiempo transcurrido como un número de punto flotante con un decimal.

  - `True`: Activa el antialiasing para que el texto se vea más suave.

  - `(255, 255, 255)`: El color del texto (blanco).

- La superficie resultante con el texto se asigna a `timer_text`.

Línea 122: screen.blit(timer_text, (10, 10))

- Dentro del bucle `while`: Dibuja la superficie `timer_text` en la `screen` en la posición de píxeles `(10, 10)` (esquina superior izquierda, con un pequeño margen).

Línea 123: (En blanco)

- Dentro del bucle `while`: Línea en blanco para separar el cronómetro de la lógica de victoria/derrota.

Línea 124: # Verificar condiciones de victoria o derrota

- Dentro del bucle `while`: Comentario que indica la sección para comprobar si el juego ha terminado.

Línea 125: if maze[player_pos[1]][player_pos[0]] == 2:

- Dentro del bucle `while`: Comprueba si el valor de la celda en la posición actual del jugador (`player_pos[1]` es la fila, `player_pos[0]` es la columna) en la matriz `maze` es `2`. Si es `2`, el jugador ha alcanzado la meta.

Línea 126: win = True

- Dentro del bucle `while` (y dentro del `if` de la meta): Si el jugador está en la meta, establece la bandera `win` a `True`.

Línea 127: game_over = True

- Dentro del bucle `while`: Establece la bandera `game_over` a `True`.

Línea 128: for e in enemies:

- Dentro del bucle `while`: Inicia un bucle `for` que itera sobre cada objeto `e` (enemigo) en la lista `enemies`.

Línea 129: if e.pos == player_pos:

- Dentro del bucle `while` (y dentro del `for e`): Comprueba si la posición del enemigo actual `e.pos` es igual a la posición del jugador `player_pos`.

Línea 130: game_over = True

- Dentro del bucle `while` (y dentro del `if e.pos == player_pos`): Si un enemigo atrapa al jugador, establece la bandera `game_over` a `True`.

Línea 131: win = False

- Dentro del bucle `while`: Asegura que la bandera `win` sea `False` (el jugador pierde si es atrapado).

Línea 132: (En blanco)

- Dentro del bucle `while`: Línea en blanco.

Línea 133: if game_over:

- Dentro del bucle `while`: Comprueba si la bandera `game_over` es `True`.

Línea 134: msg = "¡Ganaste!" if win else "¡Perdiste!"

- Dentro del bucle `while` (y dentro del `if game_over`): Determina el mensaje a mostrar. Usa una expresión condicional (operador ternario): si `win` es `True`, `msg` es "¡Ganaste!"; de lo contrario, `msg` es "¡Perdiste!".

Línea 135: txt = font.render(msg, True, (255, 255, 0))

- Dentro del bucle `while`: Renderiza el mensaje `msg` usando el objeto `font`.
  - `True`: Antialiasing.
  - `(255, 255, 0)`: Color del texto (amarillo).
- La superficie con el mensaje se asigna a `txt`.

Línea 136: screen.blit(txt, (WIDTH/2 - 60, HEIGHT/2))

- Dentro del bucle `while`: Dibuja la superficie `txt` en la `screen`.
  - `(WIDTH/2 - 60, HEIGHT/2)`: Intenta centrar el mensaje en la pantalla. `WIDTH/2 - 60` ajusta la posición horizontal basándose en una estimación del ancho del texto.

Línea 137: pygame.display.flip()

- Dentro del bucle `while`: Actualiza toda la pantalla para mostrar el mensaje de fin de juego. `flip()` es necesario aquí porque las operaciones de dibujo anteriores podrían no haberse mostrado completamente.

Línea 138: pygame.time.wait(2500)

- Dentro del bucle `while`: Pausa la ejecución del juego durante 2500 milisegundos (2.5 segundos) para que el jugador pueda leer el mensaje.

Línea 139: break

- Dentro del bucle `while`: Rompe el bucle `while True`, terminando así el bucle principal del juego y, por ende, la función `main`.

Línea 140: (En blanco)

- Dentro del bucle `while`: Línea en blanco.

Línea 141: pygame.display.flip()

- Dentro del bucle `while`: Actualiza toda la pantalla para mostrar los cambios realizados durante este fotograma (movimiento del jugador, enemigos, etc.). Esta llamada es para el caso en que `game_over` aún no es `True`.

Línea 142: clock.tick(12)

- Dentro del bucle `while`: Llama al método `tick()` del objeto `clock`. Esto limita la velocidad del juego a un máximo de 12 fotogramas por segundo (FPS). También ayuda a que el juego se ejecute a una velocidad más consistente en diferentes hardware.

