Vestigium

Fernando Álvarez de Legísima
Ana Bañobre Martínez
Iria Pardo Neira
Xian Priego Martín

16 de mayo de 2025

Índice

1. Desarrollo Artístico

1.1. Antecedentes

1.1.1. Ambientación

El videojuego se desarrolla en un mundo melancólico donde las sombras de los seres humanos persisten más allá de la muerte. Estas entidades errantes se niegan a aceptar su destino y vagan por paisajes oníricos llenos de luz que acechan con hacerlas desvanecer y de penumbra, en busca de su reencarnación.

El estilo se asocia al pixel art de 32 bits, en el caso del videojugo en 2D, y gótico y minimalista en el caso del 3D. Es fundamentalmente monocromático, destacando el uso de una combinación limitada de colores y el contraste entre luz y sombra:

- **Negro, gris y colores sutiles**: utilizados para el menú y los distintos escenarios. El personaje principal no destaca ante el paisaje pero su figura es característica.
- **Amarillos con transparencias**: el color principal de las fuentes de luz que afectan a la jugabilidad.

Una música y sonido ambiental refuerzan la atmósfera misteriosa del juego. Melodías etéreas y minimalistas envuelven al jugador y acompañan a esta historia que transcurre en diversos escenarios, explicados más profundamente en la sección 1.3, sumidos en la naturaleza, entre los que se encuentran:

- Un cementerio y su portón de salida
- El árbol de un bosque
- El fondo de un lago
- La casita del bosque

1.1.2. Historia

El juego comienza con el nacimiento de una sombra errante emergiendo del cuerpo inerte de un humano recién sepultado en un cementerio. Al despertar en su nueva forma, se da cuenta de que ha perdido su recipiente físico y siente un deseo profundo de encontrar otro cuerpo en el que habitar. Así inicia el viaje de la sombra, encapuchada como una entidad frágil que, a medida que avanza por paisajes desconocidos, se enfrenta a una revelación: su destino no es poseer otro cuerpo, si no encontrar su lugar dentro de las sombras del mundo.

A lo largo de su travesía, la sombra debe **esquivar entes iluminados** y fuentes de luz que amenazan con disolverla, refugiándose en la penumbra y utilizando el entorno a su favor. Su mayor desafío llega con la luz del día, una fuerza absoluta e inevitable. Al final del juego, la sombra comprende que no desaparecerá, sino que se fusionará con las sombras del bosque, convirtiéndose en parte de un ciclo eterno.

1.2. Personajes

1.2.1. Protagonista

- **Sombra**: Una criatura oscura y encapuchada definida por su vulnerabilidad y cuya jugabilidad experimenta variaciones constantes en función de las condiciones presentes en su entorno.

Figura 1: Sprite 2D de la sombra.

Figura 2: Modelo 3D de la sombra.

1.2.2. Enemigos

El enemigo principal del juego es la luz. Existen varios elementos y entidades que a lo largo del juego emiten luz o elementos en los que la luz es inherente a ellos:

- **Farolas**: Que se encienden y apagan intermitentemente por el cementerio.
- **Luciérnagas**: Que rondan con movimientos impredecibles a lo largo de los escenarios.
- **Enterrador**: Vigilante a la salida del cementerio que porta un farolillo pequeño y guarda la llave cerca.
- **Farolillo mágico**: Farolillo gigante cuya luz ronda por todo el cementerio.

Figura 3: Sprite del enterrador.

- **Setas luminosas**: Impulsan a la sombra cuando esta salte en ellas pero se encenderán siendo una amenaza más en el bosque.

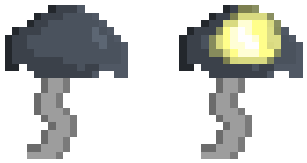


Figura 4: Sprite de las setas luminosas.

- **Arañas luminosas**: Bloquean la luz que pasa al interior del árbol pero están iluminadas.

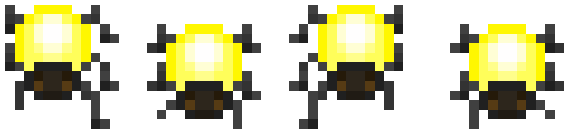


Figura 5: Sprite de las arañas.

- **Pez linterna abisal**: Su linterna emite un foco de luz para abrirse paso por la oscuridad del fondo del lago que persiguiendo a la sombra.

Figura 6: Sprite del pez linterna.

- **Medusas**: acompañan al pez iluminándose y nadando por el lago.

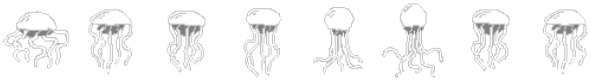


Figura 7: Sprite de las medusas.

- **Fuentes de luz dentro de la casa**: Lámparas, electrodomésticos, bombillas, fogones, televisores, etc. Estos elementos se encuentran desperdigados por todo el escenario de la casa del bosque, presente en la implementación en 3D.

1.3. Otras características de la ambientación

La ambientación del escenario es constante a lo largo de todo el videojuego. Sigue una estética antigua y descuidada, centrada en elementos de la naturaleza, pero manteniendo un ambiente misterioso.

Figura 8: Fondo del menú principal.

Figura 9: Escenario del cementerio.

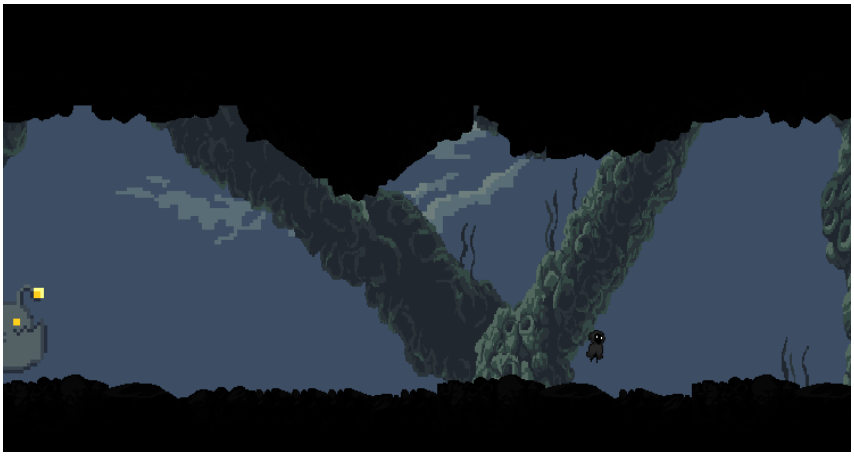


Figura 10: Escenario del lago.

1.3.1. Objetos destacados

En la siguiente sección se describe con exhaustividad las distintas fases del videojuego y sus elementos, pero cabe resaltar algunos objetos principales de cada fase.

En las escenas del cementerio destacan **lápidas y tumbas** rodeadas de vegetación (Figura 9) con **plataformas flotantes y farolas** que guían al personaje principal a la siguiente escena, la salida del cementerio. Esta, rodeada de plataformas de madera y el acechante farolillo gigante, confunde y complica los movimientos de la sombra. Una vez conseguida la **llave** que abre el viejo portón del cementerio, el personaje debe esquivar las luciérnagas que rodean su **candado**. Una vez fuera, llegará a un bosque y atravesará el **interior de un árbol lleno de huecos** que deja pasar la luz. La sombra caerá en un lago llegando a sus **profundidades musgosas** y llenas de algas iluminadas por el pez linterna (Figura 10).

Al emerger del lago, ya en el escenario 3D, se encontrará con la entrada neblinosa de la casita del árbol y deberá encontrar la **llave de la puerta principal** para entrar en ella. Atravesará las distintas salas de la casa escondíendose entre las sombras y resolviendo los distintos **acertijos** para acabar saliendo de la misma.

1.4. Guion

1.4.1. Transcurso de la acción en el videojuego

La acción del videojuego transcurre en el espacio temporal de una noche en la que el jugador nace y emprende su camino por un ambiente natural y rural. El juego comienza con una fase introductoria a la mecánica principal de esquivar la luz en el cementerio. El jugador aprende a **esquivar obstáculos fáciles** y a realizar movimientos que más tarde necesitará implementar para superar el resto de las fases. A medida que avanza, va descubriendo nuevos elementos y **formas de moverse cada vez más desafiantes**, desde saltos y dobles saltos hasta nadar, lo que requiere ser **rápido pero estratégico**.

El avance del jugador en el juego implica descubrir nuevas mecánicas que lo protegen como **esconderse tras elementos** del escenario para taparse de la luz, como en el final de la fase del cementerio. Encontrar la llave que llevará a un **pequeño puzzle** que

determinará si el jugador posee la habilidad de continuar o quedará encerrado tras el portón de salida. Su **rapidez de reacción y movimiento** decidirá si puede atravesar el árbol de un bosque por el que se cuela la luz. Ciertos enemigos como las arañas, se podrán poner en su favor durante un momento mientras tapan la luz, pero seguirá teniendo que estar alerta para evadir las luces que lo rodean. Elementos como las setas le impulsarán aún más alto que su salto regular pero entonces se iluminarán y no podrá volver a caer en ellos. Las luciérnagas a la salida del árbol, y ya presentes a lo largo de toda su travesía, complicarán su caída para llegar a la tensa fase final 2D del lago.

Finalmente, ya representada en su forma tridimensional (versión 3D), la sombra se adentrará en la casita del bosque con el objetivo de encontrar el cuerpo que debe habitar. Sin embargo, en su interior solo hallará estancias vacías, iluminadas por diversas fuentes de luz que pondrán en peligro su avance y dificultarán el descubrimiento de su verdadero propósito.

1.4.2. Fases del juego: implementación 2D

1. **Cementerio: tutorial y jefe**
   En esta fase se hace una presentación de la sombra y el por qué de su nacimiento. Se indica como debe moverse y tiene el primer contacto con la luz, unas farolas que parpadean alternadamente. Siguiendo el único camino posible, a veces bloqueado de forma intermitente por luciérnagas, deberá saltar plataformas evitando caer al vacío. Combinando su salto con el momento oportuno para pasar sin que la luz de las farolas le haga desvanecer, podrá pasar a la segunda parte de la fase, un boss que vigila la salida. En esta parte, un farolillo mágico gigante lo persigue por todo el escenario mientras que busca la llave que tiene escondida el enterrador del cementerio. Una vez encontrada la llave, si la sombra consigue pasar el enterrador y llegar a la puerta se presentará un puzzle o minijuego. En él, la sombra mueve la llave evitando las luciérnagas que rodean el candado del portón. Si consigue llevar la llave al candado, podrá desbloquear el siguiente nivel, si no, tendrá que volver a recuperar la llave.

2. **Árbol en el bosque: su interior y su salida**
   Una vez consigue salir, la sombra se encuentra con un gran árbol hueco por dentro en medio del bosque. Esta se ve obligada a atravesarlo por su interior en el que encuentra unas setas luminosas que le impulsan y le ayudarán a ir subiendo poco a poco. Más arriba descubre unas arañas con abdomen luminoso recorriendo las paredes del árbol que tapan y destapan huecos del árbol por los que pasa la luz. Esperando al momento oportuno, podrá pasar siempre que siga teniendo cuidado con no caer en una seta iluminada o acercarse demasiado a las arañas. Si consigue llegar a lo más alto del árbol, podrá continuar su camino dejándose caer para llegar a un lago. En la caída le esperan luciérnagas que, con sus movimientos impredecibles, amenazan con hacerla desaparecer y tener que volver a empezar su recorrido.

3. **Fondo del lago: huida y desenlace**
   Cuando la sombra cae al fondo del lago, se encuentra con un pez linterna que lo persigue continuamente. Tendrá que ser rápido y esquivar su luz. Pero se encontrará con unas medusas que complicarán su huida y elementos del fondo del lago que

le bloquearan el camino o le protegerán de la luz del pez. Otro pez se unirá al ya existente cuando esté a punto de salir del lago, lo que le dificultará aún más escapar. Si lo consigue, después de su intrincado y agotador viaje, descubrirá a la salida con la luz del día, su verdadero propósito.

1.4.3. Fases del juego: implementación 3D

La versión tridimensional del juego se desarrolla íntegramente en el escenario de la casa del bosque. Cada nivel corresponde a una de las estancias de la vivienda, incluida su entrada, lo que permite una progresión narrativa y espacial coherente con la estructura del entorno.

1. **Entrada a la casa**
   COMPLETAR

2. **Cocina: el gato**
   Este nivel tiene lugar en la cocina de la antigua casa. La sala contiene varias fuentes de luz —como una vieja nevera abierta y la campana extractora— que dificultan el avance de la sombra, ya que representan una amenaza directa. El jugador deberá ingeniárselas para evitar o neutralizar estas fuentes luminosas con el fin de continuar su avance. Además, se presentará un puzzle que deberá resolver para espantar a un gato que bloquea el acceso a una rejilla de ventilación, la cual funcionará como vía de escape hacia la siguiente zona.

3. **Salón: el viejo televisor**
   Tras atravesar la rejilla, la sombra aparece oculta detrás de unas cajas apiladas, que sirven de cobertura frente a la única fuente de luz del salón: un televisor encendido. Esta luz actúa como obstáculo principal, por lo que el jugador deberá moverse estratégicamente entre las sombras proyectadas por cajas de pizza, paquetes olvidados y electrodomésticos antiguos. El objetivo será alcanzar la puerta abierta al otro lado de la sala sin ser expuesto a la luz directa.

4. **Salida de la casa**
   COMPLETAR

2. Desarrollo Técnico

2.1. Videojuego en 2D

2.1.1. Descripción

El objetivo de este apartado es ofrecer una visión general de la estructura y funcionamiento del videojuego *Vestigium*, sin entrar en detalles técnicos de implementación. Se describen las mecánicas, niveles, controles y reglas generales que definen la experiencia de juego, como base previa a su análisis técnico en secciones posteriores.

- **Tipo de juego y mecánica general:** Vestigium es un videojuego de plataformas 2D desarrollado con Pygame. Está diseñado para un solo jugador y su premisa

central consiste en evitar cualquier fuente de luz. La luz es la única amenaza en el juego: si el personaje entra en contacto con ella, muere de forma inmediata y reaparece en el último punto de control (checkpoint).

El jugador controla a una sombra que puede caminar hacia los lados (con las flechas izquierda y derecha), saltar (con la tecla espacio), saltar en paredes, planear mientras cae, rebotar sobre setas, nadar bajo el agua, escalar escaleras y recoger ciertos objetos clave. El salto en paredes requiere estar en contacto con la pared y pulsar espacio, lo que impulsa al personaje hacia la dirección contraria. No se permite saltar dos veces consecutivas sobre la misma pared.

Las setas permiten al jugador impulsarse verticalmente si cae sobre ellas. Una vez activadas, emiten luz durante un tiempo limitado. Esta luz aumenta progresivamente su intensidad, luego decrece y finalmente se apaga. Algunas de estas luces pueden bloquear el avance del jugador si no se tiene en cuenta su temporización.

- **Controles:**
  - Flechas izquierda y derecha: moverse lateralmente.
  - Espacio: salto (en tierra o pared).
  - Flechas (en el aire): movimiento lateral.
  - Flechas (en el minijuego y el nivel del lago): movimiento en dos ejes.
  - ESC: pausar el juego.
- **Jugabilidad y estructura por fases:** El juego está diseñado para jugarse de una sola vez y tiene una duración estimada de unos 15 minutos, dependiendo de la habilidad del jugador. No hay sistema de selección de dificultad ni configuración de controles. La estructura está dividida en tres niveles principales y una fase adicional en forma de minijuego:
  - **Tutorial: Cementerio** — El jugador comienza en un cementerio. En este nivel se muestra la mecánica principal de esquivado de luces y salto por plataformas sencillas.
  - **Nivel 1: El enterrador** - Para avanzar, el jugador debe obtener una llave que posee el enterrador mientras esquiva la luz de su farol gigante. Al colocar la llave en la puerta, se accede a una la siguiente fase adicional.
  - **Minijuego: Inserción de la llave** — Desde una vista cenital, el jugador controla una llave que debe llevar a una cerradura mientras esquiva luciérnagas. Se dispone de tres corazones (vidas) en este segmento que, en caso de agotarlas, llevarán al jugador al inicio del Nivel 1. El movimiento es libre en dos ejes mediante las cuatro flechas.
  - **Nivel 2: Árbol del bosque** — El jugador aparece cerca de un árbol hueco. Debe ascender por él rebotando en setas (que posteriormente emiten luz) y apoyándose en ramas. Debe evitar la luz que entra por rendijas y la de arañas que emiten luz desde su abdomen. A pesar de que obstaculizan el camino, las arañas cumplen un papel clave: tapan parcialmente la luz exterior, permitiendo avanzar.

- **Nivel 3: Lago subacuático** — Tras la caída desde la cima del árbol, el personaje aterriza en un lago. A partir de aquí se desplaza bajo el agua usando las cuatro flechas. Debe evitar el contacto con peces linterna que lo persiguen y medusas que emiten luz. También debe sortear obstáculos que bloquean el camino. Al final, un segundo pez linterna aparece de frente; el jugador debe salir rápidamente hacia la superficie para escapar.

Al finalizar el último nivel, se muestra una pantalla negra con un texto narrativo que concluye la historia. A continuación, aparece un botón que permite volver al menú principal.

El juego emplea un sistema de puntos de control. Al morir, el jugador reaparece en el último checkpoint alcanzado. Las únicas muertes posibles son por contacto con la luz o caída al vacío. No hay sistema de vidas, salvo en el minijuego.

Las transiciones entre niveles y muertes están acompañadas por efectos de *fade in* y *fade out*. En cualquier momento se puede pausar el juego (tecla ESC), accediendo a un menú que permite continuar, reiniciar el nivel, ir al menú de opciones o volver al menú principal. El menú de opciones permite ajustar el volumen de la música y de los efectos de sonido.

- **Audio:** Cada nivel, así como los menús, tienen una música de fondo específica. Además, hay efectos de sonido asociados a diversas acciones, tanto del jugador como de elementos del entorno (saltos, contacto con objetos, etc.).

2.1.1.1 Personajes El personaje principal es una sombra. No tiene habilidades ofensivas ni interacción directa con otros personajes. Su funcionalidad se basa en el movimiento y evasión.

Los personajes secundarios son principalmente funcionales o ambientales:

- **Enterrador:** Aparece en el primer nivel, custodia la llave necesaria para activar el minijuego.
- **Luciérnagas:** Aparecen en distintas fases y emiten luz al desplazarse.
- **Arañas:** Emiten luz, bloquean caminos, pero también permiten avanzar al cubrir fuentes de luz.
- **Medusas:** Se encuentran bajo el agua, flotan y emiten luz continua.
- **Peces linterna:** Persiguen al jugador e iluminan su camino.
- **Setas:** Permiten al jugador rebotar sobre ellas y emiten luz amenazante.

No existe una distinción estricta entre aliados y enemigos. Algunos elementos pueden ser perjudiciales o beneficiosos según el contexto.

2.1.1.2 Enemigos No hay enemigos directos en el sentido tradicional. La única amenaza es la luz. Esta puede provenir de:

- Elementos estáticos (farolas, rendijas)
- Elementos dinámicos o interactivos (setas, criaturas, farol)

El diseño general del juego busca que el jugador interprete continuamente los elementos del entorno, ya que una fuente de ayuda puede también ser un riesgo.

2.1.1.3 Objetos

- **Llave:** Objeto necesario para desbloquear el minijuego del primer nivel.

2.1.1.4 Diseño: guion, reglas, mecánica No existe narrativa explícita. La progresión se basa en el avance a través de escenarios y cambios de ambiente. La única regla constante es: evitar la luz.

Reglas principales:

- Contacto con la luz o caída por precipicios: muerte inmediata.
- Checkpoints automáticos.
- Tres vidas únicamente en el minijuego.
- Sin vidas ni barra de vida en el resto del juego.
- Sin dificultad seleccionable.
- Controles fijos por teclado.
- Duración estimada: 15 minutos.

2.1.2. Escenas

2.1.2.1 Metodología de desarrollo y reparto de trabajo Durante la primera fase del desarrollo técnico del videojuego, todo el equipo trabajó de forma conjunta con el objetivo de familiarizarse con la librería `pygame` y establecer las bases del funcionamiento interno del juego. Esta etapa inicial sirvió como adaptación al entorno de desarrollo y permitió definir conceptualmente la arquitectura general del sistema, sin implementar aún funcionalidades concretas.

Una vez superada esta fase, el trabajo se dividió de forma modular entre los miembros del equipo. Cada persona abordaba tareas específicas, tales como la implementación del sistema de salto, la gestión de luz, la música, entre otros. Cada módulo se desarrollaba en ramas independientes dentro del repositorio `git`, y una vez completado y probado, se realizaba el *merge* a la rama principal (`main`).

El desarrollo no siguió una estrategia incremental ni iterativa basada en versiones funcionales intermedias. En su lugar, se optó por una estrategia de ensamblado final: se fueron construyendo todos los elementos necesarios para el juego de manera paralela e independiente, y posteriormente se integraron en una versión final funcional.

El reparto de niveles fue el siguiente:

- **Fernando Álvarez de Legísima y Ana Bañobre Martínez:** desarrollo del primer nivel (cementerio) y el minijuego.
- **Xian Priego Martín e Iria Pardo Neira:** desarrollo del segundo y tercer nivel (árbol y lago).

