PRÁCTICA 1: IMPLEMENTACIÓN DE UN JUEGO 2D

Índice de contenido

1. Introducción

1.1. Objetivo

El objetivo es implementar las mecánicas básica de un shmup 2D.

Contaremos con la nave del jugador, capaz de moverse con WASD y disparar pulsando Spacebar.

Tendremos un GameManager, que actuará principalmente como spawner de enemigos, spawneando distintos tipos de tropas en intervalos de tiempo y posiciones aleatorias, con distintas configuraciones de enemigos. El GameManager actualizará también la puntuación del jugador en el UI.

Por último, tendremos una DeathZone en la parte inferior de la pantalla. Si un enemigo la alcanza, la partida termina. Igualmente, la partida termina si el jugador colisiona con un enemigo.

1.2. Organización

La práctica se estructura en cuatro sesiones guiadas y una quinta sesión para terminar o preguntar las últimas dudas si fuera necesario. En la guía para cada sesión se indican los assets a crear, así como los componentes y métodos a implementar y la funcionalidad asociada a los mismos.

No es necesario añadir métodos adicionales a los indicados, aunque puede hacerse si resulta necesario. Sí que es necesario utilizar propiedades adicionales para muchos de los scripts que no aparecen indicadas en la guía. Asimismo, será necesario implementar funcionalidad adicional en el método Start para iniciliazar muchas de estas propiedades o referencias.

No olvides cachear todas las referencias necesarias en la inicialización: Será motivo de suspenso la utilización de métodos de búsqueda iterativa (GetComponent, Find...) en métodos que se ejecuten en cada frame.

2. Primera sesión: Movimiento e input

2.1. Disparo

- Crear objeto en escena, al que llamaremos Shot. Podemos partir de objeto tipo Sprite.
- Debe incluir los siguientes componentes de Unity: Sprite Renderer, Box Collider 2D, Rigid Body 2D
- Añadiremos dos componentes más, implementados por nosotros:
  - ShotMovementController: Mueve el disparo hacia arriba con velocidad configurable ShotSpeed.
  - LifetimeController: Destruye el objeto después de cierto tiempo configurable LifeTime.
  - Probar prefab instanciándolo en la escena.

2.2. Nave del jugador

- Crear objeto en escena, al que llamaremos Player. Podemos partir de objeto tipo Sprite.
- Debe incorporar los siguientes scripts de Unity: Sprite Renderer, Box Collider 2D, Rigid Body 2D.
- Añadiremos dos componentes más, implementados por nosotros:
  - ShipInputComponent: Se encarga de detectar el input del jugador y, en base al mismo, realizar las llamadas oportunas para aplicarlo a otros componentes. Implementa los siguientes métodos:
    - void Update() → Debe implementar la siguiente funcionalidad:
      - Detectar los inputs WASD, en base a ellos componer la dirección del movimiento. Llamar al método SetMovementDirection del componente ShipMovementController.
  - Ship Movement Component: Se encarga de aplicar el movimiento deseado a la nave. Implementa los siguientes métodos:
    - public void SetDirection(Vector3 movementDirection) → Establece la dirección deseada para el movimiento.
    - void Update() → Deberá aplicar la dirección establecida con la velocidad deseada.

3. Segunda sesión: Prefabs e instanciación

3.1. Disparo

- Crear prefab p_Shot. Aplicar sobre el mismo el objeto Shot creado en la sesión anterior.

3.2. Instanciación

- Añadir al objeto Player un nuevo componente:
  - Ship_AttackController: Deberá encargarse de instanciar un nuevo disparo en escena cada vez que recibe la orden de disparar. Para ello, implementa los siguientes métodos y propiedades:
    - public void Shoot() → Método que instancia un nuevo objeto en escena, tomando el prefab p_Shot como referencia.
    - Private GameObject myShot → Propiedad que guardará la referencia al prefab pShot, asignado en editor.

