Universidad Nacional De Ingeniería

Facultad de Ingeniería Eléctrica y Electrónica

Programación Orientada a Objetos (BMA15)

Condor Dash: Desarrollo de Juegos y Simulación

Integrantes:

- Borja Soto, Diego
- Montesinos Castro, Luis Antonio

Docente:

- Mg. Ing. Yuri Tello Canchapoma

LIMA – PERÚ

2024 – III

INDICE

1. Introducción

   Este proyecto en Python desarrolla un videojuego en el que un cóndor atraviesa distintos escenarios enfrentando obstáculos como contaminantes, promoviendo la concienciación ambiental. El juego cuenta con dos modos: un jugador y multijugador, ofreciendo diferentes experiencias al momento de jugarlo. Además de incluir un sistema de puntuación para motivar a los jugadores a superar los desafíos y aprender sobre el impacto ambiental de una manera interactiva.

2. Objetivos del proyecto

   a) Objetivo General

      Desarrollar un videojuego educativo en Python que fomente el aprendizaje de la programación orientada a objetos a través de un entorno interactivo, al mismo tiempo que promueve la concienciación ambiental mediante mecánicas de juego atractivas.

   b) Objetivos Específicos

      - Implementar principios de POO en el desarrollo del videojuego, asegurando modularidad y reutilización del código.
      - Diseñar una estructura flexible que facilite futuras expansiones y mejoras en el juego.
      - Implementar un sistema de configuración de dificultad para adaptar la experiencia a distintos tipos de jugadores.
      - Integrar una base de datos para el almacenamiento de puntuaciones y estadísticas de los jugadores.
      - Documentar el proceso de desarrollo para facilitar futuras colaboraciones y mejoras en el código.

3. Proyectos similares

a) Space Impact

Space Impact es un clásico videojuego de desplazamiento lateral desarrollado por Nokia en el año 2000. En el juego, el jugador controla una nave espacial que debe enfrentarse a enemigos y esquivar obstáculos mientras avanza por distintos niveles con un desplazamiento automático.

Características:

- Desplazamiento lateral automático.
- Obstáculos y enemigos que desafían al jugador.
- Mecánica de esquivar y atacar para progresar en los niveles.
- Estilo arcade con puntuaciones acumulativas.

b) Jetpack Goodride

Jetpack Goodride es un juego similar a Jetpack Joyride desarrollado en pygame. El jugador controla a Barry Steakfries, quien utiliza un jetpack para moverse y esquivar obstáculos mientras recoge monedas y potencia su equipo a lo largo del trayecto.

Características:

- Mecánica de vuelo con control sobre la altura del personaje.
- Obstáculos dinámicos que requieren reflejos rápidos.
- Posibilidad de obtener mejoras y potenciadores durante la partida.
- Estilo de juego rápido y desafiante con un enfoque arcade.

4. Clases y métodos principales

a) Condor

Personaje principal controlado por el jugador.

```
class Condor:
    def __init__(self): ···

    def ciclo(self, teclas, mouse): ···

    def render(self): ···

    def hitbox(self): ···
```

b) Servidor_local

Permite la conexión y el intercambio de datos entre el servidor y los clientes mediante un socket UDP para la detección del servidor y un socket TCP para el intercambio de datos en tiempo real.

```
class Servidor_local:
    def __init__(self, nombre_servidor, nombre_host): ···

    def cambio_datos(self, serv, host): ···

    def actualizacion_datos(self): ···

    def obtener_ip(self): ···

    def iniciar(self): ···

    def enviar_anuncios(self):      # Mandar existencia del servidor ···

    def aceptar_clientes(self):  # Aceptar jugadores ···

    def manejar_cliente(self, nombre, cliente): ···

    def enviar_lista_jugadores(self): ···

    def expulsar_jugador(self, nombre): ···

    def cerrar_servidor(self): ···

    # Juego
    def iniciar_juego_servidor(self): ···

    def enviar_estado_juego(self): ···

    def recibir_actualizaciones_jugador(self, nombre, cliente): ···

    def _enviar_datos_periodicamente(self): ···
```

c) Cliente_local

Se encarga de buscar servidores y conectarse a ellos, mostrando un menú de espera antes de iniciar la partida.

```
class Cliente_local:
    def __init__(self, nombre):···

    def obtener_ip(self):···

    def buscar_servidores(self):···

    def buscar_servidor_manual(self, ip_manual):···

    def conectar_a_servidor(self, ip, puerto, DSM, CPU, CPP, UPU):···

    def escuchar_servidor(self):···

    def salir_del_servidor(self):···

    # Juego
    def iniciar_juego_cliente(self):···

    def recibir_datos_servidor(self):···

    def enviar_datos_jugador(self):···

    def finalizar_juego(self):···

    def menu_espera_partida_local(self, DSM, CPU, CPP, UPU):···
```

