****

Módulo Profesional 08: Programación para dispositivos móviles y multimedia **M08UF1-Act2**

**2.5D Auto Runner**

**CICLO FORMATIVO DE GRADO SUPERIOR EN**

**Videojuegos y ocio digital**

**MODALIDAD PRESENCIAL**

**Nombre y apellidos del alumno**



**Introducción**





Programa un runner automático para móvil en 2.5D .

|  |
| --- |
| Objetivos |
| Aprender a utilizar el componente Character Controller.  Aprender a utilizar InputSystem para la lectura y la gestión de dispositivos de entrada habituales en móvil:   * Pantalla táctil * Acelerómetro * Giroscopio   Entender los conceptos básicos para mezclar elementos 2D con elementos 3D.  Implementar animaciones de sprites en |
| **Metodología de evaluación** |
| 15% Ejercicio 1.  15% Ejercicio 2.  15% Ejercicio 3.  15% Ejercicio 4.  15% Ejercicio 5.  25% Extras (Opcional, están descritos en los ejercicios 6 y 7)  Recuerda que todos los ejercicios obligatorios tienen que estar correctamente terminados para poder aprobar la actividad. |
| **Entrega** |
| A través del campus con nombre UF1-ACT2- NombreApellidos.zip incluyendo enunciado, proyecto Unity, APK y vídeo de la prueba en el dispositivo. |
| **Documentos de referencia** |
| Documentación de Unity |

**Situación**

Eres un programador en una pequeña empresa de videojuegos para móvil. La empresa está perdiendo fondos a espuertas, pero no queda claro si es por dirigirse a un público joven con juegos AAA que sólo funcionan en un 0.05% de los dispositivos disponibles en el mercado, o por tener como requerimiento básico el uso de ratón y teclado conectados al móvil.

Tu empresa ha decidido dar un giro radical a la política de desarrollo, y hacer un plataformas 2.5D de avance automático, utilizable tanto por gestos como por acelerómetro, con la esperanzar de alcanzar a un público mayor.

**Descripción de la Aplicación**

Se trata de una aplicación que tiene que funcionar para Android, en modo apaisado.

La aplicación presenta un juego de plataformas centrado en un personaje jugador (masculino o femenino) que dispone dos acciones:

* puñetazo
* gancho

La acción ocurre en un nivel infinito generado proceduralmente con un desarrollo mayormente lateral, compuesto por plataformas, y en el que hay una serie de peligros:

* enemigos terrestres
* enemigos voladores
* trampas de lava/pinchos
* (Opcional) enemigos gigantes (jefes)

El objetivo del juego es permanecer vivos cuanto más tiempo mejor, para obtener cuantos más puntos de experiencia mejor para (Opcional) gastarlos en habilidades como:

* (Opcional) Mayor resistencia
* (Opcional) Mayor velocidad
* (Opcional) Más fuerza en el puñetazo
* (Opcional) Más fuerza en el gancho
* (Opcional) Puños orbitales

**Controles de la Aplicación**

**Durante los Menús**

Durante los menús, la aplicación se controla a través de elementos de UI (botones, sliders…), cada uno de ellos con el funcionamiento táctil por defecto que proporciona Unity.

**Durante la Partida**

La aplicación se controla con gestos sobre la pantalla táctil. Reconoce los siguientes gestos:

* **Tap:** realiza un puñetazo, tanto desde el suelo como desde el aire
* **Slide Up:** realiza un gancho hacia arriba, sólo desde el suelo
* **Slide Down:** realiza un gancho hacia abajo, sólo desde el aire

También hay un botón de Pausa, utilizable con el funcionamiento táctil por defecto, que permite abrir un menú de pausa. Este menú de Pausa funciona como Menú, así que aplica el control descrito en la sección anterior.

**Sistemas de Juego**

Como parte imprescindible para aprobar el ejercicio, el juego estará montado sobre algunos sistemas reutilizables que tendrán que ser desarrollados.

