****

Módulo Profesional 17: Programación de Videojuegos 3D

**M17UF4 – Act1**

**3D Platformer Mejorado**

**CICLO FORMATIVO DE GRADO SUPERIOR EN**

**Videojuegos y ocio digital**

**MODALIDAD PRESENCIAL**

**Nombre y apellidos del alumno**



**Introducción**

Pantalla de juego de video

Descripción generada automáticamente con confianza media

Evoluciona el juego de plataformas que hemos creado en la anterior UF con las siguientes mejoras:

* Gráficos importados de Mixamo
* Sistema de armamento escalable
* IA escalable compuesta por sentidos, estados objeto, árbol de toma de decisiones

|  |
| --- |
| Objetivos |
| * Integración de gráficos de aspecto profesional * Preparación de sistemas basados en la derivación y composición a favor de la mantenibilidad y la escalabilidad * Preparación de una IA avanzada basada en el patrón State |
| **Metodología de evaluación** |
| 9% Ejercicio 1.  9% Ejercicio 2.  9% Ejercicio 3.  9% Ejercicio 4.  9% Ejercicio 5.  9% Ejercicio 6.  9% Ejercicio 7.  9% Ejercicio 8.  9% Ejercicio 9.  9% Ejercicio 10.  9% Ejercicio 11.  Recuerda que todos los ejercicios obligatorios tienen que estar correctamente terminados para poder aprobar la actividad. |
| **Entrega** |
| A través del campus con nombre M17UF4-ACT1-NombreApellidos.zip incluyendo enunciado, proyecto Unity, Build y vídeo final.  Importante:   * Si falta cualquier entregable la nota es 0 * Por cada día de retraso en la entrega se resta un punto |
| **Documentos de referencia** |
| * Documentación de Unity |

**Situación**

El Surfer Amrio Bruce Wolrd está publicado y ha sido un nuevo éxito de ventas; según tu jefe esto se debe a una larga tradición de esfuerzo y trabajo bien hecho; según internet, el nombre del juego era demasiado meme como para no ser tremendo éxito.

Incentivado por el olor a billetes de 500 euros, tu jefe decide que hay que trabajar aún mejor, por lo que te propone (obliga a) hacer una nueva versión del juego, con varias mejoras:

* gráficos que no estén hechos con esferas de Unity
* un sistema de mantenible y escalable de armamento
* una IA capaz de no chocarse con todas las paredes, y creada bajo estándares de escalabilidad, mantenibilidad y flexibilidad sin parangón

Tus compañeros traen las sales y consiguen reanimarte: pensar que tu jefe te ha pedido que trabajes con estándares de programación más elevados ha sido un shock demasiado grande y te has desmayado. Luego lo piensas y te das cuenta de que tu jefe, que no es tonto, sabe que a la larga le saldrá más barato y lo comprendes todo.

El mundo vuelve a girar, todo vuelve a tener sentido y te pones manos a la obra, que sigues siendo el único programador.

**Descripción del Juego**

Se trata del mismo juego que en M17UF3Act3, pero con las mejoras antes enumeradas.

|  |
| --- |
| **Parte 1**  **Mejora del Jugador y los Enemigos** |

**Ejercicio1: Mejora Gráfica del Personaje**

Visita Mixamo.com y elige un personaje adecuado al estilo de juego que te gustaría tener. Piensa que todo el sistema de ataque se hará con armas de fuego, por lo que ciertos personajes (caballeros medievales, etc.) pueden no encajar.

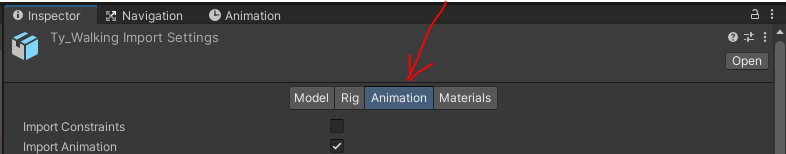
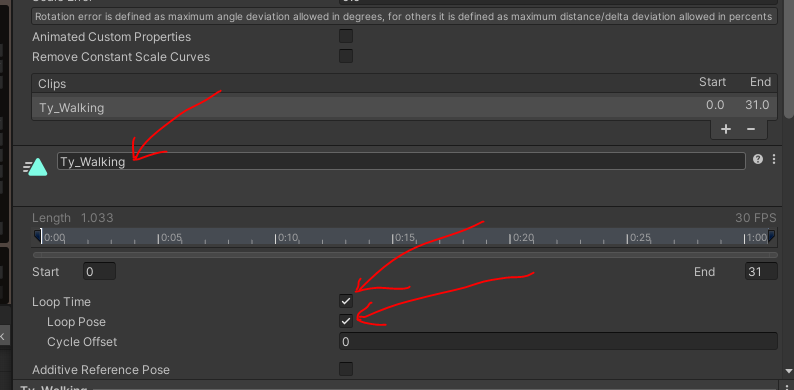
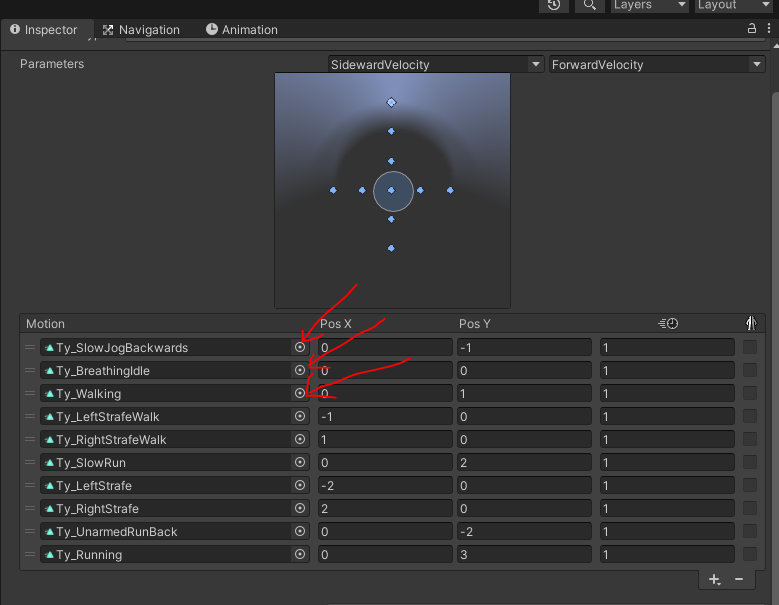
* **Descarga la Pose-T**
  + Tal cual entres a [mixamo.com](https://www.mixamo.com/) , y una vez te hayas dado de alta, verás que en esta página hay dos secciones: Personajes (Characters) y Animaciones (Animations)  
    Captura de pantalla de un celular con texto e imágenes de una persona

    Descripción generada automáticamente con confianza media
  + Entra a la sección de Characters y elige uno sin darle a ninguna animación
    - Si seleccionas una animación, perderas la Pose-T
    - Si ya le has dado a alguna animación, recarga la página (normalmente F5) para recuperar la Pose-T
  + Descarga la Pose-T con el botón de Download, utiliza las opciones que te vienen por defecto
* **Descarga las animaciones**
  + Necesitamos descargar las siguientes animaciones:
    - Caminar hacia adelante (walk)
      * ¡Importante! Todas las animaciones de caminar y correr tienen que ser exportadas con el check In Place marcado, para que no se desplacen del punto en el que estará el GameObject  
        Interfaz de usuario gráfica, Aplicación

        Descripción generada automáticamente
    - Caminar hacia atrás (walk backwards)
    - Estático (idle)
    - Caminar hacia la derecha (strafe right)
    - Caminar hacia la izquierda (strafe left)
    - Correr hacia adelante (run)
    - Correr hacia atrás (run backwards)
    - Correr hacia la derecha (strafe right run)
    - Correr hacia la izquierda (strafe left run)
  + Utiliza las opciones de descarga que te vienen por defecto
    - Es posible que pienses que las animaciones se podrían descargar sin Skin para que ocupen menos; **la teoría** es que esto es cierto, pero en el pasado he tenido problemas con animaciones sin Skin, por lo que para asegurarnos descarga todas las animaciones con Skin
* **Importa la Pose-T y las animaciones**
  + Prepara una carpeta Skins en Assets
  + Dentro de Skins, prepara una carpeta con el nombre de tu personaje (en mi ejemplo se llama Ty)
  + Prepara las siguientes carpetas dentro de la carpeta de tu personaje:  
    Texto

    Descripción generada automáticamente
  + Pon la Pose-T en models
    - Renombra la Pose-T al nombre del personaje (Ty)
  + Pon las animaciones en Animations
    - Renombra las animaciones al nombre del personaje + el nombre de la animación, como en este ejemplo:  
      Texto

      Descripción generada automáticamente con confianza baja
    - **¡No te saltes este paso!** El renombrado es importante para para no confundirnos con el enemigo, que tendrá exactamente las mismas animaciones
* **Extrae las texturas**
  + Arrastra el modelo de tu personaje a la jerarquía
  + Podrás ver que el personaje aparece blanco, como si no tuviera un material puesto
  + Extrae las texturas
    - Selecciona el modelo de tu personaje
    - En el inspector, selecciona la pestaña Materials  
      Interfaz de usuario gráfica, Aplicación

      Descripción generada automáticamente
    - Utiliza el botón **Extract Textures** que encontrarás un poco más abajo
      * Extrae las texturas en la carpeta Assets/Skins/<Tu Personaje>/Textures que habrás creado previamente
    - Tras extraer las texturas, éstas se pondrán automáticamente en tu personaje
    - **¡Recuerda cambiar el tipo de textura del mapa de normales!**
* **Extrae los materiales**
  + Si seleccionas a tu personaje en la escena, podrás ver que el Material que utiliza su MeshRenderer muestra sus propiedades en gris y no es editable
    - Esto se debe a que en estos momentos tu personaje está utilizando el material que hay dentro del modelo
  + Vuelve a seleccionar el modelo
  + Vuelve a seleccionar la pestaña Materials
  + Utiliza el botón **Extract Materials**
    - Extrae los materiales en la carpeta Assets/Skins/<Tu Personaje>/Textures que habrás creado previamente
  + Tras extraer los materiales, estos se asignarán automáticamente a tu personaje
    - Ahora puedes editar los materiales
  + **¡Importante!**
    - Ocasionalmente los materiales no se asignan correctamente, lo cual normalmente tiene que ver con el render pipeline
    - Si te ocurre esto, tendrás que cambiar en los materiales al shader correcto y probablemente también reasignar las texturas
    - Por suerte, no ocurre muy a menudo
* **Prepara el Animator Controller**
  + Renombra los AnimationClips
    - Para facilitarte la vida al preparar el AnimatorController, es conveniente que los AnimationClips tengan nombres fáciles
      * ¡Ojo! No es lo mismo el AnimationClip que el FBX que ya renombraste, por lo que necesitarás renombrar específicamente el AnimationClip
    - Para renombrar los AnimationClips
      * Selecciona el FBX de la animación
      * En el importador del FBX, escoge la pestaña Animation  
        
      * Localiza la sección de los AnimationClips
      * Utiliza el campo señalado para renombrar la animación a algo como “<NombreDelPersonaje>\_<Animación>” (en mi caso “Ty” y “Walking”)  
        
      * Si la animación es cíclica, aprovecha para marcar los checks de LoopTime y LoopPose
  + Ahora sí, prepara el AnimatorController
    - Duplica el AnimatorController del Character, creado en el ejercicio anterior
    - Ponle un nombre como <NombreDelPersonaje>AnimatorController
    - Ponlo en la carpeta Assets/Skins/<Tu Personaje>/AnimatorControllers
    - Modifica el AnimatorController para que utilice las animaciones importadas desde Mixamo
      * Gracias a que los clips están renombrados, ahora es muy fácil localizar los clips a través del buscador  
        
* **Comprueba que el personaje funciona**
  + Duplica tu Player
  + En los Visuals, pon el personaje que has creado
  + Añade un componente Animator y asigna el nuevo AnimatorController

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

|  |
| --- |
|  |

**Ejercicio 2: Mejora del Enemigo: Búsqueda de Caminos**

En este ejercicio aprenderemos a utilizar el NavMeshing y los NavMeshAgents para conseguir que los enemigos sean capaces de encontrar caminos correctos cuando persigan al jugador o hagan diferentes actividades.

