Adventure: Un juego en Pygame

Carlos Alberto López Rodríguez

Ingeniería Informática 2do Año

Grupo 21

Informe de Proyecto Final de Python



Contenido

[I. Introducción 3](#_Toc172492126)

[II. Estructura del Juego 4](#_Toc172492127)

[1. Módulo `adventure`: 4](#_Toc172492128)

[2. Módulo `camera`: 4](#_Toc172492129)

[3. Módulo `texture`: 4](#_Toc172492130)

[4. Módulo `character`: 4](#_Toc172492131)

[III. Descripción de las Clases 5](#_Toc172492132)

[1. Clase `Adventure` 5](#_Toc172492133)

[Pseudocódigo: 5](#_Toc172492134)

[Explicación: 7](#_Toc172492135)

[Explicación del código: 7](#_Toc172492136)

[2. Clase `Camera` 9](#_Toc172492137)

[Pseudocódigo: 9](#_Toc172492138)

[Explicación: 10](#_Toc172492139)

[Explicación del código: 10](#_Toc172492140)

[3. Clase `Texture` 11](#_Toc172492141)

[Pseudocódigo: 11](#_Toc172492142)

[Explicación: 11](#_Toc172492143)

[Explicación del código: 11](#_Toc172492144)

[4. Clase `Character` 12](#_Toc172492145)

[Pseudocódigo: 12](#_Toc172492146)

[Explicación: 13](#_Toc172492147)

[Explicación del código: 13](#_Toc172492148)

[5. Clase `GameClock`: 15](#_Toc172492149)

[5. Clase `main`: 16](#_Toc172492150)

[Funcionamiento de carga de niveles: 16](#_Toc172492151)

[1. Carga del Archivo JSON: 16](#_Toc172492152)

[2. Extracción de la Información: 16](#_Toc172492153)

[3. Creación del Mundo: 16](#_Toc172492154)

[4. Carga de las Configuraciones de Bloques: 17](#_Toc172492155)

[5. Creación de Objetos del Mundo: 17](#_Toc172492156)

[6. Carga del Fondo del Juego: 17](#_Toc172492157)

[7. Establecimiento del Punto de Inicio: 17](#_Toc172492158)

[IV. Conclusiones 19](#_Toc172492159)

[V. Sugerencias para el Examen 19](#_Toc172492160)

[VI. Anexos 19](#_Toc172492161)

# I. Introducción

Este documento detalla la estructura y funcionamiento del juego "Adventure: Un juego con Pygame”, desarrollado como proyecto final para la asignatura optativa de Python “Este rifle tira cualquier cosa”. El juego desarrollado es un juego de plataformas 2D desarrollado en Python utilizando la librería Pygame. El juego presenta un mundo complejo con múltiples niveles conectados a través de teletransportadores, diferentes tipos de bloques, objetos interactivos y un sistema de colisiones.

# II. Estructura del Juego

El juego se divide en diferentes módulos que se encargan de gestionar las funcionalidades principales:

## 1. Módulo `Adventure`:

Clase `Adventure`: La clase principal del juego. Gestiona la lógica general del juego, el bucle de eventos, la carga de niveles, la gestión de la cámara, la interacción del personaje y el dibujo de los elementos del juego.

## 2. Módulo `camera`:

Clase `Camera`: Se encarga de la gestión de la cámara del juego, incluyendo el seguimiento del personaje, la gestión del área visible y la renderización del juego en la pantalla.

## 3. Módulo `texture`:

Clase `Texture`: Responsable de cargar y gestionar las texturas del juego, como las de los bloques, el personaje y los elementos del entorno.

## 4. Módulo `character`:

Clase `Character`: Representa al personaje del jugador, gestionando su movimiento, interacción con el entorno y la gestión de la animación del personaje.