* **HitHurt System:** este sistema propone una manera de controlar la interacción agresiva (dañar a un oponente) entre elementos

**Elementos del Juego**

Los elementos del juego a implementar son los siguientes, y se describirán en las sucesivas secciones:

* **Luchador:** se trata de un personaje, masculino o femenino, con la habilidad de pegar puñetazos y ganchos. Estos golpes, además de afectar a los enemigos, aceleran y frenan al jugador, con lo que resultan importantes para marcar el ritmo del juego, que tiene que ser frenético.
* **Enemigos:** se trata de personajes subterráneos (orcos, duendes) con capacidad de golpear al jugador. Hay de dos tipos:
  + **Terrestre**
  + **Volador**
* **Escenario:** se trata de un entorno en el subsuelo, tipo mazmorra, caverna, o templo en las profundidades. El escenario se genera proceduralmente bajo las condiciones establecidas para ello.

**Ejercicio1: Prepara los Sistemas de Juego**

Como parte imprescindible para aprobar el ejercicio, el juego estará montado sobre algunos sistemas reutilizables que tendrán que ser desarrollados.

* **HitHurt System:** este sistema propone una manera de controlar la interacción agresiva (dañar a un oponente) entre elementos, indiferentemente de que uno de estos elementos sea el jugador.
* **Camera System:** en este juego utilizaremos Cinemachine y una cámara sencilla en modo Frame Transposer.

**Ejercicio 1.1: HitHurt System**

Implementa el **HitHurtSystem**. Este sistema se basa en dos componentes:

* **HitCollider**
  + **HitCollider** tiene los siguientes parámetros:
    - un UnityEvent ***onHitDelivered*** que reciba por parámetro un **HitCollider** y un **HurtCollider**
    - una lista de strings ***hittableTags***
  + **HitCollider** no tiene métodos propios
  + **HitCollider** reacciona a los eventos **OnTriggerEnter()** y a un **OnCollisionEnter()** de la siguiente manera:
    - comprueba si el tag del objeto colisionado esta en la lista de hittableTags
    - en ese caso
      * obtiene el componente **HurtCollider** del objeto colisionado y (si existe) llama a su método **NotifyHit()** pasando como parámetro al propio **HitCollider**
      * invoca el evento ***onHitDelivered***, pasando como parámetro al propio **HitCollider** y el **HurtCollider** Afectado
* **HurtCollider**
  + **HurtCollider** tiene los siguientes parámetros:
    - un UnityEvent ***onHitReceived*** público que reciba por parámetro un **HitCollider** y un **HurtCollider**
  + **HurtCollider** tiene un método **NotifyHit()** que recibe como parámetro el **HitCollider** que ha hecho el daño
    - Este método invoca al evento ***onHitReceived***, utilizando como parámetro el **HitCollider** especificad y al propio **HurtCollider**

**Ejercicio 1.2: Prueba del HitHurtSystem**

Para comprobar el funcionamiento del **HitHurtSystem**, prepara una escena con:

* un cubo
  + con los siguientes componentes: BoxCollider, RigidBody en modo kinematic, **HurtCollider**
  + con tag “Breakable”
  + en el editor, como respuesta al evento ***onHitReceived*** pon el GameObject 🡪 SetActive 🡪 false .
* una esfera
  + colocada dos metros por encima del cubo
  + con los siguientes componentes: SphereCollider, Rigidbody **no** kinematic, **HitCollider**
  + **el HitCollider** ha de tener el tag “Breakable” como parte de sus ***hittableTags***

Si todo está bien montado, la esfera debería desactivar el cubo en cuanto colisione contra éste.

**Ejercicio 1.3: CameraSystem**

Para este juego, importa el Package Cinemachine y utiliza una cámara en modo Frame Transposer. Asigna la esfera a la cámara.

Comprueba que la cámara sigue a la esfera preparada para la prueba.