**Prepara la Malla de Navegación**

En esta parte del ejercicio prepararemos la escena para que el NavMeshing sea capaz de encontrar caminos que eviten paredes y desniveles entre dos puntos.

* **Importa el Package del NavMeshing**
  + Abre el Package Manager
  + Descarga e instala el package AI Navigation
* **Prepara la escena para el NavMeshing**
  + El sistema de NavMeshing se basa en la existencia de una o más mallas de navegación, que pueden ser utilizadas para realizar una búsqueda de caminos
  + Existen diferentes maneras de crearlas, pero una de las más sencillas es la siguiente
    - Crea un nuevo GameObject vacío, llámalo Scenario
    - Cuelga todos los elementos que sean parte de tu escenario (aquellos sobre los cuales puedan navegar los enemigos) como hijos de Scenario
    - Añade el componente NavMeshSurface a Scenario
      * En NavMeshSurface 🡪 Object Collection 🡪 Collect Objects elige la opción Current Object Hierarchy
      * Utiliza el botón Bake para generar la malla de navegación
* **Comprueba la malla de navegación**
  + En estos momentos, ya deberías poder ver en la ventana de escena la malla de navegación, que tendrá un aspecto similar a este:  
    Imagen que contiene sobres, estacionaria

    Descripción generada automáticamente
  + Si no puedes ver la malla, prueba con los siguientes pasos:
    - Asegúrate de tener los Gizmos activados  
      Interfaz de usuario gráfica, Aplicación

      Descripción generada automáticamente
    - En el overlay de AI Navigation, asegúrate de tener Show NavMesh marcado:  
      Gráfico

      Descripción generada automáticamente
      * Si no te aparece el overlay de AI Navigation, arrastra el icono del susodicho overlay a la escena:  
        Gráfico

        Descripción generada automáticamente

**Mejora el Enemigo**

En esta parte del ejercicio modificaremos al enemigo, tanto el prefabricado como el script Enemy, para que puedan utilizar la malla de navegación. Utilizaremos el componente NavMeshAgent para automatizar tanto la búsqueda de caminos como el recorrido de éstos.

* **Añade el Componente NavMeshAgent**
  + Instancia un enemigo
  + Añade el componente NavMeshAgent
    - Los valores por defecto son correctos, pero puedes aprovechar para ajustar un poco parámetros como la velocidad, la aceleración, o la velocidad de giro
  + Aplica los cambios al Prefabricado
* **Crea un nuevo script Enemy para que utilice el NavMeshAgent**
  + Duplica el script Enemy, llama EnemyWithAI al script duplicado
  + En el prefabricado del enemigo, quita el componente Enemy y añadel el componente EnemyWithAI
  + Edita EnemyWithAI y realiza los siguientes pasos:
    - Declara una variable de tipo NavMeshAgent, llámala navMeshAgent
    - Obtén una referencia al componente NavMeshAgent utilizando GetComponent
    - En Update()
      * Dentro del caso de Wandering
        + Retira las líneas que llamaban a

UpdatePlaneMovement()

UpdateVerticalMovement()

UpdateOrientation()

* + - * + Sustitúyelas por una llamada a navMeshAgent.SetDestination en la que le pases la posición de wandering decidida en ChooseWanderPosition()
      * Dentro del caso de Chasing:
        + Retira las misma líneas que en el caso de Wandering
        + Sustitúyelas por una llamada a navMeshAgent.SetDestination en la que le pases la posición del target detectado

**Comprueba el Funcionamiento**

Si has seguido correctamente el ejercicio, en estos momentos el enemigo será capaz de buscar caminos de manera correcta para perseguirte. Pon varios enemigos, y comprueba que estos sean capaces de detectar al jugador y perseguirle incluso por áreas de acceso complicado (puentes, habitaciones, etc.)

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

|  |
| --- |
|  |

**Ejercicio 3: Mejora del Enemigo: Modelo y Animaciones de Mixamo**

En este ejercicio vamos a mejorar los visuales de nuestro enemigo utilizando otro modelo de Mixamo.

Aunque este ejercicio puede parecer complicado, no lo será tanto, ya que aprovecharemos el código del Player para ahorrar gran cantidad de tiempo y trabajo, y, aún mejor, para no tener código duplicado.

**Descarga e Importa Modelo y Animaciones para el Enemigo**

* Utiliza las instrucciones en el Ejercicio 1 para descargar, renombrar e importar un modelo en pose-T y las animaciones para un enemigo desde Mixamo.
* ¡Ojo! En lugar de hacer copy-paste del AnimatorController, utiliza el siguiente método:
  + En Assets/Skins/<Tu enemigo>/AnimatorController crea un asset de tipo AnimatorOverrideController; llamalo <TuEnemigo>\_AnimatorOverride
    - Establece como Controller el de tu personaje jugador (en el ejemplo Ty\_AnimatorController)
    - Asigna a los slots las animaciones de tu enemigo (en el ejemplo Yaya)
    - Debería quedar algo así:  
      Interfaz de usuario gráfica, Texto

      Descripción generada automáticamente
    - Fíjate que en el ejemplo hay varias animaciones que no están asignadas
      * Esto se debe a que en mi proyecto el enemigo no utiliza todas las animaciones que se importaron

**Extrae el Código Común a una Nueva Clase Entity**

Cuando se tiene un código que ya funciona en una **Clase Original**, y se pretende utilizar el mismo código en una **Clase Nueva**, una manera interesante de hacer esto sin repetir código es la siguiente:

* Extraer dicho código en la **Clase Original** a una nueva **Clase Base**
* Modificar la **Clase Original** para que ahora derive de la **Clase Base**
* Comprobar que la **Clase Original** sigue funcionando igual que antes
  + Esto nos da la seguridad de que lo que hemos hecho hasta ahora es correcto
* Crear la **Clase Nueva** derivando de **Clase Base**
* Comprobar que todo funciona
  + Si algo falla, claramente **no es culpa de la Clase Base**, puesto que ya hemos comprobado que la **Clase Original**, que ahora deriva de **Clase Base**, funciona correctamente

Esto es exactamente lo que haremos ahora: extraer la parte del código de animación de la clase PlayerController a la clase Entity, y mejorar la clase Enemy para que derive de Entity.

* **Extrae la clase Entity**
  + Crea una nueva clase abstracta Entity que derive de MonoBehaviour
  + Prepara los siguientes métodos abstractos
    - GetCurrentVerticalSpeed()
    - GetJumpSpeed()
    - IsRunning()
    - IsGrounded()
    - GetNormalizedMoveXZLocal()
    - Los métodos anteriores tendrán que ser implementadas en las clases derivadas para que se actualice correctamente la animación
  + Lleva la parte de Update() que gestiona los parámetros de animación al Update de la clase Entity
    - Utiliza los métodos anteriores para actualizar correctamente los parámetros del Animator
  + Prepara la clase Entity para que no haya conflicto entre el Update() de Entity y el Update() de la derivada
    - Cuando una clase y su derivada tienen funciones propias de Unity (Awake(), Update(), OnTriggerEnter()…), éste llamará a las funciones externas (las de la clase derivada)
      * Para conseguir que ambas sean llamadas:
        + Crea un método abstracto ChildUpdate() en Entity
        + Llama a ChildUpdate() desde el Update() de Entity
        + Cambia el nombre del método Update() de PlayerController a ChildUpdate

¡Añade también el keyword override!

* + - * Si fuese necesario, repite este mismo método para otras llamadas de Unity (Awake, etc.)
* **Comprueba que PlayerController sigue funcionando**
  + Si has hecho todo bien, podrás comprobar que PlayerController sigue funcionando como hasta ahora.
  + Arregla y ajusta lo que haga falta con tal de que así sea.
* **Deriva EnemyWithAI de Entity**
  + Haz que la clase EnemyWithAI derive de Entity en lugar de MonoBehaviour
  + Implementa todas las funciones abstractas
    - IsRunning() – devuelve false, como si el enemigo nunca corriera
    - IsGrounded() – devuelve siempre true, ya que (al menos por ahora) el enemigo nunca despegará los pies del suelo
    - GetNormalizedMoveXZLocal()
      * Para devolver un valor correcto, necesitarás almacenar una variable lastPosition que en cada LateUpdate() coja la posición actual del personaje
      * Con esto puedes devolver la (posición actual – lastPosition).normalized
    - GetCurrentVerticalSpeed() y GetJumpSpeed() – en ambos casos puedes devolver cero
* **Prepara el Prefabricado del Enemigo**
  + Pon el personaje en Pose-T como hijo de los Visuals del enemigo
  + Añadele un Animator
  + Asigna a este Animator el AnimatorOverrideController que has creado antes
  + Si es necesario, ajusta también el tamaño y la posición del NavMeshAgent
* **Comprueba que el enemigo funciona correctamente**
  + Si lo has hecho bien, el enemigo se comportará igual que antes, pero esta vez mostrará el modelo 3D y tendrá las mismas animaciones que el jugador  
    Imagen que contiene exterior, agua, hombre, sostener

    Descripción generada automáticamente

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

|  |
| --- |
|  |

|  |
| --- |
| **Parte 2**  **Sistema de Armamento Escalable** |

**Introducción: Sistema de Armamento Escalable**

Durante el desarrollo de videojuegos (y de cualquier otro software) se tiende a priorizar la funcionalidad, es decir: que el software haga aquello para lo que fue diseñado.

Esto, que a priori parece lógico, con el tiempo se demuestra que no lo es tanto: al priorizar únicamente la funcionalidad se tiende a tomar decisiones de diseño a corto plazo. Con la complejidad que alcanzan actualmente los sistemas, estas decisiones suelen ir en contra de características muy deseables en un proyecto:

* **Mantenibilidad:** un proyecto que se alarga en el tiempo necesitará que sea fácil realizar cambios y mejoras de todo tipo incluso aunque los desarrolladores originales ya no estén en la empresa
* **Legibilidad:** es parte de la mantenibilidad, e implica que nuestro código tiene que poder leerse fácilmente (CleanCode) y también poderse entender fácilmente cuál fue la intención detrás de éste
* **Flexibilidad:** nuestro proyecto debería poder recibir cambios, tanto grandes como pequeños, como parte del proceso iterativo de creación; esto es importante especialmente para los videojuegos, dónde se dice que ni el mejor diseñador del mundo es capaz de acertar en su diseño a la primera.
* **Escalabilidad:** nuestro proyecto tiene que estar listo para agregar nuevos elementos con el menor esfuerzo posible, y sin que agregar dichos elementos suponga hacer cambios en otros sistemas

En este ejercicio aplicaremos estos principios para el diseño de un sistema de armamento flexible, mantenible y escalable.

**Objetivos**

* Creación del Sistema
* Creación de cuatro armas de tipo variado utilizando este sistema

**Estructura General**

En esta sección nombraremos las clases a utilizar y describiremos la relación entre ellas.

