Capitulo 10

UD10. XOGOS 2D

Resultados de avaliación

RA1. Desenvolve programas que integren contidos multimedia, para o que analiza e emprega as tecnoloxías e as librarías específicas. **RA2.** Analiza a arquitectura de xogos 2D e 3D seleccionando e probando motores de xogos. **RA3.** Desenvolve xogos 2D e 3D sinxelos utilizando motores de xogos.

Criterios de avaliación

- CA1.4 Utilizáronse clases para crear e manipular figuras gráficas en 2D.
- CA1.5 Utilizáronse clases para reproducir e xestionar música e sons.
- CA1.6 Utilizáronse clases para a conversión de datos multimedia dun formato a outro.
- CA1.7 Utilizáronse clases para o control de eventos e excepcións, etc.
- CA1.8 Utilizáronse clases para a creación e o control de animacións.
- CA1.9 Utilizáronse clases para construír reprodutores de contidos multimedia.
- CA1.10 Depuráronse e documentáronse os programas desenvolvidos.
- CA2.1 Identificáronse os elementos da arquitectura dun xogo 2D e 3D.
- CA2.2 Analizáronse as funcións e os compoñentes dun motor de xogos.
- CA2.3 Analizáronse contornos de desenvolvemento de xogos.
- CA2.4 Analizáronse motores de xogos, as súas características e as súas funcionalidades.
- CA2.5 Identificáronse os bloques funcionais dun xogo.
- CA2.6 Definíronse e executáronse procesos de rénder.
- CA2.7 Recoñeceuse a representación lóxica e espacial dunha escena gráfica sobre un xogo existente.
- CA3.1 Deseñouse o proxecto de desenvolvemento dun xogo novo.
- CA3.2 Estableceuse a lóxica do xogo.
- CA3.3 Seleccionouse e instalouse o motor e o contorno de desenvolvemento.
- CA3.4 Creáronse obxectos e definíronse os fondos.
- CA3.5 Instaláronse e utilizáronse extensións para o manexo de escenas.
- CA3.6 Utilizáronse instrucións gráficas para determinar as propiedades finais da superficie dun obxecto ou dunha imaxe.
- CA3.7 Incorporóuselles son aos eventos do xogo.
- CA3.8 Desenvolvéronse e implantáronse xogos para dispositivos móbiles.
- CA3.9 Realizáronse probas de funcionamento e mellora dos xogos desenvolvidos.
- CA3.10 Documentáronse as fases de deseño e desenvolvemento dos xogos creados.

BC. Xogos 2D

- Reprodución e control de animacións.
- Procesamento e reprodución de obxectos multimedia: clases, estados, métodos e eventos.
- Documentación do desenvolvemento de aplicacións con contido multimedia.
- Programación de aplicacións con gráficos en dúas dimensións.
- Programación de aplicacións con música e sons.

- Conversión de información multimedia.
- Manexo de eventos e excepcións.
- Animación 2D e 3D.
- Arquitectura do xogo: compoñentes.
- Motores de xogos: tipos e uso.
- Áreas de especialización, librarías utilizadas e linguaxes de programación.
- Compoñentes dun motor de xogos.
- Librarías que proporcionan as funcións básicas dun motor 2D/3D.
- API dos gráficos 3D.
- Estudo de xogos existentes.
- Aplicación de modificacións sobre xogos.
- Proxecto de desenvolvemento: fases, estrutura e obxectivo.
- 0Incorporación de música e efectos sonoros.
- Desenvolvemento de xogos para dispositivos móbiles.
- Análise de execución. Optimización do código.
- Documentación do desenvolvemento dos xogos creados.
- Lóxica do xogo.
- Contornos de desenvolvemento para xogos.
- Integración do motor de xogos en contornos de desenvolvemento.
- Obxectos gráficos.
- Escenas e fondos.
- Propiedades dos obxectos: luz, texturas, reflexos e sombras.
- Aplicación das funcións do motor gráfico. Renderización.
- Aplicación das funcións do grafo de escena. Tipos de nodos e o seu uso.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SUBSECCIONES DE UD10. XOGOS 2D

Capitulo 1

PONG

PONG

Vamos a realizar una versión del mítico juego Pong. Para ello, abordaremos conceptos clave, como el manejo de la física del juego, la interacción entre objetos, la programación de scripts en C#, y la creación de la interfaz de usuario.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SUBSECCIONES DE PONG

Capitulo 1

INTRODUCCIÓN

INTRODUCCIÓN

Antes de comenzar con el desarrollo de nuestro videojuego, vamos a realizar un análisis de su origen, entender su dinámica de juego (gameplay) y elaborar el diseño de nuestra versión personalizada del juego.

Recursos adicionales

- Magnavox Odyssey Table Tennis (1972)
- Original Atari PONG (1972) arcade machine gameplay
- PONG
- Museum PONG
- Atari, Inc. (1972-1992)
- Allan Alcorn
- Nolan Bushnell
- Ted Dabney

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SUBSECCIONES DE INTRODUCCIÓN

REFERENCIAS HISTÓRICAS

Spacewar

Spacewar! es un videojuego de combate espacial desarrollado en 1962 por Steve Russell. Fue el primer videojuego que no estaba anclado a una máquina en concreta, con la posibilidad de que alguien de la comunidad añadiera alguna función nueva. Steve Russell aparece como el gran responsable del juego, y lo es, pero Spacewar! es significativo porque fue también un esfuerzo grupal, que crecía con el esfuerzo de su comunidad.

Tal era su éxito, que fue el primer juego que protagonizó un torneo oficial, unas olimpiadas organizadas por la revista Rolling Stones en 1972 en el laboratorio de Inteligencia Artificial de Stanford, diez años después, convirtiéndose así no sólo en un pionero de los videojuegos y de la comunidad modder, sino también en un pionero de los esports.

A pesar de que el juego estaba muy extendido para la época, su alcance directo era muy limitado. Spacewar! no fue comercializado de manera masiva, ya que estaba pensado para computadoras mainframe, se distribuyó en diversos sistemas informáticos y se realizaron adaptaciones no oficiales.

Es recordado como el primer videojuego de la historia y ha influido significativamente en la evolución de la industria, pero no fue comercializado de la misma manera que los

videojuegos contemporáneos. Su impacto radica en su contribución a la cultura de los videojuegos y su influencia en el desarrollo de títulos futuros.

Computer Space

En el año 1972, dos ingenieros visionarios, Nolan Bushnell y Steve Rusell, decidieron aventurarse en la creación de videojuegos. Inspirados por Spacewar desarrollaron el videojuego Computer Space que se convirtió en el primer videojuego **comercial** disponible para su compra.

Este juego fue una evolución de la idea original de Spacewar!, pero con el objetivo de llevar la experiencia de juego a un **público** más amplio.

La visión de Bushnell y su equipo fue plasmada en una máquina concebida completamente con tecnología electrónica analógica, bautizada como “Space.” Esta máquina, operada con monedas, se destacó como una de las primeras incursiones exitosas en el ámbito de los videojuegos comerciales. Representaba no solo una complejidad técnica notable, sino también un logro significativo en el incipiente campo del entretenimiento electrónico.

A pesar de ser un hito significativo, Computer Space no tuvo el mismo éxito comercial que el juego que lo precedió, Pong. La **complejidad del juego** y la falta de familiaridad del público con los videojuegos electrónicos pueden haber contribuido a sus ventas más modestas.

Aunque Computer Space no alcanzó el éxito esperado, fue un pionero en la industria de los videojuegos comerciales. Estableció la idea de tener **videojuegos** electrónicos disponibles para su compra y juego en lugares **públicos**, allanando el camino para futuros desarrollos en la industria del entretenimiento electrónico.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

INTRODUCCIÓN

Tras aprender de sus errores, al año siguiente Bushnell creó un juego mucho más simplificado. La idea de Pong surgió durante una visita de Nolan Bushnell a un bar donde observó a la gente jugar a un juego de tenis de mesa (ping-pong). Inspirado por la simpleza y la diversión de este juego, Bushnell concibió la idea de crear una versión electrónica.

Nolan Bushnell encargó a Al Alcorn, un ingeniero de Atari, la tarea de desarrollar una versión electrónica del tenis de mesa. Inicialmente, Pong era un experimento destinado a probar la reacción del público en una máquina de arcade ubicada en un bar local.

Lo que inicialmente se consideró un proyecto experimental reveló su verdadero potencial, convirtiéndose en un producto comercial de enorme éxito. Este logro no solo consolidó la posición de **Atari** en la incipiente industria del videojuego, sino que también estableció un precedente para futuras innovaciones y éxitos en el emocionante mundo de los videojuegos.