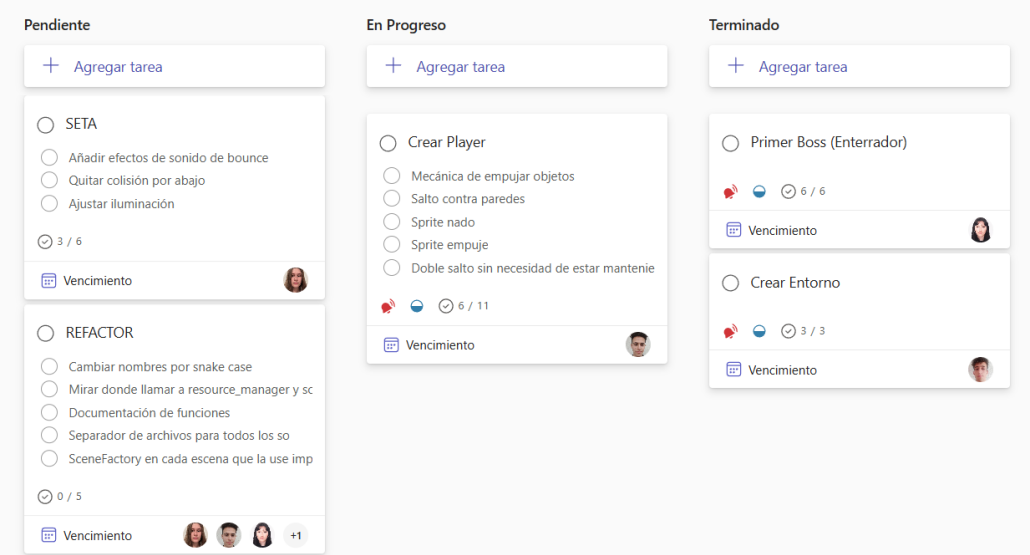
Aunque el trabajo fue mayoritariamente individual y modular, el reparto por niveles permitió que cada integrante tuviera una persona de referencia con la que colaborar directamente cuando surgieran dificultades dentro de su bloque de trabajo.

Además de las escenas específicas, existen componentes generales del juego que no pertenecen a una fase concreta, como el sistema de menús, la arquitectura base del motor del juego, el diseño de mapas, el personaje principal o los sistemas de luz y sonido. Estos elementos fueron desarrollados de forma mixta: de manera individual cuando se trataba de implementar funcionalidades específicas, y de forma conjunta en momentos de toma de decisiones de diseño.

- **Xian e Iria:** se centraron especialmente en los módulos de iluminación y el comportamiento de elementos dinámicos como las luciérnagas y las setas.
- **Fernando y Ana:** trabajaron principalmente en el sistema de construcción de niveles, incluyendo el diseño de suelos, paredes y plataformas.

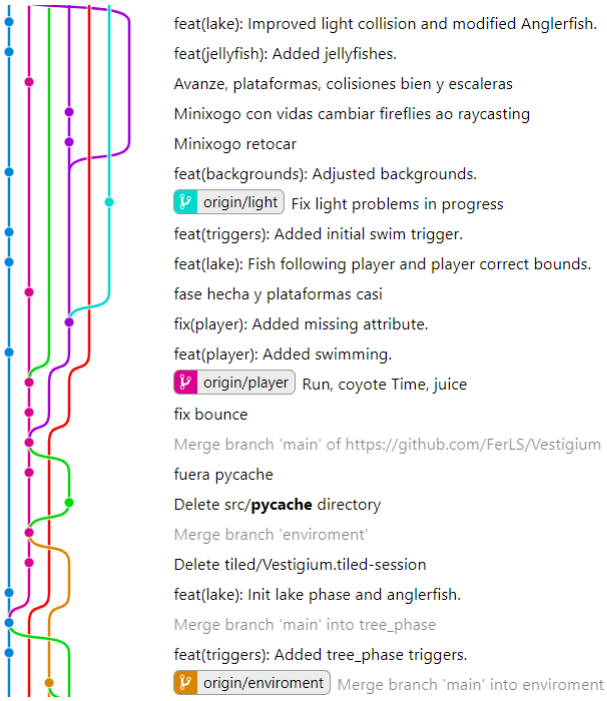
Gestión del flujo de trabajo: metodología Kanban

Para organizar las tareas y visualizar el estado de cada componente del proyecto, se empleó la metodología *Kanban* mediante un tablero digital que clasificaba las tareas en tres columnas principales: **Pendiente**, **En Progreso** y **Terminado**. Cada tarjeta contenía subtareas específicas y estaba asignada a los responsables correspondientes. Esto permitió mantener un flujo de trabajo claro, flexible y colaborativo.



Control de versiones con Git y estructura de ramas

Se utilizó `git` como sistema de control de versiones, con un enfoque basado en ramas temáticas. Cada funcionalidad o componente se desarrollaba en una rama independiente (`light`, `player`, `environment`, etc.), lo que facilitó el trabajo en paralelo y evitó conflictos durante la integración. Los *merges* se realizaban a la rama principal una vez que se completaban las pruebas correspondientes.



Calidad del código: uso de SonarLint

Para mantener un código limpio y sostenible, se utilizó `SonarLint`, una herramienta de análisis estático que ayudó a detectar errores comunes y mejorar la calidad del código. Gracias a SonarLint, se pudieron identificar variables no utilizadas, estructuras redundantes, problemas de estilo y posibles errores lógicos, permitiendo refactorizar de forma temprana y prevenir fallos en fases posteriores del desarrollo.

Un ejemplo claro de su utilidad fue la refactorización del método `update` de una clase encargada de controlar el comportamiento de una luciérnaga. Originalmente, el método contenía múltiples bloques de control con lógica embebida, lo cual dificultaba su lectura y mantenimiento.

Antes de la refactorización:

```
def update(self):
    if self.movement_type == "random":
    # Random movement
        delta = pygame.math.Vector2(
            random.uniform(-self.acceleration_change, self.
               acceleration_change),
            random.uniform(-self.acceleration_change, self.
               acceleration_change)
        )
```

```
            self.velocity += delta
            ...
        elif self.movement_type == "wave":
            self._move()
        elif self.movement_type == "vertical":
            self.rect.y += 1 * self.vertical_direction
            ...
        elif self.movement_type == "horizontal":
            self.rect.x += 1 * self.horizontal_direction
            ...
        # Light blinking
        self.blink_timer += 1
        ...
```

Después de la refactorización:

```
def update(self) -> None:
    self._update_position()
    self._update_blinking()
    self._update_light()

def _update_position(self) -> None:
    if self.movement_type == "random":
        self._update_random_movement()
    elif self.movement_type == "wave":
        self._move()
    elif self.movement_type == "vertical":
        self._update_vertical_movement()
    elif self.movement_type == "horizontal":
        self._update_horizontal_movement()

def _update_random_movement(self) -> None:
    delta = pygame.math.Vector2(...)
    ...
```

2.1.2.2 Descripción global de la Arquitectura La Figura 11 muestra la arquitectura general del sistema, centrándose en el flujo de control y las principales dependencias entre los componentes base del proyecto.

- **Main**: Es el punto de entrada del sistema. Su función principal es crear e inicializar el **Director**, quien toma el control de la ejecución del juego.
- **Director**: Es el núcleo de control de la aplicación. Su responsabilidad principal es gestionar el ciclo de vida de las **Scenes** (escenas), que representan los distintos estados del juego.
- **Scenes**: Este componente actúa como contenedor de la lógica del juego y se divide en diferentes tipos de escenas (como menús o fases del juego). Cada escena hace uso de diversos subsistemas para funcionar correctamente.

- **Dependencias de Scenes**:
  - **Entities**: Conjunto de entidades activas dentro del juego, como personajes, enemigos u objetos interactivos.
  - **Environment**: Componentes relacionados con el entorno del juego, como mapas, capas o cámaras.
  - **Utils**: Utilidades generales y funciones auxiliares que apoyan a otras partes del sistema.
  - **GUI**: Elementos de la interfaz gráfica del usuario (pantallas, botones, texto, etc.).
  - **Managers**: Componentes de gestión, como los encargados de recursos, sonido o control de escenas.

Este diagrama resume la estructura lógica principal del sistema y sirve como base para entender los distintos niveles de descomposición del proyecto.

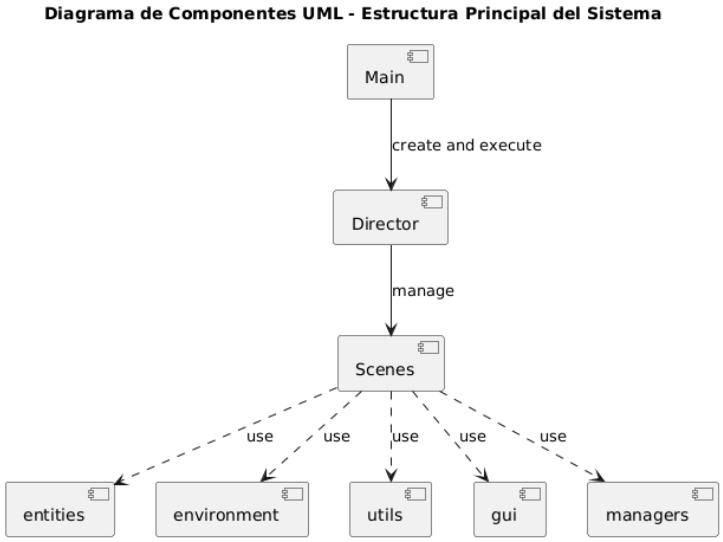


Figura 11: Diagrama de componentes UML del sistema a alto nivel.

La Figura 12 representa la organización interna del componente `Scenes`, que se divide en dos grandes ramas funcionales: `Menu` y `Phase`.

- **Scene**: Es la clase base de la que heredan todos los tipos de escena del sistema.
- **Menu**: Representa las escenas de tipo menú. Incluye:
  - `StartMenu`
  - `IntroMenu`
  - `PauseMenu`
  - `EndMenu`

Estas escenas dependen del sistema de interfaz gráfica de usuario (`gui`) y del gestor de sonido (`sound_manager`).

- **Phase**: Agrupa las escenas que representan fases jugables del juego. Incluye:
  - `CemeteryPhase`
  - `CemeteryBossPhase`
  - `MinigamePhase`
  - `TreePhase`
  - `LakePhase`

  Estas fases utilizan una mayor cantidad de componentes del sistema, incluyendo: `entities`, `environment`, `utils` y `managers`, pero no hacen uso directo de la interfaz gráfica. En una sección posterior, se definirán de forma más detallada cada una de las escenas.

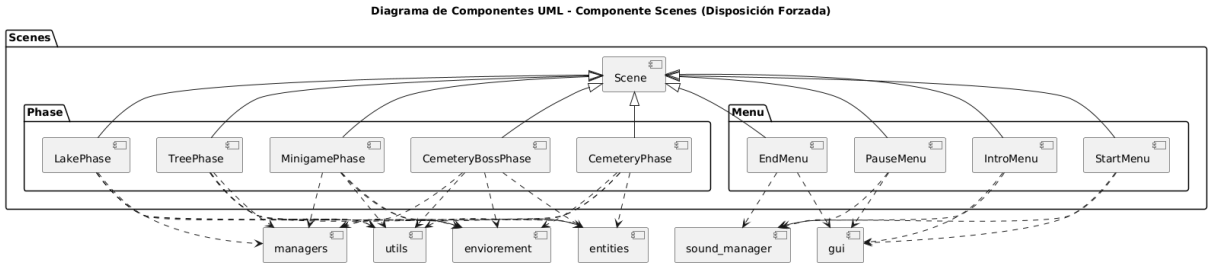


Figura 12: Diagrama de componentes UML del módulo `Scenes`.

La Figura 13 muestra la estructura del sistema de interfaz gráfica del usuario (GUI), compuesto por pantallas y elementos interactivos organizados jerárquicamente.

- **GuiScreen**: Representa una pantalla completa de la interfaz. Contiene una lista de elementos visuales heredados de `GuiElement`, y además, depende del `ResourceManager` para renderizar sus imágenes de fondo.
- **GuiElement**: Es la clase base de todos los elementos gráficos. Existen dos tipos principales:
  - `GuiText`: Elemento visual que muestra texto en pantalla. Utiliza el `ResourceManager` para cargar fuentes y estilos gráficos.
  - `GuiSlider`: Control deslizante que permite modificar valores de forma interactiva. Tiene dos subtipos:
    - `MusicVolumeSlider`
    - `SoundEffectsSlider`

    Ambos sliders se conectan al `SoundManager` para ajustar el volumen correspondiente.

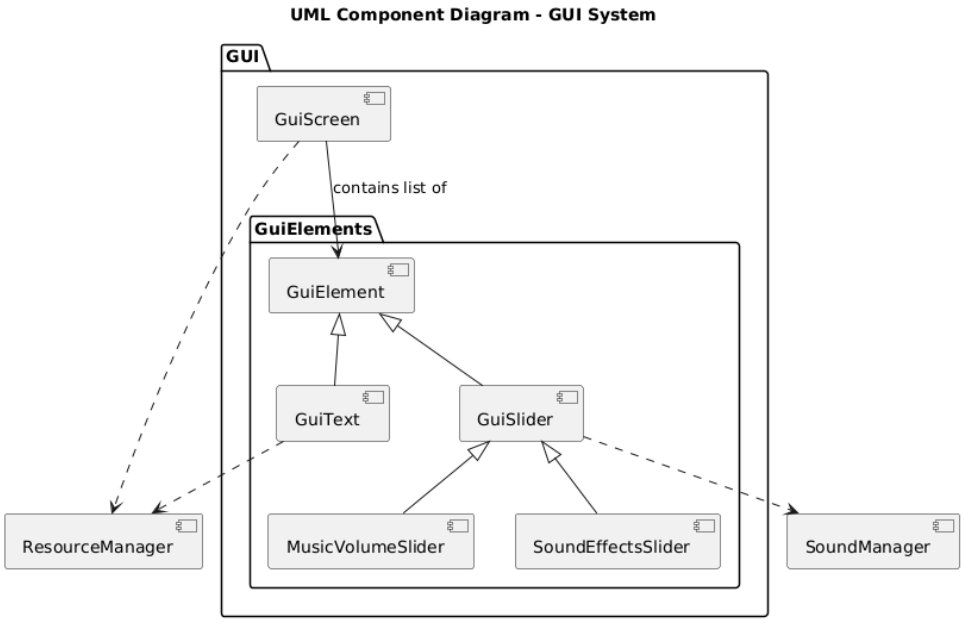


Figura 13: Diagrama de componentes UML del sistema de interfaz gráfica (GUI).

El paquete `Utils` agrupa una serie de componentes funcionales reutilizables por distintos módulos del sistema, tal y como se muestra en la Figura 14.

- `Constants`, `Images`, `Juice` y `Trigger` son módulos independientes que ofrecen, respectivamente: constantes globales, herramientas para gestión de imágenes, efectos visuales y detección de condiciones o eventos.
- `FadeTransition` define un sistema de transición visual genérico, del cual derivan dos implementaciones específicas:
  - `FadeIn`
  - `FadeOut`
- `Light` es el componente base para los efectos de iluminación, del que derivan:
  - `CircularLight`
  - `ConeLight`

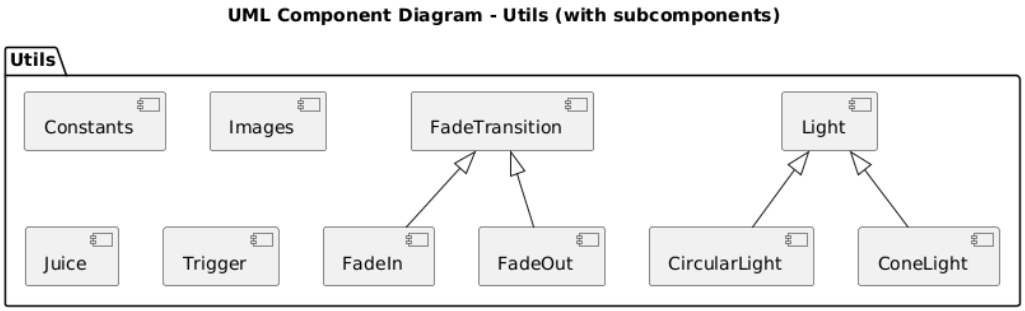


Figura 14: Diagrama de componentes UML del paquete `Utils`.

La Figura 15 muestra las relaciones entre los componentes que conforman el paquete `Managers`, los cuales desempeñan funciones clave en la gestión de recursos y escenas dentro del sistema.

- `ResourceManager` es responsable de cargar y almacenar de forma centralizada los recursos compartidos del sistema, como fuentes, sonidos e imágenes. Su objetivo principal es evitar cargas duplicadas, mejorando la eficiencia y reduciendo el uso de memoria.
- `SoundManager` gestiona la reproducción de sonidos y música, y delega la carga de dichos recursos en el `ResourceManager`, del cual depende directamente.
- `SceneManager` actúa como un envoltorio (wrapper) especializado de las funciones del `Director` relacionadas con la gestión de escenas. Su diseño responde a la necesidad de evitar dependencias circulares entre `Director` y `Scene`. Puede considerarse una implementación parcial del patrón *Factory*, ya que encapsula la lógica de creación de instancias de escenas de forma centralizada, delegando el control a `Director`.

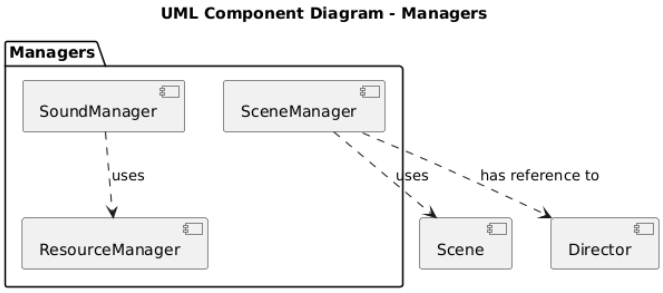


Figura 15: Diagrama de componentes UML del paquete `Managers`.

La Figura 16 muestra la estructura del paquete `Environment`, que agrupa los componentes encargados de representar el entorno gráfico y espacial del juego.

- `Tilemap` representa el mapa de tiles del nivel y contiene múltiples instancias de `Layer`, que se apilan para construir visualmente el entorno jugable. Para diseñar los niveles, se utilizó Tiled, una herramienta que permite crear mapas basados en tiles organizados en esas capas. Pueden utilizarse para separar distintos elementos del escenario, como el fondo, los obstáculos y los objetos interactuables, lo que facilita la organización y el control del entorno del juego.
- `Background` define el fondo visual de una escena y está compuesto por varias `BackgroundLayer`, que permiten construir fondos dinámicos con efecto *Parallax*.
- `Camera` es la responsable de calcular el *scroll*, es decir, el desplazamiento de la vista en función de la posición del jugador u otras entidades. Gracias a ella, se logra simular movimiento dentro de mapas de gran tamaño.
- Todos los componentes anteriores hacen uso de `Constants`, que contiene valores reutilizables como tamaños de tiles, colores, velocidades, etc.

- Además, `Background` también depende del `ResourceManager` para cargar y gestionar los recursos visuales necesarios para componer las capas del fondo.

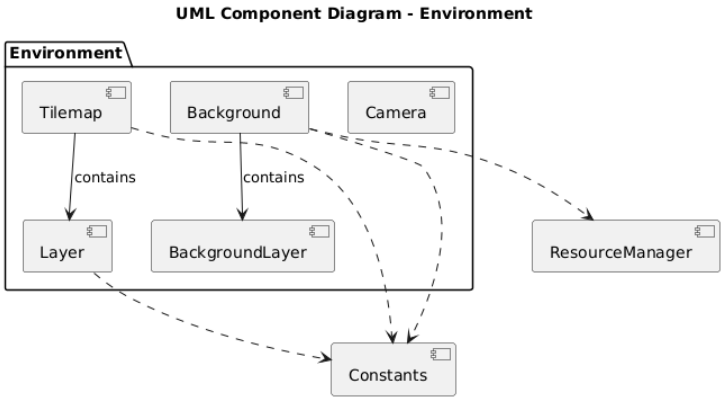


Figura 16: Diagrama de componentes UML del paquete `Environment`.

La Figura 17 muestra los distintos NPCs (*Non-Playable Characters*) definidos en el sistema, agrupados dentro del paquete `Entities.NPCs`. Todos estos personajes comparten una lógica común de interacción y dependencias.

- El grupo está formado por: `Anglerfish`, `Ant`, `Firefly`, `Gravedigger`, `Jellyfish` y `Mushroom`.
- Todos estos NPCs dependen del componente `Light`, lo que indica que emiten o reaccionan a efectos de iluminación dentro del entorno.
- También hacen uso del `SoundManager` para reproducir efectos de sonido propios de su comportamiento o interacción.
- Finalmente, cada NPC accede al `ResourceManager` para obtener los recursos gráficos y sonoros necesarios, asegurando que no se carguen múltiples veces en memoria.