**Ejercicio 1:** Incluye a continuación un enlace a un vídeo en el que demuestres el funcionamiento correcto de esta prueba.

|  |
| --- |
|  |

**Ejercicio 2: Implementa al Luchador**

El luchador es un GameObject que contiene principalmente el componente PlayerController, que tendrás que implementar. Este componente controla:

* el movimiento del jugador (ejercicio 2.1)
* el golpeado del jugador (ejercicio 2.2)
* la muerte del jugador (ejercicio 2.3)

Se da la circunstancia de que el movimiento y el golpeado del jugador están muy unidos entre ambos, así que, si lo ves necesario, realiza los ejercicios 2.1 y 2.2 a la vez.

**Ejercicio 2.1: Movimiento**

Implementa el movimiento del jugador:

* Crea un GameObject de nombre Player
* Asigna el Player a la Cinemachine Camera del CameraSystem
* Añade un componente CharacterController
* Crea un nuevo componente PlayerController con las siguientes características:
  + El PlayerController avanza hacia adelante (+Z) y ocasionalmente también se mueve en vertical (+Y / -Y)
  + Avance hacia delante (+Z)
    - El PlayerController avanza en dirección Z a una velocidad constante (Vfwd)
    - Dicha velocidad puede ser acelerada en incrementos fijos (dV)
      * Esta velocidad se incrementa hasta un máximo
    - También puede ser decrementada exactamente a la mitad
      * No hay mínimo de velocidad al decrementar
    - Existe una velocidad mínima (Vmin), tal que si V < VMin entonces el PlayerController acelerará hasta Vmin
  + Movimiento vertical
    - El PlayerController mantiene una velocidad vertical (Vvert)
    - Dicha velocidad se ve afectada por una aceleración gravitatoria (G)
    - Si PlayerController toca el suelo, Vvert tiene que reducirse a 0
    - El PlayerController permite modificar la velocidad vertical instantaneamente

**Ejercicio 2.2: Puñetazo**

Implementa la capacidad de dar puñetazos:

* Añade un GameObject hijo al Player llamado HitColliderPunch con los siguientes componentes:
  + Un collider (box, capsule) en modo trigger
  + Un HitCollider capaz de golpear a tags de tipo “Enemy”
  + Un Rigidbody en modo cinemático (para que funcione correctamente el trigger)
* Añade la funcionalidad del puñetazo al CharacterController
  + Se puede realizar cuando el Player está tanto en el suelo como en el aire
  + Inicialmente HitColliderPunch debe estar desactivado
  + Cuando el jugador pulse la tecla de puñetazo
    - se debe activar el HitColliderPunch
      * se desactivará al cabo de 0.5 segundos
    - se debe aplicar un incremento fijo a la velocidad de avance

**Ejercicio 2.3: Gancho hacia arriba (Uppercut)**

Implementa la capacidad de hacer ganchos hacia arriba:

* Añade un GameObject hijo al Player llamado HitColliderUppercut, similar a la del Punch pero apuntando hacia arriba
* Añade la funcionalidad del gancho hacia arriba al CharacterController
  + Sólo se puede realizar cuando el Player está en el suelo
  + Inicialmente HitColliderUppercut debe estar desactivado
  + Cuando el jugador pulse la tecla de puñetazo
    - se debe activar el HitColliderUppercut
      * se desactivará al cabo de 0.5 segundos
    - se debe aplicar un decremento a la mitad de la velocidad de avance
    - se modifica instantáneamente la velocidad vertical a una velocidad positiva de salto (Vjmp)

**Ejercicio 2.4: Gancho hacia abajo (Smash)**

Implementa la capacidad de hacer ganchos hacia abajo:

* Añade un GameObject hijo al Player llamado HitColliderSmash, similar a la del Punch pero apuntando hacia abajo
* Añade la funcionalidad del gancho hacia abajo al CharacterController
  + Sólo se puede realizar cuando el Player está en el aire
  + Inicialmente HitColliderSmash debe estar desactivado
  + Cuando el jugador pulse la tecla de puñetazo
    - se debe activar el HitColliderSmash
      * se desactivará al cabo de 0.5 segundos
    - se debe aplicar un decremento a la mitad de la velocidad de avance
    - se modifica instantáneamente la velocidad vertical a una velocidad positiva de smash (Vsmh)

**Ejercicio 2.5: Muerte**

Implementa la capacidad del jugador de morir:

* Añade un HurtCollider al Player
* Responde al evento onHitReceived del HurtCollider de la manera siguiente:
  + Reduce el numero de vidas restantes (inicialmente es tres)
  + Cambia la velocidad de avance a un valor negativo (-Vhitfwd)
  + Cambia la velocidad vertical para que realice un pequeño salto (Vhitjmp)
  + Si el jugador tiene más de 0 vidas
    - Desactiva los controles del jugador durante 1 segundo
    - Proporciona invulnerabilidad al jugador durante 3 segundos
  + Si el jugador tiene 0 vidas
    - Detén la aceleración de avance
    - Proporciona invulnerabilidad completa
    - Pon animación de muerte
    - Reinicia la escena al cabo de 5 segundos

**Ejercicio 2:** Incluye a continuación un enlace a un vídeo en el que demuestres el funcionamiento correcto de esta prueba.

|  |
| --- |
|  |

**Ejercicio 3: Implementa a los Enemigos**

Los enemigos son GameObjects que describen una ruta sencilla y que, si el jugador no pone remedio, le golpearán y reducirán su cantidad de vida.

Hay dos tipos de enemigos a implementar:

* Enemigos Terrestres
* Enemigos Voladores

**Ejercicio 3.1: Enemigo Terrestre**

El Enemigo Terrestre es similar al jugador, así que puedes aprovechar código de éste.

**Ejercicio 3.1.1: Movimiento**

Implementa el movimiento del Enemigo Terrestre:

* Crea un GameObject de nombre GroundEnemy
* Añade un componente CharacterController
* Crea un nuevo componente GroundEnemyController con las siguientes características:
  + El GroundEnemyController avanza hacia la izquierda (-Z) a una velocidad constante
  + Si durante este avance el GroundEnemyController choca con el jugador o con una pared, se detiene
  + El GroundEnemyController se ve afectado por la gravedad (G)

**Ejercicio 3.1.2: Ataque**

Implementa el ataque del Enemigo Terrestre:

* Añade un GameObject hijo al GroundEnemy llamado HitCollider, apuntando hacia delante
* Añade la funcionalidad del ataque:
  + Inicialmente HitCollider debe estar desactivado
  + Si el GroundEnemy tiene al jugador en rango:
    - Se detiene
    - Pone una animación de levantar el arma
      * Al cabo de dos segundos:
        + pone una animación de bajar el arma
        + activa el HitCollider
        + hace un poco de dash hacia el jugador

Sin atravesar ni al jugador ni a otros enemigos

* + - Después de pegar, realiza de nuevo el proceso (vuelve a pegar al cabo de 2 segundos)
* ¡Importante!
  + El HitCollider no debe dañar a otros enemigos, sólo al jugador
  + El rango para detenerse y atacar al jugador tiene que permitirle atacarlo incluso desde detrás de otro GroundEnemy
  + El tamaño del HitCollider también tiene que permitir atacar al jugador incluso desde detrás de otro GroundEnemy

**Ejercicio 3.1.3: Daño y Muerte**

Implementa el daño recibido por el enemigo, y la eventual muerte de este:

* Añade un HurtCollider al GroundEnemy
* Responde al evento onHitReceived del HurtCollider de la manera siguiente:
  + Reduce el número de vidas restantes (inicialmente es tres)
  + Muestra un efecto de partículas de recibir golpe
  + Si estás atacando y recibes dos puñetazos seguidos o un uppercut
    - Pone una animación de recibir daño
    - Reinicia la animación de levantar arma y renueva la cuenta de dos segundos
  + Si pierdes todas las vidas
    - Desactiva el HitCollider
    - Desactiva el HurtCollider
    - Pon una animación de muerte
    - Realiza un movimiento de salir despedido hacia arriba

**Ejercicio 3.2: Enemigo Volador**

El Enemigo Volador realiza ataques en picado, y tiene que ser abatido con el ataque adecuado en el momento preciso.