Utilizaremos las siguientes clases:

* **BarrelBase:** se trata de la clase base para el cañón de las armas (Barrel, en inglés); es capaz de disparar, tanto disparos individuales como disparos continuos
  + **BarrelByRaycast:** esta clase deriva de BarrelBase y realiza disparos instantáneos utilizando una llamada a Physics.Raycast
    - **Ejemplos:** pistola, ametralladora, escopeta
  + **BarrelByInstantiation:** esta clase deriva de BarrelBase y dispara a través de la instanciación de un objeto, el cual tiene que tener su propio sistema de impulso
    - **Ejemplos:** lanzagranadas, lanzamisiles
  + **BarrelByParticleSystem:** esta clase deriva de BarrelBase y activa/desactiva sistemas de partículas; las partículas de estos sistemas estarán preparadas para colisionar contra otros objetos y generar daño
    - **Ejemplos:** lanzallamas, pistola de pintura
* **Weapon:** una Weapon representa a un arma que tiene uno o varios Barrels
  + en nuestro caso haremos una implementación sencilla, pero normalmente esta clase consumiría el cargador o el combustible del arma para determinar si ésta puede disparar o no
  + por lo demás, su principal función es centralizar las llamadas de disparo, para que tanto el jugador como los enemigos como otros sistemas puedan montar un arma y usarla sin importar el tipo de cañón ni el tipo de uso (disparo a disparo/disparo contínuo)
* **WeaponManager:** esta clase gestiona la selección y el uso de un inventario de armas. Se utilizará para descubrir las armas de las que dispone un objeto (el jugador, un enemigo, una torreta) y poder realizar las siguientes acciones sobre éstas:
  + Realizar un disparo
  + Iniciar el disparado continuo
  + Parar el disparado continuo
  + Seleccionar un arma en concreto
  + Seleccionar el arma anterior/siguiente
* **PlayerWeaponController:** esta clase recoge los inputs del jugador y los convierte en acciones sobre el WeaponManager.

Además de las clases principales, también prepararemos otras clases que nos ayudarán a tener un mejor juego y a comprobar el funcionamiento de las clases principales:

* **Projectile** – esta clase representa a un objeto disparable
* **Explosion** – esta clase representa un objeto capaz de hacer daño a su alrededor

**Ejercicio 4: Proyectiles**

**Projectile, Grenade y Missile**

Nuestro primer objetivo es crear una clase BarrelBase y un primer Barrel, el BarrelByInstantiation. Para poder probarlo estas clases con comodidad, vamos a dedicar un poco de tiempo previamente a crear dos tipos de proyectil que puedan ser lanzados por BarrelByInstantiation:

* La Granada
* El Misil

Ambos objetos utilizarán la clase Projectile para iniciar/mantener una velocidad, detectar colisiones, y gestionar su tiempo de vida.

* **Prepara la clase Projectile**
  + Esta clase está preparada para representar proyectiles de todo tipo, con características comunes
  + Crea la clase Projectile derivada de MonoBehaviour
  + Añade las siguientes variables:
    - **speedMode** – se trata de un enumerado que indica cómo se aplicará la velocidad; puede tener uno de los siguientes valores:
      * **ApplyOnStartOnly** – la velocidad se aplicará únicamente al inicio
      * **ApplyAllTheTime** – la velocidad se aplicará en todo momento
    - **speed** – se trata de un Vector3 que indica, en coordenadas locales, en qué dirección y con qué magnitud se moverá el proyectil
    - **lifeTime** – es el tiempo de vida que tendrá el proyectil una vez lanzado
    - **resultPrefabs[]** – es un array que contiene uno o más prefabricados que deberán instanciarse cuando el proyectil termine su vida
    - **destroyOnCollision** – es un booleano que indica si debe destruirse el proyectil cuando colisiona
    - **instantiateResultsOnCollision** – es un booleano que indica si deben instanciarse los resultPrefabs cuando el proyectil colisiona
    - **instantiateResultsOnLifeTimeEnd** – es un booleano que indica si deben instanciarse los resultPrefabs cuando la vida del proyectil llega a su fin
  + Implementa las siguientes funciones
    - **Awake()**
      * Obtén una referencia al rigidbody
      * Inicia un contador remainingLifeTime a 0
    - **Start()**
      * Si el speedMode es ApplyOnStartOnly, aplica la speed al rigidbody en este momento
        + ¡Importante! Recuerda que la speed es local al proyectil, utiliza transform.TransformDirection() para pasar la dirección de local a mundo
    - **Update()**
      * Si el speedMode es ApplyAllTheTime, aplica la speed en este momento
        + Recuerda pasar la speed de local a global como se indica en Start()
      * Incrementa remainingLifeTime en Time.deltaTime
        + Si llega o supera el lifeTime

Destruye el gameObject

* + - * + Si instantiateResultsOnLifeTimeEnd es True, instancia los prefabricados contenidos en el array resultPrefabs

* + - **OnCollisionEnter()**
      * Si destroyOnCollision es true, destruye el gameObject
      * Si instantiateResultsOnCollision es True, instancia los prefabricados contenidos en el array resultPrefabs
    - **¡Atención!**
      * A la hora de instanciar los contenidos en resultPrefabs utiliza un booleano para asegurarte de que esto ocurre una única vez
        + Podría ocurrir que en el mismo fotograma se acabase el tiempo de vida y el objeto colisionase, pero sólo queremos instanciar los resultados una única vez
* **Prepara el prefabricado Grenade**
  + Crea un objeto vacío, llamalo Grenade
    - Añade un sphereCollider
    - Añádele la clase Projectile, y configura los siguientes valores:

|  |  |
| --- | --- |
| **speedMode** | ApplyOnStartOnly |
| **speed** | (0f, 0f, 10f) |
| **lifeTime** | 3 |
| **resultPrefabs** | Nada de momento |
| **destroyOnCollision** | False |
| **instantiateResultsOnCollision** | False |
| **instantiateResultsOnLifeTimeEnd** | True |

* + - Añade un rigidbody
      * Configuralo a ContinuousDynamic
    - Añade un objeto vacío a Grenade, llámalo Visuals
      * Dentro de Visuals, añade (por lo menos) una esfera con el mismo tamaño que el colisionador (quítale el collider que viene por defecto)
        + Puedes añadir más detalles que hagan que los visuales parezcan una granada, como en la imagen de ejemplo  
          Una caricatura de una persona

          Descripción generada automáticamente con confianza baja
* **Prepara el prefabricado Missile**
  + Crea un objeto vacío, llamalo Missile
    - Añade un capsuleCollider
      * ¡Importante! ¡Orienta el capsuleCollider en dirección a la Z!
    - Añádele la clase Projectile, y configura los siguientes valores:

|  |  |
| --- | --- |
| **speedMode** | ApplyAllTheTime |
| **speed** | (0f, 0f, 10f) |
| **lifeTime** | 7 |
| **resultPrefabs** | Nada de momento |
| **destroyOnCollision** | True |
| **instantiateResultsOnCollision** | True |
| **instantiateResultsOnLifeTimeEnd** | False |

* + - Añade un rigidbody
      * Configúralo a ContinuousDynamic
      * Que no le afecte la gravedad
    - Añade un objeto vacío a Grenade, llámalo Visuals
      * Dentro de Visuals, añade (por lo menos) una capsula con el mismo tamaño que el colisionador (quítale el collider que viene por defecto)
        + Puedes añadir más detalles que hagan que los visuales parezcan un misil, como en la imagen de ejemplo  
          Imagen que contiene Patrón de fondo

          Descripción generada automáticamente
* **Prueba la granada y el misil**
  + En la escena, instancia prefabricados de la granada y el misil
    - Comprueba que la granada sale en dirección a su Z, que realiza una trayectoria parabólica, y que se destruye cuando termina su tiempo de vida
    - Comprueba que el misil sale en dirección a su Z, que avanza recto, y que se destruye al chocar contra otro objeto y cuando termina su tiempo de vida
      * Puede que tengas que instanciar más misiles para realizar todas las comprobaciones

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

|  |
| --- |
|  |

**Ejercicio 5: Explosiones**

En este ejercicio prepararemos dos explosiones: la explosión lógica y la explosión gráfica. Aunque ambas van de la mano, poder disponer de ellas por separado permite crear combinaciones gráficas y lógicas.

**Explosión Lógica**

* **Prepara la clase Explosion**
  + La clase Explosion representa un objeto que, tal como se instancia, hace daño de explosión a su alrededor y empuja rigidbodies
  + Crea la clase Explosion, derivada de MonoBehaviour
    - Prepara las siguientes variables públicas
      * **radius** – es un flotante que define el alcance del daño de explosión
      * **damage** – es un flotante que define cuánto daño hace la explosión
      * **damageLayerMask** – es la layerMask que utilizaremos para detectar los colliders potencialmente afectados
      * **occlusionLayerMask** – es la layerMask que utilizaremos para ocluir el efecto de la explosión
      * **affectedTags[]** – es un array de strings que contiene los tags a los que puede afectar la explosión
      * **pushForce** – es un flotante con la fuerza máxima que hay que aplicar a los rigidbodies que se encuentren dentro del área de la explosión
    - Implementa la función Start()
      * Utiliza Physics.OverlapSphere() con damageLayerMask para obtener los colisionadores potencialmente afectados
      * Para cada uno de ellos
        + Comprueba si su tag está contenido en affectedTags,
        + En caso afirmativo, utiliza Physics.Raycast() con occlusionLayerMask para determinar si hay línea de visión entre el centro de la explosión y el collider potencialmente afectado
        + En caso afirmativo

Obtén el HurtCollider del collider, y aplica damage

Intenta obtener el rigidbody asociado al collider; si lo obtienes, aplícale un empuje por explosión utilizando el método AddExplosionForce

* + - * Llama a Destroy(), para que la explosión desaparezca al instante
* **Crea el prefabricado ExplosionRadius3**
  + Crea un nuevo GameObject, llámalo ExplosionRadius3
  + Añádele el script Explosion
  + Configura Explosion de la siguiente manera

|  |  |
| --- | --- |
| **Radius** | 3 |
| **Damage** | 1 |
| **damageLayerMask** | Puedes elegir todas las capas, o preparar una capa especial sólo para los enemigos |
| **occlusionLayerMask** | Todas las capas |
| **affectedTags** | Enemy, Untagged |
| **pushForce** | 1000 |

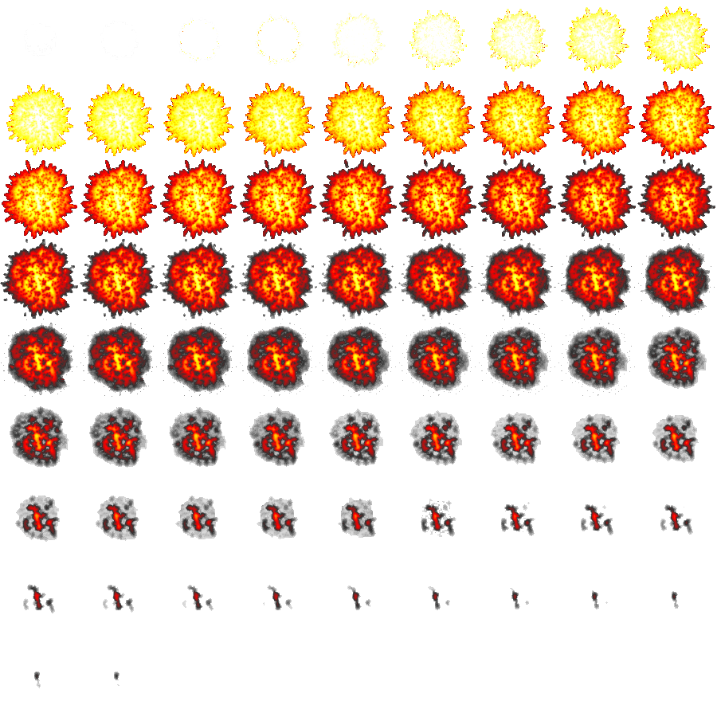
* + Convierte el GameObject en un prefabricado
* **Comprueba el funcionamiento de la Explosion**
  + Crea una nueva escena
    - Añade un cubo con escala (30, 1, 30) a modo de suelo
  + Para comprobar el daño
    - Coloca 6x6 enemigos en rejilla, separados 1m x 1m de cada uno
      * Si no haces bake del escenario, los enemigos no podrán moverse, lo cual nos favorece
    - Instancia una ExplosiunRadius3 justo en el centro
    - Desactíva la explosión
  + Para comprobar el empuje
    - Coloca 6x6 cubos en rejilla, separados 1.5m x 1.5m de cada uno
    - Instancia una ExplosiunRadius3 justo en el centro de la rejilla
    - Desactiva la explosión
  + Dale a Play
    - Activa la primera explosión y comprueba que los enemigos dentro del radio sufren daño
    - Activa la segunda explosión y comprueba que las boxes dentro del radio salen disparadas

**Explosión Gráfica**

La explosión gráfica consistirá en un efecto de partículas. Aprovecharemos para aprender algunos trucos que nos dejarán ver la potencia del sistema de partículas de Unity.