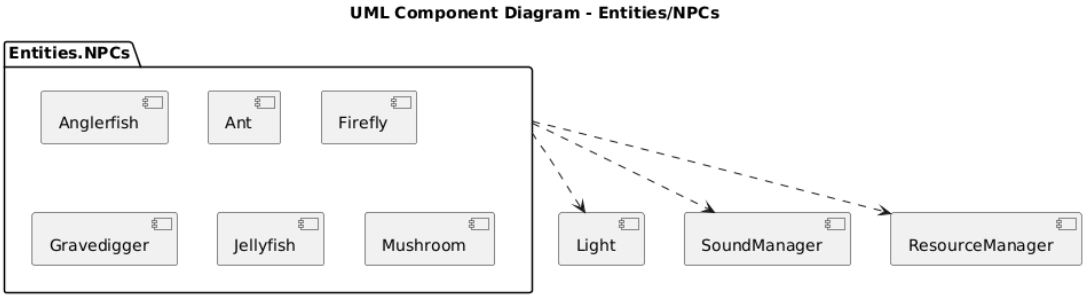


Figura 17: Diagrama de componentes UML del paquete `Entities.NPCs`.

La Figura 18 muestra los componentes que conforman el paquete `Entities.Objects`, el cual agrupa los objetos interactuables o coleccionables del juego.

- Entre los objetos representados se encuentran: `KeyItem`, `Lantern`, `Lifes` y `Lock`, todos ellos diseñados para ser utilizados por el jugador durante las distintas fases.
- Estos objetos acceden al `ResourceManager` para obtener sus recursos gráficos y sonoros, garantizando una gestión eficiente de estos.
- También dependen del `SoundManager`, lo que indica que todos ellos pueden emitir efectos sonoros al ser recogidos, utilizados o activados dentro del juego.

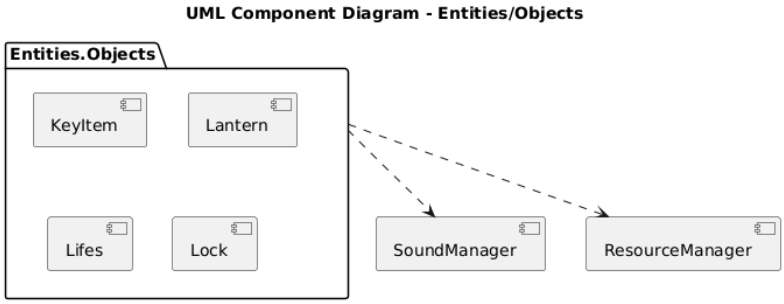


Figura 18: Diagrama de componentes UML del paquete `Entities.Objects`.

La Figura 20 muestra los componentes del paquete `Entities.Players`, el cual contiene las entidades directamente controladas por el jugador.

- `Player` representa al personaje jugable principal. Este componente es uno de los más conectados del sistema, ya que depende de múltiples elementos para poder funcionar correctamente:
  - `Tilemap`, para conocer la disposición del entorno y gestionar colisiones.
  - `Camera`, realmente, para aplicar el scroll no hace falta que se use una instancia de cámara, pero esto se hace para un uso específico de una fase que se explicará más adelante.
  - `SoundManager`, para reproducir efectos relacionados con el movimiento, daño, acciones, etc.
  - `ResourceManager`, del que obtiene sprites, sonidos y otros recursos visuales.
- `Key` representa una llave manejada por el jugador. También accede al `ResourceManager` para cargar los recursos gráficos necesarios.

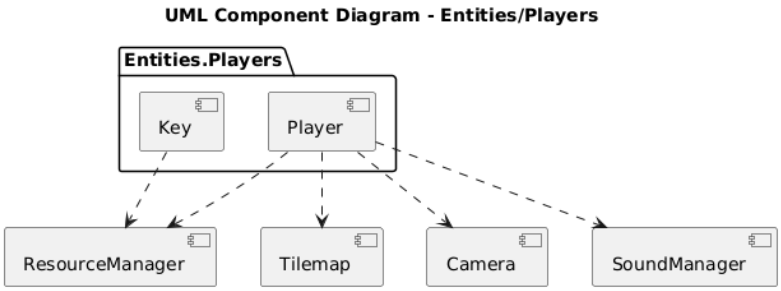


Figura 19: Diagrama de componentes UML del paquete `Entities.Players`.

2.1.2.3 Transición entre escenas El sistema de transición entre escenas está gestionado por la clase `Director`, que mantiene una pila de escenas activas. Las transiciones se realizan mediante métodos específicos para cambiar, apilar o finalizar escenas. Todas las transiciones están acompañadas por efectos visuales de entrada y salida (`fade in` / `fade out`).

El flujo principal del juego sigue la siguiente secuencia:

- Desde el **menú inicial** (`StartMenu`), el jugador puede:
  - Iniciar el juego, lo que lleva a la primera escena: `IntroMenu`.
  - Ir a opciones.
  - Salir del juego.
- En IntroMenu se muestra un texto introductorio a la historia del personaje principal, la sombra. Al pulsar el botón de Start, el usuario entrará en la primera fase: CemeteryPhase.
- En **CemeteryPhase**, al alcanzar el punto final del nivel, se accede a `CemeteryBossPhase`. Si el jugador muere durante esta fase, reaparece en el último punto de control.
- En **CemeteryBossPhase**, si el jugador consigue la llave de la puerta, se activa un nivel de minijuego adicional que debe completar para pasar a la siguiente fase de plataformas: el **MinigamePhase**.
- En **MinigamePhase**, si se supera el reto, se transiciona a `TreePhase`. Si se pierden las tres vidas, se vuelve al inicio de `CemeteryBossPhase`, siendo necesario recuperar la llave de nuevo y reiniciar el minijuego.
- Desde **TreePhase**, se accede a `LakePhase` al superar el nivel. Si el jugador muere, reaparece en el último punto de control.
- Desde **LakePhase**, al completar la fase se transiciona al `EndMenu`, que muestra una pantalla final con texto. Desde este menú se puede volver al `StartMenu`.
- En cualquier momento durante una fase jugable (`CemeteryPhase`, `CemeteryBossPhase`, `TreePhase`, `LakePhase`, `MinigamePhase`), el jugador puede pulsar la tecla `ESC` para abrir el `PauseMenu`. Desde este menú puede:
  - Retomar la partida desde el punto actual.
  - Reiniciar la fase desde el inicio.
  - Volver al menú inicial.
  - Ir a opciones.

A continuación se incluye un diagrama de estados que representa gráficamente las transiciones entre escenas.

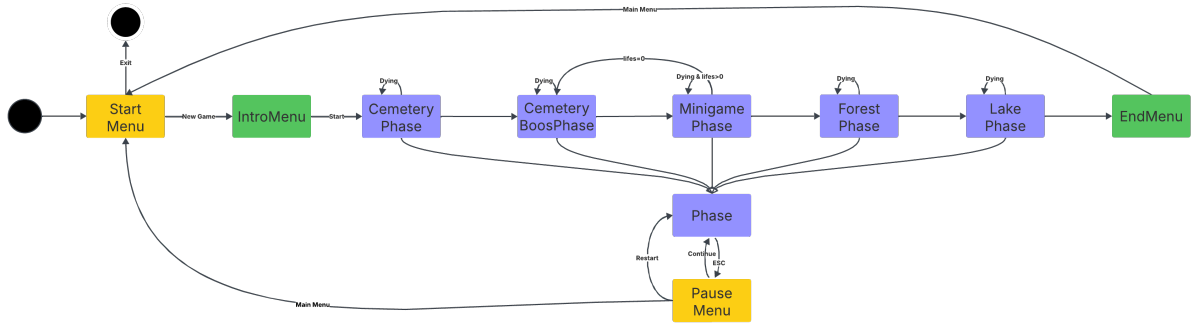


Figura 20: Diagrama de estados que muestra la transición entre escenas.

2.1.2.4 Escenas

Clases base: Phase y Menu Todas las escenas del juego heredan de una de las dos clases base: `Phase` o `Menu`, ambas derivadas a su vez de la clase abstracta `Scene`. Esta jerarquía permite encapsular comportamientos comunes, reducir duplicación de código y garantizar una interfaz uniforme para el ciclo de vida de cada escena.

- **Menu**: clase base para escenas de interfaz (como menús de pausa, inicio o final). Gestiona una lista apilable de pantallas GUI (`screen_list`) y delega en ellas las acciones de actualización, eventos y dibujo. Incorpora un `SoundManager` para controlar el audio del menú. No requiere manejo de lógica de juego, físicas ni entidades.
- **Phase**: clase base para fases jugables. Proporciona funcionalidad avanzada como gestión de recursos (`Tilemap`, `Background`), control de cámara, puntos de reaparición (checkpoints), efectos de transición (`FadeIn`, `FadeOut`), gestión de audio contextual, triggers interactivos y reaparición del jugador. Es altamente personalizable mediante la sobrescritura de sus métodos.
- Ambas clases implementan la misma interfaz de métodos: `update`, `events`, `draw` y `continue_procedure`, facilitando su manipulación uniforme por parte del `Director` y el `SceneManager`.

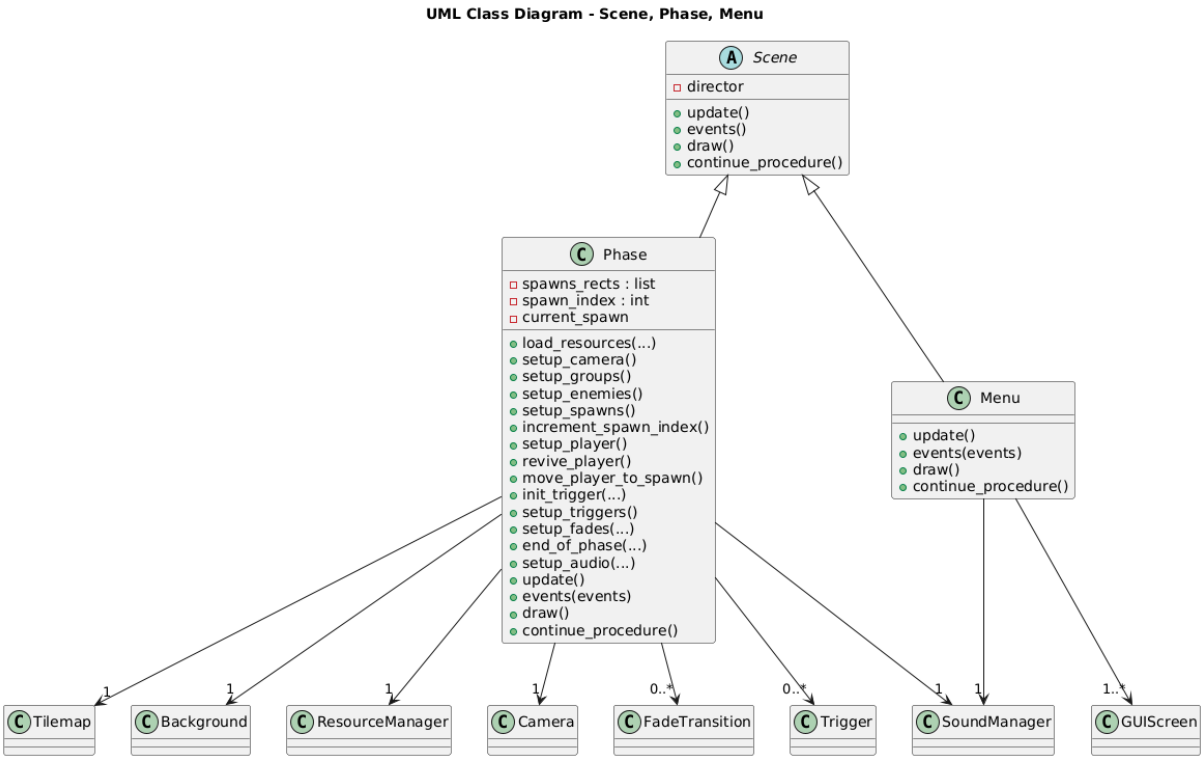


Figura 21: Diagrama de clases que muestra las relaciones de herencia entre la superclase Scene y las subclases Phase y Menu.

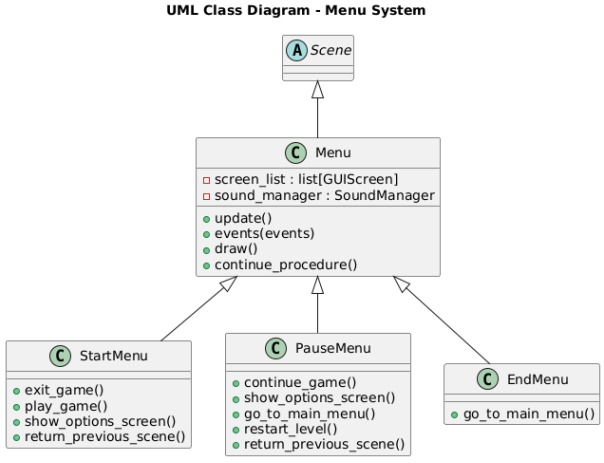


Figura 22: Diagrama de clases que muestra las relaciones de herencia concretas de la clase Menu.

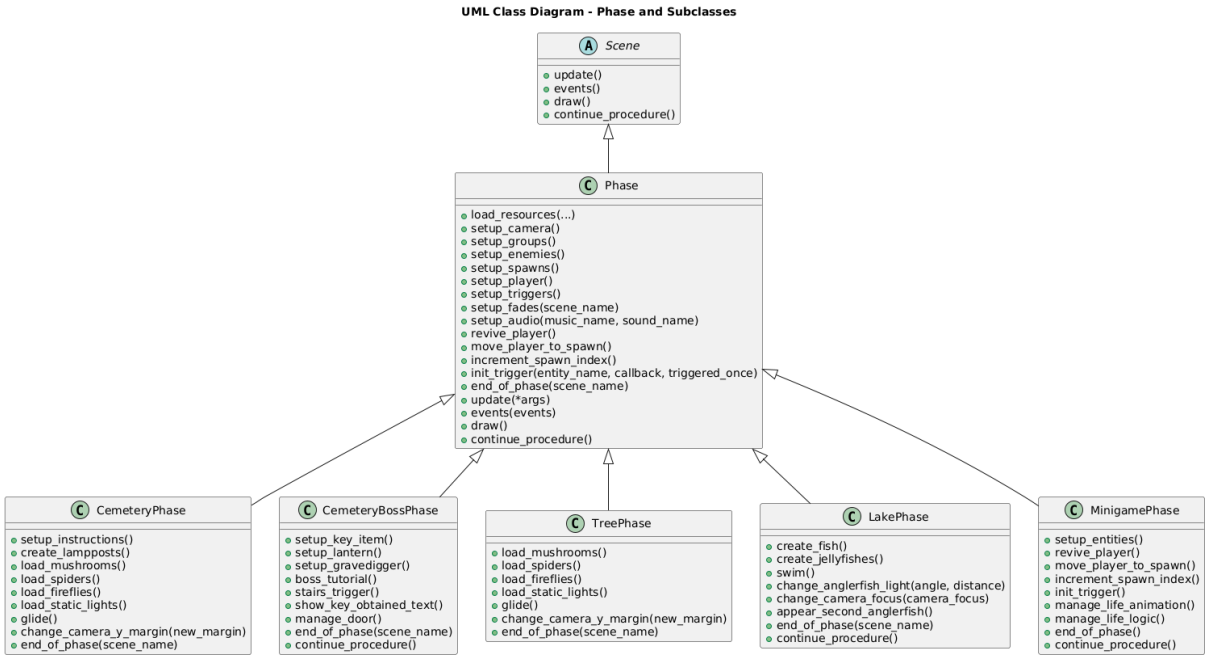


Figura 23: Diagrama de clases que muestra las relaciones de herencia de la clase Phase y sus subclases CemeteryPhase, CemeteryBossPhase, TreePhase, LakePhase y MinigamePhase.

Una de las decisiones de implementación que hay que comentar fue la incorporación del método `continue_procedure()` en la clase `Scene`. Este método permite reanudar correctamente una escena tras ser apilada y luego retomada desde el bucle principal del `Director`. En concreto, si se ha marcado una bandera de reinicio (`restart_flag`), el bucle ejecuta:

```
if self.restart_flag:
    self.restart_flag = False
    scene.continue_procedure()
```

Esto habilita comportamientos personalizados al retomar la escena, como por ejemplo reanudar efectos de sonido o reiniciar animaciones. Cada subclase puede redefinir este método para adaptar el comportamiento de reentrada a sus necesidades.

Escena 1: StartMenu

Descripción El `StartMenu` representa el menú inicial del juego. Es la primera interfaz que ve el jugador y le permite iniciar la partida, acceder a las opciones o salir del juego. También inicia la música de fondo característica de esta escena.

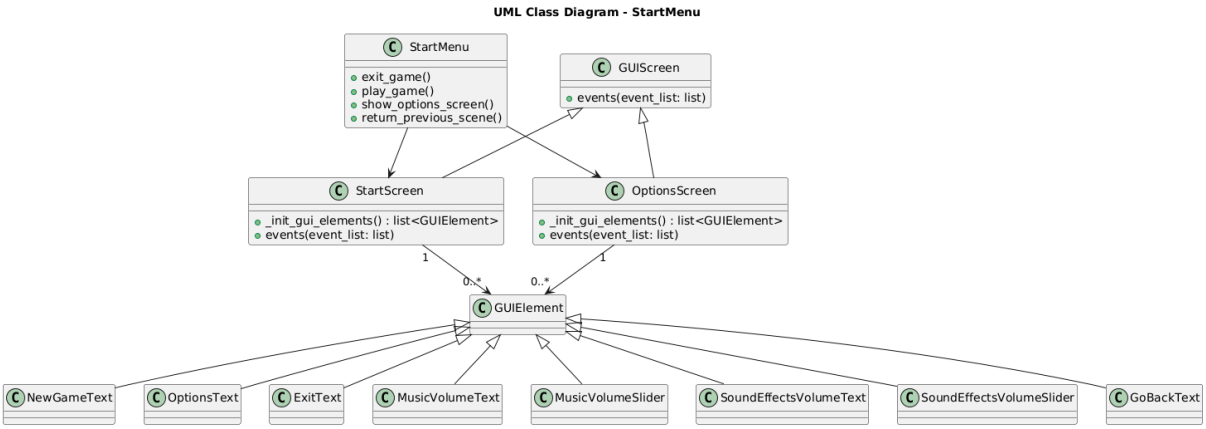


Figura 24: Diagrama de clases que muestra las relaciones de StartMenu.

Modelo

Análisis del diseño

- Hereda de `Menu`, por lo que reutiliza toda la lógica para gestionar pantallas y eventos.
- Se añade una pantalla principal (`StartScreen`) al iniciar el menú, y la música de fondo comienza automáticamente.
- Permite lanzar la escena introductoria `IntroMenu` al iniciar el juego, acceder a la pantalla de opciones o cerrar la aplicación directamente.
- Mantiene una estructura limpia, con una función por acción que llama directamente a métodos del `Director`.

Detalles de implementación

- Se reproduce música automáticamente al iniciar el menú:

```
self.sound_manager.play_music("start_menu.mp3", "assets\\music",
    -1)
```

- Se apila la pantalla principal del menú sobre la lista de pantallas activas:

```
self.screen_list.append(
    StartScreen(self, "assets\\images\\backgrounds\\
       main_menu_background")
)
```

- Se inicia el juego apilando la escena inicial:

```
self.director.scene_manager.stack_scene("IntroMenu")
```

- Se accede a las opciones añadiendo otra pantalla sobre la actual:

```
self.screen_list.append(
    OptionsScreen(self, "assets\\images\\backgrounds\\
        options_menu_background")
)
```

Escena 2: IntroMenu

Descripción La escena `IntroMenu` marca el inicio de la partida. Muestra una pantalla inicial con el mensaje que cuenta la historia, y permite al jugador empezar el juego. Actúa como introducción del juego y marca el principio de la nueva partida.

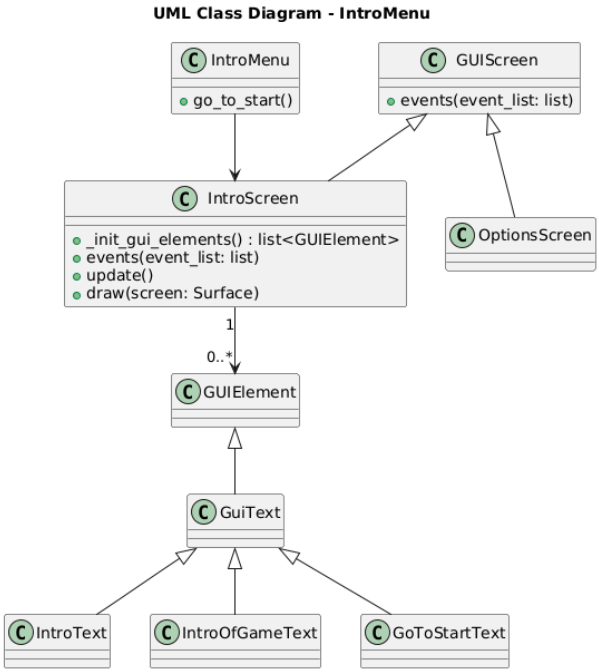


Figura 25: Diagrama de clases que muestra las relaciones de IntroMenu.

Modelo

Análisis del diseño

- `IntroMenu` hereda de la clase `Menu`, aprovechando toda la lógica compartida para la gestión de pantallas de menú y la herencia de `Scene`.
- Al inicializarse, añade una música ambiental mediante el `SoundManager`, creando un ambiente enigmático que sella el inicio de la experiencia de juego
- La escena incorpora un único `IntroScreen`, que contiene el mesaje de la historia inicial. No permite interacción más allá del botón que permite empezar la partida.
- La interacción con el `Director` y el `SceneManager` permite transicionar a la siguiente escena jugable (`CemeteryPhase`) mediante el método `stack_scene()`, evitando dependencias directas entre escenas promoviendo un diseño desacoplado.