**Ejercicio 3.2.1: Movimiento**

Implementa el movimiento del Enemigo Volador:

* Crea un GameObject de nombre AirEnemy
* Añade un componente CharacterController
* Crea un nuevo componente AirEnemyController con las siguientes características:
  + El AirEnemyController avanza hacia la izquierda (-Z) a una velocidad constante
  + Si durante este avance el AirEnemyController choca contra cualquier cosa (un enemigo, una pared) **huye**
    - Huir quiere decir dar media vuelta y avanzar en +Z, +Y sin chocar contra nada
  + El EnemyController **no** se ve afectado por la gravedad

**Ejercicio 3.2.2: Ataque**

Implementa el ataque del Enemigo Volador:

* Añade un GameObject hijo al AirEnemy llamado HitCollider
  + Debe tener forma esférica y cubrir por completo al enemigo
* Añade la funcionalidad del ataque:
  + Inicialmente HitCollider debe estar desactivado
  + Si el EnemyController tiene al jugador en rango:
    - Se lanza en línea recta hacia éste
    - Durante un 5 segundos, intenta alcanzarlo
      * No huye aunque choque contra otros elementos
    - Si consigue dañar al jugador, **huye**
* ¡Importante!
  + El HitCollider no debe dañar a otros enemigos, sólo al jugador
  + El rango de ataque será variable, de manera que el ángulo desde el que llega el ataque sea diferente de un AirEnemy a otro

**Ejercicio 3.2.3: Daño y Muerte**

Implementa el daño recibido por el enemigo, y la eventual muerte de este. El enemigo volador solo aguanta un golpe:

* Añade un HurtCollider al GroundEnemy
* Responde al evento onHitReceived del HurtCollider de la manera siguiente:
  + Muestra un efecto de partículas de recibir golpe
  + Desactiva el HitCollider
  + Desactiva el HurtCollider
  + Pon una animación de muerte
  + Realiza un movimiento de salir despedido hacia arriba

**Ejercicio 3:** Incluye a continuación un enlace a un vídeo en el que demuestres el funcionamiento correcto de esta prueba.

|  |
| --- |
|  |

**Ejercicio 4: Implementa el Escenario**

El escenario se generará de manera procedural. Esta generación tendrá dos vertientes:

* La generación del escenario
* La generación de los enemigos

La gestión de esta generación se hará a través del componente ScenarioGenerator, que tendrás que preparar de la siguiente manera:

* Crea una clase ScenarioGenerator que derive de MonoBehaviour
* Implementa acceso estático (“singleton”) a dicha clase

La implementación de esta clase está detallada en las siguientes secciones.

**Ejercicio 4.1: Generación del Escenario**

La Generación del Escenario se hará en base a Piezas de Escenario. Estas piezas de escenario tendrán una estructura muy concreta que permitirá:

* Saber cuándo ha terminado el escenario y así generar un nuevo escenario
* Saber en que posiciones puede o no aparecer un enemigo
* Saber dónde hay que generar la siguiente pieza

**Ejercicio 4.1.1: Antes de Empezar**

En esta sección explicaremos cómo son las Piezas de Escenario, y que estructura tienen que hace posible que las utilicemos para generar el escenario.

**Tamaño**

Las Piezas de Escenario tienen el tamaño de dos o tres pantallas horizontales. Generalmente, tienen una sola pantalla de altura, pero se pueden tener más si se busca una verticalidad.

**Detección de Salida**

Todas las Piezas de Escenario disponen de un Trigger con el Tag “ScenarioPieceEnd” al final de este. El tamaño de este trigger tiene que ser inevitable, para que, de ninguna manera, el jugador pueda evitar tocarlo en su avance.

Importante: una vez tocado **por el jugador** este objeto se desactivará, para impedir que se toque más de una vez. Esto es crucial de cara a gestionar correctamente la aparición/desaparición de las Piezas del escenario.

**Punto de Siguiente Pieza**

Todas las Piezas de Escenario dispondrán de un objeto que servirá como posición para instanciar la siguiente pieza. Para que sea fácil de encontrar, este objeto dispondrá de un Script NextPiecePosition que podrá ser buscado fácilmente con un GetComponent().