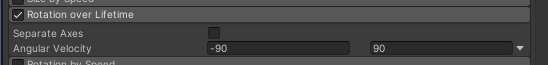
Imagen que contiene luz

Descripción generada automáticamente

* **Prepara el Material**
  + Entra en internet, busca una textura tipo “Sprite-sheet” transparente de explosión; la encontrarás fácilmente buscando “explosión sprite sheet”.  
    
  + Importa la textura en Unity
  + Crea un nuevo material de tipo HDRP/Lit
    - Activa el alpha clipping, ponlo a 0.5f
    - Pon la textura que has bajado en la propiedad Albedo
    - Pon la misma textura en la propiedad Emissive
    - Marca el check de intensidad de emisión y pon dicha intensidad a unos 100000 nits
      * Según la iluminación en tu escena puedes necesitar más o menos emisión
* **Prepara la explosión visual principal**
  + Crea un nuevo Particle System, llámalo VisualExplosion
  + Asígnale el material que has creado
  + Utiliza los siguientes parámetros para los módulos:
    - Main:  
      Interfaz de usuario gráfica

      Descripción generada automáticamente
    - Emission:  
      Una captura de pantalla de un celular

      Descripción generada automáticamente
    - Shape:  
      Interfaz de usuario gráfica

      Descripción generada automáticamente
    - Rotation Over Lifetime:  
      
    - Texture Sheet Animation:  
      Interfaz de usuario gráfica, Aplicación

      Descripción generada automáticamente
    - Renderer:  
      Interfaz de usuario gráfica

      Descripción generada automáticamente
  + Fíjate en que el efecto, una vez terminado, se autodestruirá
  + Convierte el GameObject VisualExplosion en un prefabricado
* **(Opcional) Prepara la luz de la explosión**
  + Opcionalmente, puedes crear un sub-emitter que:
    - Emita una única partícula
      * Esta partícula será invisible: en el módulo Render, cambia a RenderMode None
    - Tenga el módulo Light preparado para asociar una luz (que tendrás que crear previamente) a dicha partícula
    - Puedes utilizar los valores de tamaño y alfa de la partícula para hacer que la luz cambie de alcance e intensidad a lo largo del tiempo
* **(Opcional) Prepara los brazos de la explosión**
  + Aprovechando el material, puedes crear brazos para la explosión de la siguiente manera
    - Utiliza un sub-emitter que emita entre tres y cinco partículas invisibles con una trayectoria parabólica
    - Dentro de este sub-emitter, utiliza otro sub-emitter que:
      * Utiliza el mismo material que en la explosión base
      * Utiliza la misma configuración que en la explosión visual principal, pero:
        + Emitiendo por distancia recorrida
        + Emitiendo desde una esfera relativamente pequeña
        + Sin gravedad

**Integración con la Granada y el Misil**

* Modifica el prefabricado de la granada y del misil
  + En ambos casos, añade a resultPrefabs
    - El prefabricado de la explosión lógica
    - El prefabricado de la explosión visual
* Repite la prueba de funcionamiento de la granada y del misil
  + En este caso, ambos deberían explotar (visual y lógicamente) al alcanzar sus respectivas condiciones de final

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

|  |
| --- |
|  |

**Ejercicio 6: Barrels**

En este ejercicio prepararemos los Barrels que nos permitirán componer cualquier tipo de arma.

**BarrelBase, BarrelByInstantiation**

Empezaremos por el BarrelBase y el BarrelByInstantiation; para este último podremos utilizar la granada y/o el misil para hacer pruebas.

* **Prepara la clase BarrelBase**
  + Prepara la funcionalidad
    - Crea la clase abstracta BarrelBase, derivada de MonoBehaviour
    - Prepara los siguientes métodos abstractos
      * Shot()
      * StartShooting()
      * StopShooting()
  + Prepara el depurado (debugging)
    - Prepara las siguientes variables miembro:
      * debugShot
      * debugStartShooting
      * debugStopShooting
    - Prepara la función OnValidate()
      * Utiliza debugShot para llamar a Shot()
      * Utiliza debugStartShooting para llamar a StartShooting()
      * Utiliza debugStopShooting para llamar a StopShooting()
* **Prepara la clase BarrelByInstantiation**
  + Crea la clase BarrelByInstantiation, derivala de BarrelBase
  + Prepara las siguientes variables
    - **projectilePrefab –** es una referencia al prefabricado del proyectil que tiene que ser disparado
  + Implementa **Shot()**
    - Shot() tiene que instanciar un projectilePrefab en transform.position, y con rotación transform.rotation
  + Implementa **StartShooting()** y **StopShooting()**
    - Dado que BarrelByInstantiation no soporta el disparo contínuo, ambas funciones han de lanzar una **excepción** avisando de esto
* **Comprueba el funcionamiento del BarrelByInstantiation**
  + En la escena, crea un objeto vacío, llámalo BarrelByInstantiation
  + Establece el prefab Grenade como projectilePrefab
  + Dale a Play
  + Utiliza debugShot sobre el Barrel para lanzar granadas
    - Comprueba que cada vez que pones a True debugShot se instancia una granada en la posición de BarrelByInstantiation y que sale disparada en la dirección del forward de BarrelByInstantiation

**BarrelByRaycast, BarrelByRaycastWithCadence**

Continuamos con la clase BarrelByRaycast, que representa aquellos disparos tan instantáneos que se tienen que resolver en el mismo fotograma, normalmente armas de fuego.

También prepararemos la clase BarrelByRaycastWithCadence, que hace exactamente lo mismo, pero con funcionamiento de disparo continuo.

* **Prepara la clase BarrelByRaycast**
  + Crea la clase BarrelByRaycast, derivada de BarrelBase
  + Añade las siguientes propiedades:
    - **range** – es un flotante que indica hasta dónde alcanza el disparo
    - **damage** – es el daño que hace un disparo
    - **layerMask** – es una LayerMask que indica tanto la layer que puede afectar como las layers que ocluyen los disparos
    - **affectedTags[]** – contiene los tags que se verán afectados por un disparo
  + Prepara el método **InternalShot()**
    - Utiliza Physics.Raycast para lanzar un rayo desde transform.position en dirección transform.forward, con un alcance de range y discriminando por layerMask
      * Si hay Collider y su tag está contenido en affectedTags, intenta obtener un HurtCollider
        + Si lo obtienes, haz el daño establecido en damage
  + Implementa **Shot()**
    - Llama a InternalShot()
  + Implementa **StartShooting()** y **StopShooting()**
    - Dado que BarrelByRaycast no soporta el disparo contínuo, ambas funciones han de lanzar una **excepción** avisando de esto
* **Prepara la clase BarrelByRaycastWithCadence**
  + Crea la clase BarrelByRaycastWithCaden, derivada de BarrelByRaycast
  + Añade las siguientes propiedades:
    - timeBetweenShots – es un flotante que indica el tiempo en segundos entre disparo y disparo
  + Añade las siguientes variables privadas:
    - isShooting – es un booleano que indica si el Barrel está disparando
    - lastShotTime – es un flotante que contiene el tiempo en segundos desde que se inició la aplicación en el que se realizó el último disparo
  + Implementa Shot()
    - Dado que BarrelByRaycastWithCadence no soporta el disparo individual, lanza una excepción avisando de esto
  + Implementa **StartShooting()**
    - Pon isShooting a True
  + Implementa **StopShooting()**
    - Pon isShooting a False
  + Prepara la función Update()
    - Si isShooting está a True, y ha pasado suficiente tiempo desde lastShotTime
      * Llama a InternalShot()
      * Asigna a lastShotTime el valor de Time.time
* **Comprueba el funcionamiento de ambos barrels**
  + **BarrelByRaycast**
    - Crea un GameObject vacío, llámalo BarrelByRaycast
      * Añádele un componente BarrelByRaycast con rango 100, daño 1, layerMask todas, y affectedTags Enemy, Untagged
    - Pon un enemigo inmóvil delante del GameObject
    - Utiliza debugShot para comprobar que el enemigo recibe daño
  + **BarrelByRaycastWithCadence**
    - Crea un GameObject vacío, llámalo BarrelByRaycastWithCadence
      * Añádele un componente BarrelByRaycastWithCadence con rango 100, daño 1, layerMask todas, affectedTags Enemy, Untagged, y 0.25f de tiempo entre disparos
    - Pon varios enemigos inmóviles en fila india frente al GameObject
    - Utiliza debugStartShooting y debugStopShooting para comprobar que todos los enemigos reciben daño

**BarrelByParticleSystem**

Finalmente, prepararemos el BarrelByParticleSystem. Este Barrel se basa en la capacidad de los ParticleSystems de generar partículas con colisión.

Esta capacidad es muy interesante, ya que permite crear armas como el lanzallamas, además de patrones de disparo al estilo de los shooters de naves.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza baja

Sin embargo, también tiene sus límites, pues las partículas únicamente pueden tener colisión esférica, y no soportan detección de colisión continua (pueden saltarse objetos) y no hay detección de partículas contra partículas.

También hay que tener cuidado con la cantidad de partículas que colisionan, pues pueden impactar en el rendimiento; pero si tenemos todo esto en cuenta y aún así nos parece bien, son cómodas y potentes.

* **Prepara la clase BarrelByParticleSystem**
  + Crea la clase BarrelByParticleSystem, derivada de BarrelBase
  + Añade las siguientes propiedades:
    - **damage** – es el daño que hace cada partícula que impacta
  + Implementa **Shot()**
    - Dado que BarrelByParticleSystem no soporta el disparo individual, lanza una excepción avisando de esto
  + Implementa **StartShooting()**
    - Activa la emisión del ParticleSystem
  + Implementa **StopShooting()**
    - Desactiva la emisión del ParticleSystem
  + Crea el método OnParticleCollision()
    - Intenta obtener un HurtCollider del GameObject que viene por parámetro
    - Si lo consigues, haz el daño especificado
* **Prepara un ParticleSystem que haga daño**
  + Crea el ParticleSystem por defecto, llamalo BarrelByParticleSystem
  + Oriéntalo hacia el suelo
    - Comprueba que en estos momentos no colisiona
  + Activa el modulo Collision
    - En type, establece World
      * Comprueba que ahora las partículas chocan contra el mundo
    - Establece LifeTimeLoss a 1, para que las partículas mueran al chocar
    - Marca el check Send Collision Messages para que tanto el GameObject que contiene el ParticleSystem como el GameObject impactado puedan recibir mensajes de OnParticleCollision
  + Comprueba que en estos momentos las partículas colisionan y desaparecen al hacerlo
  + Añade el componente BarrelByParticleSystem
    - Establece un damage de 1
* **Comprueba el funcionamiento**
  + Coloca varios enemigos inmóviles delante del objeto que has creado
  + Utiliza el debugStartShooting y el debugStopShooting para comprobar que el ParticleSystem dispara cuando se establece, y que hace daño como se espera

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

|  |
| --- |
|  |

**Ejercicio 7: Weapon, WeaponManager y PlayerWeaponController**

En este ejercicio implementaremos tanto las Weapon como el WeaponManager.