Detalles de implementación

- No se apilan múltiples pantallas ni se ofrece navegación dentro del menú. Esto mantiene la escena simple, directa y con un solo propósito.

  ```
  self.screen_list.append(IntroScreen(self))
  ```

- La transición a la primera fase se realiza con `change_scene(ÇemeteryPhase")`.

  ```
  def go_to_start(self):
      self.director.scene_manager.change_scene("CemeteryPhase")
  ```

Escena 3: PauseMenu

Descripción El `PauseMenu` permite pausar el juego en cualquier momento durante una fase mediante la tecla ESC. Desde este menú, el jugador puede reanudar la partida, reiniciar la fase actual, acceder a las opciones o regresar al menú principal. Además, detiene todo el sonido del juego para reforzar la sensación de pausa.

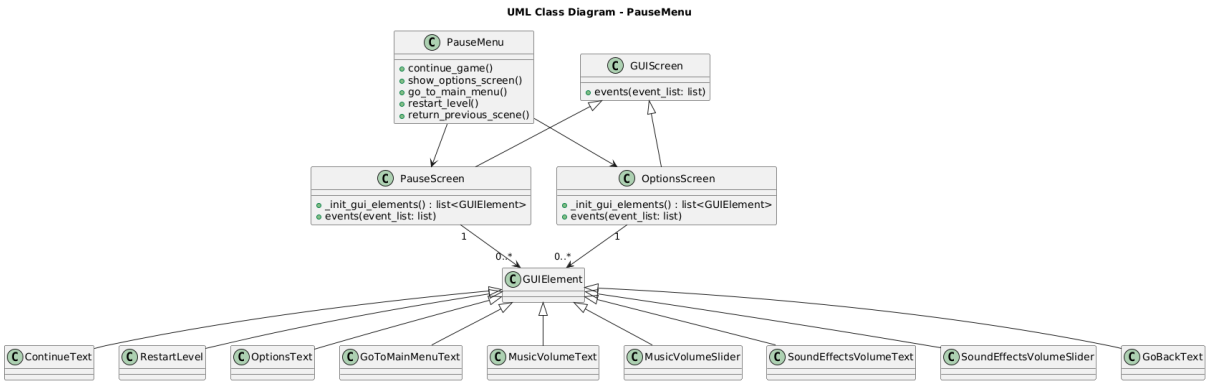


Figura 26: Diagrama de clases que muestra las relaciones de PauseMenu.

Modelo

Análisis del diseño

- Reutiliza la lógica de `Menu` para manejar pantallas, pero introduce acciones específicas ligadas al flujo de las fases.
- La música y los sonidos se detienen al entrar y se reanudan al salir del menú.
- Proporciona una opción para reiniciar la escena actual reutilizando su identificador dinámicamente.

Detalles de implementación

- Se detienen todos los sonidos y música al abrir el menú:

```
self.sound_manager.pause_music()
self.sound_manager.stop_all_sounds()
```

- Se reanuda la música y se cierra la escena al continuar la partida:

```
self.sound_manager.resume_music()
self.director.finish_current_scene()
```

- Se reinicia la escena actual usando su nombre:

```
scene_name = self.director.get_current_scene_name()
self.director.scene_manager.change_scene(scene_name)
```

- Se permite volver al menú principal directamente:

```
self.director.scene_manager.change_scene("StartMenu")
```

Escena 4: EndMenu

Descripción El `EndMenu` es el último menú del juego. Se muestra al completar la última fase y presenta una pantalla de cierre. Desde él, el jugador puede regresar al menú inicial para reiniciar el juego si lo desea.

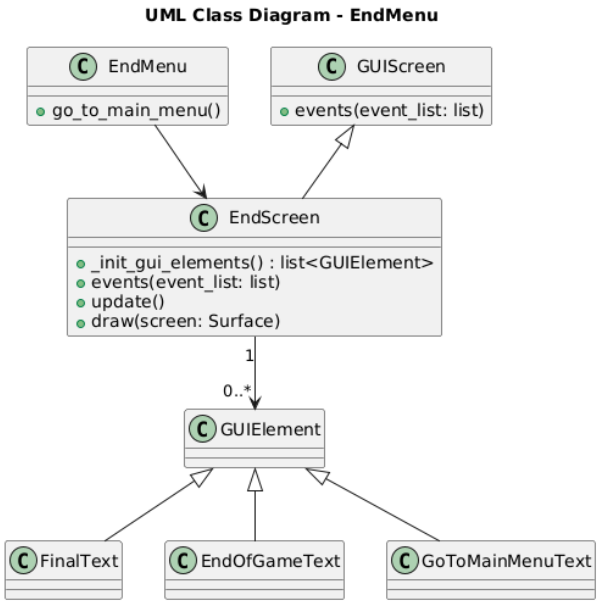


Figura 27: Diagrama de clases que muestra las relaciones de EndMenu.

Modelo

Análisis del diseño

- Al igual que otros menús, hereda de `Menu` y utiliza `screen_list` para gestionar la interfaz.
- Como `IntroMenu`, no incluye opciones múltiples: solo muestra una pantalla totalmente negra pero permite volver al menú principal.
- La música de fondo es la misma que la del nivel final, para mantener la sensación de continuidad. Sin embargo, todos los sonidos ambientales activos se detienen al llegar a esta escena para generar una atmósfera de cierre.

Detalles de implementación

- Se detienen todos los sonidos al iniciar la escena:

```
self.sound_manager.stop_all_sounds()
```

- Se apila la pantalla final del juego:

```
self.screen_list.append(EndScreen(self))
```

- El jugador puede volver al menú principal con una llamada directa:

```
self.director.scene_manager.change_scene("StartMenu")
```

Escena 5: CemeteryPhase

Descripción La escena `CemeteryPhase` es la primera fase jugable tras el menú de introducción. Presenta un entorno oscuro y misterioso donde el jugador debe superar obstáculos luminosos, aprender las mecánicas básicas de movimiento y enfrentarse a enemigos elementales como las luciérnagas. Introduce el sistema de puntos de control y condiciones de muerte ambiental (luces estáticas o caída al vacío).

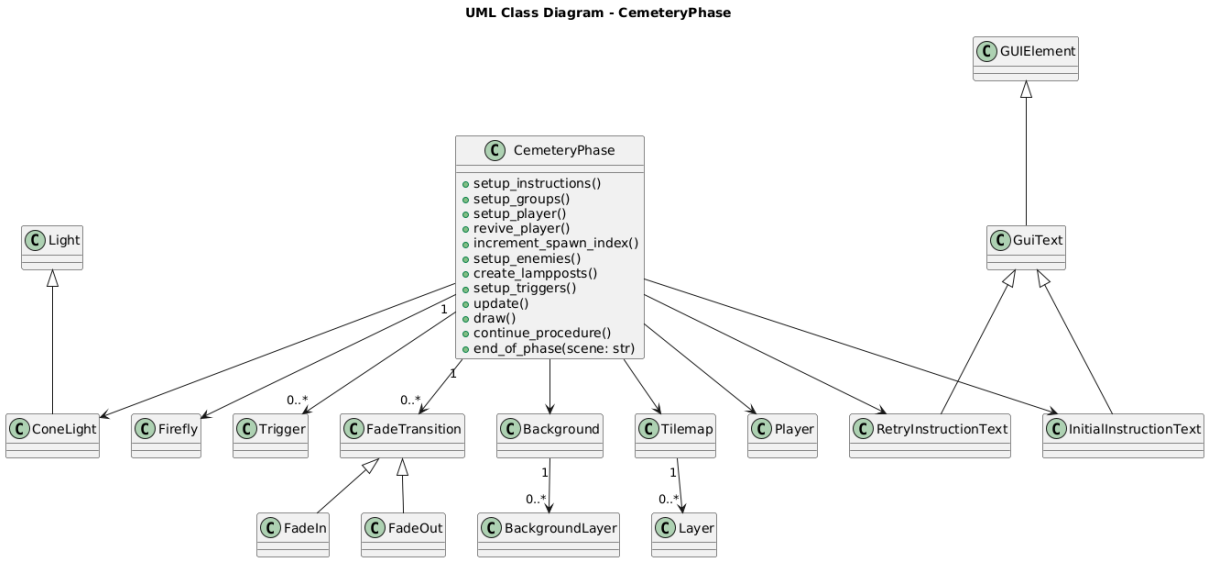


Figura 28: Diagrama de clases que muestra las relaciones de CemeteryPhase.

Modelo

Análisis del diseño

- La clase hereda de `Phase`, por lo que reutiliza el sistema genérico de gestión de recursos, cámara, audio, triggers y respawns.
- Define un método propio `setup_instructions` para mostrar mensajes visuales guiando al jugador. Este se actualiza dinámicamente según el progreso.
- La fase introduce un sistema de **luciérnagas** que reaccionan. Se dividen en dos grupos (movimiento vertical y horizontal) y están integradas con el sistema de colisiones letales.
- Implementa un efecto de parpadeo personalizado para los postes de luz con temporizador y desfase individual, utilizando la clase `ConeLight`.
- Sobrescribe la lógica de `update` para manejar el comportamiento de colisiones, parpadeos de luz y condiciones de muerte, combinando elementos visuales, efectos y estado del jugador.
- Utiliza triggers personalizados conectados con los fades para definir tanto el final de la fase como la condición de muerte.

Detalles de implementación

- El texto de instrucciones cambia según el progreso del jugador, sobrescribiendo `revive_player` e `increment_spawn_index`:

```
self.instruction_text = RetryInstructionText(self.screen, (50,
    100))
super().revive_player()

self.instruction_text.visible = True
super().increment_spawn_index()
```

- Se añade un efecto de parpadeo a cada poste de luz con un desfase específico por índice:

```
adjusted_timer = (self.lamppost_blink_timer - light.blink_offset
    ) % 6
if 0 <= adjusted_timer < 1:
     light.intensity = adjusted_timer
...
```

- Se comprueba la intensidad de luz para matar al jugador si hay colisión:

```
if light.intensity > 0.4 and self.player.
    check_pixel_perfect_collision(light):
     self.player.dying()
     self.fades['death_fade_out'].start()
```

- Se utilizan triggers mapeados desde el tilemap que inician efectos de fade:

```
self.init_trigger("death", lambda: self.fades['death_fade_out'].
    start(), triggered_once=False)
self.init_trigger("end_of_phase", lambda: self.fades['fade_out'
    ].start())
```

- El final de la fase lanza una transición definida con el nombre de la siguiente escena:

```
def end_of_phase(self, scene: str):
    self.director.scene_manager.change_scene(scene)
```

Escena 6: CemeteryBossPhase

Descripción La escena `CemeteryBossPhase` es una fase de transición con estructura de reto, previa al minijuego. Introduce nuevos elementos jugables como enemigos más complejos (`Gravedigger`), un objeto clave a recoger (`KeyItem`) y una linterna móvil que funciona como amenaza. Su diseño mezcla exploración, plataformas y desbloqueo condicional.

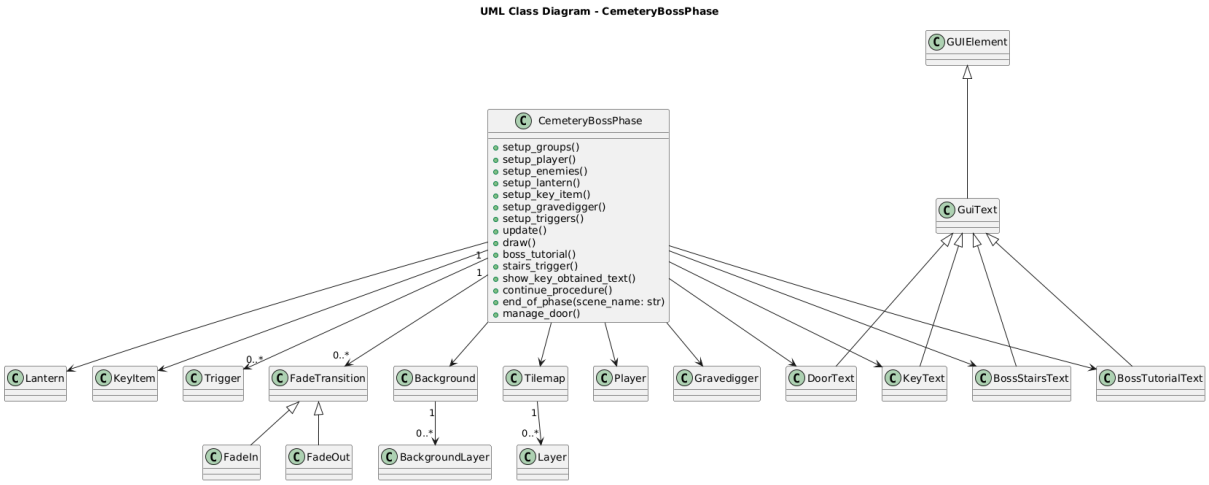


Figura 29: Diagrama de clases que muestra las relaciones de CemeteryBossPhase.

Modelo

Análisis del diseño

- Hereda de `Phase`, reutilizando la lógica de cámara, carga de recursos, triggers, audio y efectos de transición.
- Implementa enemigos y objetos personalizados como la `Lantern` (con ruta), el `Gravedigger` y un `KeyItem` que debe recogerse para desbloquear la puerta final.
- El jugador tiene un comportamiento personalizado al entrar en esta fase: su salto es limitado para aumentar la dificultad en las secciones de plataformas.

- Se definen múltiples triggers de eventos únicos, como mostrar tutoriales o controlar si el jugador puede pasar por la puerta final.

Detalles de implementación

- Se modifica la potencia de salto del jugador al iniciar la escena, para aumentar la dificultad de las plataformas:

```
self.player.jump_power_coyote = -4 * SCALE_FACTOR
```

- Se construye una ruta para la linterna enemiga a partir de puntos definidos en el mapa:

```
path_points = {
    int(obj.name): (obj.x, obj.y)
    for obj in self.foreground.tmx_data.objects
    if obj.type == "Point"
}
path_points = [path_points[i] for i in sorted(path_points.keys()
    )]
self.lantern = Lantern(position=path_points[0], path=path_points
    , speed=5)
```

- El objeto clave `KeyItem` se posiciona en el mapa mediante el método del tilemap:

```
key_spawn = self.foreground.load_entity("key_spawn")
self.key = KeyItem(key_spawn.x, key_spawn.y)
```

- Se utiliza un sistema de triggers para mostrar textos contextuales, tutoriales o permitir el paso por la puerta si el jugador tiene la llave:

```
self.init_trigger("tutorial_trigger", lambda: self.boss_tutorial
    ())
self.init_trigger("stairs_trigger", lambda: self.stairs_trigger
    ())
self.init_trigger("end_of_phase", lambda: self.manage_door(),
    triggered_once=False)
```

- El método `manage_door` verifica si la llave ha sido recogida para permitir la transición a la siguiente escena:

```
if self.key.picked:
    self.fades["fade_out"].start()
    self.player.jump_power_coyote = -6 * SCALE_FACTOR
```

Escena 7: MinigamePhase

Descripción La escena `MinigamePhase` actúa como un minijuego autónomo dentro del flujo principal. Introduce una mecánica de precisión y evasión: el jugador, representado por una llave, debe alcanzar una cerradura mientras evita luciérnagas hostiles. El jugador tiene un número limitado de vidas, y la dificultad está centrada en reflejos y control fino.

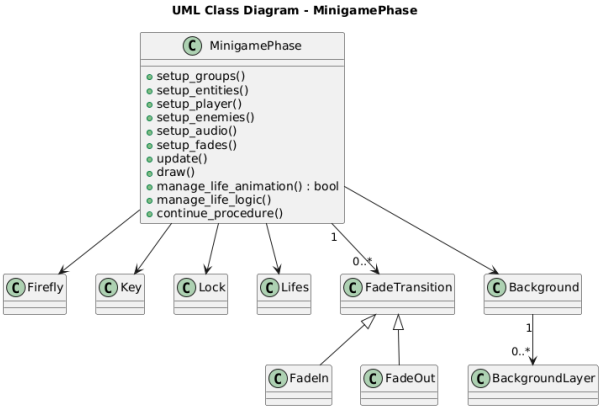


Figura 30: Diagrama de clases que muestra las relaciones de MinigamePhase.

Modelo

Análisis del diseño

- Aunque hereda de `Phase`, esta escena ignora muchos de sus mecanismos base: no usa triggers, spawns ni transición estándar, lo que indica un diseño aislado y personalizado.
- Los enemigos (`Firefly`) siguen trayectorias predefinidas o aleatorias, y sus colisiones afectan a la barra de vidas (`Lifes`), en lugar de aplicar muerte directa.
- Se incorporan entidades personalizadas como `Key`, `Lock` y `Lifes`, que encapsulan la lógica del reto.
- Utiliza dos tipos de transición diferenciadas (`fade_out_win` y `fade_out_loose`) según si el jugador tiene éxito o pierde todas las vidas.

Detalles de implementación

- La escena sobrescribe e inutiliza varios métodos de la clase base, como la gestión de puntos de control, triggers o respawns:

```
def setup_spawns(self):
    pass

def init_trigger(self, entity_name, callback, triggered_once=
   True):
    pass
```

- La lógica central gira en torno a una llave que el jugador controla, y su interacción con una cerradura:

```
if pygame.sprite.collide_mask(self.key, self.lock):
    for firefly in self.fireflies_group:
       firefly.stop()
    if self.lock.end:
        self.fades['fade_out_win'].start()
```

- Si el jugador colisiona con luces mientras tiene vidas, se reinicia su posición y se reduce la cantidad:

```
if self.lifes.ammount > 0:
    self.lifes.decrease()
    self.key.reset()
    for firefly in self.fireflies_group:
        firefly.reset("life_decreased", delay=True)
```

- Si pierde todas las vidas, se lanza la transición correspondiente:

```
if self.lifes.ammount == 0 and not self.lifes.animating:
    for firefly in self.fireflies_group:
        firefly.stop()
    self.fades['fade_out_loose'].start()
```

- Las luciérnagas se generan con posiciones y trayectorias definidas manualmente:

```
fireflies = {
    "1": SimpleNamespace(x=100, y=100),
    ...
}
firefly = Firefly(firefly.x, firefly.y, None, "wave")
firefly_extra = Firefly(500, 150, None, "random")
```

Escena 8: TreePhase

Descripción `TreePhase` es una fase vertical y compleja que introduce una variedad de enemigos, plataformas especiales, colisiones avanzadas con luces y mecánicas de planeo. El jugador debe escalar un entorno con múltiples obstáculos, haciendo uso de nuevos movimientos y reaccionando a patrones de luz y enemigos móviles.

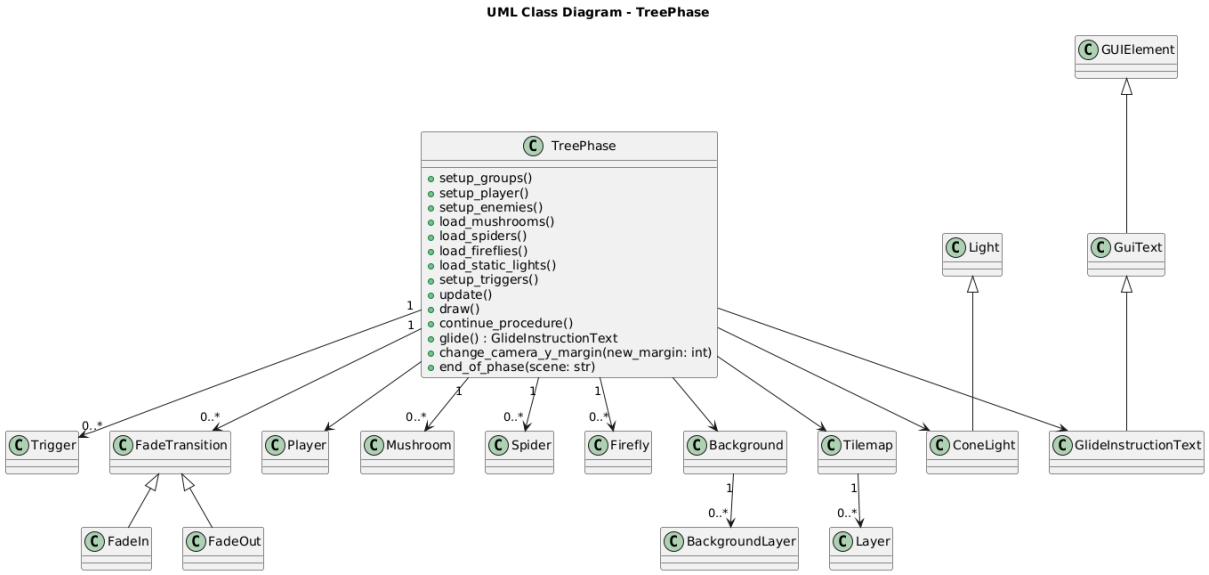


Figura 31: Diagrama de clases que muestra las relaciones de TreePhase.