En esta época (pre-microprocesador), los juegos se implementaban mediante **hardware** (no software). Esto limitaba las posibilidades creativas de los videojuegos

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

HARDWARE

Aunque ya en el año anterior, 1971, se introdujo el primer microprocesador, el Intel 4004, que era un procesador de cuatro bits, y en ese mismo año se lanzó el Intel 8008, marcando el inicio de los microprocesadores de ocho bits, no fue sino hasta 1975 cuando se desarrolla el primer videojuego utilizando **microprocesadores** en concreto, fue una desarrolladora americana que se llamaba **Midway** a partir de otro juego desarrollado por la empresa japonesa Taito con la que colaboraban. Estas dos empresas crean un juego que se llamaba **"Gun Fight"**. Este juego marcó un antes y un después al incorporar el Intel 4004 en su diseño. Hasta ese momento, los videojuegos se implementaban principalmente mediante circuitos electrónicos sin la complejidad de los microprocesadores. "Gun Fight" se convirtió en un referente al simultáneamente combinar los mercados estadounidense y japonés. Esto fue en el año 1975, pero hasta ese momento, todos los videojuegos se desarrollaban mediante electrónica.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

DISEÑO

Al igual que la mayoría de los videojuegos de aquella época, el diseño integral del juego se fundamenta en dos principios fundamentales:

1. **Simplicidad:**

Este principio, defendido por el propio Nolan Bushnell, se basa en la premisa de que todos los grandes juegos deben ser fáciles de aprender pero difíciles de dominar. La idea es que deben ser accesibles desde el primer momento, recompensando tanto al jugador novato que invierte su primera moneda como al experimentado que continúa desafiándose en el juego. Como él mismo decía: "Todos los grandes juegos deben ser fáciles de aprender y difíciles de dominar. Deben recompensar la primera moneda y la centésima"

2. **Inicio Inmediato del Juego:**

Otro principio crucial es la rapidez con la que el jugador puede sumergirse en la acción. Nolan Bushnell estableció la directriz de que el tiempo dedicado a instrucciones, secuencias de corte (cut-scenes), y menús previos al inicio de una partida no debería exceder el 5% del tiempo promedio de juego. Dado que, en los arcades de los años 80, las partidas solían durar alrededor de tres minutos, esto implicaba que las introducciones e instrucciones no debían ocupar más de 9-10 segundos. Este enfoque buscaba mantener la atención del jugador y proporcionar una experiencia de juego inmediata y envolvente.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

GDD

GDD son las siglas de “Game Design Document” o, en español, “Documento de Diseño de Videojuego”. Es un documento integral que sirve como guía y referencia clave durante el proceso de desarrollo de un videojuego.
En nuestro caso podría ser:

Título: PONG **Plataforma:** PC **Género:** Arcade, Deportes, Simulación **Cámara:** Vista cenital en juego 2D **Descripción:** Juego para dos jugadores que simula una partida de Ping Pong. Cada jugador maneja una pala intentando devolver la pelota al campo contrario para ganar puntos. Ganará el partido el jugador que más puntos consiga.

Mecánicas

- **Control de Palas:** Cada jugador maneja su pala de manera vertical.
- **Movimiento Continuo de la Pelota:** La pelota está en constante movimiento, rebotando sobre las palas y los laterales superior e inferior que delimitan la zona de juego.
- **Puntuación por Toque en la Pared Lateral:** Alcanzar la pared lateral del campo del jugador contrario suma un punto al adversario.
- **Relanzamiento de la Pelota:** Al inicio y después de cada punto, la pelota se relanza desde el centro del campo con posición vertical y ángulo aleatorios, apuntando al perdedor del punto.
- **Primer Saque Aleatorio:** El sentido del primer saque se determina de manera aleatoria.
- **Visualización de Puntuaciones:** El juego muestra continuamente las puntuaciones actuales de los jugadores.
- **Condición de Fin de Juego:** La partida concluye cuando uno de los jugadores alcanza 10 puntos.
- **Ganador de la Partida:** El jugador que llega primero a los 10 puntos se proclama como el ganador de la partida.
- **Reinicio del Juego:** Se establecerá un mecanismo para reiniciar la partida.
- **Efectos Sonoros Integrados:** Se reproducirán sonidos tanto en los rebotes como al conseguir puntos, añadiendo elementos auditivos a la experiencia de juego.

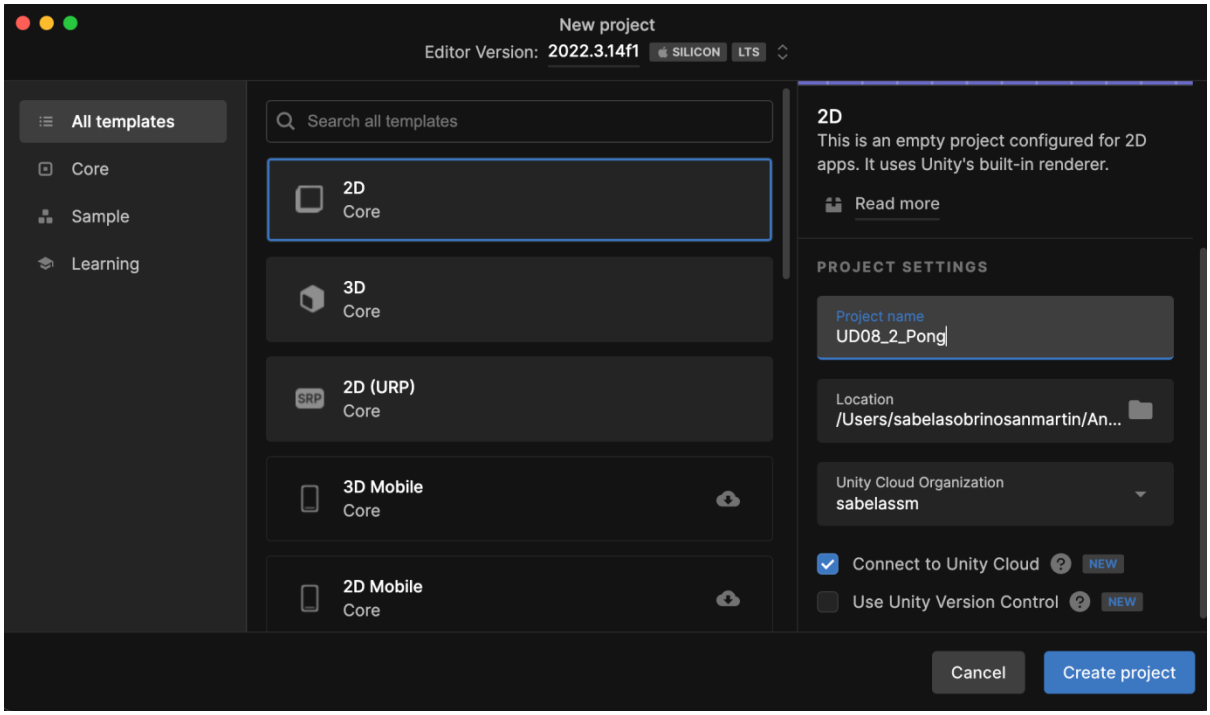
Autor/a: Sabela Sobrino Última actualización: 08.02.2024

Capitulo 2

PROYECTO

PROYECTO

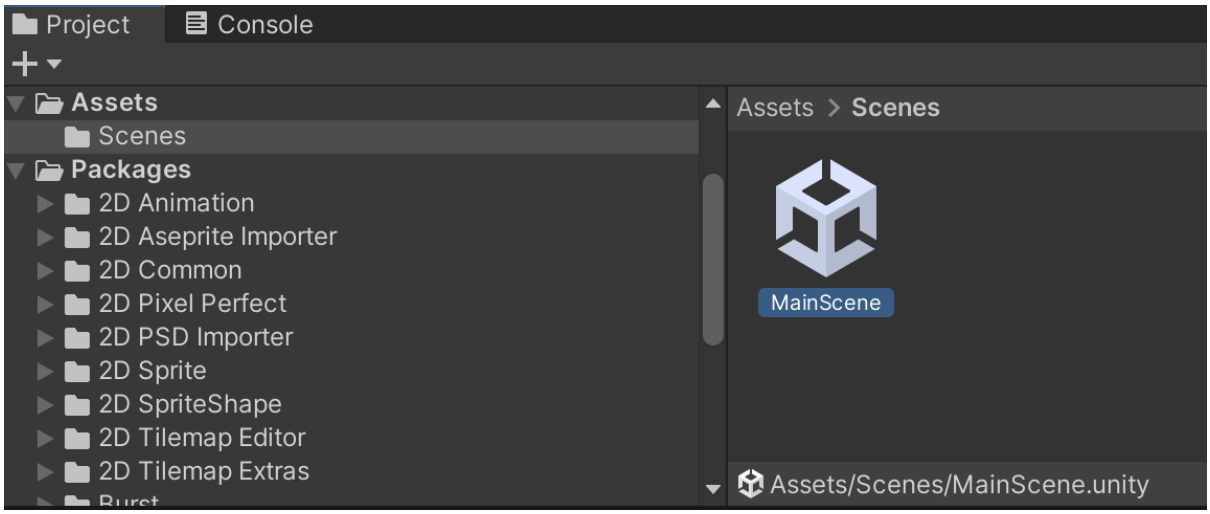
Para iniciar nuestro proyecto en Unity Hub, seleccionaremos la plantilla “2D Core” y le asignaremos el nombre “UD08_2_Pong”, junto con la ubicación deseada para su almacenamiento.



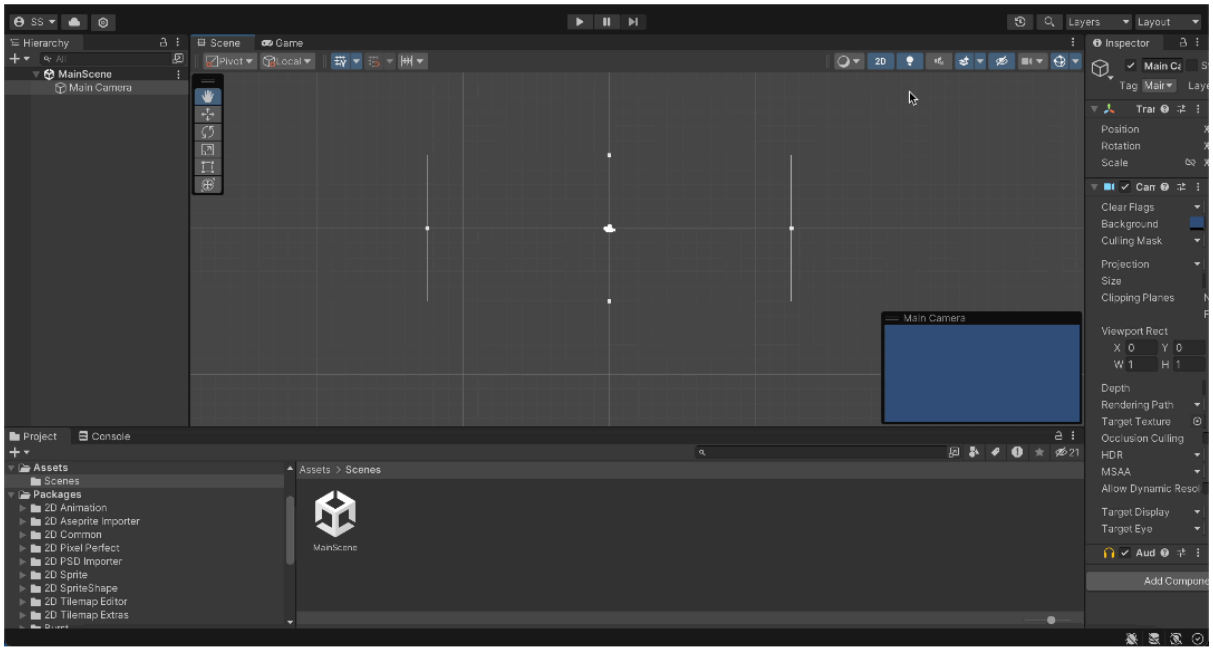
Una vez iniciado el proyecto, ajustaremos las diversas pantallas según nuestras preferencias.

