COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI ICADE CIHS

Proyecto Final
Warehouse Escape

Javier Mendoza Guerrero

Lucía Tamarit Barberán

Grupo B

Paradigmas y Técnicas de la Programación Avanzada – 3º
Ingeniería Matemática e Inteligencia Artificial

1. Resumen general y Objetivo principal

Versión de Unity: 6000.2.6f2

Objetivo del juego:
El jugador controla a un ladrón infiltrado en un almacén el cual debe encontrar un objeto y escapar por la salida, en cada partida cambia la localización del objeto, la localización de la salida y la localización de las alarmas. El juego incluye varios niveles de dificultad, donde cambian la velocidad del robot y el número de alarmas.

Mecánicas principales:
El robot utiliza una IA basada en una máquina de estados (patrulla y persecución).
Las alarmas detectan al ladrón y activan al robot.
El jugador debe moverse sin pisar ninguna alarma y alcanzar el objeto y la salida. Si es descubierto este debe intentar llegar a la salida antes de que le alcance el robot.

Condiciones de Victoria/Derrota:

- Victoria: escapar con el objeto.
- Derrota: el robot atrapa al ladrón.

Repositorio Github:

https://github.com/JavierMendozaGuerrero/Warehouse_Escape_JavierLucia.git

2. Arquitectura del proyecto

La arquitectura del videojuego Warehouse Escape se ha diseñado siguiendo una estructura modular reflejada en el diagrama UML del proyecto. El objetivo principal de esta arquitectura es separar las responsabilidades de cada sistema, facilitar la comprensión del código y permitir la extensión del juego sin necesidad de modificar su núcleo principal.

GameManager

El eje central de la arquitectura es el GameManager, que actúa como coordinador del flujo de la partida. Esta clase es responsable de mantener el estado global del juego, controlar las condiciones de victoria y derrota y notificar el final de la partida al resto del sistema. El GameManager no implementa directamente comportamientos específicos de otros módulos, sino que se comunica con ellos mediante eventos.

Personajes

En cuanto a los personajes, el jugador está representado por la clase. El enemigo está representado por la clase GuardRobot.

La inteligencia artificial de los robots guardianes se implementa mediante una máquina de estados, compuesta por la clase StateMachine, la interfaz IRobotState y los estados concretos PatrolState y ChaseState. La máquina de estados mantiene una referencia al estado actual y delega en él el comportamiento del robot en cada actualización. Gracias a esta estructura, el comportamiento del guardia es extensible, permitiendo añadir nuevos estados en el futuro sin modificar la lógica existente del robot.

La IA en sí es un algoritmo simple que se aplica en el estado de persecución. Este busca el vector director que lleve al robot hasta el Thief siguiendo el camino más cercano posible. Si el robot choca con un objeto o detecta que su avance está bloqueado, realiza comprobaciones laterales mediante SphereCast para buscar una ruta alternativa. En caso de quedar atascado durante un breve periodo de tiempo, activa un comportamiento de escape que fuerza un pequeño retroceso y un giro pronunciado, permitiéndole despegarse de paredes o esquinas y continuar la persecución.

Sistema de Alarmas

El sistema de alarmas se gestiona a través de la clase AlarmFactory, que implementa el patrón Factoría. Esta clase es responsable de crear las alarmas en el escenario, controlar cuántas se instancian y gestionar su eliminación cuando es necesario. Cada alarma se representa mediante la clase Alarm, que detecta la entrada del jugador y notifica a los guardias para que cambien su comportamiento. Este diseño desacopla la creación de alarmas del resto del sistema y permite ajustar la dificultad del juego sin modificar el GameManager.

Generación de escenario

Las clases ObjectiveSpawner y ExitSpawner generan el objetivo y la salida en posiciones aleatorias que cambian en cada partida, mientras que ObjectiveItem y ExitZone notifican al GameManager cuando se cumplen las condiciones del juego.

Sistema de dificultad
La clase DifficultySettings, aplicada desde MainMenu, encapsula los parámetros de dificultad para desacoplar su configuración de la lógica del juego.

Interfaz de usuario
Las clases ScoreManager, ScoreUI y GameOverUI muestran información al jugador reaccionando a eventos emitidos por el GameManager.

3. Patrones de diseño

El proyecto Warehouse Escape se diseñó con una arquitectura basada en patrones de diseño para asegurar una correcta separación de responsabilidades y facilitar la escalabilidad, manteniendo estos principios durante toda la implementación final. A continuación, se resumen los patrones utilizados, la mayoría de los cuales ya han sido mencionados previamente en el informe.

El patrón Factory se implementa mediante la clase AlarmFactory, que centraliza la creación y gestión de alarmas según la dificultad, evitando el acoplamiento con el GameManager.

El componente ScoreManager se implementa siguiendo el patrón Singleton, ya que gestiona un estado global único (la puntuación) que debe ser accesible desde distintos componentes del juego, garantizando la existencia de una única instancia.

La inteligencia artificial de los robots se desarrolla siguiendo el patrón State, utilizando una máquina de estados que organiza los distintos comportamientos y permite alternar entre ellos.

El sistema de dificultad mantiene el principio del patrón Bridge mediante el uso de ScriptableObjects, que desacoplan la configuración de los niveles de la lógica del juego y facilitan la modificación de parámetros sin cambios de código.

El GameManager actúa como elemento central de coordinación y utiliza un sistema de eventos que sigue el patrón Observer, permitiendo que otros sistemas, como la interfaz, reaccionen al estado de la partida sin dependencias directas.