d) Menú

Conjunto de bucles activables que muestran las interfaces del juego.

```
class Menu:
    def __init__(self, puntuacion, puntaje_maximo):···

    def reinicio(self):···

    def menu_principal(self):···

    def menu_jugar(self, estado):···

    def menu_multijugador(self, estado):···

    def menu_crear_partida_local(self, estado):···

    def menu_unirse_partida_local(self, estado):  ···

    def menu_fin_un_jugador(self, estado):···

    def menu_fin_un_jugador_data(self):···
```

e) Puntuación

Calcula y envía el puntaje de la partida en curso.

```
class Puntuacion:
    def __init__(self, velocidad):···

    def ciclo(self):···

    def reinicio_puntaje(self):···

    def reinicio_puntaje_maximo(self):···

    def puntaje(self):···

    def puntuacion_maxima(self):···

    def mostrar_puntaje(self):···
```

f) Obstáculo (superclase)

Base para todos los obstáculos del juego.

```
class Obstaculo:
    def __init__(self, posicionX, velocidad): ···

    def ciclo(self): ···

    def update(self): ···

class Paloma(Obstaculo): ···

class CFC(Obstaculo): ···

class Tuberias(Obstaculo): ···
```

- **Paloma y CFC**
  Obstáculos en movimiento lineal de derecha a izquierda que buscan colisionar con el jugador.
- **Tuberías**
  Generación compleja de tuberías con desplazamiento lineal de derecha a izquierda.

g) Escenario (superclase)

Fondos mostrados durante el avance del juego.

```
class Escenario:
    def __init__(self, posicionX, velocidad, condor):   # 2 ult. ···

    def generar_obstaculo(self): ···

    def ciclo_basico(self): ···

    def ciclo(self): ···

    def fin(self): ···

    def valor_colision(self): ···

    def render(self): ···

    def update(self): ···

class Transicion(Escenario): ···

class Valle(Escenario): ···

class Ciudad(Escenario): ···

class Industria(Escenario): ···

class Mina(Escenario): ···
```

- **Escenario**
  Fondo con movimiento lateral y obstáculos específicos por nivel.
- **Transición**
  Animación que marca el cambio entre escenarios.

h) Juego

Ejecuta las clases *Menú, Condor, Obstáculo* y *Escenario*, combinándolas para crear la jugabilidad principal.

```
class Juego:
    def __init__(self): ···

    # Reinicio de valores
    def reinicio_valores(self): ···

    def aparicion_transicion(self): ···

    def puntuacion_actual(self): ···

    def puntuacion_maxima(self): ···

    # Juego
    def juego(self, estado_juego): ···
```

i) Juego_multijugador

Variante de *Juego* que utiliza la biblioteca pickle para enviar y recibir paquetes de datos.

```
class Juego_multijugador:           # Servidor
    def __init__(self): ···

    # Reinicio de valores
    def reinicio_valores(self): ···

    def aparicion_transicion(self): ···

    def puntuacion_actual(self): ···

    def puntuacion_maxima(self): ···

    # Juego
    def juego_servidor(self, estado_juego): ···

    def juego_cliente(self, estado_juego): ···

    # Metodos de paquetes de datos
    def empaquetado_datos(self): ···

    @staticmethod
    def desempaquetado_datos(data): ···

    def lectura_datos(self, datos_juego): ···

    def crear_obstaculo(self, data): ···

    def crear_escenario(self, data): ···

    def posicion_personaje(self, nombre_jugador, x, y): ···

    def nuevo_objeto(self, objeto): ···
```

- **juego_servidor**
  Ejecuta la lógica del juego en el host, empaquetando y enviando datos a los clientes.
- **juego_cliente**
  Desempaqueta los datos recibidos y renderiza el estado del juego, enviando la posición del jugador al servidor.

j) Run

Método que inicia la ejecución del juego.