Cámara y Proyección

En la carpeta “Escenas” de la sección “Assets”, encontraremos una única escena que renombraremos como “MainScene”.



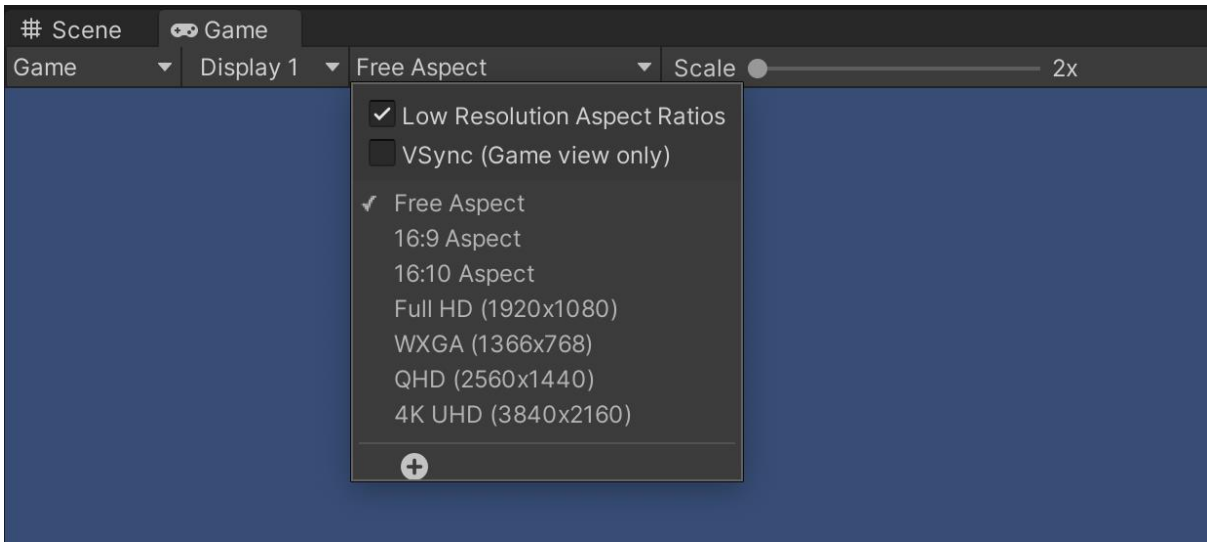
Es importante tener en cuenta que, aunque nuestro proyecto esté etiquetado como 2D, Unity opera en un motor 3D. Si seleccionamos la cámara y pulsamos el botón “2D”, desactivando esta vista, descubrimos que la escena es tridimensional. Esto nos permite ajustar los planos de corte del volumen capturado por la cámara.



Todo lo contenido dentro de este volumen será renderizado por la cámara.

Previsualización

En relación con la previsualización del juego, encontramos un menú desplegable en la pestaña “Game” que nos permite seleccionar diversas opciones. Dependiendo de la plataforma para la cual estemos desarrollando, podemos ajustar la apariencia a diferentes configuraciones, como Full HD, 4K, entre otras.



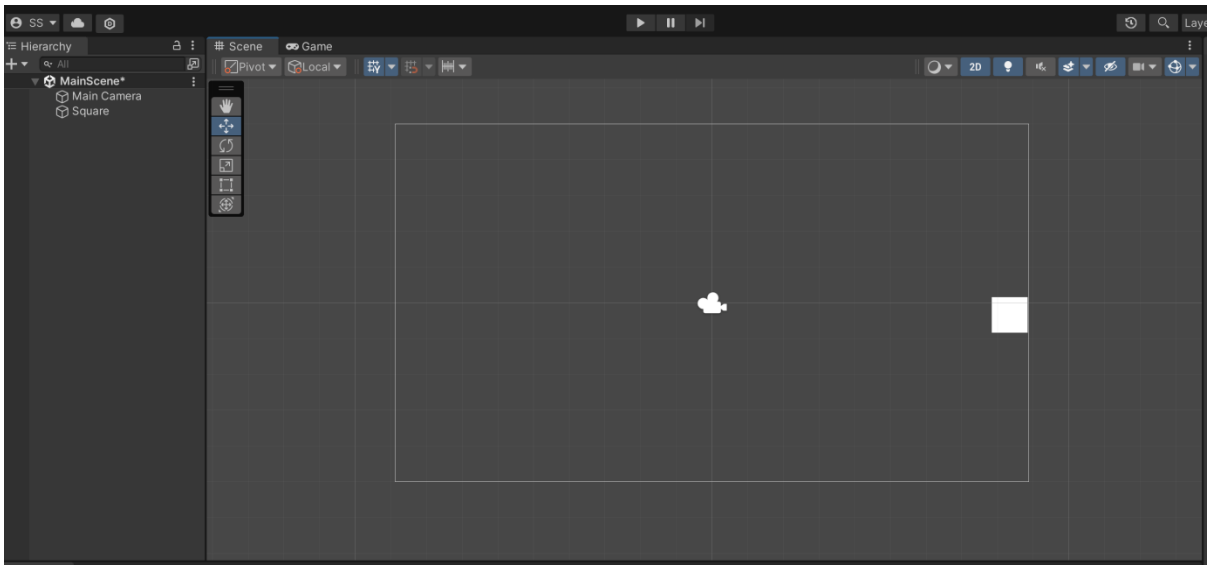
En este caso, optaremos por la opción 16:9 para la previsualización de nuestro juego. Con estos pasos, establecemos las bases para comenzar el desarrollo de nuestro proyecto Pong en Unity.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

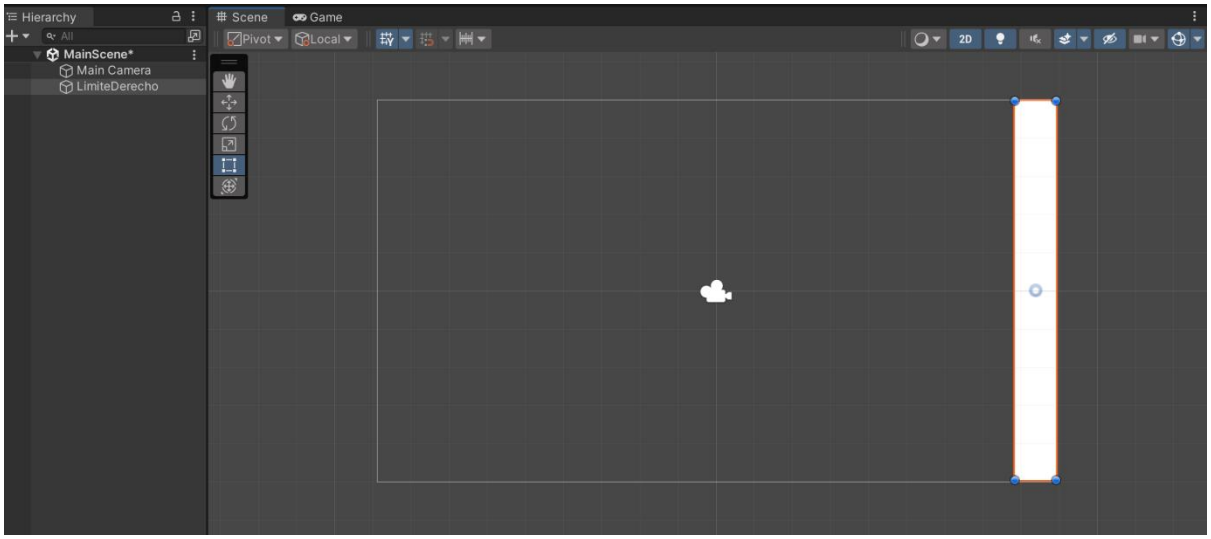
SUBSECCIONES DE PROYECTO

ZONA DE JUEGO

Comencemos creando la zona de juego donde se desplazarán las palas y se moverá la pelota. Dirígete a la jerarquía de objetos y selecciona un “Cuadrado” de la sección de objetos 2D:



Este cuadrado será nuestro límite lateral derecho. Cambiemos su nombre a “limiteDerecho” y ajustemos sus dimensiones para que se convierta en un lateral de nuestro juego:



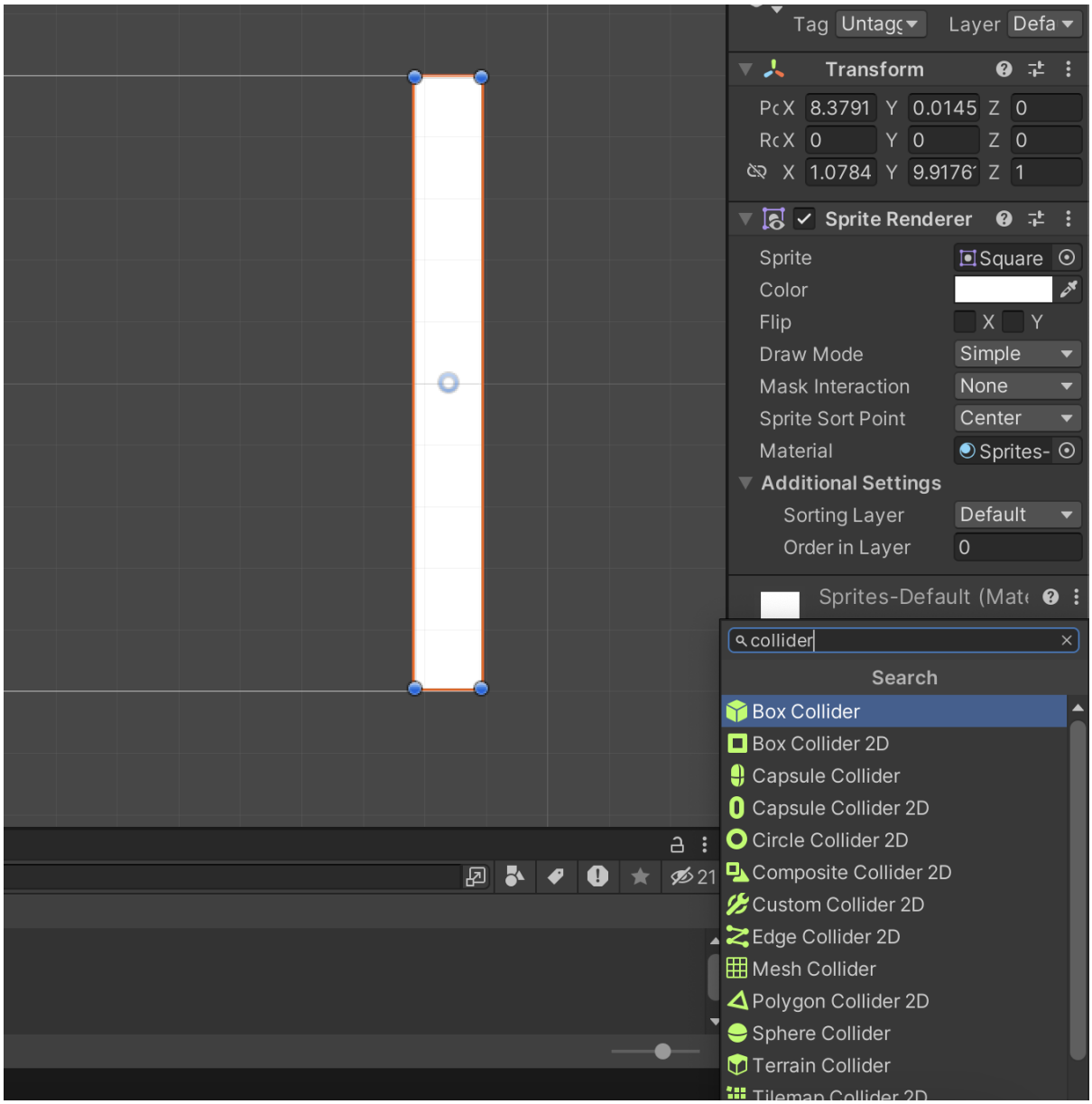
En este momento, la alineación precisa no es crítica; podremos realizar ajustes posteriormente. De hecho, en el juego original, estos límites no estaban predefinidos. Sin embargo, en el desarrollo de videojuegos, es común crear objetos tridimensionales que más adelante serán refinados por el equipo de arte para generar nuevas texturas y mejorar la estética del juego.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

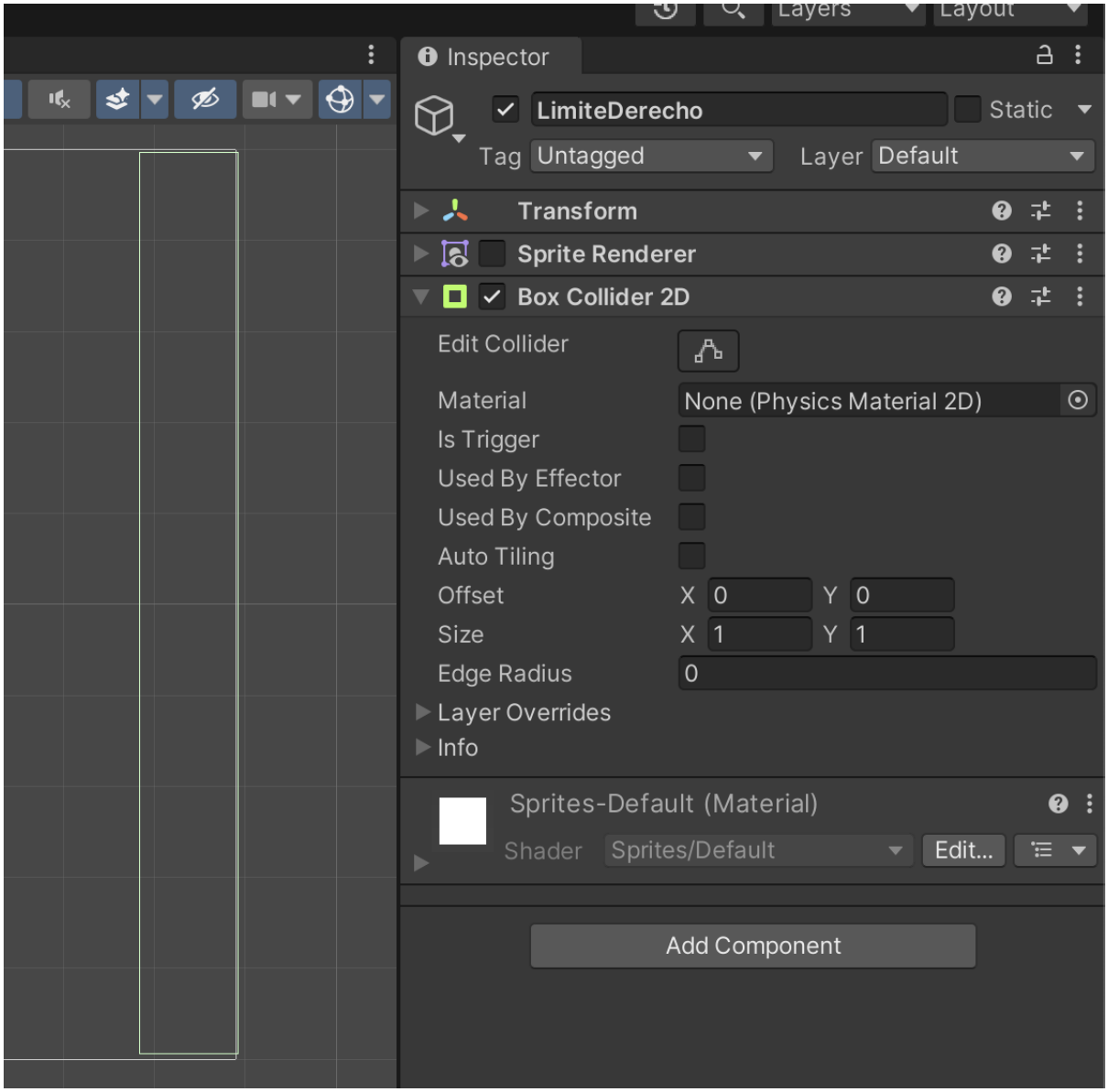
COLLIDER 2D

Generaremos un nuevo componente que permitirá al motor **detectar** cuando la pelota interactúa con las áreas designadas. Para habilitar la detección de **colisiones**, necesitamos agregar a estos componentes un *"collider"*, el cual establecerá una región que el motor utilizará para identificar dichas colisiones.

Primero, seleccionaremos el objeto y, al hacer clic en "Añadir componente", se desplegará un menú. En el buscador, ingresaremos "collider":

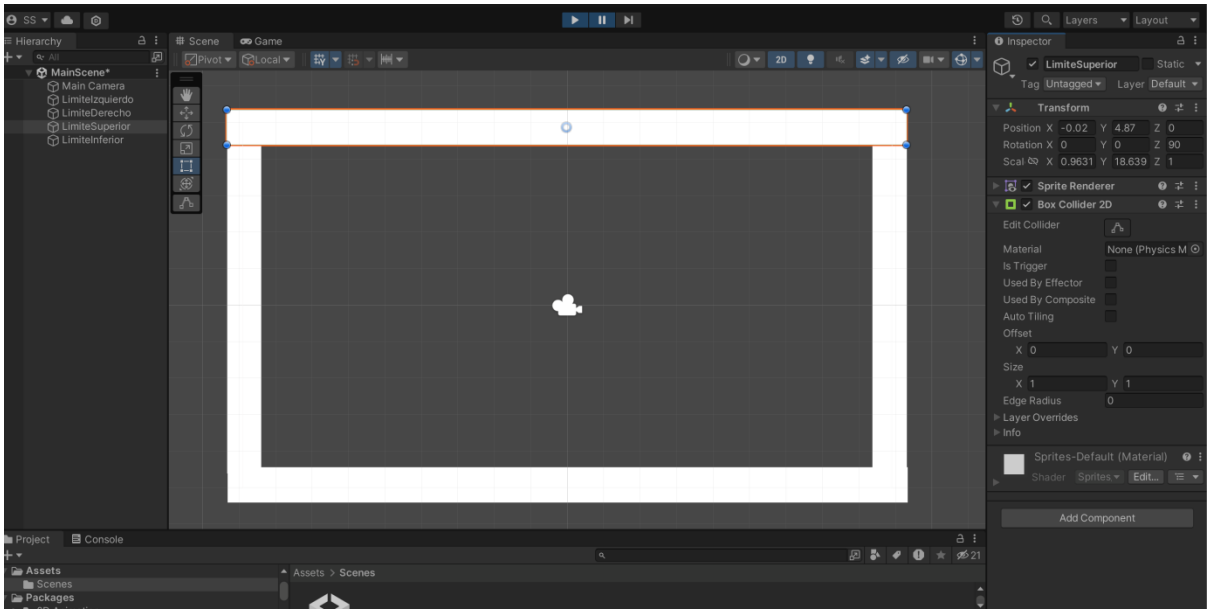


Dentro de las opciones disponibles, seleccionaremos "Box Collider 2D" para nuestro ejemplo. Si desactivamos el "Sprite Render", observaremos una línea verde que se ajusta al tamaño de la figura:

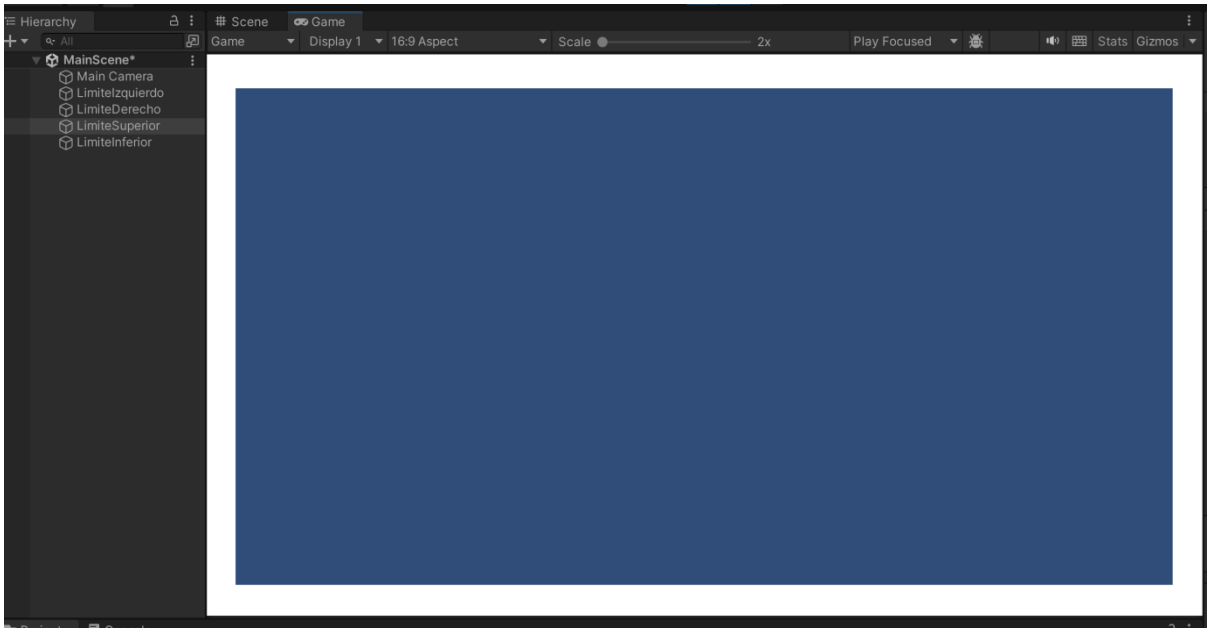


El motor detectará las colisiones entre los colliders de diferentes objetos. En el componente, encontraremos el botón “Editar Collider”, que nos permite ajustar la zona de detección de colisiones. En este caso, queremos que sea el propio objeto.

Importante: Replicaremos este proceso tres veces para crear los objetos “LimiteIzquierdo”, “LimiteSuperior” y “LimiteInferior”. Para los límites superior e inferior, los rotaremos 90 grados sobre el eje Z.



En la ventana de juego, observaremos el siguiente esquema:



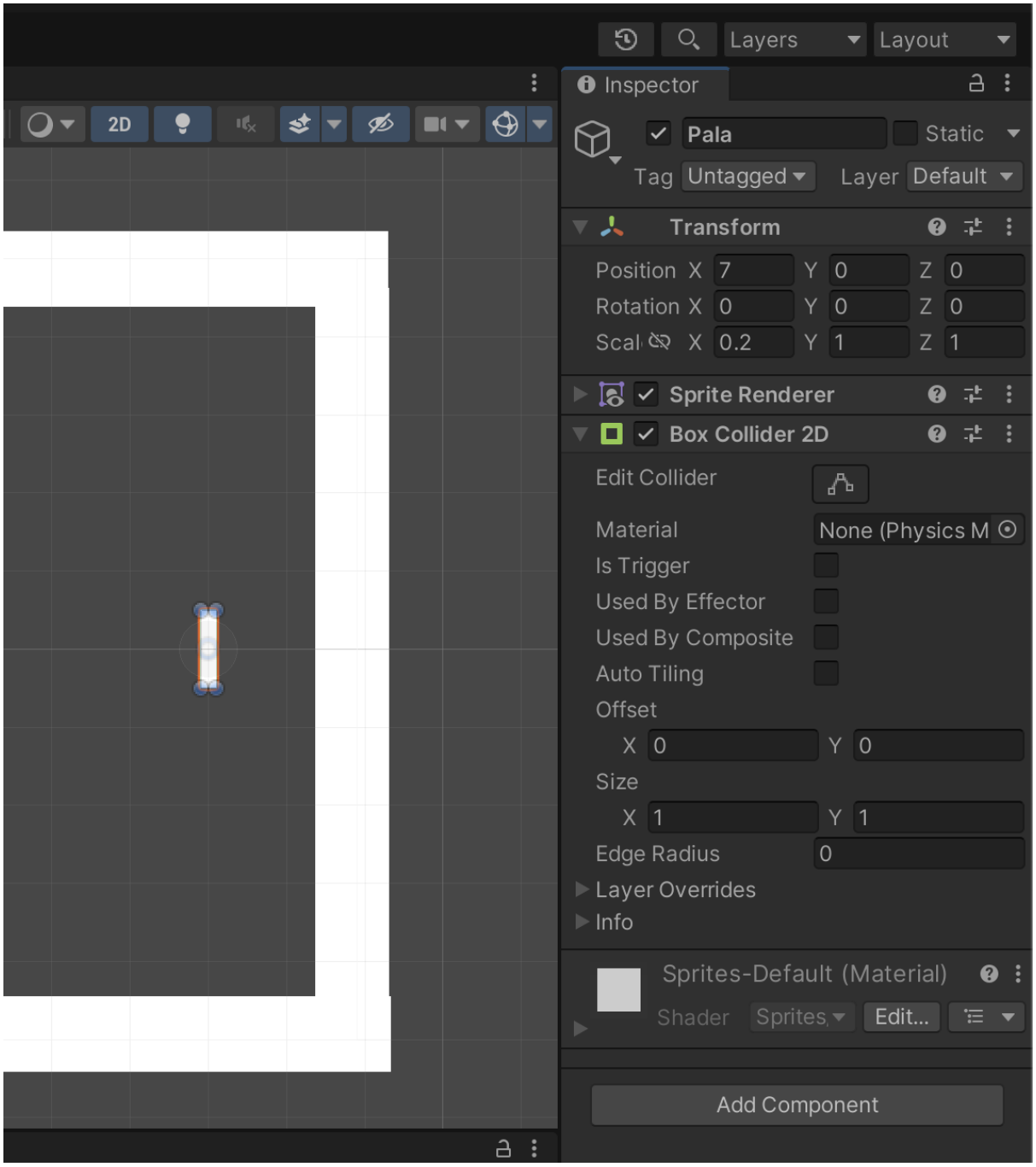
A continuación, agruparemos los elementos actuales en un objeto vacío denominado “Zona de juego”. Restableceremos su posición (Transform - reset) y colocaremos todos los elementos dentro de este objeto:

PALA

PALA

Vamos a incorporar una pala en nuestro juego. Para lograr esto, añadiremos un nuevo cuadrado y lo posicionaremos en el eje X a 7 y en el eje Y a 0. Además, ajustaremos sus dimensiones a X: 0.2 y Y: 1.

Observamos un patrón de cuadrícula en el fondo, que avanza en incrementos de una unidad, lo que nos permite verificar que las dimensiones se ajustan correctamente.



Al igual que en el caso de los límites, la pala también experimentará colisiones. Por lo tanto, necesitamos agregar un “collider” para que el motor identifique cuando la pala colisiona con la pelota.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

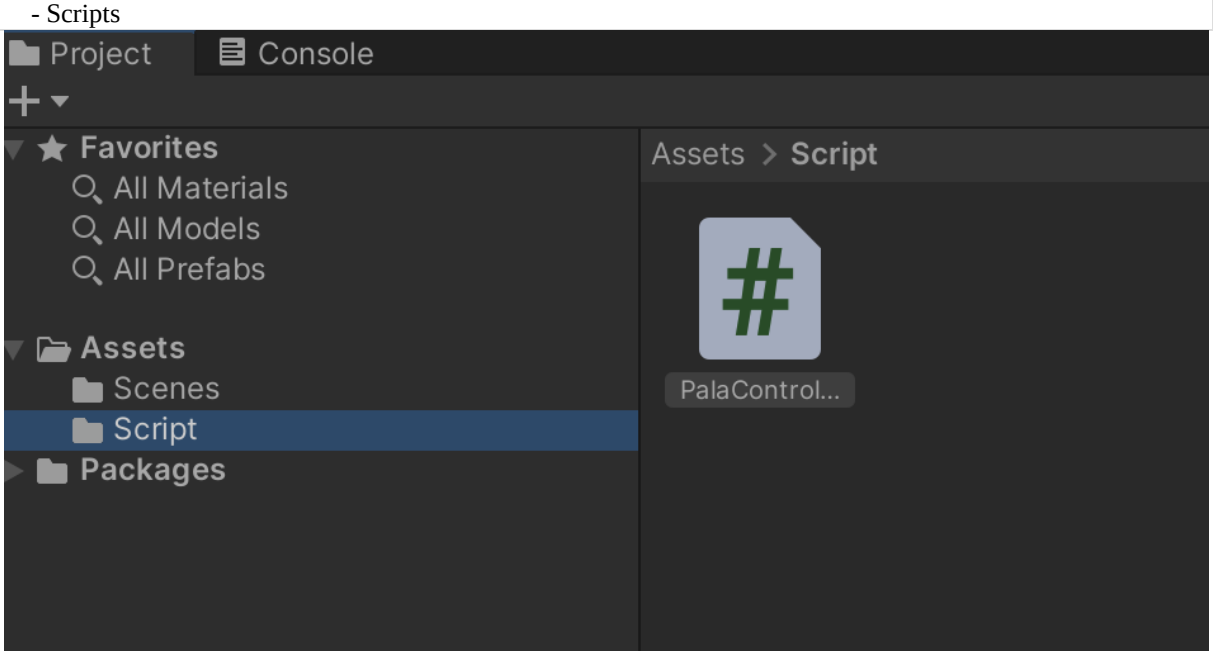
SUBSECCIONES DE PALA

CONTROLADOR

Vamos a implementar la capacidad de respuesta de la pala a las acciones del usuario, que en este caso serán las flechas **arriba y abajo** del teclado. Para ello, necesitaremos crear un script que maneje la lógica de este comportamiento.