4. Problemas y soluciones durante el desarrollo

Durante el desarrollo del videojuego surgieron varios problemas técnicos que requirieron ajustes respecto a la planificación inicial, como la mejora del movimiento visual del personaje jugador. Para conseguir una animación de la acción de correr, fue necesario crear y configurar manualmente un Animator en Unity, lo que permitió sincronizar correctamente las animaciones con el movimiento del personaje.

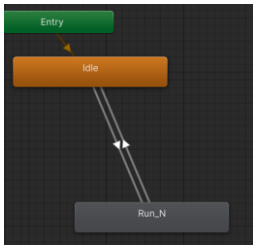

Figura 1: Animator del jugador en Unity

Otro problema relevante surgió al implementar el seguimiento de cámara. Inicialmente, siguiendo lo aprendido en clase, la cámara se configuró como hija del jugador, pero esta solución provocaba rotaciones no deseadas. Por ello, se optó por desacoplarla y utilizar un script de seguimiento independiente.

Esto son solo un par de ejemplos de los muchos cambios y retos que se afrontaron durante el proyecto.

5. Algunas justificaciones de relaciones en el UML final:

La relación entre ExitSpawner y ExitZone se define como una composición, ya que el spawner crea, gestiona y controla la existencia de la salida, estando su ciclo de vida completamente ligado a él.

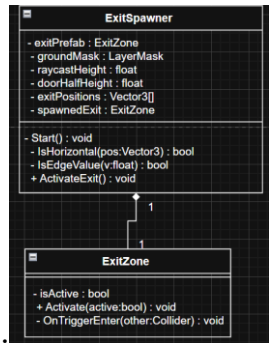

Figura 2: relación entre ExitSpawner y ExitZone

En cambio, la relación entre ObjectiveSpawner y ObjectiveItem se define como una asociación, ya que el spawner solo crea y coloca el objetivo, mientras que el objeto funciona de forma autónoma y gestiona su comportamiento sin depender del spawner.

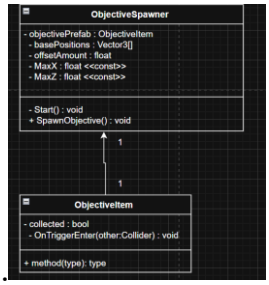

Figura 3: relación entre ObjectiveSpawner y ObjectiveZone

La máquina de estados del robot se representa en el UML mediante una relación entre StateMachine y la interfaz IRobotState, mientras que los estados concretos PatrolState y ChaseState mantienen una relación de dependencia con dicha interfaz como vemos en la figura 4.

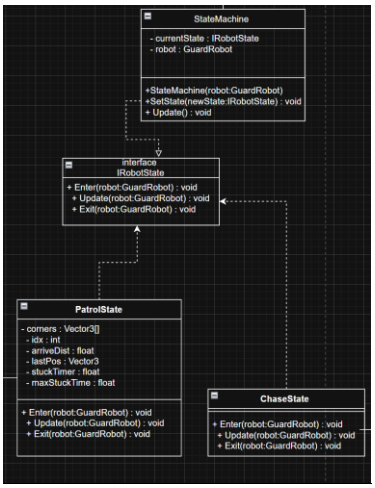

Figura 4: relaciones de la interfaz IRobotState

El MainMenu mantiene una relación de asociación con GuardRobot, ya que es capaz de modificar sus parámetros en función del nivel de dificultad seleccionado por el jugador. De manera similar, existe una asociación entre MainMenu y AlarmFactory, puesto que el menú decide cuántas alarmas deben generarse según la dificultad elegida. En ambos casos, el menú no crea ni destruye estos objetos, sino que interactúa con ellos para configurar su comportamiento al inicio de la partida.

Por último, las clases Thief, ObjectiveItem y ExitZone mantienen una relación de asociación unidireccional con el GameManager. Estas clases conocen al GameManager y le notifican eventos relevantes del juego, como la recogida del objetivo, la captura del jugador o la salida del almacén. Sin embargo, no controlan el ciclo de vida del GameManager ni forman parte de él, y este tampoco es responsable de su creación o destrucción. Por este motivo, la relación se representa como una asociación y no como una composición o agregación.

6. Resultados

Figura 5: Menu Principal

Figura 6: Búsqueda de diamante

Figura 7: Alarma activada

Figura 8: Game Over

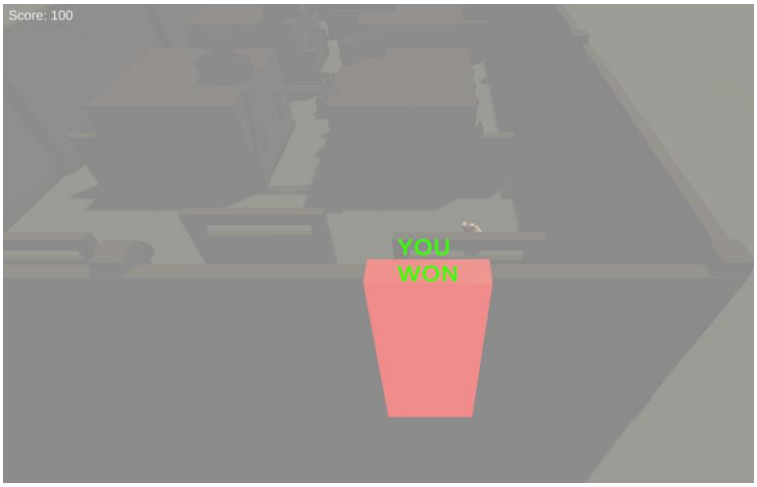

Figura 9: Ladrón alcanza salida con el diamante