**Ejercicio 4.1.2: Funcionamiento del Generador**

Prepara un Componente ScenarioGenerator con las siguientes características:

* Tendrá una lista de referencias a Piezas de Escenario

Funcionará de la siguiente manera:

* Dispondrá de una funcionalidad para generar una nueva pieza al final de la última que se generó
  + Para ello necesitará una referencia a la última Pieza generada (si es Null quiere decir que se va a generar la primera Pieza)
  + Utilizará un GetComponentInChildren<NextPiecePosition>() para encontrar el punto en el que generar la siguiente pieza
* Inicialmente, se llamará tres veces a esta función, para disponer de suficiente mundo para recorrer
* Cada vez que el jugador toque el trigger con el tag “ScenarioPieceEnd”, deberá notificarse al Generador, que
  + incrementará la cantidad de triggers tocados en 1
  + a partir de 2 triggers tocados, generará una nueva pieza y destruirá la más antigua
    - Esto implica que, una vez superados estos dos triggers, siempre habrá 5 piezas en el escenario

**Ejercicio 4:** Incluye a continuación un enlace a un vídeo en el que demuestres el funcionamiento correcto de esta prueba.

|  |
| --- |
|  |

**Ejercicio 5: Condiciones de Final de Partida y Reinicio**

Este autorunner es un juego sin fin, en el que la única manera de que se termine es que el jugador muera. En un mundo ideal, con los puntos conseguidos compraríamos mejoras que nos ayudasen a llegar más lejos, pero no es lo que vamos a hacer.

* Implementa el final de Partida
  + El jugador inicia la partida con 3 vidas
  + Cada vez que recibe un golpe, pierde una de estas
  + Cuando pierde las 3 vidas
    - Se ralentiza el tiempo para conseguir un efecto dramático
    - Deja de colisionar con el escenario
    - Sale despedido en vertical y hacia atrás
    - Aparece un cartel en pantalla indicando el final de la partida
    - Junto con el cartel, aparecen dos botones para
      * Reiniciar
      * Salir

**Ejercicio 5:** Incluye a continuación un enlace a un vídeo en el que demuestres el funcionamiento correcto de esta prueba.

|  |
| --- |
|  |

**Ejercicio 6: Gráficos y Sonido (Opcional)**

Puedes implementar gráficos tanto para el escenario como para los elementos del juego y el feedback.

* **Escenario:**
  + Puedes obtener elementos gráficos 3D de la AssetStore de Unity o de otros lugares
    - TurboSquid
  + Vigila que el poligonaje de los gráficos no sea exagerado
* **Elementos de juego**
  + Utiliza personajes extraídos de Mixamo o de la AssetStore
* **Feedback**
  + En la AssetStore de Unity encontrarás efectos especiales de partículas
  + Junto con dichos efectos, también puedes utilizar sonidos que obtengas en sitios como <https://freesound.org>
* **Música**
  + Existen varios sitios desde los que descargar música. Sobretodo, asegúrate de que esta música es CC0 o que tienes permiso para utilizarla en proyectos personales.

**Ejercicio 6:** Incluye a continuación un enlace a un vídeo en el que demuestres el funcionamiento correcto de esta prueba.

|  |
| --- |
|  |

**Ejercicio 7: Menús (Opcional)**

Integra un sistema de menús cómo el del ejercicio que (probablemente) hayas realizado para M17UF3.

**Ejercicio 7:** Incluye a continuación un enlace a un vídeo en el que demuestres el funcionamiento correcto de esta prueba.

|  |
| --- |
|  |

**Extra (1/3 cuenta como actitud): Características Opcionales**

Cada una de las siguientes líneas se considera una característica extra. Implementa por lo menos dos de ellas:

* **Característica 1:** Gráficos para los personajes y elementos de juego
* **Característica 2:** Gráficos para los escenarios
* **Característica 3:** Sonidos
* **Característica 4:** Música
* **Característica 5:** Efectos especiales de Feedback
* **Característica 6:** Menús de juego

**Opcionales:** Incluye a continuación un enlace a un vídeo en el que demuestres el funcionamiento.

|  |
| --- |
|  |