Comencemos organizando nuestros scripts. Crearemos una **carpeta** dentro de la sección “Assets” llamada “Scripts”:

- Assets
  - Scripts



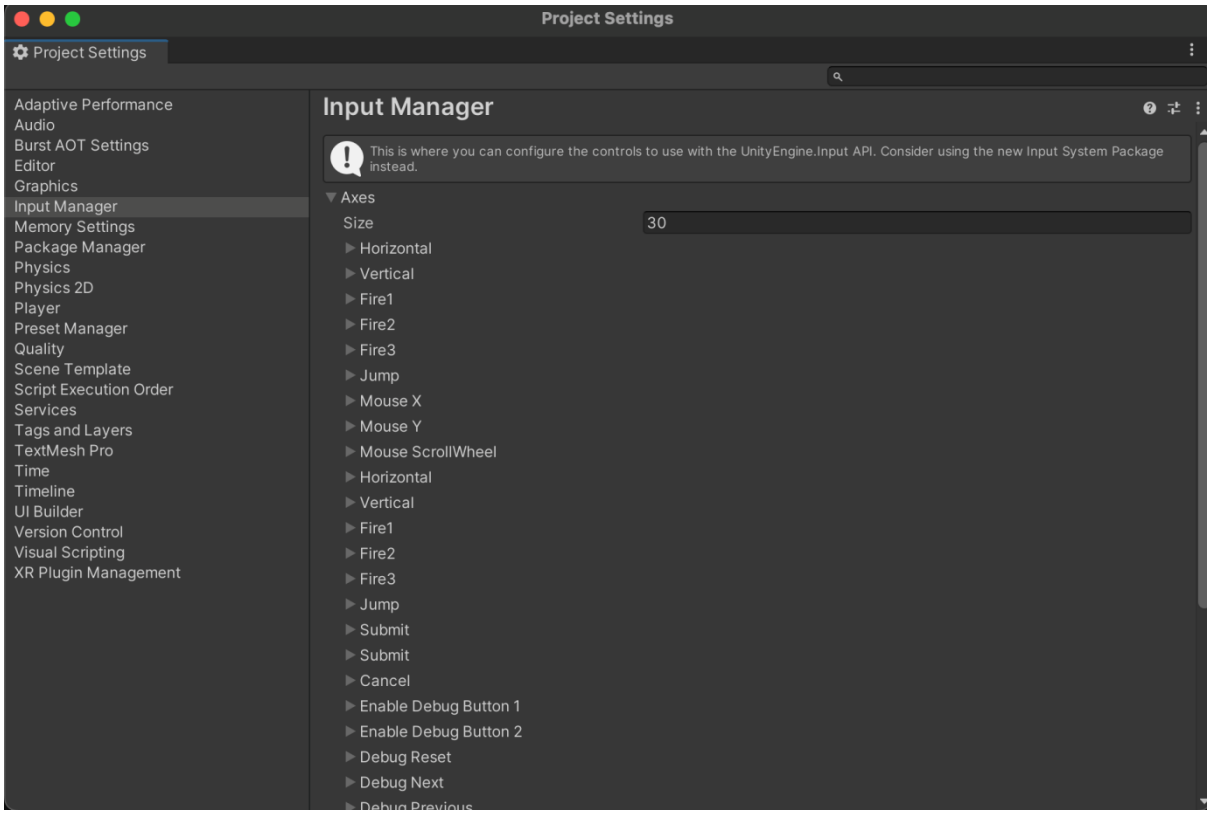
Dentro de esta carpeta, generaremos un nuevo script al que llamaremos “PalaController”. Para lograr que nuestra pala responda a los eventos del usuario, implementaremos la lógica dentro del método `Update()`, asegurándonos de que el script esté constantemente verificando si se presiona alguna tecla.

Unity ya proporciona funciones dentro de su API que nos permiten gestionar estas acciones. Utilizaremos el objeto `Input`, que tiene un método llamado `GetKey` que indica qué tecla ha sido presionada. En este caso, para verificar la tecla de arriba, utilizaremos el código “up”. Si la tecla está siendo presionada, el método nos devolverá un valor verdadero.

```
void Update()
{
    if (Input.GetKey("up")) {
        // Lógica para movimiento hacia arriba
```

```
    }
    if (Input.GetKey("down")){
        // Lógica para movimiento hacia abajo
    }
}
```

Dentro de la ventana “Project Settings”, encontramos la configuración del “Input Manager”:

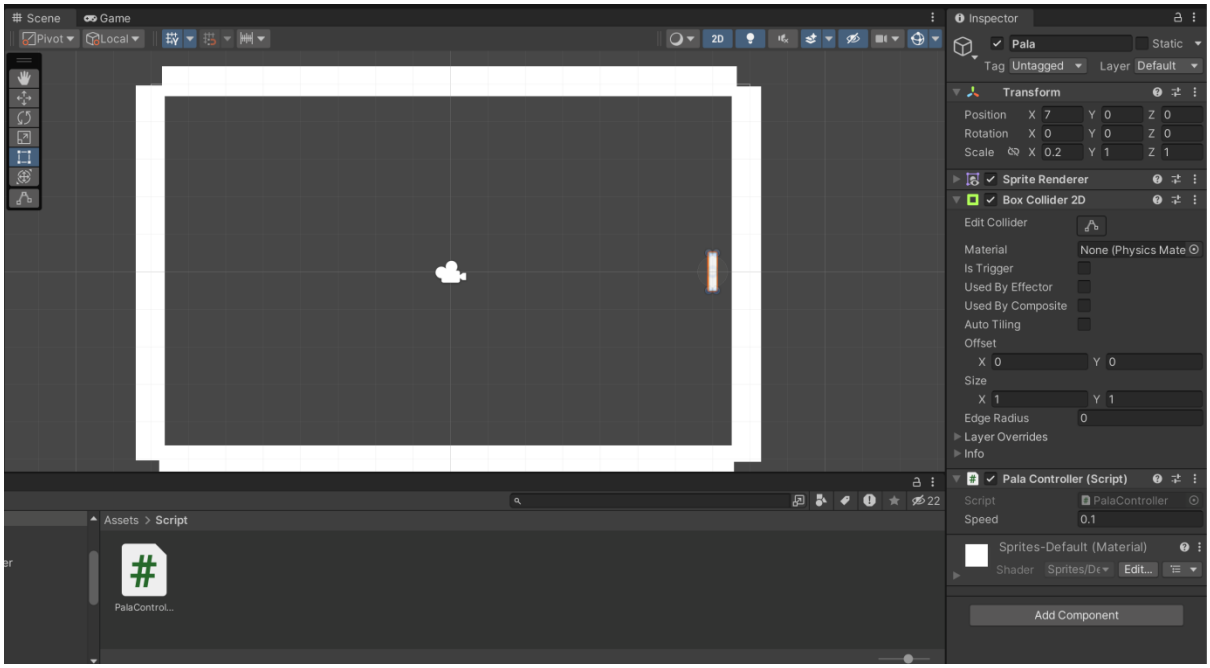


Aquí podemos visualizar la **configuración estándar** para las diferentes teclas, y aunque podríamos ajustar estos parámetros, por ahora, dejaremos la configuración predeterminada para mantener la familiaridad del usuario.

Continuando con el movimiento de la pala, queremos que esta se desplace dentro de nuestro juego. Para ello, debemos consultar la API de Unity y acceder al componente `Transform` para modificar su posición en el eje X. Utilizaremos la propiedad `Translate` para realizar este desplazamiento.

```
void Update()
{
    if (Input.GetKey("up") && transform.position.y < MAX_Y) {
        // Desplazamiento hacia arriba
        transform.Translate(Vector3.up * speed);
    }
    if (Input.GetKey("down") && transform.position.y > MIN_Y) {
        // Desplazamiento hacia abajo
        transform.Translate(Vector3.down * speed);
    }
}
```

Asegurémonos de que este script esté asociado a un objeto en el juego, y ahora, al ejecutar el juego, observamos que la pala se mueve. Sin embargo, aún no respeta los límites del juego.



Este comportamiento se debe a que hemos creado el script pero no lo hemos vinculado a ningún objeto. Necesitamos arrastrar el script hasta el objeto correspondiente.

LIMITAR MOVIMIENTO

Para restringir el **movimiento de la pala** y evitar que se salga de la zona de juego, podemos aprovechar la funcionalidad de los ***colliders***. Al agregar un componente llamado "***Rigidbody***" al objeto, activamos el motor de física, permitiendo que se verifique constantemente si estamos colisionando con otros objetos en la escena y aplicando las leyes físicas correspondientes. Esto asegura que un objeto **sólido no atraviese otro objeto sólido**, lo cual sería útil para nuestras palas.

Sin embargo, para un juego tan simple, utilizar el motor de física puede resultar un enfoque un tanto excesivo. Podemos optar por un método más directo, simplemente controlando que la pala no supere ciertas **posiciones** tanto en el extremo superior como en el inferior.