Modelo

Análisis del diseño

- Hereda de `Phase`, pero extiende su funcionalidad en varios aspectos: movimientos verticales, colisiones más precisas, enemigos con comportamiento variado y múltiples tipos de luces.
- Introduce nuevos grupos de sprites: `mushrooms`, `spiders`, `fireflies`, así como luces estáticas para colisión por máscara mediante `pixel_perfect_lights_group`.
- Los hongos actúan como plataformas reboteadoras, modificando el comportamiento del jugador y añadiendo dinamismo al desplazamiento vertical.
- Usa triggers personalizados para desbloquear el planeo (`glide`), ajustar el margen vertical de la cámara y finalizar la fase.
- La colisión letal está implementada tanto con máscaras como por bounding box, según el tipo de luz involucrada.

Detalles de implementación

- Se asigna a los hongos una propiedad reboteadora que se activa al detectar colisión con el jugador:

```
if self.player.rect.colliderect(mushroom.platform_rect):
    mushroom.glow = True
    mushroom.bounce = True
```

- Se configura colisión pixel-perfect para luces estáticas, verificando que haya contacto real con la forma proyectada:

```
if any(self.player.check_pixel_perfect_collision(light) for
   light in self.pixel_perfect_lights_group):
    self.player.dying()
    self.fades['death_fade_out'].start()
```

- También se comprueba colisión con luces de tipo rectángulo (bounding box):

```
if pygame.sprite.spritecollideany(self.player, self.lights_group
   ):
    self.player.dying()
    self.fades['death_fade_out'].start()
```

- El trigger `glide_trigger` habilita una nueva mecánica de planeo en el jugador, y muestra un texto contextual:

```
def glide(self):
    text = GlideInstructionText(self.screen, (100, 100))
    self.player.can_glide = True
    self.hide_lights = True
    return text
```

- Se modifica el margen vertical de la cámara para ajustar la visibilidad según el avance del jugador:

```
self.init_trigger("camera_y_margin_trigger",
    lambda: self.change_camera_y_margin(self.camera.
       screen_height // 2.2))
```

- El final de la fase reproduce un sonido específico y transiciona a la siguiente:

```
self.sound_manager.play_sound("water-splash.ogg", ...)
self.director.scene_manager.change_scene(scene)
```

Escena 9: LakePhase

Descripción `LakePhase` es la última fase jugable del juego. Ambientada en un entorno submarino, introduce nuevas mecánicas como la natación y la cámara centrada inicialmente en un pez acompañante. El jugador debe navegar por un escenario hostil, evitando criaturas luminosas y reaccionando a eventos activados por triggers, hasta alcanzar el final de la historia.

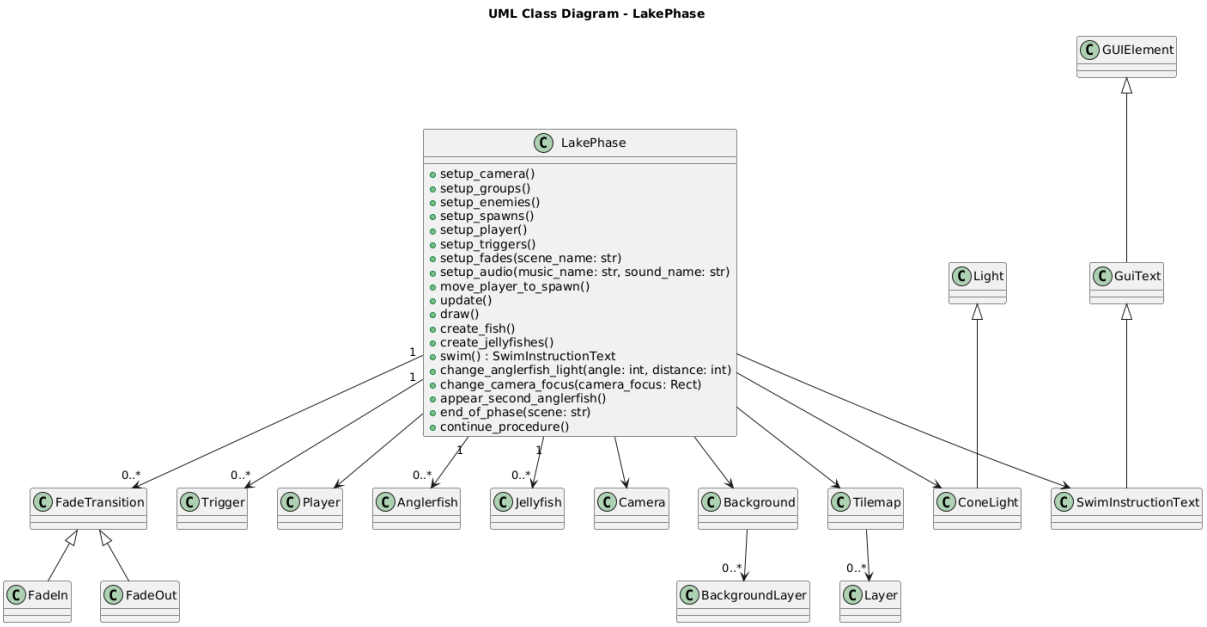


Figura 32: Diagrama de clases que muestra las relaciones de LakePhase.

Modelo

Análisis del diseño

- Hereda de `Phase` y sobrescribe muchos de sus comportamientos estándar: el foco inicial de la cámara no está en el jugador, sino en un `Anglerfish`, y la lógica de reaparición afecta también a dicho pez.

- Se implementan múltiples tipos de enemigos: `Anglerfish` con linternas y movimiento dinámico, y `Jellyfish` que se mueven en varios ejes.
- Se utilizan triggers para alterar el comportamiento de luces, cambiar el foco de la cámara o introducir enemigos adicionales.
- Se permite al jugador nadar desde el principio, y se usan colisiones luminosas (máscaras y bounding boxes) como elementos letales.

Detalles de implementación

- Se configura la cámara para seguir al pez en lugar del jugador:

```
self.camera_focus = self.anglerfish.rect
self.camera.update(self.camera_focus)
```

- El método `move_player_to_spawn` también reposiciona al pez y restablece su estado:

```
self.anglerfish.rect.center = (self.current_spawn[0] - 1000,
    self.current_spawn[1])
```

- El trigger `swim` habilita la mecánica de nado y muestra instrucciones:

```
text = SwimInstructionText(self.screen, (WIDTH // 4, WIDTH //
    1.4))
self.player.is_swimming = True
```

- La luz del pez puede alterarse dinámicamente mediante triggers, cambiando su ángulo y distancia:

```
self.anglerfish.light = ConeLight(..., angle=angle, distance=
    distance)
```

- Se añade un segundo pez mediante el trigger `appear_second_anglerfish`:

```
self.anglerfish_2 = Anglerfish(...)
self.anglerfishes_group.add(self.anglerfish_2)
```

- El final de la fase lanza una transición directa a `EndMenu`:

```
self.director.scene_manager.change_scene(scene)
```

2.1.3. Patrones de Diseño

En el desarrollo del juego en `pygame`, se han identificado varios patrones de diseño que estructuran la lógica del motor de juego y los comportamientos de entidades. A continuación, se describen tres de ellos con ejemplos extraídos del código.

Director: Coordinación de Escenas y Estados La clase `Director` actúa como una implementación del patrón **Director** común en el desarrollo de videojuegos. Su función es coordinar y orquestar las escenas del juego, gestionando el flujo entre menús, niveles y estados como pausa o fin de partida.

```
class Director:
    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls.scenes_stack: list[Scene] = []
```

El `Director` contiene una pila de escenas activas y permite hacer transiciones entre ellas, delegando la lógica de cada escena a instancias específicas como `CemeteryPhase` o `StartMenu`. Este patrón facilita una separación clara entre el flujo del juego y la lógica particular de cada escena, además de permitir un control centralizado del ciclo de vida de cada parte del juego.

Singleton: Gestores de Recursos El patrón **Singleton** se utiliza para asegurar que ciertas clases tengan una única instancia global, accesible desde cualquier parte del código. En este proyecto, el patrón se aplica claramente a clases como `ResourceManager`, que centraliza el acceso a recursos como imágenes, sonidos y fuentes.

```
class ResourceManager:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls.images = {}
        return cls._instance
```

Factory: Creación Centralizada de Escenas en `SceneManager` El patrón **Factory** proporciona una interfaz para crear objetos en una superclase, pero permite que las subclases alteren el tipo de objetos que se crearán. En este proyecto, el `SceneManager` actúa como una **fábrica especializada de escenas**.`SceneManager` actúa como un envoltorio (*wrapper*) especializado de las funciones del `Director` relacionadas con la gestión de escenas.

```
class SceneManager:
    def __init__(self):
        self.scene_classes = {
            "intro": IntroMenu,
            "pause": PauseMenu,
            "cemetery": CemeteryPhase,
            "lake": LakePhase,
        }

    def create_scene(self, scene_name: str, *args, **kwargs) -> Scene
        :
        scene_class = self.scene_classes.get(scene_name)
```

```
        if scene_class:
            return scene_class(*args, **kwargs)
        raise ValueError(f"Scene '{scene_name}' not found")
```

Esta estructura permite al `Director` solicitar una escena por su nombre simbólico, sin acoplarse a las clases concretas. La creación de instancias queda encapsulada, lo que facilita la escalabilidad (añadir nuevas escenas) y el mantenimiento del código. Además, refuerza el principio de responsabilidad única, manteniendo la lógica de creación separada de la lógica de control del flujo del juego.

Template Method: Entidades flotantes El patrón **Template Method** define el esqueleto de un algoritmo en una clase base, permitiendo que las subclases redefinan pasos específicos sin alterar la estructura general. En este proyecto, la clase `FloatingEntity` actúa como plantilla para NPCs que flotan en el mapa (como arañas o medusas).

```
class FloatingEntity(pygame.sprite.Sprite, ABC):
    def __init__(self, x, y, sprite_sheet_path, ...):
        # Lógica común: cargar sprites, animación, luz, movimiento
        self.light = CircularLight(x, y, light_radius) if draw_light
           else None
        ...

    def update(self):
        self.animate()
        self.move()
```

Las subclases como `Spider` o `Jellyfish` extienden esta lógica común agregando su comportamiento único. Esto reduce la duplicación de código y permite mantener coherencia entre entidades similares. Un caso que es necesario comentar es que la clase Firefly, aunque en un principio pueda parecer implementable mediante esta clase, esto no es posible, pues, al aparecer en muchas fases distintas su comportamiento es errático y de difícil extracción.

Component: Sistema de Luz Aunque no se implementa como un patrón de componentes formal completo, se observa una aproximación al **Component Pattern** en el uso de `CircularLight`. Esta clase encapsula la lógica de iluminación, y puede ser agregada opcionalmente a entidades como una sub-función desacoplada.

```
if draw_light:
    self.light = CircularLight(x, y, light_radius)
```

Este patrón permite extender las funcionalidades de las entidades (en este caso, dotarlas de iluminación) sin tener que modificar la clase base o crear múltiples jerarquías heredadas. Las luces se comportan como un componente extraíble, facilitando una arquitectura más flexible.

2.1.4. Aspectos destacables

- **Sistema de iluminación con trazado de rayos**

Una de las características técnicas que aporta un valor diferencial al videojuego es el sistema de iluminación dinámica implementado mediante trazado de rayos.

Técnica general implementada El sistema se basa en lanzar múltiples rayos desde una fuente de luz en distintas direcciones. Estos rayos detectan colisiones con obstáculos del entorno, lo que permite delimitar con precisión la zona iluminada. Para ello, cada fuente de luz genera una *máscara* que representa gráficamente el área afectada por la luz. Esta máscara se renderiza sobre la pantalla con transparencia, simulando el efecto de iluminación.

Listing 1: Lanzamiento de rayos en CircularLight

```
for i in range(self.segments):
    angle = math.radians(i * (360 / self.segments))
    direction = pygame.Vector2(math.cos(angle), math.sin(angle))
    end_point = self._cast_ray(self.position, direction,
       nearby_obstacles)
    relative_point = (end_point - self.position) + center
    points.append(relative_point)
```

Optimización del rendimiento Debido al coste computacional que implica este sistema, se han introducido dos mejoras clave para optimizar su rendimiento:

- **Creación de la máscara sólo una vez:** para cada fuente de luz, la máscara se genera una única vez, evitando su regeneración constante a menos que sea estrictamente necesario.
- **Regeneración condicional:** la máscara sólo se vuelve a generar si la fuente de luz es visible en pantalla. Para ello se utiliza una verificación mediante el rectángulo de cámara.

Listing 2: Generación condicional de la máscara

```
if self.mask is None:
    self._generate_mask(obstacles or [])

if not camera_rect or light_area.colliderect(camera_rect):
    self._generate_mask(obstacles or [])
```

Uso de luces en los distintos niveles Cada fase del juego plantea desafíos distintos que influyen en cómo se usan las luces:

- **Tutorial: CemeteryPhase**
  En esta fase, las lámparas utilizan `ConeLight` con `use_obstacles = True` para conseguir una colisión pixel-perfect con el jugador. En cambio, las luciérnagas usan `CircularLight` sin detectar obstáculos, ya que su detección se basa en colisiones rectangulares.

```
lampara = ConeLight(pos, direction, angle, distance,
   use_obstacles=True)
luciernaga = CircularLight(pos, radius, use_obstacles=False)
```

- **CemeteryBossPhase**
  En esta fase, la lámpara tiene un radio tan grande que el sistema tradicional de máscaras y rayos resultaba inviable. Para solucionarlo, se implementó una clase alternativa basada en sprites animados, que simula la luz mediante efectos visuales sin máscaras ni trazado de rayos.

```
class Lantern(pygame.sprite.Sprite):
    # Simula la luz mediante una imagen estática con canal
       alpha
    def __init__(self,
                 position: tuple[int, int],
                 path: list[tuple[int, int]],
                 scale: int = 4,
                 speed: float = 2):
        ...

    distance_to_player: float = self.pos.distance_to((player
       .rect.x, player.rect.y))
        if distance_to_player <= self.radius and not player.
           is_dying and not player.dead:
            safe_tiles = tilemap.get_safe_rects()
            player_in_safe_zone = any(
                player.rect.colliderect(safe_tile) for
                   safe_tile in safe_tiles
            )
            if not player_in_safe_zone:
                player.dying()
```

- **TreePhase**
  Esta fase fue clave para introducir la optimización que impide renderizar luces fuera del campo visual. Dado que hay arañas que se interponen entre la fuente de luz y el jugador, era imprescindible mantener colisiones perfectas en cada frame. La máscara se evita regenerar si la luz no es visible.

- **LakePhase**
  Contiene una única luz cónica, de tamaño razonable, que emite desde un pez, además de las simples luces circulares que no usan obstáculos. Aquí no hubo necesidad de optimizaciones adicionales.

```
pez_luz = ConeLight(pos, direction, angle, 80, use_obstacles=
   True)
```

Parámetros de optimización: `segments` y `ray_step` Los parámetros `segments` y `ray_step` permiten ajustar la precisión y coste computacional del sistema de raycasting:

- `segments`: determina el número de rayos lanzados desde la fuente de luz. Un valor más alto mejora la calidad visual del área iluminada, pero incrementa el coste.
- `ray_step`: define la precisión del avance de cada rayo. A menor valor, más precisa es la detección de colisiones con obstáculos, pero también más costosa.

Ajustando estos dos valores en función del contexto del nivel se logra un equilibrio entre calidad visual y rendimiento.

- **Sistema dinámico de control del jugador**

Otro de los aspectos clave que aporta valor y complejidad técnica al proyecto es la implementación de un sistema de control del jugador que varía en función del entorno. Este diseño permite al personaje adaptarse de forma dinámica a diferentes escenarios del juego, modificando tanto su comportamiento como sus animaciones en tiempo real.

Entorno terrestre En la mayoría de niveles, el jugador se comporta como un personaje de plataformas tradicional. Puede caminar, correr, saltar e incluso rebotar contra paredes (wall-jump). Se ha incorporado la mecánica de **salto coyote**, que ofrece un breve margen de tiempo para saltar tras abandonar el suelo, mejorando la jugabilidad.

Listing 3: Salto coyote y salto desde pared

```
if self._coyote_timer > 0 and not self.jumped:
    self.velocity_y = self.jump_power_coyote
    ...
elif self.on_wall_left:
    self.velocity_x = self.lateral_jump_power
    self.velocity_y = self.jump_power * 0.75
```

Además, la fuerza de salto puede ajustarse en función del nivel, permitiendo controlar la dificultad sin necesidad de rehacer mecánicas.

Entorno aéreo: TreePhase Durante la fase del árbol (*TreePhase*), el jugador puede **planear** si se mantiene en el aire pulsando el salto. Esto reduce la velocidad de caída y permite esquivar los obstáculos con mayor precisión.

Listing 4: Habilidad de planear

```
if self.can_glide and not self.on_ground and self.velocity_y >
   0:
    self.velocity_y = MAX_FALL_SPEED // 2
    self.is_gliding = True
    self.set_animation("glide")
```

Entorno acuático: LakePhase En los niveles acuáticos, como *LakePhase*, el control cambia completamente. El jugador pierde la capacidad de correr o saltar, y en su lugar puede nadar en todas direcciones. La física se ajusta para simular gravedad reducida, y las animaciones también se adaptan automáticamente.

Listing 5: Controles y animación al nadar

```
if self.is_swimming:
    self.set_animation("swim")
    if keys[pygame.K_UP]:
        self.velocity_y = self.swim_ascend_speed
    ...
```

Cambio dinámico de entorno El entorno actual del jugador se ajusta automáticamente mediante triggers colocados en el mapa. Estos triggers determinan si el jugador puede nadar, planear, rebotar o simplemente caminar. Al activarse, modifican tanto la física como las animaciones en tiempo real, sin necesidad de recargar el personaje.

Este sistema flexible permite que el mismo personaje se comporte de forma completamente distinta en cada fase del juego, enriqueciendo la experiencia del jugador.

- **Creación dinámica de mapas usando Tiled**

Otro aspecto a destacar en cuanto a la parte técnica es el uso de `Tiled` (figura 33), un editor de niveles 2D como herramienta para crear los mapas del videojuego. A partir de una cuadrícula de tiles y un tileset, se dibujaron los diferentes elementos de los niveles y se dividieron en capas tanto de Tiles como de objetos. Gracias a eso, se pudieron definir:

- Las posiciones de los checkpoints, los finales de fase y aparición de enemigos.
- Las posiciones de triggers, como los de aparición de textos en pantalla.
- El terreno con el que se colisiona y elementos de decoración.
- Propiedades personalizadas para cada tile u objeto.
- Una organización más dinámica y flexible del entorno.

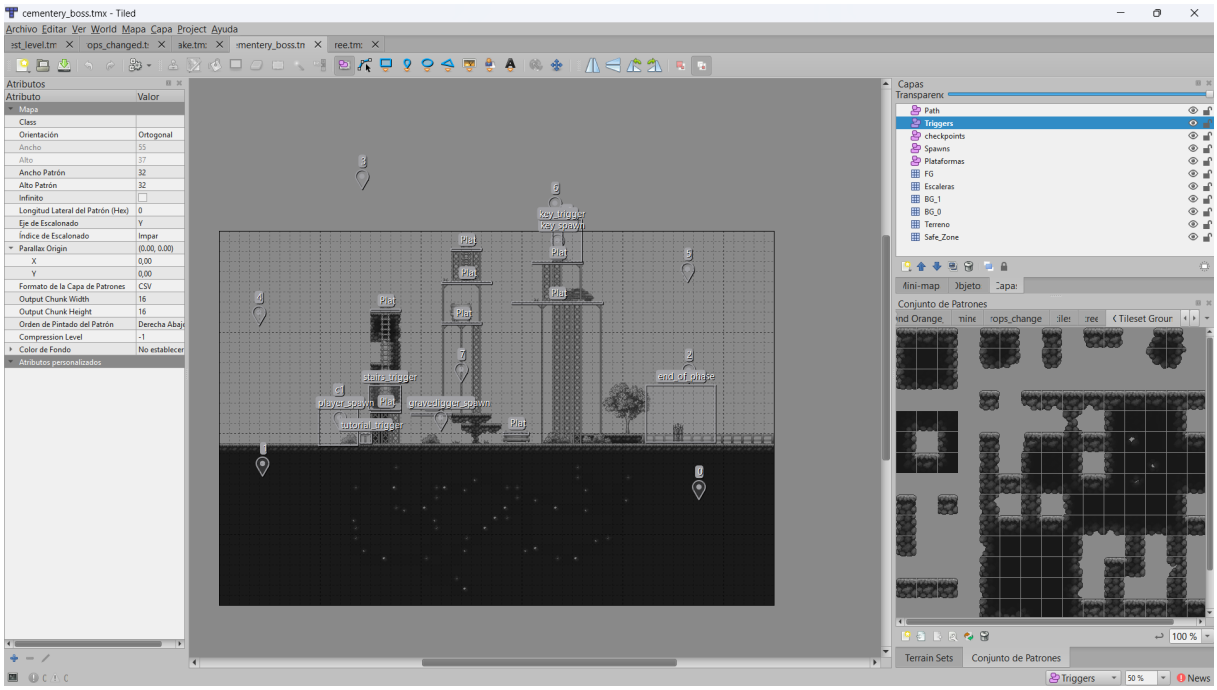


Figura 33: Tiled para el boss de CemeteryPhase.

- **Originalidad en el diseño visual y artístico**

  Otro de los elementos que diferencian claramente este videojuego respecto a otras producciones similares es el enfoque original en el diseño visual y artístico. A diferencia de muchos proyectos que recurren a recursos gráficos predefinidos o bancos de sprites, gran parte de los elementos visuales de este juego fueron **dibujados, coloreados y animados a mano**, contribuyendo a una identidad estética única y cohesionada.