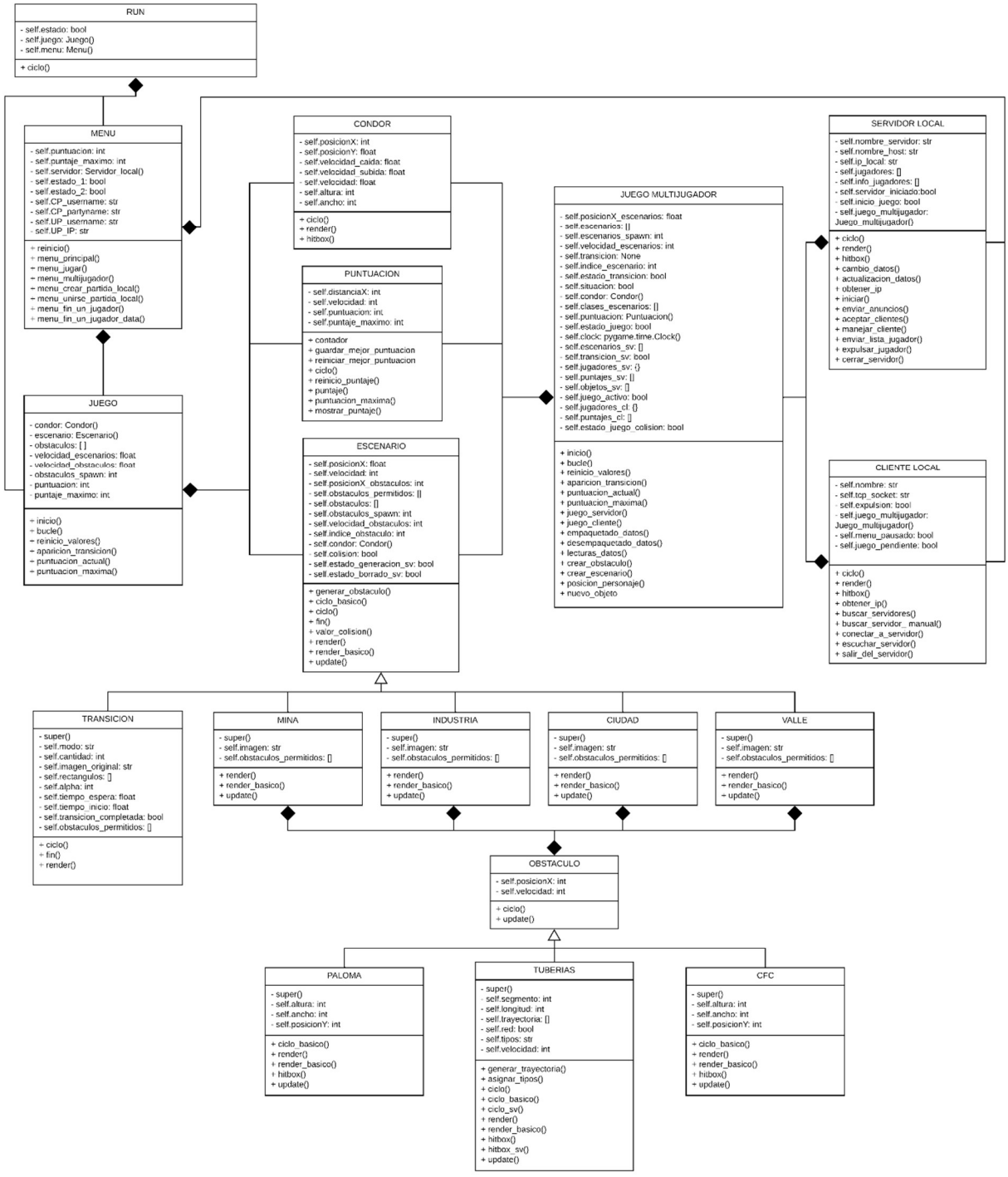
```python
# Ejecucion del juego
class Run:
    def __init__(self):···

    def ciclo(self):
        while self.estado:
            # Datos
            self.menu.estado_1 = True
            # Menu principal - juego "monojugador"
            self.menu.menu_principal()
            self.juego.juego(self.menu.estado_1)    # Un jugador

# Inicializar objetos
run = Run()

if __name__ == "__main__":
    run.ciclo()
```

5. Diagrama UML



6. Conclusiones

El desarrollo de Cóndor Dash ha permitido demostrar la aplicación práctica de la Programación Orientada a Objetos en la creación de videojuegos, asegurando modularidad y escalabilidad en su código. A través de este proyecto, se ha logrado integrar distintos elementos clave, como la generación de escenarios dinámicos, la gestión de colisiones y la implementación de un modo multijugador funcional.

El juego no solo ofrece una experiencia entretenida y desafiante, sino que también cumple con su propósito educativo y de concienciación ambiental, al presentar obstáculos que representan problemáticas reales como la contaminación. Además, la implementación de sistemas de puntuación y dificultad ajustable permite una experiencia de usuario adaptable y motivadora.

Asimismo, el enfoque modular del desarrollo facilita futuras expansiones, lo que permite la integración de nuevos escenarios, personajes y mecánicas de juego sin afectar la estabilidad del sistema. Además, la implementación del modo multijugador representa un avance significativo en la interactividad del juego, brindando una experiencia más dinámica y colaborativa.

A futuro, se podrían explorar mejoras como la optimización del rendimiento, la ampliación de las funcionalidades de red para mejorar la estabilidad del modo multijugador y la incorporación de un sistema de logros y recompensas para aumentar la retención de jugadores. También sería viable la integración de herramientas de análisis para evaluar el impacto educativo del juego y su efectividad en la concienciación ambiental.

7. Bibliografía

Sweigart, A. (2015). Invent Your Own Computer Games with Python. No Starch Press.

Van Rossum, G., & Drake Jr, F. L. (2009). The Python Language Reference Manual. Network Theory Ltd.

Stevens, W. R. (1998). Unix Network Programming, Volume 1: The Sockets Networking API. Addison-Wesley.

Goerzen, J. (2004). Foundations of Python Network Programming. Apress.

Van Rossum, G. (2022). Python Standard Library Documentation – pickle module. Python Software Foundation. Disponible en: https://docs.python.org/3/library/pickle.html