Durante la ejecución, podemos observar cómo varían los distintos atributos y las posiciones. Estos valores podrían servirnos para establecer límites en el movimiento de la pala. En nuestro código, definiremos dos constantes:

```
public class PalaController : MonoBehaviour
{
    const float MAX_Y = 4.2f;
    const float MIN_Y = -4.2f;
```

Luego, en las condiciones para las teclas de arriba y abajo, comprobaremos que no excedemos los límites superior e inferior respectivamente:

```
void Update()
{
    if (Input.GetKey("up") && transform.position.y < MAX_Y) {
```

```
        // Movimiento hacia arriba
    }
    if (Input.GetKey("down") && transform.position.y > MIN_Y) {
        // Movimiento hacia abajo
    }
}
```

Estas condiciones aseguran que la pala solo se moverá hacia arriba si no ha alcanzado el límite superior (`MAX_Y`) y hacia abajo si no ha llegado al límite inferior (`MIN_Y`). Este enfoque más directo evita el uso excesivo del motor de física para un movimiento simple en nuestro juego.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SERIALIZEFIELD

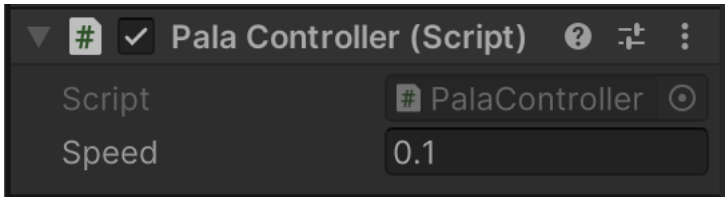
Evitemos el uso de literales en el código, por lo que definiremos una nueva constante para determinar el movimiento de la pala:

```
public class PalaController : MonoBehaviour
{
    const float MaxY = 4.2f;
    const float MinY = -4.2f;
    [SerializeField] float speed = 0.1f;
    // ...
}
```

Estos atributos definidos en el script ahora pueden aparecer en el editor de Unity, lo cual es beneficioso ya que podemos observar el valor en tiempo real. Tenemos dos opciones para lograr esto:

- Hacer la variable pública (no recomendado).
- Hacerla accesible desde Unity con `[SerializeField] float speed = 0.1f;`.

Ahora, podremos ver y ajustar la velocidad en tiempo real en Unity:



TASA DE REFRESCO

Cuando desarrollamos un videojuego, es crucial procurar que sea lo más independiente de la plataforma posible. Esto implica asegurar que el juego funcione de manera consistente en diferentes configuraciones de computadoras. La meta principal es garantizar que, aunque la tasa de refresco varíe considerablemente, el movimiento de la pala sea uniforme.

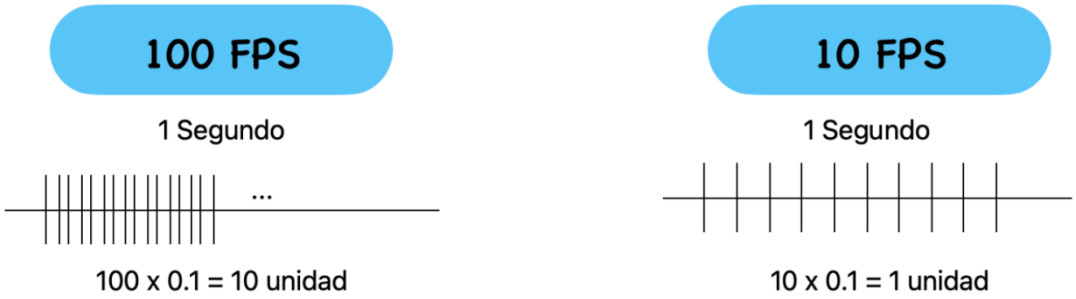
El método `Update` se ejecuta cada vez que se carga un fotograma (similar a una película). Si mantenemos pulsada una tecla, nuestra pala se moverá 0.1 unidades. Sin embargo, en un ordenador más potente que carga más fotogramas por segundo, nuestro código se ejecutará más veces, y en cada ejecución nos moveremos 0.1 unidades. Esto significa que, en el mismo intervalo de tiempo, nos habremos desplazado una distancia mucho

mayor en un sistema más potente. En términos simples, lo que buscamos es que la velocidad de la pala sea consistente, independientemente del rendimiento del ordenador. Claro está, existen límites, especialmente en computadoras muy lentas, pero este enfoque es efectivo en la gran mayoría de los casos.

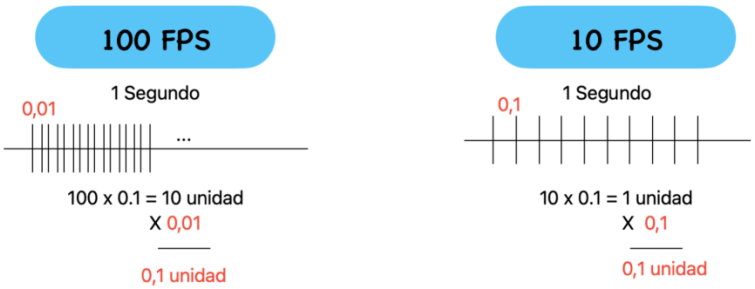
Time.deltaTime

Vamos a emplear una propiedad clave de Unity llamada `deltaTime` (Documentación). Esta propiedad nos proporciona el intervalo en segundos desde el último fotograma hasta el actual, y este valor es dependiente de la tasa de refresco del equipo. En un ordenador más lento, este lapso de tiempo entre dos fotogramas será mayor. Utilizando `deltaTime`, podemos lograr que el movimiento de nuestra pala sea independiente de la velocidad del equipo.

Como hemos visto anteriormente, nos desplazamos 0.1 unidades dentro de nuestro juego. Dependiendo de los fotogramas por segundo que se generen, experimentaremos un desplazamiento mayor o menor:



Al utilizar `Time.deltaTime`, aprovechamos el tiempo que transcurre entre los fotogramas, de modo que en un ordenador más lento este tiempo será mayor, provocando un desplazamiento mayor o menor según la capacidad del equipo:



La diferencia radica en que en cada actualización en un ordenador más rápido, nos desplazaremos menos. Esto se logra multiplicando la unidad de desplazamiento por el delta, asegurando que nos movamos la misma distancia independientemente de la velocidad del ordenador:

```
void Update()
{
    if (Input.GetKey("up") && transform.position.y < MAX_Y) {
        // Desplazamiento hacia arriba
        transform.Translate(Vector3.up * speed * Time.deltaTime);
    }
    if (Input.GetKey("down") && transform.position.y > MIN_Y) {
        // Desplazamiento hacia abajo
        transform.Translate(new Vector3(0, -speed * Time.deltaTime, 0));
    }
}
```

Es posible que necesitemos ajustar la velocidad para que sea un poco más rápida:

```
[SerializeField] float speed = 10f;
```

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

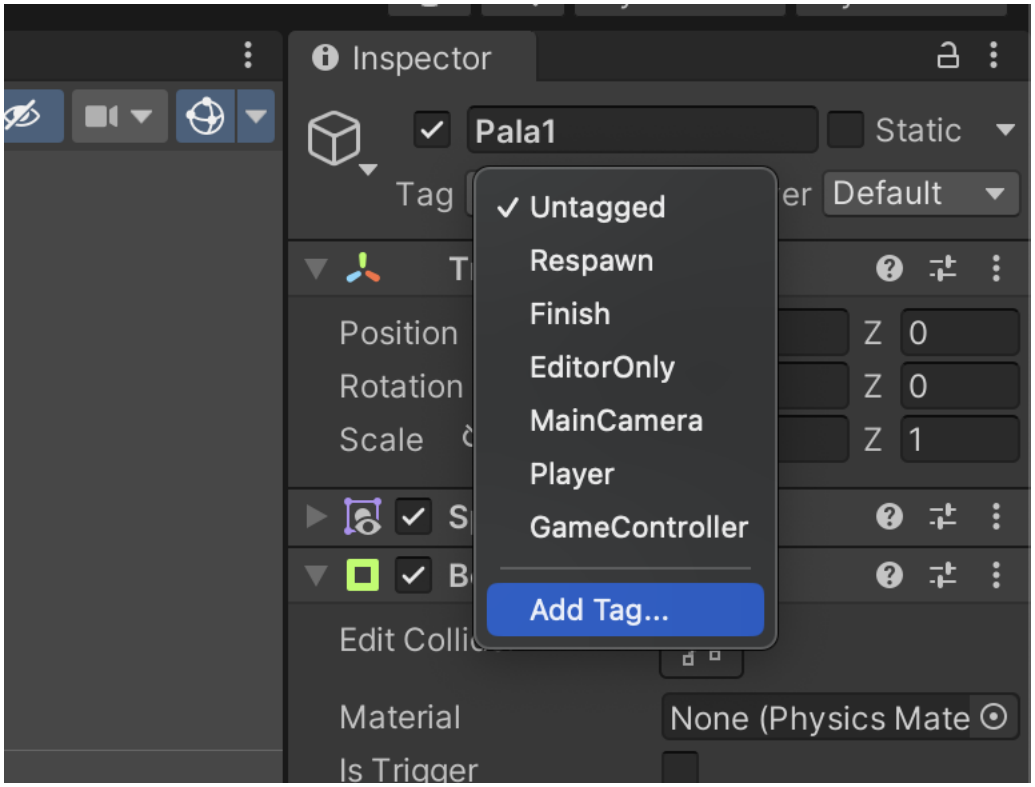
OTRA PALA

Ahora procederemos a agregar la segunda pala para el segundo jugador. Si bien podríamos copiar la pala existente con el script asociado, al ejecutarlo observaríamos que ambas palas se desplazan simultáneamente, ya que comparten el mismo script.