  Sprites originales Se han diseñado personajes y elementos desde cero, creando cada uno de sus cuadros de animación manualmente. Algunos ejemplos destacados incluyen las clases Jellyfish, Mushroom o Gravediger.

  Ampliación de sprites existentes En lugar de conformarse con sprites estáticos o limitados, se ha trabajado sobre bases ya existentes para adaptarlas al estilo del juego:

  - Se añadieron nuevas animaciones al sprite del **jugador**, como la animación de *planeo*, ausente en su versión original.
  - Personajes como el **pez**, el propio **jugador** y las **arañas** fueron **recoloreados cuidadosamente** para mantener una paleta coherente y estilizada que refuerce la atmósfera del juego.

Ensamblado de sprites complejos Existen elementos visuales que fueron compuestos ensamblando manualmente varias capas gráficas para conseguir efectos únicos, como la **linterna del nivel del jefe final**.

2.1.5. Manual de Usuario

- **Requisitos del sistema**:
  - Sistema Operativo: Windows
  - Python: Versión 3.8 o superior.
    - pygame (versión 2.0 o superior)

      ```
      pip install pygame
      ```

    - pytmx (para manejar mapas en formato TMX)

      ```
      pip install pytmx
      ```

  - Librerías estándar de python (os, math, random, etc.)
- **Instrucciones para arrancar el juego**:
  1. Abrir una terminal o símbolo del sistema y navegar al directorio src del proyecto:

     ```
     cd C:\...\Vestigium\src
     ```

  2. Ejecutar el archivo principal del juego:

     ```
     python3 main.py
     ```

  3. El juego se abrirá en una ventana de Pygame.
- **Controles del juego**: Al iniciar el juego se verá el menú principal. Bajo el nombre del videojuego se encuentra la opción `New Game` (Figura 34). Para empezar a jugar desde el primer nivel se debe hacer click sobre esta opción. Sin embargo, si lo que se quiere es cambiar el volumen de la música, los efectos de sonido o salir, están las opciones `Options` y `Exit` (Figura 35). También, durante la partida tendremos un menú de pausa como el de la Figura 36.

Figura 34: Pantalla de inicio

Figura 35: Opciones del menú

Figura 36: Pantalla de pausa

2.1.6. Reporte de Bugs

A continuación se detalla una lista de errores conocidos en el videojuego, junto con sus posibles soluciones o recomendaciones para mitigarlos.

Respawn inesperado con salto En la segunda fase, si el jugador reaparece después de morir, durante unos milisegundos se mantiene la animación o el estado del último movimiento antes de morir. Esto puede provocar que el personaje reaparezca en el aire realizando un salto, colisionando inmediatamente con una fuente de luz cercana y muriendo de nuevo. **Solución:** Reiniciar completamente el estado del jugador al reaparecer, asegurando que esté en tierra y sin velocidad acumulada.

Inconsistencia en el cambio de dirección Si el jugador mantiene pulsado `DERECHA` y luego pulsa `IZQUIERDA`, el personaje cambia correctamente de dirección. Sin embargo, si se hace al revés (primero `IZQUIERDA` y luego `DERECHA`), el jugador permanece en dirección izquierda. **Solución:** Ajustar la prioridad de teclas en el sistema de input para que siempre se tenga en cuenta la última dirección pulsada.

Pausa interrumpe el salto Si el juego se pausa mientras el jugador está saltando, al reanudar se reinicia el salto, generando inconsistencias en la trayectoria. **Solución:**

Guardar el estado físico del jugador (velocidad, animación y acción actual) al pausar, y restaurarlo tal cual al continuar.

Dirección del jugador no restaurada tras morir Tras morir y reaparecer (excepto en la segunda fase), se restauran correctamente la posición y el movimiento, pero no la dirección en la que el personaje estaba mirando. **Solución:** Guardar el estado del jugador antes de morir, incluyendo también la orientación.

Colisiones de luz inexactas En algunos casos, las colisiones entre el jugador y la luz no son completamente precisas. Por ejemplo, en el tutorial, si el jugador se coloca justo en la esquina de una farola, no muere. En otros casos, colisiona prematuramente con el Rect de la luz de una luciérnaga. **Solución:** Ajustar las máscaras de colisión y el tamaño de los Rects de las luces para mejorar la precisión del sistema.

Frame sin efectos en transición de fases Al realizar un cambio de fase con los efectos de *FadeIn* y *FadeOut*, aparece un frame intermedio donde la pantalla se muestra completamente nítida, rompiendo la fluidez del efecto. **Solución:** Revisar el sistema de transición para evitar que se renderice el juego sin efectos entre ambos procesos. Posiblemente incorporar un buffer de frames o extender el efecto hasta que la nueva fase esté completamente cargada.

2.2. Videojuego en 3D

2.2.1. Descripción

El objetivo de este apartado es ofrecer una visión general de la estructura y el funcionamiento del videojuego *Vestigium 3D*, continuación directa del título original. Esta versión traslada la experiencia a un entorno tridimensional, conservando la esencia del juego previo pero adaptando sus mecánicas, narrativa y ambientación al nuevo formato espacial. Se describen los niveles, controles, dinámicas de juego y normas generales que configuran esta nueva entrega, sin profundizar aún en los aspectos técnicos de implementación, que se abordarán en secciones posteriores.

- **Tipo de juego y mecánica general:** La segunda entrega de *Vestigium*, titulada *Vestigium 3D*, es un videojuego de plataformas y puzles en tres dimensiones, desarrollado en Unity. Esta nueva versión conserva la premisa principal del título original: la luz actúa como una amenaza letal, de modo que cualquier contacto del personaje con una fuente luminosa provoca su muerte inmediata.

  El jugador controla una sombra humanoide de aproximadamente treinta centímetros de altura, capaz de desplazarse libremente en los tres ejes del espacio, así como de correr, saltar, trepar, empujar objetos y realizar acciones de interacción básica con el entorno.

- **Controles:**

  - **W, A, S, D**: Movimiento sobre los ejes X y Z.
  - **Espacio**: Saltar desde el suelo.
  - **Shift (izquierdo)**: Correr.

- **Botón derecho del ratón**: Interactuar con objetos (ej. apagar luces).
- **Botón derecho del ratón (mantenido)**: Transportar objetos en la espalda o empujarlos y tirar de ellos.
- **ESC**: Pausar el juego y acceder al menú de opciones.

- **Jugabilidad y estructura por niveles:** El juego está diseñado para ser completado en una única sesión, con una duración estimada de unos 15 minutos, dependiendo de la habilidad del jugador. No dispone de sistema de selección de dificultad ni opciones de configuración avanzada de controles.

  La estructura del juego se organiza en tres niveles, cada uno correspondiente a una estancia distinta de la casa:

  - **Nivel 1: Entrada de la casa** — El nivel comienza en un bosque tenebroso, justo en el borde del lago que marca el final de la parte anterior. Cerca de este se puede ver una antigua casa de madera. Sin embargo, para llegar allí, el jugador debe tener cuidado con las farolas que iluminan el jardín, ya que cualquier contacto con ellas resultará en la muerte del personaje. Para acceder a la casa, primero debe apagar la luz que ilumina la entrada, resolviendo un pequeño rompecabezas de interruptores. Después, deberá encontrar la llave escondida en el tejado de la casa para abrir la puerta. Todo esto debe hacerse con astucia, esquivando las luces y apartando los obstáculos que se encuentren en el camino a través de la densa niebla.
  - **Nivel 2: Cocina** — El jugador debe recorrer la cocina, un espacio sombrío y aparentemente abandonado, hasta alcanzar un conducto de ventilación situado en la parte superior de las encimeras. Al localizar la salida, descubre que un gato negro bloquea el acceso. Para desplazarlo, deberá colocar comida en su comedero, inicialmente inaccesible debido a una fuente de luz generada por una nevera antigua con la puerta abierta. Una vez logre cerrar la nevera y apagar la luz proveniente de la campana de la cocina, podrá depositar el pescado en el recipiente y así liberar el paso hacia el siguiente nivel.
  - **Nivel 3: Salón** — Tras atravesar los conductos de ventilación, la sombra llega a un amplio salón cubierto de moqueta y lleno de cajas y objetos desordenados. En este espacio, el único peligro lo representa la luz de un televisor antiguo que proyecta destellos aleatorios debido a su mal funcionamiento. El jugador deberá desplazarse con precaución entre las sombras proyectadas por cajas de pizza, electrodomésticos y otros objetos hasta alcanzar la puerta abierta al otro extremo de la sala.

    Al completar este nivel, se mostrará una secuencia narrativa de texto progresivo que transmite el cierre argumental del juego, una explicación simbólica del propósito final de la sombra.

    Para incrementar la dificultad con respecto a la versión 2D, en esta entrega no se utilizan puntos de control intermedios. En caso de que el jugador muera, reaparecerá al inicio del nivel actual, pero el estado del escenario (objetos movidos, luces apagadas, etc.) se mantendrá tal como estaba justo antes de la muerte.

Las transiciones entre niveles, así como las muertes del personaje, están acompañadas de efectos visuales de *fade in* y *fade out*. En cualquier momento se puede pausar la partida (tecla ESC) para acceder a un menú con las opciones de continuar, reiniciar el nivel, volver al menú principal o ajustar el volumen de la música y los efectos de sonido.

- **Audio:** Cada nivel cuenta con una pista musical ambiental propia, diseñada para reforzar la atmósfera de cada estancia. Además, se incluyen efectos de sonido vinculados a acciones del jugador y del entorno, como saltos, desplazamientos, aperturas de objetos o cambios en las fuentes de luz. REVISAR

2.1.1.1 Personajes El personaje principal sigue siendo la sombra, que mantiene su naturaleza pacífica y no ofensiva. Su comportamiento se basa en la evasión, el desplazamiento sigiloso y la interacción con objetos del entorno, lo cual define su estilo de juego.

En esta nueva entrega, los personajes secundarios adoptan un carácter principalmente ambiental: su función no es activa ni narrativa, sino que contribuyen a reforzar la atmósfera general y el contexto simbólico del mundo del juego. Sin embargo, algunos de ellos sí representan un obstáculo o amenaza indirecta para el jugador.

- **Gato negro:** Animal pasivo que bloquea la salida de la cocina. Aunque no supone un peligro directo, obliga al jugador a resolver un puzle para lograr que se aparte, incrementando así el desafío del nivel.
- **Televisor averiado:** Aparato antiguo situado en el centro del salón. Aunque se trata de un objeto inanimado, su emisión de luz intermitente y errática lo convierte en una amenaza activa para la sombra. Su presencia constante, impredecible y luminosa le otorga una cualidad casi antropomórfica y perturbadora.

2.1.1.2 Enemigos Al igual que en el *Vestigium* original, el juego no presenta enemigos tradicionales. En su lugar, cualquier fuente de luz funciona como amenaza directa para el jugador. De este modo, se redefine el concepto de enemigo, otorgando esa condición a elementos del entorno que emiten luz y representan un riesgo mortal.

Ejemplos de ello en esta entrega son la luz proveniente de objetos como las **farolas** del jardín de la casa, la **nevera abierta** en la cocina, la iluminación de la **campana extractora** y los destellos impredecibles del **televisor** en el salón.

2.1.1.3 Objetos

- **Llave (obtenible):** Elemento esencial para avanzar en el primer nivel (la entrada de la casa). Permite al jugador acceder a las salas interiores, funcionando como puerta de entrada a la experiencia principal del juego.
- **Troncos (arrastrable):** Ubicados en el área de entrada. Estos troncos huecos pueden ser desplazados por el jugador para utilizarlos como refugio temporal frente a las lámparas que iluminan el camino. Su colocación estratégica es fundamental para avanzar sin exponerse a la luz.

- **Piedra (obtenible):** Complemento al desafío anterior. El jugador debe cargar con una piedra y colocarla dentro de uno de los troncos huecos, lo cual permite fijarlo en su posición para usarlo como cobertura estable.
- **Pez (obtenible):** Se encuentra sobre una tabla de cortar en la cocina. Es necesario transportarlo hasta el cuenco del gato para distraerlo y liberar el acceso al conducto de ventilación que conduce al siguiente nivel.
- **Interruptores (interactuables):** En la escena de la cocina se presenta un interruptor que se puede accionar para apagar la luz de la campana extractora y permitir el paso de la sombra.

2.1.1.4 Diseño: guion, reglas, mecánica El juego carece de narrativa explícita o diálogos; su historia se transmite de forma ambiental, a través del recorrido del jugador y los cambios de atmósfera. La mecánica principal se basa en evitar la luz, que actúa como amenaza letal constante.

Reglas principales:

- **Contacto con la luz:** produce la muerte inmediata del personaje.
- **Puntos de control:** ubicados al inicio de cada nivel (sin checkpoints intermedios).
- **Sistema de vidas:** vidas ilimitadas con reaparición automática tras la muerte.
- **Dificultad:** no existe opción para modificar el nivel de dificultad.
- **Controles:** configurados de forma fija mediante teclado y ratón.
- **Duración estimada:** aproximadamente 15 minutos por partida, dependiendo de la habilidad del jugador.

2.2.2. Escenas

2.1.2.1 Metodología de desarrollo y reparto de trabajo El desarrollo de *Vestigium 3D* se llevó a cabo utilizando el motor Unity, lo que supuso un cambio significativo con respecto a la primera entrega. Desde el inicio, se mantuvieron reuniones conjuntas para definir las escenas que compondrían el juego, así como los elementos artísticos necesarios. Se seleccionaron y adaptaron los *assets* buscando mantener una coherencia visual y estilística entre todos los niveles.

Dada la dificultad de gestionar eficazmente el control de versiones de proyectos Unity a través de GitHub, se decidió que cada integrante trabajase de forma aislada en una escena independiente del juego. Esta decisión evitó conflictos entre ramas y facilitó un flujo de trabajo más ordenado y sin solapamientos.

Antes de iniciar la implementación individual de escenas, se definieron en común los componentes clave compartidos entre todas ellas, en particular:

- **SceneManager:** encargado de gestionar las transiciones entre escenas.
- **AudioManager:** responsable de controlar el sistema de sonido global y la música ambiental en todas las fases del juego.

Con estos componentes definidos, cada miembro del equipo pudo integrarlos en su escena siguiendo una lógica común.

La base de la jugabilidad del personaje se estableció en primer lugar, siendo desarrollada por Fernando, quien contaba con experiencia previa en motores como Unity. Se encargó de implementar el sistema de movimiento (caminar, correr, saltar) y el mecanismo de agarre. Esto derivó en la creación de una clase abstracta de objetos interactuables, diseñada para ser modular y extensible, permitiendo añadir diferentes tipos de interacción entre el personaje y el entorno. También se desarrolló una primera versión del sistema de detección de muerte por colisión entre el personaje y la luz.

Posteriormente, se amplió esta clase para incluir nuevos tipos de objetos:

- **Pickable:** objetos que el jugador puede recoger.
- **Switchable:** elementos que el jugador puede activar o desactivar, como interruptores de luz.
- **Closable:** elementos que el jugador puede cerrar, como la puerta de la nevera.

Posteriormente, Xian tuvo que refinar esta implementación en el nivel del salón, ya que se requería una mayor precisión. Para ello, se optó por utilizar un sistema basado en raycasting.

En cuanto al reparto de trabajo:

- **Fernando:** desarrollo del sistema de movimiento del personaje, creación de la clase de objetos interactuables y diseño de la primera escena (entrada de la casa).
- **Iria:** implementación de la escena de la cocina, con lógica de interacción y resolución de puzles.
- **Xian:** desarrollo de la escena del salón, incluyendo el comportamiento aleatorio de la fuente de luz.
- **Ana:** diseño e implementación de los menús (inicio, pausa, créditos), así como los mensajes tutoriales sobre interacción y evasión de luz.

Una vez finalizadas las escenas individuales, se procedió a la integración de los gestores comunes (`SceneManager` y `AudioManager`) en cada una de ellas, asegurando una experiencia de usuario unificada y coherente.

Gestión del flujo de trabajo: metodología Kanban y coordinación

Se utilizó la metodología *Kanban* para la gestión de tareas, mediante un tablero digital con columnas para tareas **Pendientes**, **En progreso** y **Completadas**. Cada tarea se desglosaba en subtareas y estaba asignada a los miembros responsables. Esta organización permitió una planificación clara, evitando solapamientos y asegurando la trazabilidad del progreso.

Además, el equipo realizó dos reuniones semanales para poner en común los avances, resolver dudas y coordinar las decisiones técnicas necesarias. Estas sesiones fueron fundamentales para garantizar que las nuevas funcionalidades se implementasen de forma coherente en todas las escenas y según los estándares comunes definidos al inicio del desarrollo.

2.1.2.2 Descripción global de la Arquitectura y Escenas

- **Diagrama de Arquitectura General**
- **Diagrama de flujo de datos (DFD)**
- **Diagrama de clases (UML)**
- **Diagrama de estados/escenas**
- **Diagrama de componentes**

2.1.2.3 Detalles de implementación de las clases generales

- Managers (diagrama de clases de los managers) YA HECHO
  - Scene Manager (diagrama de componentes + diagrama de secuencia) YA HECHOS
  - Audio Manager (diagrama de componentes + diagrama de secuencia)
- **SceneManager.** Esta clase se encarga de gestionar el cambio de escenas dentro del juego, centralizando la lógica de transición y combinándola con efectos visuales de entrada y salida. Para ello, hace uso de corrutinas asíncronas y de una clase auxiliar llamada `TransitionManager`, que coordina las animaciones de transición.
  - **Patrón Singleton.** Para asegurar que solo exista una instancia del gestor durante toda la ejecución, se implementa un patrón Singleton sencillo. Esto permite mantener el gestor activo entre escenas mediante `DontDestroyOnLoad`, evitando duplicados y facilitando su acceso desde cualquier parte del proyecto.

    ```
    if (Instance == null) {
        Instance = this;
        DontDestroyOnLoad(gameObject);
    } else {
        Destroy(gameObject);
    }
    ```

  - **Gestión de escenas.** El `SceneManager` implementa tres métodos públicos para gestionar escenas:
    - `ChangeToNextScene()` — Avanza a la siguiente escena en el build index.
    - `ChangeToScene(int)` — Carga una escena concreta según su índice.
    - `ReloadCurrentScene()` — Recarga la escena actual.

    Todos ellos se apoyan en el método base `ChangeToScene()`, que realiza la transición visual mediante una animación y, tras su finalización, ejecuta la carga real de escena:

    ```
    await TransitionManager.Instance.PlayTransition(
       transitionType);
    UnitySceneManager.LoadScene(sceneIndex);
    ```

- **Transiciones visuales.** Antes de realizar cualquier cambio de escena, se lanza una animación de fundido (por ejemplo, `FadeOut`) gestionada por el `TransitionManager`. Del mismo modo, tras cargarse la nueva escena, se lanza una animación de entrada (`FadeIn`) que suaviza la aparición del nuevo contenido en pantalla.
- **Flujo completo del cambio de escena.** El diagrama de la figura 37 muestra con detalle el flujo de ejecución del sistema de cambio de escena. Se distinguen tres partes:
  1. **Activación por trigger:** un evento de colisión llama a `ChangeToNextScene()`, se obtiene la escena actual, se calcula la siguiente, y se inicia una transición con animación de salida.
  2. **Activación por selección de menú:** un evento UI llama a `ChangeToScene(sceneInde` que inicia directamente la transición y luego carga la escena deseada.
  3. **Inicio de la nueva escena:** Unity invoca automáticamente el método `Start()` tras la carga, donde se lanza la animación de entrada para cerrar visualmente la transición.

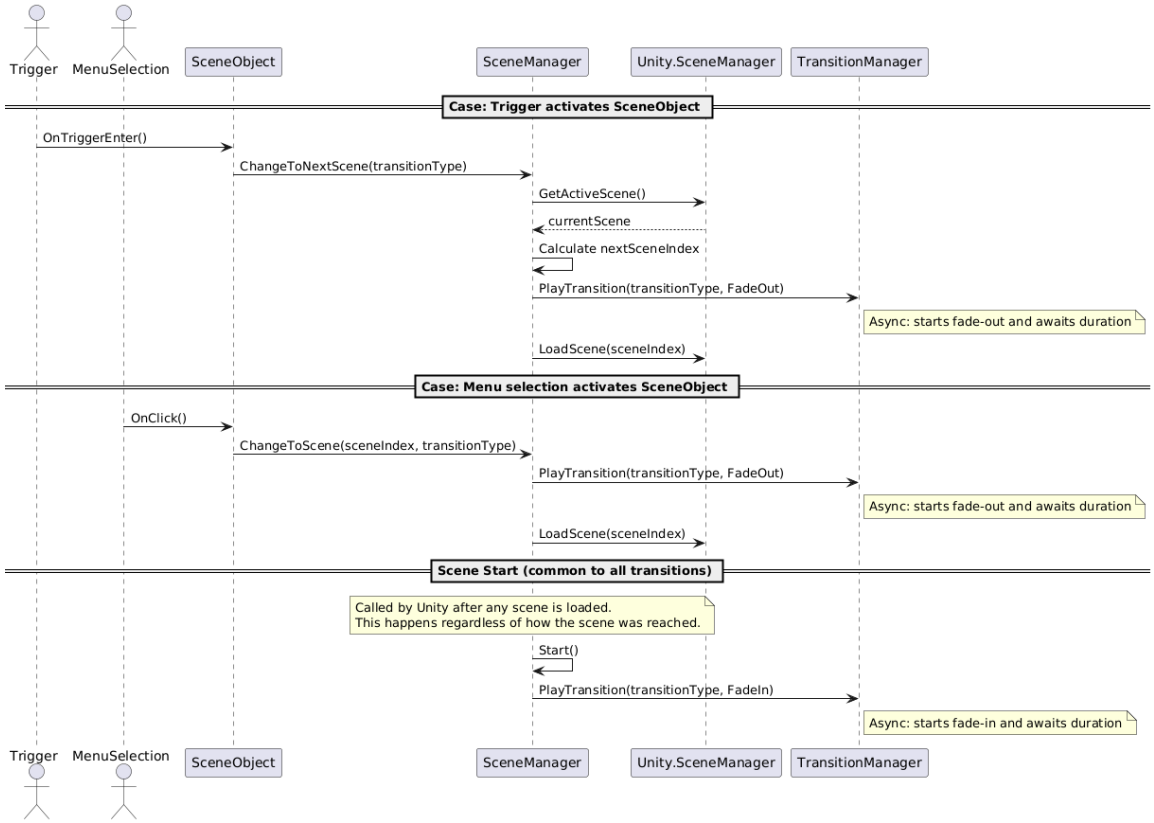


Figura 37: Diagrama de secuencia del flujo de transición entre escenas con animación.