3.3. Enemigo

- Crear prefab p_Enemy.
- Deberá incluir los siguientes componentes de Unity: SpriteRenderer, BoxCollider2D.
- Además, deberá incluir el siguiente componente, implementado por nosotros:
  - Enemy_LifeComponent: Se encarga de gestionar lo que ocurre cuando un enemigo recibe daño. Para ello, implementa el siguiente método:
    - public void Damage() → Destruye el GameObject del propio enemigo.

3.4. Tropas

- Crear prefab p_Squad_01.
  - Añadir varias instancias de prefab p_Shot dentro de p_Squad_01 y establecer sus posiciones relativas a nuestro gusto.
  - El prefab deberá incluir los siguientes componentes, implementados por nosotros:
    - SquadMovementComponent: Implementará un movimiento descendente, combinado con desplazamiento lateral alternativamente a izquierda y derecha, regulado por una función sinusoidal. En función del valor del seno, mayor o menor que cero, el movimiento lateral será hacia la izquierda o hacia la derecha. Para ello, implementará los siguientes métodos y propiedades:
      - void Update() → Método que actualiza la posición de la Squad en base al algoritmo descrito.
      - Private int _initialDirection → Movimiento lateral inicial a izquierda o derecha.
      - Private float _verticalSpeed → Velocidad de desplazmiento hacia abajo.

        - Private float _horizontalSpeed → Velocidad de desplazamiento lateral.
        - Private float _frequency → Velocidad angular de la sinusoide que regula la dirección.
    - WorldVerticalDeadlineComponent: Se encargará de que las tropas se autdestruyan al alcanzar cierta cota vertical. Para ello, implementa los siguientes métodos:
        - void Update() → Se encargará de comprobar la posición en y para destruir el GameObject si se alcanza la cota establecida.
- Parametrizar SquadMovementComponent a nuestro gusto.
- Crear prefab p_Squad_02, siguiendo el mismo proceso. La configuración de enemigos y la parametrización del componente deberán ser diferentes.
- Podemos repetir el proceso añadiendo más squads (prefabs p_Squad_03, p_Squad_04... tampoco perdamos mucho tiempo en eso)

4. Tercera sesión: GameManager, UI y patrón singleton

4.1.UI

- Añadir a la escena objeto de tipo UI → Text. Lo llamaremos GameOver y lo configuraremos para mostrar el texto "Game Over!".
- Añadir a la escena objeto de tipo UI → Text. Lo llamaremos Score y lo configuraremos para mostrar el texto "Score:".

4.2. GameManager

- Crear objeto vacío.
- Deberá incluir el siguiente componente, implementado por nosotros:
  - GameManager: Aunque el GameManager tendrá algunas funcionalidades típicas de cualquier GameManager (por ejemplo, instancia única y una mínima gestión de la partida y el flujo de juego) será en este caso, sobre todo, quien gestione el spawn de enemigos. Para ello, deberá implementar los siguientes métodos y propiedades:
    - private bool _isGameRunning → Variable booleana que indica si el juego está o no en marcha.
    - static private GameManager _instance → Instancia única del GameManager.
    - static public GameManager Intance → Accesor que permite acceder a dicha instancia desde otros objetos y componentes.
    - private GameObject[] _squads → Array de objetos, en este caso prefabs, que coresponden a los distintas squads que el GameManager podrá instanciar. Configurable desde editor.
    - void Update() → Deberá implementar temporizador para que, cada cierto tiempo, se instancie una nueva squad, elegida aleatoriamente de entre las disponibles en el array _squads. Deberá instanciar el squad en un punto distinto elegido aleatoriamente en cada ocasión dentro de una determinada área válida.
    - Public void OnEnemyDies(int scoreToAdd) → Método llamado por los enemigos cuando mueren. Recibe los puntos a añadir a la puntuación y los añade, actualizando el valor en el objeto Score.
    - Public void OnEnemyReachesBottomline() → Debe llamar al método GameOver, también perteneciente al GameManager.
    - Public void OnPlayerDies() → Debe llamar al método GameOver, también perteneciente al GameManager.
    - Private void GameOver() → Debe activar el objecto GameOver, desactivar el objeto Player, y setear la variable _isGameRunning a false, para indicar que el juego ya no está en marcha.
- Incluir en el método Damage() del Enemy_LifeComponent llamada al método OnEnemyDies(int scoreToAdd) previa a la destrucción del enemigo.