# III. Descripción de las Clases

## 1. Clase `Adventure`

### Pseudocódigo:

```python

class Adventure:

Inicializa variables del juego (tamaño de la ventana, fuentes, etc.)

def init(self, config\_obj):

Carga las configuraciones del juego desde un archivo JSON

Inicializa Pygame y crea la ventana del juego

Inicializa la fuente de texto

Carga el nivel inicial

Establece el juego como inicializado

Inicia el bucle principal del juego

def start(self):

Bucle principal del juego:

Maneja los eventos del juego (teclado, ratón, etc.)

Actualiza el estado del juego (personaje, cámara, etc.)

Dibuja los elementos del juego en la pantalla (personaje, bloques, fondo, etc.)

Actualiza la pantalla del juego

Controla la velocidad del juego (FPS)

Fin del bucle principal

Carga un nivel del juego

def load\_level(self, level):

Carga el archivo JSON del nivel

Extrae información del nivel (tamaño del mundo, bloques, etc.)

Crea la superficie del canvas para el juego

Crea las cámaras (por defecto y para debug)

Establece el punto de inicio del personaje

Crea el personaje

Inicializa el mundo del juego (matriz de bloques)

Carga los bloques del nivel

Carga el fondo del juego

Establece el juego como listo

Crea un nuevo bloque en el mundo del juego

def new\_block(self, block, block\_id):

Inserta el bloque en la matriz del mundo

Si el bloque tiene configuración de generación de objetos, genera un objeto aleatorio

Obtiene el ID del bloque en una posición específica

def get\_block\_id(self, x, y):

Busca el ID del bloque en la matriz del mundo

Dibuja los bloques del nivel en el canvas

def draw\_blocks(self):

Dibuja los bloques del nivel en el canvas, teniendo en cuenta el área visible de la cámara

Dibuja el fondo del juego en el canvas

def draw\_background(self):

Dibuja los elementos de fondo del juego

Dibuja el fondo de la cámara en el canvas

def draw\_camera\_background(self):

Dibuja los elementos de fondo específicos de la cámara

Interpolación lineal entre dos valores

def lerp(self, v1, v2, f):

Fórmula de interpolación lineal

Reinicia el juego al estado inicial

def restart(self):

Restablece la posición y velocidad del personaje

```

### Explicación:

La clase `Adventure` es el núcleo del juego. Se encarga de:

Inicializar el juego, incluyendo la configuración de la ventana, la fuente de texto y la carga del nivel inicial.

Gestionar el bucle principal del juego, manejando los eventos, actualizando el estado del juego y dibujando los elementos en la pantalla.

Cargar niveles del juego a partir de archivos JSON.

Gestionar la creación y dibujo de los bloques del mundo del juego.

Gestionar la creación y dibujo de los elementos de fondo del juego.

Implementar la interpolación lineal para efectos de movimiento suave.

Reiniciar el juego al estado inicial.

### Explicación del código:

`start(self)`:

Inicia el bucle principal del juego que se ejecuta hasta que `self.done` es `True`.

Maneja los eventos del juego, como las pulsaciones de teclas y los clics del ratón.

Activa/desactiva el modo de depuración utilizando F8.

Reinicia el juego utilizando R.

Interpola la posición de la cámara hacia la posición del personaje.

Dibuja el fondo de la cámara, los bloques, el fondo del juego, el personaje y los elementos de depuración (si está en modo de depuración).

Dibuja la superficie de la cámara en la pantalla y actualiza la pantalla.

Limpia la pantalla si la cámara tiene un desplazamiento.

Dibuja el fondo del canvas.

Actualiza el reloj del juego y obtiene el tiempo transcurrido.

`load\_level(self, level)`:

Carga un nivel del juego.

Lee el archivo JSON del nivel y obtiene la información sobre el número de filas y columnas del mundo, el tamaño de los bloques y la escala del canvas.

Crea la superficie del canvas con las dimensiones del mundo del juego.

`load\_level(self, level)`:

Carga el nivel del juego desde un archivo JSON.

Obtiene las propiedades del nivel del archivo JSON, como el número de filas y columnas del mundo, el tamaño de los bloques y la escala del canvas.

Crea las superficies del canvas y las cámaras.

Obtiene el punto de inicio del personaje del archivo JSON y crea el objeto `Character`.

Posiciona la cámara en el punto de inicio del personaje.

Crea un árbol binario para almacenar los bloques del mundo.

Crea los bloques del nivel y agrega los objetos generados a la lista `background`.

Carga el fondo de la cámara.

`new\_block(self, block, block\_id)`:

Crea un nuevo bloque en el mundo del juego a partir de la información del diccionario `block`.

Recorre las filas y columnas del bloque, insertando el ID del bloque en el árbol binario para el mundo.

Si el bloque tiene una configuración para generar objetos, genera objetos aleatorios con una probabilidad determinada y los agrega a la lista `background`.

`get\_block\_id(self, x, y)`:

Obtiene el ID del bloque en la posición especificada en el mundo.

Busca el ID del bloque en el árbol binario `self.world` utilizando las coordenadas x e y.