Este sistema modular desacopla completamente el cambio de escena de la lógica de animación, permitiendo una ampliación sencilla de nuevos tipos de transiciones (fundido, desenfoque, barrido, etc.), así como facilitar pruebas unitarias del control de flujo de escenas.

- **AudioManager.** El `AudioManager` actúa como componente centralizado para la reproducción y gestión del audio en el juego. Permite controlar tanto la música como los efectos de sonido, con soporte para audio en 2D y 3D, así como bucles y transiciones. Su diseño facilita tanto la personalización de la experiencia auditiva como la integración con el sistema de opciones del usuario.

  - **Uso de AudioMixer y grupos.** Para mantener una separación clara entre los distintos tipos de audio, se emplea un `AudioMixer` de Unity. Este se divide en dos grupos: `musicGroup` y `sfxGroup`, que agrupan respectivamente la música de fondo y los efectos sonoros. Esto permite aplicar configuraciones y controles de volumen por categoría. La asignación de grupos se realiza al inicializar las fuentes:

    ```
    sfxSource.outputAudioMixerGroup = sfxGroup;
    bgmSource.outputAudioMixerGroup = musicGroup;
    ```

  - **Control de volumen con persistencia.** El sistema incluye dos deslizadores para controlar los niveles de música y efectos. Estos valores se guardan con `PlayerPrefs` para mantenerse entre sesiones. Se aplica una conversión logarítmica al volumen para una percepción más natural:

    Aplicación del volumen al mixer:

    ```
    float volume = Mathf.Max(sfxSlider.value, 0.0001f);
    mixer.SetFloat("SFXVolume", Mathf.Log10(volume) * 20f);
    ```

  - **Sonidos puntuales y en bucle.** El sistema diferencia entre sonidos que se reproducen una única vez (OneShot) y sonidos que deben mantenerse activos de forma continua (Loop):

    - *OneShot*: se reproducen directamente mediante `PlayOneShot()` y no requieren control posterior.

      ```
      sfxSource.PlayOneShot(clip);
      ```

    - *Loop*: se crean dinámicamente y se reproducen en bucle. Estos sonidos se registran en un diccionario con clave única para poder detenerlos cuando sea necesario:

      ```
      looped3DSounds[name] = source;
      ...
      looped3DSounds[name].Stop();
      ```

  - **Audio 2D y 3D.** El sistema también distingue entre sonidos que deben sonar de forma global (2D) y aquellos que deben percibirse desde una posición concreta en el espacio (3D):

    - *Audio 2D*: se reproduce a través de una fuente fija y se percibe igual desde cualquier punto de la escena. Se usa para interfaz, clics, o eventos no espaciales.
    - *Audio 3D*: se reproduce desde un objeto posicionado en la escena, y su volumen depende de la distancia al oyente. Para ello, se crea dinámicamente un objeto con un `AudioSource` configurado con las propiedades espaciales adecuadas:

```
source.spatialBlend = 1f;
source.minDistance = 1f;
source.maxDistance = 15f;
source.rolloffMode = AudioRolloffMode.Logarithmic;
```

Además, cuando el sonido 3D es en bucle (por ejemplo, el motor de una máquina), el objeto desde el que se reproduce se hace hijo del objeto emisor. Esto asegura que el sonido se mueva con el objeto, manteniendo su posicionamiento relativo actualizado dinámicamente:

```
loopGO.transform.parent = transform;
loopGO.transform.localPosition = Vector3.zero;
```

Esta combinación permite tener una experiencia auditiva espacial coherente, con fuentes móviles que afectan al jugador de forma realista según su distancia y posición.

- **TransitionManager.** Esta clase se encarga de gestionar las transiciones visuales entre escenas, centralizando los efectos de entrada y salida mediante un fundido a negro. Su propósito es mejorar la fluidez visual en los cambios de contexto y evitar cortes bruscos entre escenas.

  - **Patrón Singleton.** Para garantizar que solo exista una única instancia del sistema de transiciones en todo el ciclo de vida del juego, se implementa el patrón Singleton. Esto permite que cualquier otro componente (como el `SceneManager`) pueda invocar transiciones sin necesidad de referencias explícitas:

    ```
    Instance = this;
    ```

  - **Transiciones asincrónicas con DoTween.** Las transiciones se implementan utilizando la librería externa `DoTween`, que permite interpolaciones fluidas sobre propiedades visuales. En este caso, se utiliza para modificar la opacidad de una imagen UI que cubre toda la pantalla.
  - **Tipos de transición.** El sistema actualmente admite dos tipos de transición definidos por un enumerado: `FadeIn` y `FadeOut`. Ambos modifican el alfa de una imagen, de forma que se oscurezca o revele progresivamente el contenido de la escena.
  - **Gestión del raycast.** Durante las transiciones, se activa el atributo `raycastTarget` de la imagen para bloquear cualquier interacción con la interfaz o elementos del juego mientras la pantalla está opaca. Este valor se desactiva al finalizar la transición de entrada:

    ```
    fadeImage.raycastTarget = true;
    await fadeImage.DOFade(0, duration)
        .OnComplete(() => fadeImage.raycastTarget = false)
        .AsyncWaitForCompletion();
    ```

  - **Transición de entrada automática.** Al iniciar una nueva escena, el `TransitionManager` reproduce automáticamente una transición de tipo `FadeIn` en su método `Start()`, asegurando así una entrada progresiva en la escena cargada:

```
private async void Start() {
    await PlayTransition(TransitionType.FadeIn);
}
```

- **Control directo de opacidad.** Para garantizar un estado visual coherente al inicio, se fuerza el valor alfa a 1 en el método `Awake()`, de modo que al lanzar el `FadeIn`, el efecto visual parte desde una pantalla completamente negra:

```
var color = fadeImage.color;
color.a = 1f;
fadeImage.color = color;
```

- **Player.** La clase `PlayerController` encapsula el comportamiento completo del jugador, incluyendo movimiento, interacción con el entorno, control mediante input personalizado, animaciones, lógica de muerte y reacciones a eventos externos como la detección por parte de luces. El modelo del jugador ha sido extraído desde *SketchUp* y las animaciones han sido aplicadas mediante la plataforma *Mixamo*, lo cual ha permitido una rápida integración de un esqueleto animado con transiciones coherentes entre estados.

  - **Movimiento y control.** El movimiento se basa en fuerzas físicas aplicadas mediante un componente `Rigidbody`. Se adapta dinámicamente a la entrada del usuario (caminar, correr, empujar, saltar). Se utiliza interpolación esférica para suavizar las rotaciones del jugador en la dirección del movimiento:
    Interpolación de rotación hacia la dirección de movimiento:

```
transform.rotation = Quaternion.Slerp(
    transform.rotation, targetRotation,
    Time.fixedDeltaTime * 10f
);
```

    En combinación con la fuerza aplicada, esto permite que el jugador se desplace de forma fluida en cualquier dirección.

  - **Sistema de agarre, empuje y tracción.** El jugador puede interactuar con objetos físicos en la escena a través del sistema de "Push/Pull". Cuando entra en estado de agarre, el movimiento queda restringido a una dirección concreta, representada por el vector `grabDirection`. Esto limita el desplazamiento a adelante/atrás respecto al objeto:
    Cálculo de proyección del movimiento en la dirección del agarre:

```
float forwardAmount = Vector3.Dot(moveDirection,
   grabDirection.normalized);
Vector3 projectedMove = grabDirection.normalized *
   forwardAmount;
rb.AddForce(projectedMove.normalized * acceleration * vel,
   ForceMode.Acceleration);
```

  - **Sistema de salto.** El salto está condicionado por el estado del jugador (si está en el suelo, agarrando, etc.). Se utiliza una ligera espera asíncrona con `Task.Delay` para coordinar animaciones y física de forma natural.

- **Animaciones reactivas.** Las animaciones del jugador están directamente ligadas al estado interno y a los eventos de input. Por ejemplo, el parámetro `isMoving` se actualiza automáticamente en función del movimiento leído del input.
- **Interacciones con el entorno.** La clase permite gestionar diferentes tipos de interacciones mediante un sistema basado en `InteractionTrigger` y `InteractionType`. Por ejemplo, el método `OnStartInteraction()` permite iniciar comportamientos distintos según el tipo de interacción detectada (empujar, recoger, accionar interruptores...).
- **Sistema de detección por luz.** En cada `Update()`, el jugador consulta el grupo de detectores de luz para comprobar si está siendo iluminado. Si lo está y no ha sido detectado previamente, se ejecuta la lógica de muerte:

  Consulta y activación del estado de muerte:

```
if (!_isDead && LightCollisionDetectorGroup.CollidesWithAny(
    transform)) {
     KillPlayer();
}
```

- **Muerte y reinicio.** Cuando el jugador muere, se reproducen animaciones, se desactiva el control, se reproduce un sonido, y tras un retardo se destruye el objeto. La escena se recarga de forma automática:

  Lógica de muerte del jugador:

```
_isDead = true;
anim.CrossFade("Diying", 0.1f);
rb.isKinematic = true;
_onTransition = true;
inputActions.Disable();
AudioManager.Instance.PlaySFX(deathSound);
Destroy(gameObject, 3f);
await Task.Delay(2000);
SceneManager.Instance.ReloadCurrentScene();
```

- **Sonido contextual.** En cada paso del jugador, se detecta la superficie bajo sus pies y se reproduce un sonido asociado a dicha superficie, utilizando el sistema `SurfaceToSound`. Para sincronizar este sonido con el momento exacto del paso, se ha configurado un evento en la propia animación del personaje, de forma que el método `OnFootstep()` es llamado justo cuando el pie entra en contacto con el suelo.

  La lógica de selección de sonido se basa en la detección del material físico del suelo mediante raycasting hacia abajo desde la posición del jugador:

  Detección de superficie y obtención del sonido:

```
string surface = "Default";
if (Physics.Raycast(transform.position + Vector3.up * 0.1f,
    Vector3.down, out RaycastHit hit, 2f)) {
     if (hit.collider.sharedMaterial != null)
          surface = hit.collider.sharedMaterial.name;
```

```
}
AudioClip sound = SurfaceToSound.Instance?.GetSurfaceSound(
   surface);
AudioManager.Instance.PlaySFX(sound);
```

Este enfoque permite una reproducción sonora dinámica y coherente con el tipo de suelo (por ejemplo, metal, madera, piedra...), sin necesidad de codificar manualmente los sonidos por escena.

- **Sistema de Cámara con Cinemachine**
  Para el sistema de cámara del proyecto se ha utilizado el paquete **Cinemachine**, una herramienta de Unity diseñada para facilitar la creación de cámaras dinámicas, inteligentes y cinematográficas sin necesidad de programar manualmente el comportamiento de seguimiento o rotación. Cinemachine permite definir múltiples cámaras virtuales (*Virtual Cameras*) que controlan la posición y orientación de la *Main Camera* a través del componente `CinemachineBrain`.

  Cada escena del juego incorpora una cámara principal (*Main Camera*) acompañada por una Cinemachine Camera. Aunque las configuraciones básicas son similares en todas las escenas —como el uso del `Binding Mode` en `World Space`, o el uso del `Rotation Composer` para un encuadre suave— se han aplicado variaciones sutiles que responden a las necesidades narrativas y artísticas específicas de cada entorno.

  Los elementos comunes entre las cámaras de todas las escenas incluyen:

  - **Seguimiento al jugador** mediante la propiedad `Follow`, vinculada al transform del personaje.
  - **Control de posición y rotación** a través de los módulos `Cinemachine Follow` y `Cinemachine Rotation Composer`.
  - **Damping** en posición y rotación para suavizar los movimientos de cámara.
  - **Offset vertical y posterior** para mantener una vista en tercera persona ligeramente elevada.

  Aunque la base técnica es consistente, la configuración específica se ajusta según el diseño del nivel y el tono narrativo que se desea transmitir. A continuación, se detallan las particularidades más relevantes:

  Escena 1: Entrada de la casa Este nivel tiene un diseño más abierto y natural, con un terreno irregular que podría afectar la experiencia visual si se utiliza una cámara demasiado dinámica. Por ello, se opta por una configuración más estable, con un *follow offset* ligeramente más bajo y sin *damping* vertical, asegurando una visión clara del entorno sin interferencias por los desniveles del terreno.

  Desde un punto de vista artístico, esta elección favorece una sensación de exploración y descubrimiento, invitando al jugador a observar con detenimiento el paisaje.

Escenas 2 y 3: Transiciones narrativas y llegada al interior Estas escenas buscan reforzar una atmósfera más narrativa y emocional. Se incrementa el *position damping* tanto en los ejes X como Y, lo que genera una sensación de cámara flotante, más cinematográfica. Además, se eleva el *follow offset* en altura y distancia, ampliando el campo de visión y resaltando el espacio escénico.

Esto permite enfatizar el recorrido del personaje dentro del mundo y otorga un ritmo visual más pausado y deliberado, adecuado para momentos de transición o impacto visual.

Un detalle técnico importante es la ubicación del componente `Audio Listener`. Tradicionalmente, se sitúa en la `Main Camera`, sin embargo, en este proyecto se reubicó al personaje jugador. Esta decisión se tomó porque la cámara, al mantenerse distante o con rutas de movimiento propias, no siempre atravesaba las zonas de audio espacial (como áreas de eco), lo que provocaba errores perceptivos. Al colocar el `Audio Listener` en el jugador, se garantiza una correcta interacción con el entorno sonoro y se mejora la inmersión del usuario.

- **Sistema de Luces**
  El componente `Light` de Unity se utiliza como base para construir múltiples comportamientos personalizados que amplían su funcionalidad de manera modular. En lugar de heredar de esta clase, se emplea un enfoque compositivo donde scripts individuales operan sobre ella.

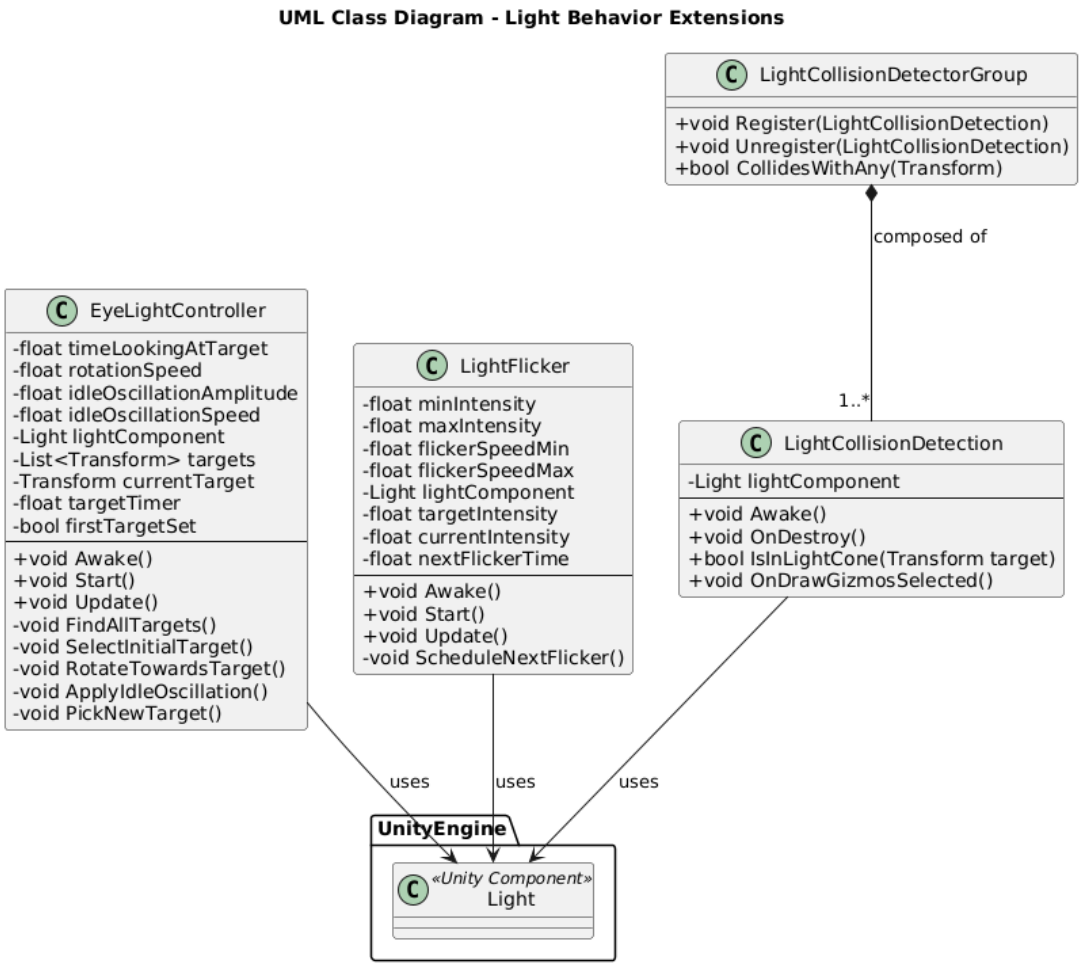


Figura 38: Diagrama de clases de los comportamientos que extienden la funcionalidad del componente `Light`.

- **Comportamientos asociados al componente Light.** La figura 38 ilustra cómo varias clases se apoyan en el componente `Light` para implementar comportamientos específicos. Estos scripts acceden al componente mediante referencias serializadas o a través de `GetComponent<Light>()`, y permiten componer funcionalidades sin acoplamiento rígido. A continuación se describen brevemente:
  - **EyeLightController**: orienta dinámicamente la luz hacia objetivos en la escena, alternando entre ellos con rotaciones suaves y una leve oscilación que simula vigilancia activa.
    Interpolación suave hacia el objetivo actual:

    ```
    Quaternion targetRotation = Quaternion.LookRotation(
       direction);
    transform.rotation = Quaternion.Slerp(
        transform.rotation, targetRotation,
        Time.deltaTime * rotationSpeed
    );
    ```

    Oscilación sutil mientras mantiene el enfoque:

```
float x = Mathf.Sin(Time.time * idleOscillationSpeed) *
   idleOscillationAmplitude;
float y = Mathf.Cos(Time.time * idleOscillationSpeed *
   0.8f) * idleOscillationAmplitude;
transform.rotation *= Quaternion.Euler(x, y, 0f);
```

- **LightFlicker**: modifica de forma aleatoria la intensidad de la luz en cada fotograma, simulando parpadeos o inestabilidad lumínica.
  Interpolación hacia una intensidad aleatoria:

  ```
  currentIntensity = Mathf.Lerp(currentIntensity,
     targetIntensity, Time.deltaTime * 20f);
  lightComponent.intensity = currentIntensity;
  ```

- **LightCollisionDetection**: permite determinar si un objeto se encuentra dentro del cono de luz, aplicando cálculos de distancia, ángulo y visibilidad (raycasting).
  Verificación de colisión mediante raycast:

  ```
  if (Physics.Raycast(lightComponent.transform.position,
     toTarget.normalized, out RaycastHit hit, distance)) {
      return hit.transform == target;
  }
  ```

- **LightCollisionDetectorGroup**: actúa como agrupador y punto de consulta centralizado. Contiene varias instancias de `LightCollisionDetection` y permite verificar si un objeto colisiona con alguna de ellas. Mantiene una relación de composición 1:N con dichas instancias.
  Consulta global desde cualquier agente:

  ```
  public static bool CollidesWithAny(Transform target) {
      foreach (var detector in detectors) {
          if (detector != null && detector.IsInLightCone(
             target))
              return true;
      }
      return false;
  }
  ```

- **Patrón de detección de colisión con la luz.** El sistema de detección de colisión con luces implementa un patrón de diseño que centraliza la lógica de detección en un grupo compartido, permitiendo que cualquier agente de la escena (como el jugador) pueda consultar si está siendo afectado por alguna luz. La figura 39 presenta una visión abstracta del patrón utilizado. En él, el componente `LightCollisionDetectorGroup` mantiene una lista de detectores registrados. Los agentes externos no interactúan directamente con los detectores, sino que realizan consultas globales mediante el método `CollidesWithAny(transform)`. Esto permite una arquitectura extensible, limpia y desacoplada.