**Weapon**

Implementaremos la clase Weapon, que representa a un arma individual. Este arma estará compuesta por uno o más barrels, y ofrecerá información sobre cómo ser utilizada a otras clases.

* **Crea la clase Weapon**
  + Crea una clase Weapon que derive de MonoBehaviour
    - Prepara las siguientes propiedades:
      * **type** – es un enumerado que puede tener dos valores:
        + ShotByShot
        + ContinuousShot
    - Prepara también estas propiedades para debug:
      * **debugShot** – booleano
      * **debugStartShooting** – booleano
      * **debugStopShooting** – booleano
    - Implementa **Awake()**
      * Guarda una caché con todos los barrels que sean hijos del GameObject
    - Implementa **Shot()**
      * Llama a la función Shot() de todos los barrels en la caché
    - Implementa **StartShooting()**
      * Llama a la función StartShooting() de todos los barrels en la caché
    - Implementa **StopShooting()**
      * Llama a la función Shot() de todos los barrels en la caché
    - Implementa **NotifySelected()**
      * De momento no hace nada
    - Implementa **NotifyDeselected()**
      * Comprueba si type es de tipo ContinuousShot; en ese caso llama a StopShooting para asegurar que un arma que no está seleccionada no esté disparando
* **Crea 5 Weapons funcionales**
  + **Antes de empezar**
    - Los gráficos del arma tienen que apuntar hacia el forward (la Z) del arma
    - Si los has hecho correctamente, los barrels apuntan y disparan en dirección forward de su transform
      * Cuando los coloques como hijos, colocalos en el punto desde el que tienen que disparar y apuntando al forwarde del arma
  + **La Pistola**
    - Crea un GameObject vacío, llámalo Pistol
    - Añade la clase Weapon a este objeto, establece type a ShotByShot
    - Añade un GameObject hijo, llámalo Visuals
      * Dentro de Visuals, añade algo de geometría de Unity en este estilo  
        Imagen que contiene oscuro, reloj, tabla

        Descripción generada automáticamente
    - Añade un GameObject hijo, llámalo Barrels
      * Añade a Barrel un GameObject hijo, llámalo BarrelByRaycast
        + Añade a este nuevo hijo el componente BarrelByRaycast, puéblalo con parámetros interesantes
  + **La Ametralladora**
    - Crea un GameObject vacío, llámalo MachineGun
    - Añade la clase Weapon a este objeto, establece type a ContinuousShot
    - Añade un GameObject hijo, llámalo Visuals
      * Dentro de Visuals, añade algo de geometría de Unity en este estilo  
        Imagen que contiene cuarto, oscuro, tabla

        Descripción generada automáticamente
      * ¡Recuerda que los gráficos del arma tienen que apuntar hacia el forward (la Z) del GameObject MachineGun!
    - Añade un GameObject hijo, llámalo Barrels
      * Añade a Barrel un GameObject hijo, llámalo BarrelByRaycastWithCadence
        + Añade a este nuevo hijo el componente BarrelByRaycastWithCadence, puéblalo con parámetros interesantes
  + **El LanzaGranadas**
    - Crea un GameObject vacío, llámalo GrenadeLauncher
    - Añade la clase Weapon a este objeto, establece type a ShotByShot
    - Añade un GameObject hijo, llámalo Visuals
      * Dentro de Visuals, añade algo de geometría de Unity en este estilo  
        Imagen que contiene computadora, oscuro, tabla, iluminado

        Descripción generada automáticamente
      * ¡Recuerda que los gráficos del arma tienen que apuntar hacia el forward (la Z) del GameObject GrenadeLauncher!
    - Añade un GameObject hijo, llámalo Barrels
      * Añade a Barrel un GameObject hijo, llámalo BarrelByInstantiation
        + Añade a este nuevo hijo el componente BarrelByInstantiation, puéblalo con parámetros interesantes
  + **El Lanzallamas**
    - Crea un GameObject vacío, llámalo FlameThrower
    - Añade la clase Weapon a este objeto, establece type a ContinuousShot
    - Añade un GameObject hijo, llámalo Visuals
      * Dentro de Visuals, añade algo de geometría de Unity en este estilo  
        Imagen que contiene oscuro, tabla, grande, hombre

        Descripción generada automáticamente
      * ¡Recuerda que los gráficos del arma tienen que apuntar hacia el forward (la Z) del GameObject GrenadeLauncher!
    - Añade un GameObject hijo, llámalo Barrels
      * Añade a Barrel un GameObject hijo, llámalo BarrelByInstantiation
        + Añade a este nuevo hijo el componente BarrelByInstantiation, puéblalo con parámetros interesantes
  + **Opcionalmente**, puedes crear otras armas como:
    - **La Escopeta** – dispone de varios BarrelByRaycast apuntando en diferentes direcciones que hacen mucho daño pero tienen poco alcance  
      Imagen que contiene objeto, oscuro, remoto, luz

      Descripción generada automáticamente
    - **El LanzaMisiles** – similar al LanzaGranadas, pero de éste salen misiles  
      Imagen en blanco y negro

      Descripción generada automáticamente con confianza baja
    - **El Rifle de Francotirador** – a efectos prácticos es una pistola con mucho alcance y poca dispersión; en este ejercicio no hemos implementado valores de dispersión, pero no es difícil hacerlo  
      Imagen que contiene oscuro, béisbol, tabla, iluminado

      Descripción generada automáticamente
* **Comprueba que las Weapons funcionan**
  + Instancia las 4 weapons
  + Coloca enemigos frente a estas
  + Utiliza las propiedades de debug para comprobar que las 4 armas funcionan correctamente

**WeaponManager**

La clase WeaponManager gestiona el arma que actualmente está utilizando un objeto (personaje jugador, enemigo, torreta…), permitiendo elegir entre varias armas disponibles. Para facilitar la vida, la clase WeaponManager “descubre” las armas disponibles haciendo una búsqueda hacia abajo.

* **Prepara la clase WeaponManager**
  + Crea la clase WeaponManager, derivada de MonoBehaviour
    - Prepara las siguientes propiedades:
      * **initialWeaponToSelect** – es el índice del arma disponible en el WeaponManager
    - Prepara las siguientes propiedades para Debug – son todas booleanas
      * **debugSelectNextWeapon**
      * **debugSelectPrevWeapon**
      * **debugShot**
      * **debugStartShooting**
      * **debugStopShooting**
    - Añade las siguientes variables privadas:
      * **weapons[]** – es el array de armas en el que guardaremos las armas descubiertas
      * **currentWeaponIndex** – es el índice del arma actualmente seleccionada
        + Si vale -1, quiere decir que no hay arma seleccionada
    - Implementa los siguientes métodos:
      * **Awake**()
        + Crea una caché de las Weapons disponibles buscando con un GetComponentsInChildren()
      * **Start()**
        + Desactiva los GameObjects de todas las armas
        + Llama a la función SelectWeapon() (a continuación) pasando por parámetro el initialWeaponToSelect
      * **SelectWeapon()** – esta función se encarga de cambiar el arma seleccionada, recibe un entero **weaponIndex** que indica el índice del arma a seleccionar
        + Comprueba que weaponIndex esté en rango

Si weaponIndex es menor que -1, cicla al indice de la última arma disponible

Si weaponIndex es mayor que la cantidad de armas disponibles, cicla a -1

* + - * + Si ya hay un arma seleccionada:

Notifica a dicha arma que ya no está seleccionada

Desactiva el objeto del arma

* + - * + Si el weaponIndex es diferente de -1

Activa el objeto de la nueva arma

Notifica a la nueva arma correspondiente que está seleccionada

* + - * **HasSelectedWeapon()** – devuelve true si el índice del arma seleccionada es diferente de -1
      * **SelectedWeaponIsShotByShot()** – devuelve true si hay arma seleccionada y es de tipo ShotByShot
      * **Shot()** – si hay arma seleccionada, llama al Shot() de ésta; si no, lanza una excepción.
      * **StartShooting()** – si hay arma seleccionada, llama al StartShooting() de ésta; si no, lanza una excepción.
      * **StopShooting()** – si hay arma seleccionada, llama al StopShooting() de ésta; si no, lanza una excepción.
      * **SelectNextWeapon()** – llama a SelectWeapon con el índice del arma siguiente
      * **SelectPreviousWeapon()** – llama a SelectWeapon con el índice del arma anterior
      * **OnValidate()** – utiliza los booleanos de debug que hemos declarado antes para llamar a las diferentes funciones y comprobar su correcto funcionamiento
        + **debugSelectNextWeapon** – SelectNextWeapon()
        + **debugSelectPrevWeapon** – SelectPrevWeapon()
        + **debugShot** – Shot()
        + **debugStartShooting** – StartShooting()
        + **debugStopShooting** – StopShooting()
* **Prepara el Prefab WeaponManager**
  + Crea un nuevo objeto vacío, llámalo WeaponManager
    - Añádele la clase WeaponManager
    - Instancia y cuelga de este objeto todas las armas que hemos creado hasta ahora
    - Convierte el objeto WeaponManager en un prefab
* **Comprueba el funcionamiento**
  + Instancia un prefab WeaponManager
  + Instancia varios enemigos delante de este
  + Utiliza los booleanos de debug para comprobar que se puede
    - Cambiar de arma
    - Realizar disparos individuales
    - Realizar disparado continuo
    - Dañar a los enemigos creados

**PlayerWeaponController**

Ahora que tenemos toda la potencia de las Weapon y el WeaponManager a mano, es hora de integrarlo con el player. Para ello, prepararemos un componente intermedio para que el GameObject Player pueda utilizar las funciones de WeaponManager, e integraremos este, junto con el prefab WeaponManager en el personaje jugador.

* **Deshabilita el disparo de PlayerController**
  + Para que no haya colisión de controles, comenta el código de PlayerController que actualmente se encarga de disparar
* **Prepara las InputActions para las armas**
  + Añade las siguientes actions para en el InputActionAsset que creamos en la actividad anterior
    - **UnSelectWeapon** – botón asociado a la tecla 0
    - **SelectWeapon1** – botón asociado a la tecla 1
    - **SelectWeapon2** – botón asociado a la tecla 2
    - **SelectWeapon3** – botón asociado a la tecla 3
    - **SelectWeapon4** – botón asociado a la tecla 4
    - **SelectWeapon5** – botón asociado a la tecla 5
    - **CycleWeapon** – value de tipo delta asociada a la rueda del ratón
* **Prepara la clase PlayerWeaponController**
  + Crea la clase PlayerWeaponController, derivada de MonoBehaviour
    - Añade las siguientes propiedades:
      * **shot** – es la InputActionReference que recogerá la action de disparo
      * **weapon1** – es la InputActionReference que recogerá la action de cambio al arma 1
      * **weapon2** – es la InputActionReference que recogerá la action de cambio al arma 2
      * **weapon3** – es la InputActionReference que recogerá la action de cambio al arma 3
      * **weapon4** – es la InputActionReference que recogerá la action de cambio al arma 4
      * **weapon5** – es la InputActionReference que recogerá la action de cambio al arma 5
      * **cycleWeapon** – es la InputActionReference que recogerá la action de cambio de arma por ciclado
      * **weaponManager** – es una referencia al WeaponManager que estará controlado por este PlayerWeaponController
    - Prepara las siguientes funciones:
      * **OnEnable()** y **OnDisable()**
        + Utiliza estas funciones para habilitar (Enable()) y deshabilitar (Disable()) todas las actions que hemos recibido por parámetro
      * **Update()**
        + Llamará a **UpdateShooting()**, **SelectWeaponByNumber()** y **SelectWeaponByCycle()**
        + Estas funciones las veremos a continuación
      * **UpdateShooting()**
        + Si el WeaponManager tiene arma seleccionada, gestiona el disparo de ésta según el tipo