5. Cuarta sesión: Triggers, matriz de colisión y duck typing

5.1. Matriz de colisión

- Definir las siguientes capas de colisión:
  - Player: Para Player y p_Shot
  - Enemy: Para p_Enemy.
- Configurar la matriz de colisión:
  - Player sólo debe colisionar con Enemy.

5.2. Utilización de Triggers

- Debemos hacer que el componente ShotController de p_Shot cause daño a un enemigo cuando choque con él. Para ello, implementaremos el siguiente método:
  - private void OnTriggerEnter2D(Collider 2D collision) → Este método debe llamar al método Damage del enemigo colisionado. Utilizará GetComponent para comprobar si el GameObject colisionado es un enemigo (duck typing), y si lo es, llamará a su método Damage().
- Debemos hacer que Player se autodestruya si colisiona con un enemigo, informando de ello al GameManager. Para ello, agregaremos el componente Ship_LifeComponent al Player, que implementará el siguiente método:
  - private void OnTriggerEnter2D(Collider 2D collision) → Este método debe llamar al método OnPlayerDies() del GameManager.
- Debemos hacer que el jugador pierda la partida si un enemigo alcanza la zona inferior.
  - Añadiremos un objeto a la escena al que llamaremos WorldLimitDown:
    - Debe inlcuir los siguientes componentes de Unity:
      - SpriteRenderer, BoxCollider2D, RigidBody2D.
    - Además, le añadiremos el siguiente componente, implementado por nosotros:
      - DeathZone: Este componente se encarga de detectar la entrada de otro collider en el mismo, informando de ello al GameManager para que termine la partida. Para ello implementa el siguiente método:
        - private void OnTriggerEnter2D(Collider2D collision) → Debe llamar al método OnEnemyReachesBottomLine del GameManager.

5.3. Configuración de Colliders y RigidBody

En el caso de que algo no funcione correctamente, puede que no hayamos configurado Colliders o RigidBodies correctamente:

- Player:
  - Collider: No es trigger.
  - RigidBody: BodyType = Dynamic, Freeze Rotation Z, Gravity Scale = 0
- p_Shot:
  - Collider: Es Trigger
  - RigidBody: BodyType = Dynamic, FreezePosition X, Y, Freeze Rotation Z, Gravity Scale = 0
- p_Enemy:
  - Collider: Es Trigger
- WorldLimitDown:
  - Collider: Es Trigger
  - RigidBody: BodyType = Dynamic, FreezePosition X, Y, Freeze Rotation Z, Gravity Scale = 0

6. Quinta y última sesión: Preguntas finales a responder

- En el método OnTriggerEnter2D del shot controller hemos utilizado GetComponent para comprobar si el objeto colisionado era un enemigo antes de llamar a sus métodos.
  - ¿Conoces alguna forma más sencilla de hacer esta comprobación?
  - ¿Cuál?
  - ¿Y por qué es preferible hacerlo como lo hemos hecho nosotros?
- En el caso de DeathZone no hemos hecho esa comprobación. ¿Por qué no era necesario hacerlo?
- Hemos utilizado multitud de componentes para implementar comportamientos muy sencillos que podrían haberse incluído en un único componente. ¿Por qué es mejor hacerlo así?
- Hablemos de prefabs anidados:
  - Si cambias el color de p_Enemy, ¿Qué ocurre con los prefabs p_Squad_01, p_Squad_02... ?
  - Y si cambias el color de un enemigo en p_Squad_01, ¿Qué ocurre con p_Enemy?
- ¿Qué método has utilizado para implementar el movimiento de Player, p_Shot y p_Squads? ¿Por qué?
- ¿Qué método has utilizado para que el resto de GameObjects pudieran acceder al GameManager? ¿Cuántas veces en la ejecución de tu juego se busca el GameManager? ¿Es una forma correcta de implementarlo?