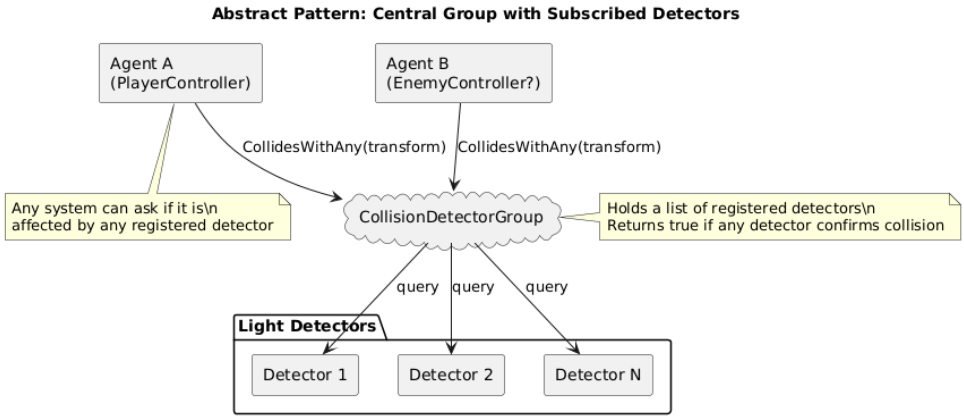


Figura 39: Patrón abstracto: grupo central con detectores suscritos.

La lógica concreta que se ejecuta en tiempo de juego se detalla en la figura 40. En cada actualización del jugador, se realiza una consulta al grupo de detectores. Este grupo evalúa en orden cada uno de los detectores registrados llamando a su método `IsInLightCone(transform)`. Si alguno de ellos devuelve `true`, el grupo responde afirmativamente al jugador, quien, si no ha sido ya detectado anteriormente, ejecuta su lógica de muerte. Si ninguno de los detectores activa una colisión, el jugador continúa normalmente.

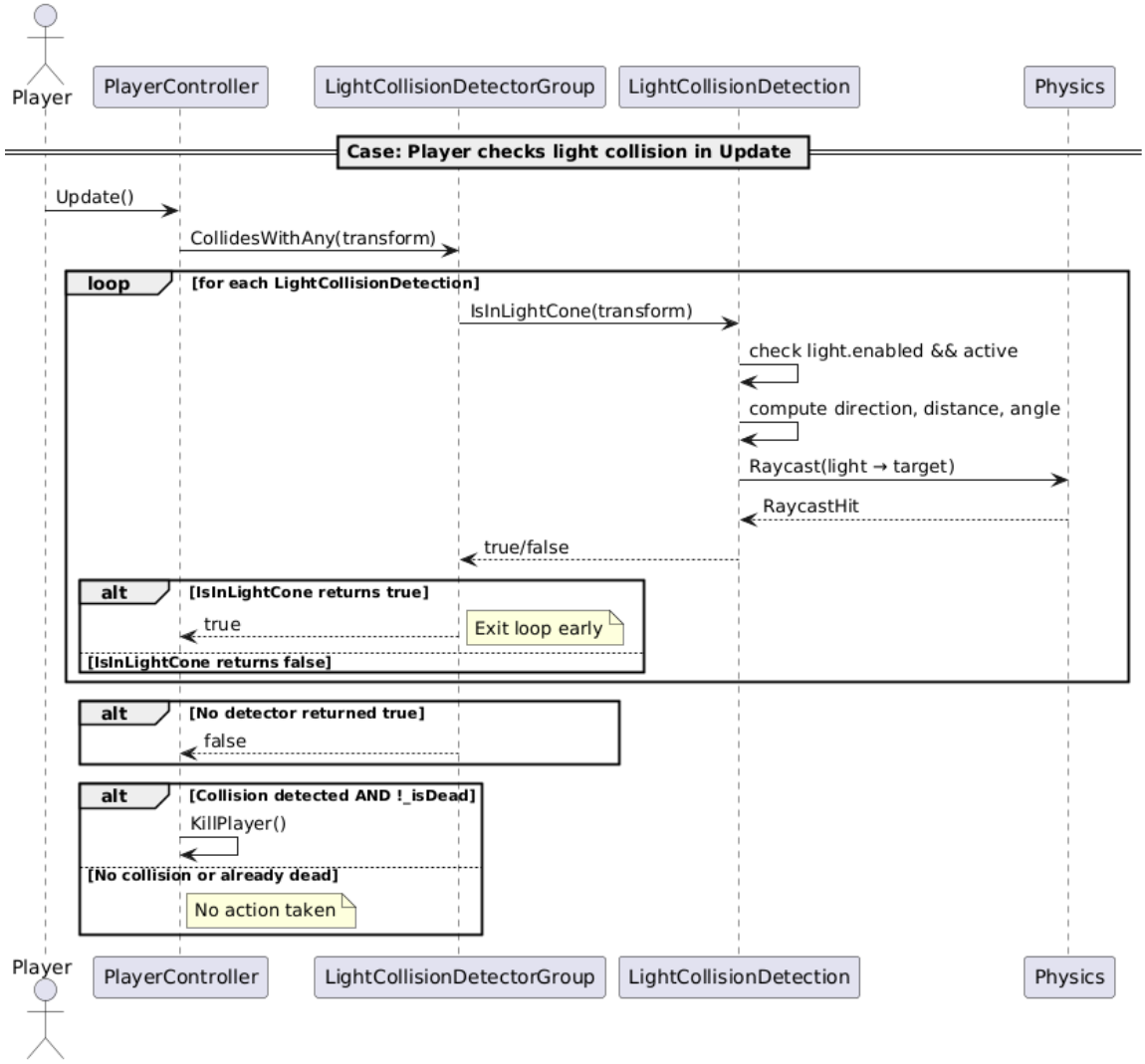


Figura 40: Diagrama de secuencia del proceso de detección de colisión con la luz.

La adopción de esta arquitectura modular basada en composición y registro ofrece importantes ventajas frente a soluciones más acopladas. En una versión inicial del sistema, la detección de colisiones con la luz se encontraba embebida directamente en la lógica del jugador, con clases como `LightPlayerDetector`, que contenían referencias explícitas al jugador y estaban fuertemente acopladas a él. Aunque esta aproximación podía parecer suficiente para el caso concreto —donde el único afectado por la luz era el jugador—, limitaba considerablemente la escalabilidad y la reutilización del sistema.

En contraste, el enfoque presentado en esta sección permite un diseño más limpio y mantenible. Cada luz simplemente se registra en un grupo centralizado y encapsula su propia lógica de detección, lo que permite añadir nuevas luces sin modificar el código del jugador. Además, si en el futuro se desea que otros agentes (como enemigos, cámaras o NPCs) puedan ser detectados por las luces, el sistema ya está preparado para ello: bastaría con que dichos agentes realicen consultas al grupo de detectores del mismo modo que lo hace el jugador. Este

nivel de desacoplamiento y extensibilidad es fundamental para mantener un código base robusto, adaptable y coherente con buenas prácticas de diseño.

2.1.2.4 Detalles de implementación de las escenas

2.1.2.4.5 Menú Sistema de menús con un diseño modular y reutilizable, con el objetivo de mantener una estructura escalable a nivel funcional.

- **Menú Principal:** Escena independiente que se carga al iniciar el juego. Es un Menú de Inicio que contiene los botones `New Game` que realiza un cambio de escena a la primera escena jugable, `Options`, implementado como Prefab reutilizable, que muestra el panel de opciones de volumen y `Exit` que cierra la aplicación. Tiene un controlador central `MainMenuManager.cs`, que gestiona las transiciones.
- **Menú de Pausa:** Prefab integrado en cada escena jugable que incluye los botones `Continue` para reanudar el juego, `Restart` para volver a empezar la escena que se esté jugando, el mismo `Options` que el del Menú Principal y `Main Menu` para iniciar de nuevo el juego desde el Menú Principal. Se accede y se sale pulsando ESC.

El Menú de Opciones tiene implementado un botón `Back` que permite regresar al menú correspondiente, ya sea el Principal o el de Pausa. También permite mantener los ajustes de volumen de la música y de los efectos de sonido. Estos son dos `Sliders` asociados al `AudioMixer` mediante `AudioManager.cs`, el cual permite mantener los ajustes de volumen en cualquier escena o menú del juego. En los cambios de escena se usan los métodos que dependen de `SceneManager` como `ChangeToNextScene()` y `ChangeToScene()`.

2.1.2.4.6 Fases

- **Escena 1: La Entrada de la Casa**
  COMPLETAR Introducir uso de pickable(llave), draggable (silla), puzle del cuadro de luces, la llave.
- **Escena 2: La Cocina**
  La primera estancia que atraviesa el jugador al cruzar el umbral de la puerta principal es una cocina deteriorada y parcialmente iluminada, diseñada específicamente para generar una atmósfera de tensión mediante el uso narrativo de la iluminación. En esta habitación, las principales fuentes lumínicas son dos focos: uno intermitente proveniente del interior de una antigua nevera y otro más intenso situado en la campana extractora ubicada sobre los fogones.

  El diseño funcional de esta escena requirió la implementación de dos nuevos scripts que heredan de la clase abstracta `Interactable`: `Closable.cs` y `Switchable.cs`. Ambos scripts implementan el método `OnStartInteraction`, mientras que el método `OnStopInteraction` se mantiene vacío de forma deliberada.

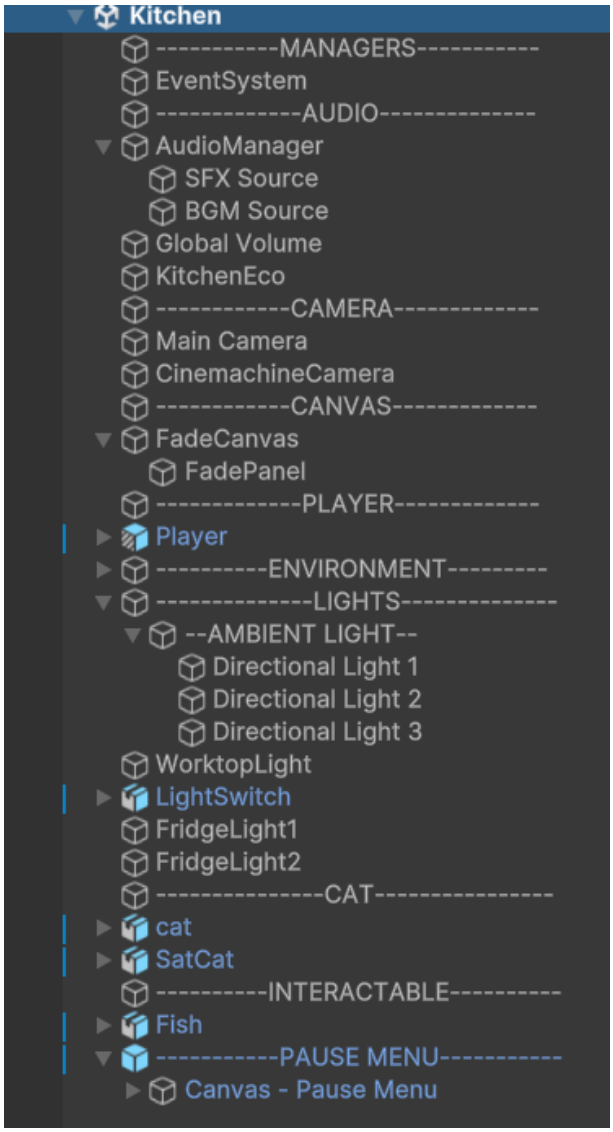


Figura 41: Distribución de la jerarquía para la escena de La Cocina.

- `Closable.cs`: Este script gestiona el cierre de la puerta de la nevera utilizando un componente `Animator`. La nevera incluye dos fuentes de luz: una interna (`"FridgeLight1"`) que simula la iluminación habitual del electrodoméstico, y otra (`"FridgeLight2"`) que incorpora el comportamiento de parpadeo a través del script `LightFlicker.cs`. Durante la interacción, se desactiva la primera luz mediante `SetActive(false)`, lo que impide que la puerta siga emitiendo luz una vez cerrada. A continuación, se activa una animación mediante `anim.SetTrigger(.ºpenDoor")`, aunque esta se reproduce en sentido inverso, ya que originalmente fue diseñada como animación de apertura. Gracias al uso de sombras suaves (*soft shadows*) en la luz, el cierre de la puerta bloquea completamente el paso de la iluminación, permitiendo que el entorno quede a oscuras y evitando que el jugador resulte dañado por la exposición directa a la luz.
- `Switchable.cs`: Este script gestiona el encendido y apagado de las luces asociadas a un objeto interactuable. Durante la interacción, y siempre que el jugador se encuentre en el suelo (`PlayerController.Instance.IsGrounded()`), se alterna el estado de todos los componentes `Light` hijos del objeto principal mediante un bucle sobre la colección `lightsToToggle`. Esta lógica permite apagar la luz de la campana extractora, la cual es hija del objeto `lightSwitch` que contiene este script, desbloqueando así zonas previamente inaccesibles por su alto nivel de exposición.

El puzle central de esta escena gira en torno a tres elementos clave: una sardina, un cuenco de comida y un gato que bloquea el acceso a la siguiente sala. Tanto la sardina como el gato están asociados a scripts específicos que definen su comportamiento: `Sardine.cs` y `CatWalk.cs`.

- `Sardine.cs`: Asociado al objeto de la sardina, que además es un `Pickable`. Este script implementa el método `OnTriggerEnter`. Cuando la sardina entra en contacto con el cuenco etiquetado como `"DogBowl"`, se detecta la colisión mediante el sistema de *triggers*. Esto activa el método `Activate()` del script `CatWalk`, lo que inicia la animación y el movimiento del gato.
- `CatWalk.cs`: Este script define el comportamiento dinámico del gato. Al activarse, se desactiva el modelo estático del gato (`catSat`) y se activa el modelo animado (`cat`), que inicia una animación de carrera mediante `animator.SetTrigger("Run")`. El gato se desplaza progresivamente hacia un destino predefinido (`destino`), correspondiente a la posición del cuenco. Dado que no fue posible obtener un modelo que integrase tanto la animación de salto desde la encimera como la caminata hacia el cuenco, se utilizan dos prefabs diferentes: uno para el gato sentado y otro para el modelo en movimiento. El sistema alterna entre estos prefabs mediante activación y desactivación en la jerarquía de objetos. La disposición estratégica de la cámara contribuye a reforzar la ilusión de continuidad, haciendo creer al jugador que el gato ha saltado y caminado hasta el cuenco, cuando en realidad el modelo original ha desaparecido de escena. El movimiento se implementa mediante `Vector3.MoveTowards`, y se detiene automáticamente al alcanzar una distancia mínima respecto

al destino. En ese momento, se activa la animación de parada mediante `animator.SetTrigger("Stop")`.

La resolución del puzle se plantea de la siguiente manera: el jugador comienza frente a una encimera desde la que puede observar un cuenco parcialmente iluminado por la luz parpadeante de la nevera. Para progresar, debe utilizar una silla como plataforma para alcanzar la parte superior de las encimeras. Sin embargo, la luz directa de la campana extractora representa un obstáculo que impide el avance seguro. Para sortear este problema, el jugador debe escalar una estantería ubicada en la cocina y activar un interruptor que apaga dicha luz, desbloqueando así una nueva ruta que conduce a la entrada de un conducto de ventilación. No obstante, este acceso se encuentra bloqueado por un gato que impide el paso.

El jugador debe entonces retroceder y atravesar la zona anteriormente expuesta a los fogones (ahora a oscuras), para obtener la sardina. Debido a que el cuenco de comida permanece bajo la luz de la nevera, no es posible colocar la sardina sin ser detectado. Por tanto, es necesario cerrar primero la puerta de la nevera, lo cual requiere un salto sobre el gato para acceder al punto de interacción. Una vez completada esta acción, el jugador puede colocar la sardina en el cuenco. Esto desencadena el comportamiento del gato, que se desplaza hacia la comida, despejando así el camino hacia la siguiente sala.

- **Escena 3: El Salón**

Tras abandonar la cocina mediante un polvoriento conducto de ventilación, el jugador accede a una nueva estancia: un salón más amplio pero igualmente deteriorado y en desuso. El personaje emerge desde una rejilla, oculta tras unas viejas cajas de cartón situadas en una esquina del suelo. La atmósfera general mantiene la estética decadente del nivel anterior, pero introduce una nueva amenaza principal: una intensa luz rojiza que emana de un televisor encendido al fondo de la habitación.

El objetivo del jugador en esta sección es claro: atravesar la sala y alcanzar la puerta situada en el lado opuesto, evitando en todo momento ser alcanzado por el haz de luz. A nivel de diseño, esta mecánica se apoya en el comportamiento programado en el script `EyeLightController.cs`, descrito en una sección previa del documento.

La luz proyectada por el televisor realiza un barrido constante de la habitación, desplazándose a lo largo de una ruta compuesta por distintos obstáculos. Entre estos se encuentran cajas de documentos, restos de cartones de pizza y un ventilador eléctrico averiado, todos ellos colocados de forma estratégica para proporcionar coberturas temporales al jugador. La navegación por la sala se convierte así en un juego de sigilo y posicionamiento, donde el jugador debe observar cuidadosamente los patrones de movimiento de la luz y planificar sus desplazamientos de un punto seguro a otro.

Además de la luz que emite el televisor, existe una segunda amenaza que puede sorprender al jugador si decide tomar un atajo bordeando la habitación por el lado más expuesto. Justo frente al televisor hay una ventana desde la que

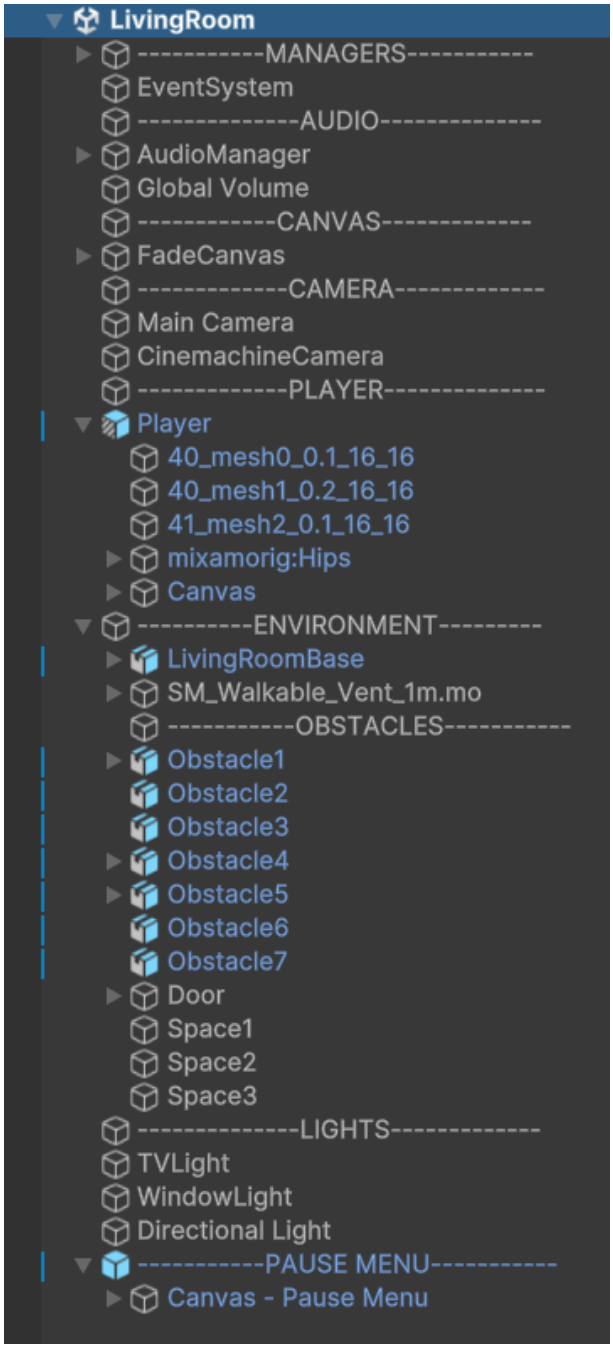


Figura 42: Distribución de la jerarquía para la escena del Salón.

se cuela una intensa luz exterior, igualmente letal. De este modo, el jugador no tiene otra opción que utilizar el mobiliario y los objetos del entorno como cobertura, evitando en todo momento exponerse a cualquiera de las dos fuentes de luz.

Esta escena supone un cambio en el ritmo del juego, ya que introduce una nueva dinámica basada en el sigilo y la observación. El jugador debe prestar atención a los patrones de luz y buscar los momentos adecuados para avanzar, lo que le obliga a adoptar una actitud más cuidadosa y estratégica para poder superar el desafío.

- Menú
- Fases
  - Captura de jerarquia
  - Detalles de implementación de las clases que no se explicaran (por ejemplo el gato, decir que la tele tiene el movimiento aleatorio de la luz, etc.) + scripts
  - Lista de prefabs
  - Mapa esquemático o layout
  - Escena 1: Cocina

Para ejecutar el juego **Vestigium 3D**:

1. Abrir la carpeta del juego y hacer doble click sobre el archivo `Vestigium3D.exe`.
2. Pulsar `New Game` para iniciar una partida.