Si el tipo del arma es ShotByShot, dispara en cuanto la acción de disparo tenga lugar

Si el tipo del arma es ContinuousShot, llama a StartShooting() cuando se pulse la acción de disparo, y a StopShooting() cuando se suelte

* + - * **SelectWeaponByNumber()**
        + Gestiona la selección del arma a través de las acciones asociadas a las teclas numéricas

Llama a SelectWeapon() con el índice correspondiente a la tecla pulsada

* + - * **SelectWeaponByCycle()**
        + Gestiona la selección del arma por ciclado

Lee el valor de la acción **cycleWeapon**

Utiliza la componente y:

Si es mayor de 0, pasa a la siguiente arma

Si es menor de 0, pasa al arma anterior

* **Integra el WeaponManager y el PlayerWeaponController en el jugador**
  + Instancia un prefab WeaponManager dentro del PlayerGroup
    - Añádele un ParentConstraint para asegurarte que se mueve junto con el Player
  + Añade al Player el Componente PlayerWeaponController
    - Asigna las actions que has creado anteriormente
    - Asigna la referencia al componente WeaponManager
* **Comprueba que todo funciona**
  + Si lo has hecho correctamente, en estos momentos dispones de un jugador con capacidad para seleccionar entre varias armas y disparar con todas ellas, tanto si se trata de armas disparo a disparo como si son armas de disparo contínuo

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

|  |
| --- |
|  |

**Ejercicio 8: Grab Weapons**

En este ejercicio prepararemos a nuestro personaje jugador y sus armas para que pueda cogerlas utilizando técnicas de cinemática inversa disponibles en el Package Animation Rigging.

Una caricatura de una persona

Descripción generada automáticamente con confianza baja

Realizaremos este ejercicio en tres pasos:

* Preparar dos Rigs para que los brazos puedan dirigirse hacia cualquier punto que deseemos
* Prepararemos las armas para que dispongan de **puntos de agarre** para las manos y “**pistas” (hints)** para la colocación de los codos del personaje
* Prepararemos el WeaponManager para que pueda gestionar el agarrado de las armas si está disponible

**Preparación de los Rigs**

En Animation Rigging, Rig es una estructura que nos permite utilizar una o más “restricciones” (constraints) de los diferentes tipos que hay. Nosotros utilizaremos restricciones de dos huesos, que simulan articulaciones como el codo o las rodillas:

* **Antes de empezar**
  + Instala el Package Animation Rigging, lo encontrarás en el PackageManager
* **Configura un BoneRenderer para el personaje**
  + Selecciona el modelo (en mi ejemplo Ty)
  + En la barra de menús, selecciona la opción Animation Rigging 🡪 Bone Renderer Setup  
    Interfaz de usuario gráfica, Sitio web

    Descripción generada automáticamente
  + Gracias al Bone Renderer puedes ver y seleccionar los huesos del personaje fácilmente
* **Prepara los rigs**
  + Añade un componente RigBuilder al modelo del personaje, allí donde tiene el Animator y el Bone Renderer
    - El componente RigBuilder permite establecer el orden en el que se evalúan los Rigs que puedan afectar al modelo

Interfaz de usuario gráfica

Descripción generada automáticamente

* + Prepara la estructura de los Rigs:  
    Texto

    Descripción generada automáticamente
    - Crea un GameObject Rigs vacío
      * Dentro de éste, crea el GameObject HoldWeaponRig
        + Añade a éste último un componente Rig

El componente Rig sirve para establecer el peso de las Constraints que estén bajo su jerarquía

Añade este Rig al RigBuilder que has creado anteriormente

* + - * + Ponle el peso al máximo (1)  
          Interfaz de usuario gráfica

          Descripción generada automáticamente
        + Añade un GameObject hijo, llámalo LeftArm

En LeftArm, añade un componente TwoBoneIK

Ayudándote del BoneRenderer, configura el TwoBoneIK para que controle el hueso del brazo, del antebrazo, y la mano  
Interfaz de usuario gráfica

Descripción generada automáticamente

Crea dos esferas

Dales escala 0.05f

Ponlas como hijas de LeftArm

Llámalas Hint y Target

Añade a ambas el componente **RigTransform**, necesario para su correcto funcionamiento con los Rigs

Añade a ambas un componente ParentConstraint

Asígnalas en los SourceObjects  
Interfaz de usuario gráfica

Descripción generada automáticamente

**Comprueba el Rig funciona**

Dale a Play

Si lo has hecho todo bien, el brazo se estirará hacia la esfera Target, y el codo intentará apuntar hacia la esfera Hint

* + - * + Repite estos mismos pasos para crear la restricción para el brazo derecho

**Preparación de las Armas**

En estos momentos, tenemos control sobre dónde se colocan las manos gracias a los Targets y los Hints.

A continuación, prepararemos una serie de “GrabPoints” en las armas que nos servirán para enganchar los TargetsYHints, dando así la sensación de que el personaje ha cogido el arma.

* **Modifica la clase Weapon**
  + Añade la propiedad grabPointsParent, una transform que permitirá asignar el parent de los GrabPoints
* **Prepara el prefabricado GrabPoints**
  + Para facilitarnos la vida, asumiremos que los GrabPoints están siempre en el mismo orden
    - Para ayudarnos con esto, prepararemos un prefabricado que añadiremos a todas las armas, pudiendo configurar posiciones y rotaciones para cada una de ellas
  + Crea la siguiente estructura de objetos:  
    Interfaz de usuario gráfica, Texto

    Descripción generada automáticamente
    - Conviértela en un prefabricado
* **Prepara todas las armas**
  + Añade el prefabricado GrabPoints a todas ellas
  + Coloca los objetos dentro de GrabPoints en las posiciones que consideres adecuadas
    - Orienta los dos Targets para que las manos queden bien orientadas cuando estén cogiendo el arma
  + En el componente Weapon del arma, arrastra a la propiedad grabPointsParent el GrabPoints añadido al arma
  + Actualiza los prefabricados

**Preparación del WeaponManager**

Finalmente, haremos algunos cambios en el WeaponManager para que sea éste el que gestione el “enganchado” de los Targets y los Hints de los rigs a los GrabPoints de las armas.

* **Mejora la clase WeaponManager**
  + Añade las siguientes propiedades de tipo ParentContraint:
    - **leftTarget**
    - **leftHint**
    - **rightTarget**
    - **rightHint**
  + Añade la propiedad **weaponHoldingRig**, de tipo Rig
  + ***¡Importante!*** *Dado que el WeaponManager no es exclusivo de personajes que dispongan de Targets y Hints, de ahora en adelante asegúrate de que las propiedades anteriores no son null antes de utilizarlas*
  + Añade la función **OnEnable()**
    - Pon el peso del weaponHoldingRig a 0f, para que (de momento) el rig esté inactivo
  + Prepara el método **SetupConstraint()**
    - Esta función se encarga de configurar una constraint para que funcione con un grabPoint
      * Cuando se cambie de arma, este método hará que los Targets y Hints del Rig queden enganchados a los GrabPoints del arma
    - Dado que el método es un poco complejo, utiliza el código a continuación:  
      Texto

      Descripción generada automáticamente
  + Prepara el método **UnsetupConstraint()**
    - Esta función “libera” una constraint
    - Utiliza el siguiente código:  
      Texto

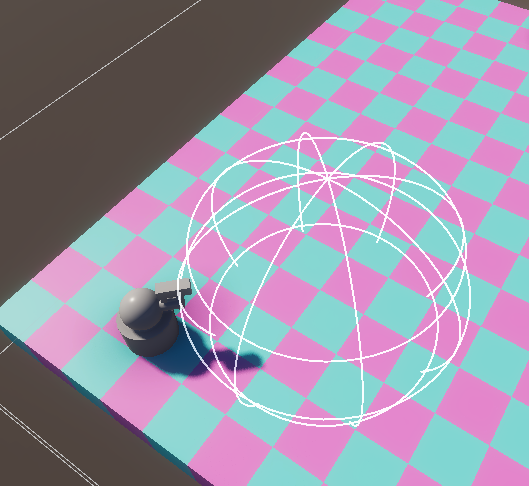
      Descripción generada automáticamente
  + Modifica la función SelectWeapon()
    - En caso de que ya hubiese arma, prepara código para soltarla:
      * Llama a UnsetupConstraint() para todas las constraints
      * Pon el peso de weaponHoldingRig a 0
    - En caso de que se esté poniendo un arma con índice diferente de -1
      * Llama a SetupConstraint() para todas las constraints
      * Pon el peso de weaponHoldingRig a 1
* **Configura el WeaponManager del jugador**
  + Asigna las nuevas referencias en el WeaponManager del jugador
* **Comprueba que todo funcione**
  + Si lo has hecho bien, ahora tu personaje, además poder disparar, también cogerá el arma correctamente

**Ejercicio 8:** Incluye a continuación un enlace a un vídeo en el que demuestres el funcionamiento.

|  |
| --- |
|  |

|  |
| --- |
| **Parte 3**  **Sistema de IA Escalable** |

**Ejercicio 9: Vista y Torreta**



En los inicios del desarrollo de videojuegos, las IA se basaban en un principio inmutable: existía un único jugador, por lo que acciones como el acercamiento o el disparo utilizaban la posición de dicho jugador.

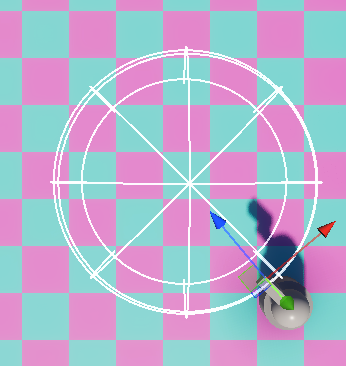
En la actualidad, sin embargo, utilizamos técnicas de detección que imitan los sentidos de los seres vivos. Esto tiene varias ventajas:

* **Realismo** – el jugador entiende que las IAs están limitadas del mismo modo que en la realidad, lo que ayuda a la sensación de inmersión
* **Flexibilidad** – no sólo el jugador puede ser detectado: los aliados del jugador, u objetos de interés también pueden serlo, dando lugar a un abanico superior de posibilidades

En este ejercicio prepararemos un sistema a imitación de la vista, y lo utilizaremos junto con el WeaponManager para implementar una torreta. Aunque no lo haremos, no es difícil imaginar que también podemos implementar otros sentidos y utilizar algún tipo de criterio para decidir nuestro objetivo.

**Sentido de la Vista**

Prepararemos un sentido de la vista muy sencillo y limitado, pero fácil de implementar precisamente por eso mismo. Para lo que normalmente sería el cono de visión utilizaremos una simple pero eficiente esfera.