Devuelve el ID del bloque o `None` si no hay ningún bloque en esa posición.

`draw\_blocks(self)`:

Dibuja los bloques del nivel en el canvas.

Recorre la lista `self.blocks` que contiene la información de los bloques.

Obtiene la textura del bloque utilizando `self.texture.get\_texture(block["name"])`.

Dibuja la textura del bloque en el canvas utilizando `self.canvas.blit()`.

Si el bloque se dibuja de forma repetida (`block['draw'] == "repeat"`), dibuja la textura de forma repetida en el canvas para cubrir todo el área del bloque.

`draw\_background(self)`:

Dibuja el fondo del juego en el canvas.

Recorre la lista `self.background` que contiene la información de los elementos de fondo.

Obtiene la textura del elemento de fondo utilizando `self.texture.get\_texture(key)`.

Dibuja la textura del elemento de fondo en el canvas utilizando `self.canvas.blit()`.

`draw\_camera\_background(self)`:

Dibuja el fondo de la cámara en el canvas.

Recorre la lista `self.camera\_background` que contiene la información de los elementos de fondo de la cámara.

Obtiene la textura del elemento de fondo utilizando `self.texture.get\_texture(key)`.

Dibuja la textura del elemento de fondo en el canvas utilizando `self.canvas.blit()`.

`lerp(v1, v2, f)`:

Calcula la interpolación lineal entre dos valores utilizando la fórmula `v1 + ((v2 - v1) f)`.

`restart(self)`:

Reinicia el juego, restableciendo la posición y velocidad del personaje.

## 2. Clase `Camera`

### Pseudocódigo:

```python

class Camera:

Inicializa la cámara con sus propiedades (escala, dimensiones, etc.)

def \_\_init\_\_(self, scale, canvas\_size, window\_size, visible\_area, block\_size):

Inicializa las propiedades de la cámara

Calcula el área visible de la cámara

Actualiza el rectángulo de la cámara (área visible)

def update\_camera\_rect(self):

Calcula el rectángulo de la cámara en base a la posición actual

Obtiene la posición del ratón en la cuadrícula del mundo

def get\_mouse\_hover\_point(self):

Calcula las coordenadas del ratón en la cuadrícula del mundo

Dibuja la cuadrícula de la cámara en el canvas

def draw\_camera\_gird(self, canvas):

Dibuja la cuadrícula de la cámara sobre el canvas

Ajusta la escala de la cámara

def set\_scale(self, scale):

Cambia la escala de la cámara y actualiza el área visible

Obtiene el rectángulo del área visible de la cámara

def get\_camere\_block(self):

Devuelve el rectángulo del área visible de la cámara

```

### Explicación:

La clase `Camera` se encarga de gestionar la visualización del juego. Se encarga de:

Definir el área visible del juego en la pantalla.

Seguir al personaje, moviendo la cámara para que el personaje siempre esté dentro del área visible.

Ajustar la escala de la cámara para que se pueda ver más o menos del mundo del juego.

Dibujar la cuadrícula del mundo del juego para fines de depuración.

Gestionar la posición del ratón en la cuadrícula del mundo para facilitar la interacción del usuario.

### Explicación del código:

`init(self, scale, pos, window, canvas, block\_size)`: Inicializa la cámara con la escala, posición, tamaño de ventana, canvas y tamaño de bloque.

`set\_scale(self, scale)`: Establece la escala de la cámara, ajustando el tamaño de la superficie de la cámara y el rectángulo de la cámara.

`update\_offset(self)`: Calcula el desplazamiento horizontal y vertical de la cámara para centrarla en la ventana.

`update\_camera\_rect(self)`: Calcula el rectángulo de la cámara, que determina el área visible del mapa.

`get\_mouse\_hover\_point(self)`: Obtiene las coordenadas del bloque en el que se encuentra el ratón, teniendo en cuenta la escala y el desplazamiento de la cámara.

`get\_camere\_block(self)`: Obtiene las coordenadas del bloque de la cámara (el área del mapa que se está mostrando).

`draw\_camera\_gird(self, canvas)`: Dibuja una cuadrícula sobre el canvas, mostrando los límites de los bloques que están dentro del área visible de la cámara.