Línea 143: (En blanco)

- Dentro del bucle `while`: Línea en blanco para separar la actualización de pantalla del movimiento de los enemigos.

Línea 144: for e in enemies:

- Dentro del bucle `while`: Inicia un bucle `for` que itera sobre cada objeto `e` (enemigo) en la lista `enemies`.

Línea 145: e.move()

- Dentro del bucle `while` (y dentro del `for e`): Llama al método `move()` de cada enemigo para actualizar su posición.

Línea 146: (En blanco)

- Dentro del bucle `while`: Línea en blanco para separar el movimiento de los enemigos del manejo de eventos.

Línea 147: # Eventos

- Dentro del bucle `while`: Comentario que indica la sección de manejo de eventos de Pygame.

Línea 148: for event in pygame.event.get():

- Dentro del bucle `while`: Inicia un bucle `for` que itera sobre todos los eventos que han ocurrido desde la última vez que se llamó a `pygame.event.get()`. Los eventos pueden ser pulsaciones de teclas, movimientos del ratón, el evento de cerrar la ventana, etc.

Línea 149: if event.type == pygame.QUIT:

- Dentro del bucle `while` (y dentro del `for event`): Comprueba si el tipo de evento (`event.type`) es `pygame.QUIT`. Este evento ocurre cuando el usuario hace clic en el botón de cerrar la ventana.

Línea 150: pygame.quit()

- Dentro del bucle `while` (y dentro del `if event.type == pygame.QUIT`): Llama a `pygame.quit()`. Esta función desinicializa todos los módulos de Pygame.

Línea 151: sys.exit()

- Dentro del bucle `while`: Llama a `sys.exit()`. Esto termina la ejecución del programa Python.

Línea 152: if event.type == pygame.KEYDOWN:

- Dentro del bucle `while` (y dentro del `for event`): Comprueba si el tipo de evento es `pygame.KEYDOWN`. Este evento ocurre cuando una tecla es presionada.

Línea 153: if event.key == pygame.K_LEFT: move_player(-1, 0)

- Dentro del bucle `while` (y dentro del `if event.type == pygame.KEYDOWN`): Si la tecla presionada (`event.key`) es la flecha izquierda (`pygame.K_LEFT`), llama a `move_player(-1, 0)` para mover al jugador una celda a la izquierda.

Línea 154: if event.key == pygame.K_RIGHT: move_player(1, 0)

- Dentro del bucle `while`: Si la tecla presionada es la flecha derecha (`pygame.K_RIGHT`), llama a `move_player(1, 0)` para mover al jugador una celda a la derecha.

Línea 155: if event.key == pygame.K_UP: move_player(0, -1)

- Dentro del bucle `while`: Si la tecla presionada es la flecha arriba (`pygame.K_UP`), llama a `move_player(0, -1)` para mover al jugador una celda hacia arriba (disminuyendo la fila).

Línea 156: if event.key == pygame.K_DOWN: move_player(0, 1)

- Dentro del bucle `while`: Si la tecla presionada es la flecha abajo (`pygame.K_DOWN`), llama a `move_player(0, 1)` para mover al jugador una celda hacia abajo (aumentando la fila).

Línea 157: (En blanco)

- Dentro del bucle `while`: Línea en blanco para separar el manejo de eventos de la actualización del frame del jugador.

Línea 158:  global player_frame

- Dentro del bucle `while`: Declara que se va a modificar la variable global `player_frame`. Esta línea es redundante si `player_frame` ya fue declarada global dentro de `draw_player`, pero no causa error si está en el mismo ámbito (la función `main`). Sin embargo, la modificación directa debería idealmente hacerse solo en un lugar o mediante funciones que encapsulen la lógica. Aquí, se incrementa directamente.

Línea 159:  player_frame += 1

- Dentro del bucle `while`: Incrementa la variable `player_frame` en 1. Esto se usa para la animación del jugador, aunque con un solo frame de animación, su efecto visible en el cambio de imagen es nulo.

Línea 160: (En blanco)

- Línea en blanco: Separa el final de la función `main` de la comprobación `if __name__ == "__main__"`.

Línea 161: if __name__ == "__main__":

- Esta es una construcción estándar en Python. Comprueba si el script se está ejecutando directamente (es decir, no está siendo importado como un módulo en otro script).

  - `__name__` es una variable especial que Python establece. Si el script es el principal, `__name__` es `"__main__"`.

Línea 162:  main()

- Dentro del `if __name__ == "__main__"`: Si el script se está ejecutando directamente, llama a la función `main()` para iniciar el juego.

Línea 163: (En blanco)

- Línea en blanco: Fin del archivo.


Laberinto con Muñeco Animado, Enemigos y cronómetro
Tiempo: 13.4s
¡Ganaste!