* **Prepara el Componente Visible**
  + Todos los objetos que sean visibles deberán tener este mismo componente
  + Crea la clase Visible, derivada de MonoBehaviour
  + ¡Y ya está! Se trata de una clase vacía que, al menos por ahora, sólo sirve para determinar qué objeto es visible o no
    - En el futuro, puedes añadir datos para refinar si un objeto se ve o no
      * Si está de pie, agachado o estirado
      * Cuál es su bando
      * Etc.
* **Prepara el sentido de la vista**
  + Crea una nueva clase Sight, derivada de MonoBehaviour
  + Añádele las siguientes propiedades:
    - **radius** – es el radio de la esfera que utilizaremos a modo de visión
    - **visibleTags[]** – es un array de strings que contiene los tags de aquellos objetos que detectaremos como visibles
    - **layerMask** – es una layerMask para reducir la cantidad de objetos que recibiremos en las llamadas de física en busca de visibles
  + Añádele las siguientes variables miembro:
    - **visiblesInSight** – es una lista que contiene los Visibles que están a la vista
      * Si mantienes esta lista pública, te vendrá bien para depurar el funcionamiento de Sight, pero idealmente es un miembro privado
  + Implementa **Update()**
    - Limpia visiblesInSight()
    - Utiliza OverlapSphere() para obtener una lista de Colliders que potencialmente puedan ser visibles; utiliza layerMask para descartar los objetos que no
      * La idea es que el objeto que contenga la vista esté en los ojos de un personaje, y que el overlap Sphere se coloque por delante de su posición, en dirección hacia donde está mirando  
        
    - Para todos los colliders que recibas:
      * Comprueba si el tag del collider está en visibleTags
        + Si es así, intenta obtener el componente Visible
        + Si lo consigues, añade dicho Visible a los visiblesInSight
* **Comprueba que el Sentido de la Vista funciona**
  + Crea un nuevo GameObject vacío
  + Añádele el componente Sight
  + Crea varios cubos con layers y tags específicos
    - Configura el componente Sight para que pueda ver algunos de ellos, pero otros no
  + Comprueba utilizando la lista de visiblesInSight que la vista puede ver los cubos en la layer y tag necesarios
    - Haz visibilesInSight pública, al menos durante esta prueba
  + Opcionalmente, pero ayuda mucho, utiliza el Window 🡪 Analysis 🡪 Physics Debugger para poder visualizar las queries del OverlapSphere

**Torreta**

La torreta será sencilla. Implementaremos una clase encargada de:

* Esperar a que algún objetivo esté a la vista
* Orientarse continuamente hacia ese objetivo
* Abrir fuego/dejar de disparar gracias al WeaponManager

Una torreta avanzada sería capaz de girar para apuntar y también de apuntar arriba/abajo, pero esta torreta la mantendremos bastante sencilla para facilitar el ejercicio.

* **Prepara los Visuales de la torreta**
  + Crea un nuevo GameObject, llámalo TorretaVisuals
  + Utiliza la siguiente imagen como inspiración para crear unos gráficos sencillos para una torreta  
    Imagen que contiene interior, tabla, pequeño, colorido

    Descripción generada automáticamente
    - ¡Ojo! En esta imagen se puede ver el arma montada en el cañón
      * Una buena idea es ocultar la geometría del arma y aún así disponer de las ventajas de poder colocar y dispararla donde necesitemos
    - Si quieres, puedes bajarte un gráfico de internet, pero procura no perder mucho tiempo en este paso
  + Convierte TorretaVisuals en un prefabricado
* **Prepara el GameObject Torreta**
  + Crea un nuevo objeto vacío, llámalo Torreta
  + Asegúrate de que está apoyado sobre el suelo
  + Crea un hijo de Torreta llamado Visuals
    - Dentro de Visuals, cuelga una nueva instancia de To
  + Añade el prefabricado WeaponManager como hijo de Torreta
    - Colócalo en una posición de modo que, al activar el arma inicial, ésta quede en una posición correcta respecto a la torreta
      * Es decir, que pueda disparar con la esperanza de acertar a un objetivo
    - Configúralo para que inicialmente utilice una de las armas de tipo ContinuousShot que tenga disponible
  + Crea un hijo llamado Sight, colócalo un poco por delante de la posición en la que aparecerá el arma
    - Añade a este hijo un componente Sight
    - Configura el componente Sight para que pueda detectar al jugador
    - Ponle un radio limitado (3 metros será suficiente)
* **Prepara la clase Torreta**
  + Crea una clase Torreta, derivada de MonoBehaviour
  + Prepara las siguientes propiedades
    - **sight** – es una referencia al componente Sight que utilizaremos para localizar
    - **weaponManager** – es una referencia al WeaponManager que la torreta utilizará para disparar
  + Prepara los siguientes métodos:
    - **Update()**
      * Si sight detecta un nuevo objetivo…
        + Activa el arma actual (llama a weaponManager.StartShooting())
      * Si había objetivo detectado y se pierde…
        + Desactiva el arma actual
      * Si hay objetivo actualmente…
        + Gira alrededor del eje vertical hacia la dirección en la que se encuentra el objetivo
        + Puedes utilizar el mismo código de orientación que has creado para el personaje jugador
* **Completa el GameObject Torreta y haz un Prefabricado**
  + Añade la clase Torreta al objeto Torreta
    - Asigna referencias a sight y weaponManager
  + Convierte Torreta en un Prefabricado
* **Comprueba que la torreta funciona**
  + Si has configurado correctamente la vista y has creado correctamente la torre, deberías poder mover a un personaje jugador y la torreta debería girar en dirección a éste y disparar cuando el jugador entre dentro de su vista
  + Alternativamente, puedes crear una torre que dispare a enemigos

**Ejercicio 9:** Incluye a continuación un enlace a un vídeo en el que demuestres el funcionamiento.

|  |
| --- |
|  |

**Ejercicio 10: IA por Estados-Objeto**

En la UF anterior hemos preparado un enemigo con estados. El resultado fue correcto, pero hay unas cuantas pegas:

* El código de los estados está en la clase Enemy, a pesar de que es totalmente válido para representar otros tipos de personajes (NPCs, variantes de enemigos)
* El código de la toma de decisión que lleva al enemigo a estar en un estado u otro también está en la clase Enemy, provocando un **acoplamiento** que en el futuro puede hacer el código difícil de mantener

En este ejercicio convertiremos los estados de la IA de los que ya dispone el Enemy en nuevos estados de IA desacoplados, representados cada uno de ellos por una clase, y aplicables tanto al Enemy como a muchos otros objetos.

También haremos nuevos estados:

* Patrullaje
* Ir a visitar el último punto en el que vimos a un objetivo

**El Patrón de Diseño State**

Este ejercicio está basado en implementar el patrón de diseño State.

|  |
| --- |
| En Programación Orientada a Objetos, un patrón de diseño es una solución a un problema recurrente (el tipo de problema que podemos encontrar en varios proyectos). Los patrones de diseño cumplen con varios criterios:   * Son fáciles de implementar * Son desacoplados, ya que solucionan un único problema |

Dado que ya tenemos el código de dos estados (Wandering y Chasing), simplemente extraeremos dicho código a nuevas clases.

|  |
| --- |
| El Patrón de Diseño State propone que los Estados sean clases con los siguientes métodos:   * **OnEnter()** – la máquina de estados llamará a este método para notificar al Estado que ahora es el estado que está en funcionamiento   + El estado puede utilizar esta llamada para activar/desactivar objetos o reservar recursos que necesite para su funcionamiento * **Update()** – la máquina de estados llamará a este método a cada ciclo (fotograma) para que el estado tenga la oportunidad de actualizar el * **OnExit()** – la máquina de estados llamará a este método para notificar al Estado que ha dejado de estar en funcionamiento   + El estado puede utilizar esta llamada para devolver recursos y activar/desactivar objetos que ya no necesite   En el caso de Unity, utilizaremos como equivalentes los métodos OnEnable(), Update(), y OnDisable(), y nuestra máquina de estados se encargará simplemente de activar/desactivar el estado que considere necesario, **produciendo de manera automática el mismo funcionamiento que tendríamos que implementar nosotros en la máquina de estados**. |

**Extracción de los Estados Existentes**

* **Prepara la clase AIStateBase**
  + Esta es la clase base para todos los estados del juego
  + Crea la clase AIStateBase derivada de MonoBehaviour
    - Crea variables protegidas para cachear el NavMeshAgent y el Sight
    - Crea el método virtual Awake()
      * En Awake(), cachea los componentes previamente mencionados
* **Extrae el estado Wandering** 
  + Crea la clase AIStateWandering, derivada de AIStateBase
    - Trae las siguientes propiedades que ya tenías en EnemyWithAI
      * maxWanderDistance
      * wanderReachDistance
    - Trae también las siguientes variables privadas que ya tenías en EnemyWithAI
      * originalPosition
      * wanderPosition
    - Trae los siguientes métodos de los que ya disponías:
      * ChooseWanderPosition()
    - Implementa los siguientes métodos:
      * **Awake()** 
        + Aprovecha para guardar la originalPosition
        + ¡No olvides llamara a base.Awake() para que AIStateBase pueda hacer su trabajo!
      * **OnEnable()**
        + Llama a ChooseWanderPosition()
      * **Update()**
        + Traete del Update() de EnemyWithAI el código relevante del caso State.Wandering:

Llamada al NavMeshAgent para ir a la wanderPosition

Comprobación de si se ha llegado a ésta, en cuyo caso se llama a ChooseWanderPosition()

* **Extrae el estado Chasing**
  + Crea la clase AIStateSeekTarget, derivada de AIStateBase
  + Implementa los siguientes métodos
    - **Update()**
      * Traete el código relevante del Update() de EnemyWithAI para el estado State.Chasing:
        + Si hay Target

Llamada al NavMeshAgent para que se dirija a la posición de dicho Target

* + - * + Aprovecha que ahora disponemos del sentido de la vista para obtener el target de ésta
        + ¡No es necesario código para salir de este estado! La toma de decisiones se encargará de ir a un estado o a otro

**Adaptación de EnemyWithAI y Pruebas**

* **Actualiza la clase EnemyWithAI**
  + Agrega las siguientes variables
    - allStates[] – es un array de tipo AIStateBase
    - aiStateWander – es una referencia al estado de Wander
    - aiStateSeekTarget – es una referencia al estado de persecución (el antiguo estado de Chasing)
  + Actualiza **Awake()**
    - Cachea todos los estados disponibles en allStates[] con un GetComponents()
    - Cachea aiStateWander con GetComponent()
    - Cachea aiStateSeekTarget con GetComponent()
  + Actualiza **Start()**
    - Deshabilita todos los estados en allStates[]
    - Llama a SetState() con aiStateWander como parámetro
      * En breve veremos la función SetState()
  + Prepara la función **SetState()**
    - Esta función recibe por parámetro una referencia a un AIStateBase, llamado nextState
    - Opera de la siguiente manera
      * **Paso 1:** desactivar el estado que esté actualmente activado
        + Recorre toda la lista de estados

Para todos los estados que no sean nextState, si está habilitado, deshabilítalo

* + - * **Paso 2:** activar nextState
        + Si está deshabilitado, habilita nextState
  + “Limpia” la función **Update()**
    - Retira todo el código del Switch()
    - En su lugar, prepara una toma de decisiones muy simple:
      * Si hay un objetivo a la vista (Sight) llama a SetState() con aiStateSeekTarget
      * Si no, llama a SetState() con aiStateWander
* **Actualiza el Prefabricado**
  + Añade los estados AIStateWander y AIStateSeekTarget al Enemy en el prefabricado
  + Aplica los cambios
* **Prueba que el EnemyWithAI funciona**
  + Si lo has hecho correctamente, el enemigo funcionará exactamente igual que antes.