## 3. Clase `Texture`

### Pseudocódigo:

```python

class Texture:

Inicializa la clase Texture

def \_\_init\_\_(self, path):

Almacena la ruta de las texturas

Obtiene una textura a partir de un nombre

def get\_texture(self, name):

Carga la textura desde el archivo con el nombre dado

Devuelve la textura cargada

```

### Explicación:

La clase `Texture` se encarga de gestionar las texturas del juego, como las de los bloques, el personaje y los elementos del entorno. Se encarga de:

Cargar las texturas desde los archivos de imagen.

Proporcionar acceso a las texturas a través de su nombre.

### Explicación del código:

`texture\_dict`: Diccionario que almacena las texturas cargadas, utilizando una tupla de `(key, reverse)` como clave.

`path`: Ruta del directorio donde se encuentran las texturas.

`init(self, path)`:

Inicializa la clase `Texture` con la ruta del directorio.

`get\_texture(self, key, reverse=False)`:

Busca la textura en el diccionario `texture\_dict` utilizando la clave `(key, reverse)`.

Si la textura no está en el diccionario:

Construye la ruta completa del archivo de imagen (`image\_path`).

Carga la imagen utilizando `pygame.image.load()`.

Si se especifica `reverse`, invierte la imagen horizontalmente utilizando `pygame.transform.flip()`.

Almacena la textura en el diccionario `texture\_dict`.

Si la textura ya está en el diccionario, devuélvela.

En resumen:

La clase `Texture` facilita la carga y gestión de las texturas del juego. A través del método `get\_texture`, se puede acceder a las texturas utilizando una clave y la posibilidad de invertirla horizontalmente. Esto permite cargar las texturas de forma eficiente, evitando la necesidad de cargarlas repetidamente.

## 4. Clase `Character`

### Pseudocódigo:

```python

class Character:

Inicializa el personaje con sus propiedades (posición, tamaño, etc.)

def \_\_init\_\_(self, x, y, w, h):

Inicializa las propiedades del personaje

Carga la textura del personaje

Inicializa la velocidad del personaje

Maneja las entradas del jugador

def handle(self, keys):

Gestiona la dirección del movimiento del personaje en base a las teclas presionadas

Actualiza el estado del personaje

def update(self, delay):

Aplica la gravedad al personaje

Actualiza la posición del personaje en base a su velocidad

Dibuja el personaje en el canvas

def draw(self, canvas):

Dibuja el personaje en el canvas

```

### Explicación:

La clase `Character` representa al personaje del jugador. Se encarga de:

Gestionar el movimiento del personaje en base a las entradas del jugador (teclado).

Aplicar la gravedad al personaje para que caiga.

Actualizar la posición del personaje en base a su velocidad.

Dibujar el personaje en el canvas del juego.

### Explicación del código:

`init(self, x, y, w, h)`: Inicializa el personaje con su posición, tamaño y estado inicial.

`handle(self, inputs)`: Maneja las entradas del usuario (movimiento izquierda/derecha).

`update(self, delay)`: Actualiza la posición y el estado del personaje, incluyendo colisiones con los bloques.

`draw(self, canvas)`: Dibuja el personaje en el canvas usando el sprite correspondiente a su estado actual.

Puntos Clave:

El código usa una estructura de estados para controlar la animación del personaje, lo que permite que se mueva, salte, caiga, etc., de manera fluida.

Se incluye una comprobación de colisiones con los bloques del juego para que el personaje no se pueda mover a través de ellos.

Se utiliza una técnica de caching para la textura del personaje, lo que permite que se cargue la textura solo una vez y se reutilice en las diferentes animaciones.

`get\_current\_block(self)`: Calcula el rectángulo del bloque en el que se encuentra el personaje.

`get\_surrounding\_block(self, test)`: Obtiene los bloques que están alrededor del bloque actual del personaje, según la dirección de movimiento `test`.

`update\_target(self, test)`: Encuentra el bloque objetivo al que se dirige el personaje, teniendo en cuenta su dirección de movimiento y la posición de los bloques.

`on\_collision(self, b\_id)`: Maneja las acciones que se deben realizar cuando el personaje colisiona con un bloque (por ejemplo, morir, saltar o teletransportarse).

Clase `StatusIdle`:

Define el estado "idle" del personaje, donde está quieto.

`handle(self, instance, inputs=None)`: Maneja las entradas del usuario cuando el personaje está en estado "idle" (saltar o correr).

`update(self, instance, delay)`: Actualiza el estado "idle", animando el personaje y comprobando si está cayendo.

Puntos clave:

Detección de Colisiones: La clase `Character` utiliza la detección de colisiones para que el personaje no se pueda mover a través de los bloques.

Movimiento del Personaje: La clase `Character` maneja el movimiento del personaje en los diferentes estados, como correr, saltar y caer.

Estados del Personaje: La clase `Character` utiliza una estructura de estados para controlar el comportamiento del personaje en diferentes situaciones.

Clase `StatusJump`: Define el estado de salto del personaje.

`handle(self, instance, inputs=None)`: Maneja las entradas del usuario durante el salto, incluyendo la posibilidad de realizar un doble salto.

`update(self, instance, delay)`: Actualiza el estado de salto, aplicando la gravedad y comprobando si el personaje ha aterrizado o si se encuentra en un bloque.

`update\_target(self, instance)`: Encuentra el bloque con el que el personaje va a colisionar durante el salto.

Clase `StatusFall`: Define el estado de caída del personaje.

`handle(self, instance, inputs=None)`: Maneja las entradas del usuario durante la caída, incluyendo la posibilidad de realizar un doble salto.

`update(self, instance, delay)`: Actualiza el estado de caída, aplicando la gravedad y comprobando si el personaje ha aterrizado o si se encuentra en un bloque.

`update\_target(self, instance)`: Encuentra el bloque con el que el personaje va a colisionar durante la caída.

Clase `StatusRun`: Define el estado de carrera del personaje.

`handle(self, instance, inputs=None)`: Maneja las entradas del usuario mientras corre, cambiando de estado si se presiona la tecla de salto o si se suelta la tecla de movimiento.

`update(self, instance, delay)`: Actualiza el estado de carrera, animando el personaje y comprobando si está cayendo.

Clase `StatusDJump`: Define el estado de doble salto del personaje.

`handle(self, instance, inputs)`: No se manejan entradas del usuario en este estado.

`update(self, instance, delay)`: Actualiza el estado de doble salto, aplicando la gravedad y comprobando si el personaje ha aterrizado o si se encuentra en un bloque.

`update\_target(self, instance)`: Encuentra el bloque con el que el personaje va a colisionar durante el doble salto.

En resumen:

Estas clases implementan los estados de carrera y doble salto del personaje, incluyendo la animación, la detección de colisiones y la transición a otros estados. El código controla el movimiento del personaje durante la carrera y el doble salto, asegurando un comportamiento realista y fluido.

## 5. Clase `GameClock`:

`init(self, max\_fps)`:

Inicializa el `GameClock` con el máximo FPS deseado.

Define el FPS mínimo a 30.

Calcula el tiempo objetivo por frame (`fps\_cost\_time`).

Inicializa las variables de tiempo, contador y FPS.

`tick(self)`:

Calcula el tiempo transcurrido desde el último frame.

Ajusta la velocidad de frame esperando si es necesario.

Actualiza el tiempo total transcurrido y los FPS.

Devuelve el tiempo transcurrido desde el último frame.

`get\_fps(self)`:

Devuelve el FPS actual.

`sleep(delay\_time)`:

Método estático para pausar la ejecución durante un tiempo específico.

Utiliza `time.perf\_counter()` para obtener el tiempo actual y pausar la ejecución hasta que se alcance el tiempo de finalización.

En resumen:

La clase `GameClock` es una herramienta útil para controlar el bucle de juego y el FPS en un juego. Permite que el juego se ejecute con la frecuencia deseada, mientras que se asegura de que no se ejecute demasiado rápido. El método `tick` se llama en cada frame del juego para actualizar el temporizador y controlar la velocidad de frame. El método `get\_fps` se puede usar para obtener el FPS actual.

## 6. Clase `main`:

`import adventure`: Importa el módulo `adventure` que contiene la clase `Adventure`.

`import json`: Importa el módulo `json` para trabajar con archivos JSON.

`if \_\_name\_\_ == '\_\_main\_\_':`: Esta condición verifica si el script se está ejecutando directamente (no como un módulo importado).

Si se cumple la condición, se ejecuta el código dentro del bloque.

`config\_file = open("config/config.json", "r")`: Abre el archivo `config.json` en modo lectura.

`config\_obj = json.load(config\_file)`: Carga el contenido del archivo JSON en el objeto `config\_obj`.

`instance = adventure.default`: Obtiene la instancia de la clase `Adventure` que se creó en el archivo `adventure.py` y la asigna a la variable `instance`.

`instance.init(config\_obj)`: Inicializa el juego con las configuraciones del archivo JSON.

`instance.start()`: Inicia el bucle principal del juego.