**Nuevos Estados**

* **Prepara la nueva funcionalidad de patrullado**
  + Crea la clase AIStatePatrol derivada de AIStateBase
    - Añade las siguientes propiedades
      * **patrolPointsParent** – es una referencia a una transformada que contendrá varios objetos que se considerarán puntos de ruta
      * **startPatrolPointIndex** – es un entero que indica
* **Prepara la nueva funcionalidad de ir al último punto en que se vio al jugador**
  + Modifica el sentido de la vista (clase **Sight**)
    - Añade las siguientes variables:
      * **lastVisiblePosition** – Vector3 que almacenará la última posición en la que se vio al enemigo
      * **lastVisiblePositionIsForgotten** – booleano que empieza siendo true y que indica si hay lastVisiblePosition ya ha sido visitada
    - Actualiza la función **Update()**
      * Al final de todo, y si es que hay visibles en la lista de visibles,
        + Actualiza lastVisiblePosition a la posición del primero de ellos
        + Pon lastVisiblePositionIsForgotten a false
    - Prepara nuevas funciones públicas
      * **IsThereALastVisiblePosition()** – devuelve cierto si lastVisiblePositionIsForgotten es false, es decir, si la vista aún recuerda la última posición de un objetivo
      * **GetLastVisiblePosition()** – devuelve lastVisiblePosition
      * **ForgetLastVisiblePosition()** – pone lastVisiblePositionIsForgotten a cierto
  + Crea la clase **AIStateGoToLastTargetPosition**
    - Añade las siguientes variables
      * **reachThreshold** – es un flotante que indica la distancia a la posición a la que tenemos que llegar para determinar que hemos alcanzado la LastTargetPosition
    - Implementa los siguientes métodos
      * **Update()**
        + Utiliza el NavMeshAgent para acercarte a Sight.GetLastVisiblePosition()
        + Si el GameObject está lo suficientemente cerca de la LastVisiblePosition, entonces llama a Sight.ForgetLastVisiblePosition()

Esto hará que, en el siguiente fotograma, la toma de decisiones abandone el estado AIStateGoToLastTargetPosition

* **Actualiza el Prefab Enemy**
  + Edita el Prefab EnemyGroup:
    - Añade al GameObject Enemy los componentes
      * AIStatePatrol
      * AIStateGoToLastTargetPosition
    - Añade al EnemyGroup un nuevo GameObject hijo, llámalo PatrolPoints
      * Añade varios objetos hijos a este que serán los puntos de patrullaje
      * En el AIStatePatrol de Enemy, pasa una referencia
  + Modifica el EnemyWithAI
    - Sustituye el estado Wandering por el estado de Patrullaje
      * Modifica EnemyWithAI para que tome la decisión de hacer patrullado en lugar de Wandering
    - Añade el estado de ir a la última posición en la que se ha visto un objetivo
      * Modifica EnemyWithAI para que, en caso de que no haya objetivo a la vista, consulte al Sight si recuerda la última posición del objetivo a la vista
        + Si la recuerda, entonces entramos al estado de ir al último lugar de objetivo
        + Si no, entonces que vuelva al patrullado
* **Comprueba que todo funciona bien**
  + Si lo has hecho correctamente, tu enemigo ahora es capaz de patrullar; si ve al jugador irá hacia él, y si deja de verlo irá al punto en el que lo perdió de vista

**Ejercicio 10:** Incluye a continuación un enlace a un vídeo en el que demuestres el funcionamiento.

|  |
| --- |
|  |

**Ejercicio 11: Árbol de Decisiones en GameObject**

Ahora que disponemos de estados que siguen el Patrón de Diseño State, estamos mejor pero seguimos teniendo varios problemas:

* La clase Enemy, que toma todas las decisiones, necesita conocer todos los estados
  + Esto provoca una dependencia de Enemy de todos estos estados, con lo que si se añade un nuevo estado o se quita un estado, hay que cambiar también la clase Enemy
    - A esto se le llama **solapamiento**, y queremos evitarlo
* La toma de decisiones en la clase Enemy es una y única; si queremos implementar una nueva toma de decisiones (para un enemigo cobarde, o para un enemigo que no patrulla ni hace wander) tendremos que…
  + … derivar la clase Enemy…
  + … o crear un método alternativo en la clase Enemy…
  + … pero ambas opciones son poco deseables

Por ese motivo vamos a preparar un árbol de decisiones basado en nodos.

**El Árbol de Decisiones**

Un árbol de decisiones es una estructura de datos formada por dos tipos de nodos principalmente:

* Nodos de Condición
* Nodos de Estado

Esta estructura se inicia en un Nodo de Condición binario (cierto/falso) que conecta con otros nodos, lo que da lugar a un árbol que, una vez recorrido, tiene que terminar obligatoriamente en uno u otro Nodo de Estado.

El árbol de decisiones se recorre cada fotograma o cada cierto tiempo, aunque también se puede poner como condición que el estado actual permita cambiar de estado.

**Implementación del Árbol de Decisiones en Unity**

De nuevo, aprovecharemos que estamos trabajando con Unity para implementar nuestro árbol apoyados en Unity. Para ello, implementaremos los nodos como GameObjects, para lograr una estructura como la siguiente:

Texto

Descripción generada automáticamente

Como se puede intuir, los GameObjects que se llaman Condition\_XXX implementan una condición, mientras que los nodos SetState\_XXX establecen el nuevo estado.

Además de los nodos, utilizaremos un componente DecissionMaker que se encargará de recorrer el árbol y dar opción a que los nodos hagan su trabajo.

**Prepara los Nodos Base y el DecissionMaker**

* **Prepara el componente AIDecissionBase**
  + Crea una clase abstracta AIDecissionBase que derive de MonoBehaviour
    - Añadele el método abstracto Execute()
* **Prepara el componente AIConditionBase**
  + Crea una clase abstracta AIConditionBase que derive de MonoBehaviour
    - Crea un método abstracto **Evaluate()** que devuelva un booleano
      * Lo utilizarán las derivadas para implementar las condiciones
    - Crea el método **Run()**
      * Implementa el siguiente algoritmo:
        + Si Evaluate() devuelve cierto

Obtén el **primer** hijo de la transformada

Si este hijo tiene un componente AIConditionBase, llama a su método Run()

En caso contrario, si este hijo contiene un componente AIDecissionBase, llama a su método Execute()

* + - * + Si Evaluate() devuelve falso

Obtén el **segundo** hijo de la transformada

Si este hijo tiene un componente AIConditionBase, llama a su método Run()

En caso contrario, si este hijo contiene un componente AIDecissionBase, llama a su método Execute()

* **Prepara el DecissionMaker**
  + La clase DecissionMaker es muy parecida al EnemyWithAI, pero centrada en la ejecución del árbol de toma de decisiones y en habilitar a las decisiones la posibilidad de cambiar el estado
  + Crea la clase DecissionMaker derivada de MonoBehaviour
    - Añade las siguientes propiedades:
      * **startingCondition** – es la ConditionBase que se utilizará como raíz del árbol de toma de decisiones
    - Añade también las siguientes variables:
      * **allStates[]** – es el array que contendrá todos los estados posibles
    - Implementa los siguientes métodos
      * **Awake()**
        + Cachea todos los estados con GetComponents()
      * **Start()**
        + Deshabilita todos los estados en la caché
      * **Update()**
        + Llama al Run() de la startingCondition
      * **ChangeState()**
        + Exactamente igual que el de EnemyWithAI

**Prepara las Condiciones y Decisiones Efectivas**

* **Prepara el componente AIDecissionSetState**
  + Crea una clase AIDecissionSetState derivada de AIDecissionBase
    - Añádele las siguientes propiedades:
      * **decissionMaker** – es una referencia al DecissionMaker del enemigo
      * **stateToSet** – es una referencia al AIStateBase que establecer cuando se ejecute esta decisión
    - Implementa el método **Execute()**
      * Simplemente llama al ChangeState() del Decissionmaker con el parámetro stateToSet
* **Prepara el componente AIConditionHasCurrentTarget**
  + Crea una clase AIConditionHasCurrentTarget derivada de AIConditionBase
    - Añádele las siguientes propiedades:
      * **sight** – es una referencia a la vista del enemigo
    - Implementa el método **Evaluate()**
      * Llama a GetCurrentTarget() de sight, devuelve cierto si hay target o falso si no
* **Prepara el componente AIConditionHasLastTarget**
  + Crea una clase AIConditionHasLastTarget derivada de AIConditionBase
    - Añádele las siguientes propiedades:
      * **sight** – es una referencia a la vista del enemigo
    - Implementa el método **Evaluate()**
      * Devuelve el resultado de llamar a Sight.IsThereALastVisiblePosition()

**Prepara la clase EnemyEntity**

En el momento en el que tenemos el DecissionMaker, la clase EnemyWithAI pierde casi el 100% de su funcionalidad. Por eso la extraeremos del prefabricado final del enemigo.

Sin embargo, esto nos deja también sin la animación que nos proporcionaba a través de las funciones abstractas implementadas para Entity. Por ello prepararemos una nueva clase EnemyEntity para sustituir finalmente a EnemyWithAI.

* **Prepara la clase EnemyEntity**
  + Crea la clase EnemyEntity, derivada de Entity
  + Implementa los métodos abstractos de Entity
    - **GetCurrentVerticalSpeed()** – devuelve 0f, ya que el navMeshAgent no se despegará del suelo
    - **GetJumpSpeed()** – devuelve 0f
    - **IsGrounded()** – devuelve true
    - **IsRunning()** – devuelve true si la velocidad es mayor que la runSpeed
    - **GetNormalizedMoveXZLocal()**
      * Crea una variable Vector3 en la clase, llámala lastPosition
      * En la llamada a GetNormalizedMoveXZLocal()
        + Calcula un deltaMovement restando lastPosition a la posición actual
        + Asigna la posición actual a lastPosition
        + Calcula un deltaMovementLocal utilizando transform.InverseTransformDirection
        + Devuelve deltaMovementLocal normalizado

**Prepara el Prefab Enemigo y Comprueba que Funcione**

* **Prepara el Prefab Enemigo**
  + Prepara los nodos del árbol de decisiones
    - Utiliza el esquema que ya hemos visto para realizarlo  
      Texto

      Descripción generada automáticamente
  + Añade el componente DecissionMaker al objeto Enemy
    - Asigna la condición inicial a Condition\_HasTarget
  + Retira el componente EnemyWithAI
  + Añade el componente EnemyEntity
  + Aplica todos los cambios al Prefab
* **Comprueba que el Enemigo funciona**
  + Si has hecho las tareas correctamente, ahora tu enemigo puede hacer ruta, localizar al jugador, acercarse a éste, e ir al último punto en que lo vio

**Ejercicio 11:** Incluye a continuación un enlace a un vídeo en el que demuestres el funcionamiento.

|  |
| --- |
|  |

|  |
| --- |
| **Parte 4**  **Extras Opcionales** |

**Extras Opcionales**

Puedes realizar diversos extras para obtener mejor nota.

**Extras Clásicos**

Puedes realizar las siguientes mejoras para obtener puntos extra:

* Incluye un Sistema de Menús
* Incluye sonidos
* Incluye música
* Realiza un diseño de nivel con enemigos distribuidos de manera interesante
* Viste con gráficos interesantes el nivel realizado

**Extras sobre los Enemigos**

También puedes realizar alguna de las siguientes extensiones de los enemigos:

* **Capacidad de disparar**
  + Añade una condición de “objetivo está a distancia de disparo”
  + Añade un estado disparar que:
    - Al entrar en el estado empiece a disparar
    - Al salir del estado deje de disparar
    - Se oriente hacia el target en todo momento
  + Utiliza la condición en el Árbol de Toma de Decisiones para crear un enemigo que utiliza el arma que tenga en el WeaponManager para atacar al jugador

Con esta capacidad disponible, ya puedes hacer varios tipos de enemigo:

* **Enemigo Normal**
  + Patrulla o hace Wander
  + Cuando te ve te persigue
  + Cuando estás a distancia de disparo, se detiene y te dispara
* **Guardián**
  + Permanece siempre en el mismo punto
  + Cuando te ve, te sigue con la mirada
  + Si entras en su rango de disparo, te dispara
  + Es ideal para ponerlo al final de pasillos o en otros sitios
* **Valiente**
  + Patrulla o hace Wander
  + Cuando te ve te persigue
  + Cuando estás a distancia de disparo, te dispara pero sin detenerse
  + Cuando está a corta distancia, te sigue disparando mientras se orienta hacia ti y camina hacia la derecha,
    - Esto hace que gire alrededor tuyo