## Funcionamiento de carga de niveles:

La interpretación de los datos del archivo JSON para crear el mundo del juego se realiza en la clase `Adventure` del módulo `adventure`.

### 1. Carga del Archivo JSON:

Al inicio del juego, la clase `Adventure` recibe un objeto de configuración que contiene la ruta al archivo JSON del mundo.

Se utiliza la librería `json` de Python para cargar el contenido del archivo JSON en un diccionario.

### 2. Extracción de la Información:

Se extraen los valores de los campos principales del diccionario:

`worldCol` y `worldRow`: Se utilizan para definir el tamaño del mundo del juego en bloques (ancho y alto).

`blockSize`: Define el tamaño de cada bloque en píxeles.

`scale`: Define la escala del mundo (para zoom).

### 3. Creación del Mundo:

Se crea una matriz bidimensional (una lista de listas) para representar el mundo del juego. Las dimensiones de la matriz se definen por `worldCol` y `worldRow`.

Cada elemento de la matriz representa un bloque del mundo y contiene un identificador (ID) que indica el tipo de bloque.

### 4. Carga de las Configuraciones de Bloques:

Se itera sobre el array `blocks` del diccionario JSON.

Para cada configuración de bloque:

Se crea una instancia de un objeto de bloque con los datos del JSON (nombre, modo de dibujo, coordenadas, tamaño, probabilidad de generación y lista de objetos a generar).

Se genera el bloque en el mundo del juego, teniendo en cuenta la probabilidad de generación y la posición.

Se crean los objetos alrededor del bloque, utilizando los datos de `gen`.

### 5. Creación de Objetos del Mundo:

Para cada objeto a generar (definido en `gen` de las configuraciones de bloques), se busca la textura correspondiente en el módulo `texture`.

Se crea un objeto con la textura cargada y se posiciona en el mundo del juego según el offset definido en `gen`.

### 6. Carga del Fondo del Juego:

Se itera sobre el array `camera\_background` del diccionario JSON.

Para cada objeto de fondo:

Se busca la textura correspondiente en el módulo `texture`.

Se crea un objeto con la textura cargada y se posiciona en el fondo de la cámara.

### 7. Establecimiento del Punto de Inicio:

Se utiliza el objeto `start` del diccionario JSON para obtener las coordenadas X e Y del punto de inicio del personaje.

Se crea el personaje del jugador en esas coordenadas.

El mundo del juego se define en archivos JSON. Cada archivo JSON representa un nivel del juego. La información del mundo se organiza en las siguientes secciones:

`worldCol` y `worldRow`: Definen el tamaño del mundo en bloques (ancho y alto).

`blockSize`: Define el tamaño de cada bloque en píxeles.

`scale`: Define la escala del mundo (para zoom).

`blocks`: Contiene un array de objetos que representan los diferentes tipos de bloques del mundo. Cada objeto de bloque tiene los siguientes atributos:

`name`: Nombre del tipo de bloque (textura).

`draw`: Define el modo de dibujo del bloque (repeat, once, single).

`collision`: Define el tipo de colisión con el bloque (jump, died, etc.).

`power`: Define la fuerza del bloque (para empujar al jugador).

`transport`: Define el archivo JSON del mundo al que se teletransporta el personaje (si es un teletransportador).

`prob`: Define la probabilidad de generar el bloque (0-100).

`gen`: Define los objetos a generar alrededor del bloque (offset y tipo de objeto).

`camera\_background`: Contiene un array de objetos que representan los elementos de fondo del juego. Cada objeto de fondo tiene los siguientes atributos:

`x`: Posición X relativa del objeto (0-1).

`y`: Posición Y relativa del objeto (0-1).

`key`: Nombre del objeto de fondo.

`start`: Define las coordenadas X e Y del punto de inicio del personaje.

# IV. Conclusiones

El juego "Adventure: Un juego con Pygame" es una aplicación práctica de los conceptos de programación en Python aprendidos en la asigntaura, utilizando Pygame para el desarrollo de juegos. Se implementaron funcionalidades como gestión de la cámara, carga de niveles, interacción del personaje, gestión de texturas y dibujo de elementos del juego.

El juego desarrollado presenta una estructura modular y escalable, permitiendo la creación de mundos complejos y niveles conectados. El uso de archivos JSON facilita la creación y edición de niveles, mientras que la implementación de las diferentes clases permite la organización y reutilización del código.

# 

# V. Anexos