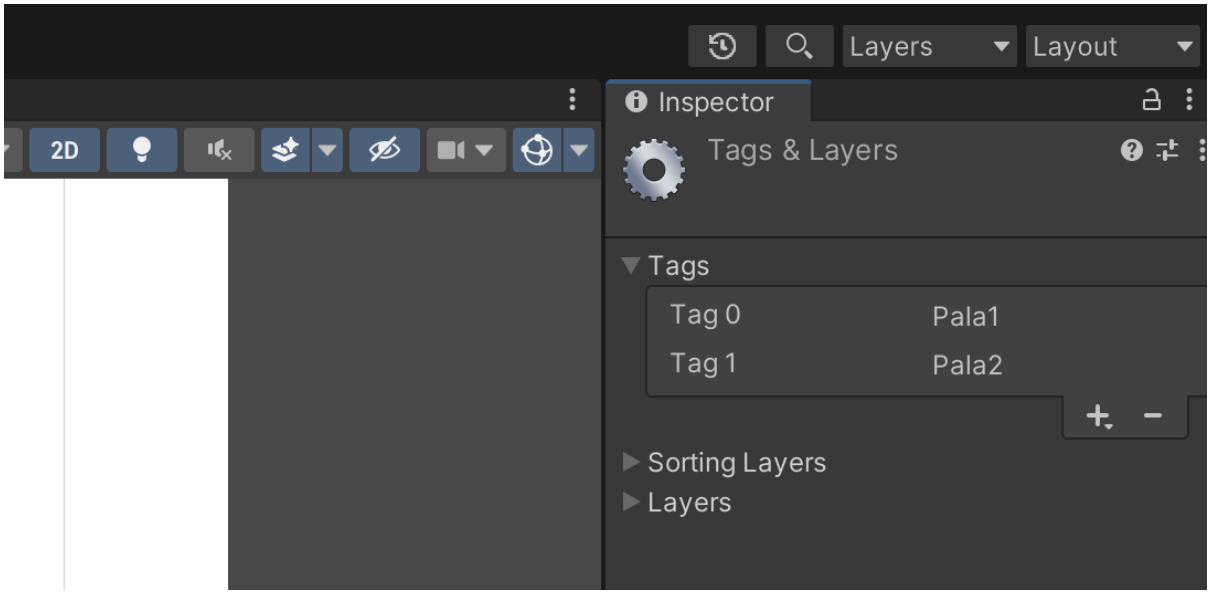
Nuestra intención es que cada pala sea independiente y responda a las teclas correspondientes. Para lograr esto, necesitaremos distinguir qué objeto queremos mover según las teclas presionadas. Haremos que la pala 1 responda a las teclas “w” y “s”, mientras que la pala 2 lo hará con las teclas “up” y “down”.

Desde el propio script, podemos determinar si se está asociando a un objeto u otro. Así, podemos tener un script asociado a varios objetos en nuestra escena, y la clave es saber en qué objeto nos encontramos para ejecutar la acción adecuada.

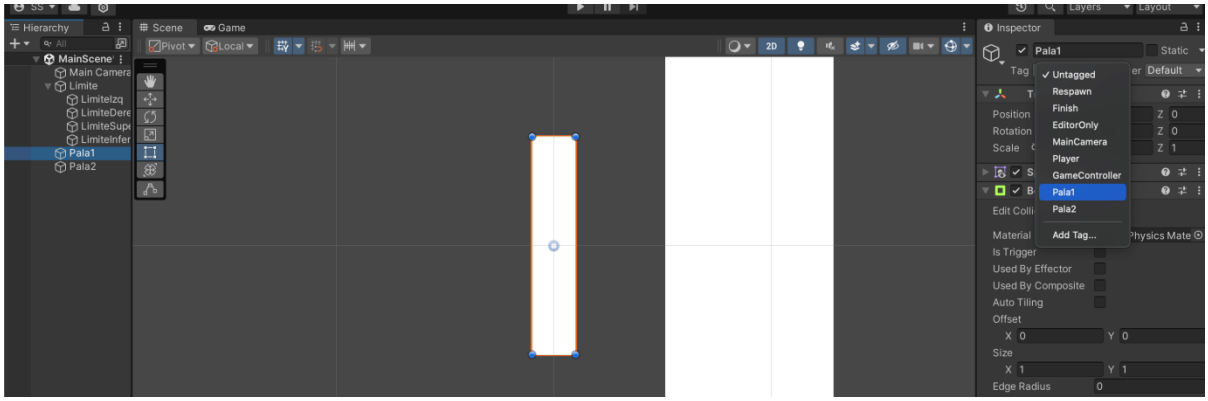
Para lograrlo, añadiremos etiquetas a nuestros objetos:



Aunque hay algunas etiquetas predefinidas, crearemos dos etiquetas personalizadas: “Pala1” y “Pala2”.



Si bien hemos creado las etiquetas, aún no las hemos asociado a ningún objeto. Volvamos a los objetos Pala1 y Pala2 para añadir estas etiquetas.



Ahora, en nuestro código, necesitaremos identificar en qué objeto nos encontramos. Para hacer esto, accederemos a la propiedad `gameObject` (https://docs.unity3d.com/ScriptReference/GameObject.html) y dentro de este objeto, revisaremos la etiqueta:

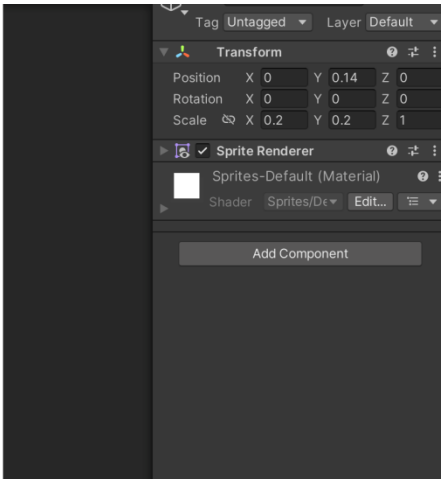
```
void Update()
{
    if (gameObject.CompareTag("Pala2"))
    {
        if (Input.GetKey("up") && transform.position.y < MAX_Y)
        {
            // Movimiento hacia arriba
            transform.Translate(Vector3.up * speed * Time.deltaTime);
        }
        if (Input.GetKey("down") && transform.position.y > MIN_Y)
        {
            // Movimiento hacia abajo
            transform.Translate(new Vector3(0, -speed * Time.deltaTime, 0));
        }
    }
    else if (gameObject.CompareTag("Pala1"))
    {
        if (Input.GetKey("w") && transform.position.y < MAX_Y)
        {
            // Movimiento hacia arriba
            transform.Translate(Vector3.up * speed * Time.deltaTime);
        }
        if (Input.GetKey("s") && transform.position.y > MIN_Y)
        {
            // Movimiento hacia abajo
            transform.Translate(new Vector3(0, -speed * Time.deltaTime, 0));
        }
    }
}
```

Ahora, si estamos en la Pala1, nos moveremos con las teclas “w” y “s”, y si estamos en la Pala2, nos moveremos con las teclas “up” y “down”.

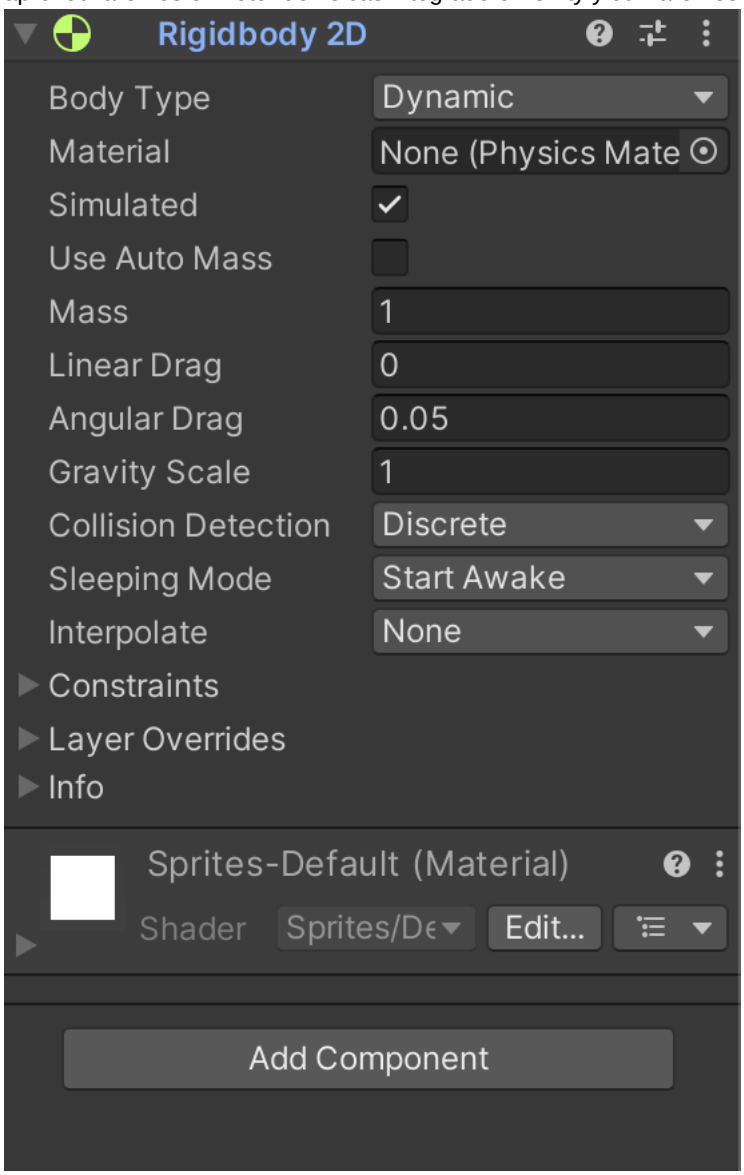
Autor/a: Sabela Sobrino Última actualización: 08.02.2024

PELOTA

Vamos a incorporar ahora la pelota al juego. Para ello, añadiremos un nuevo objeto cuadrado y ajustaremos su tamaño a 0.2 unidades:

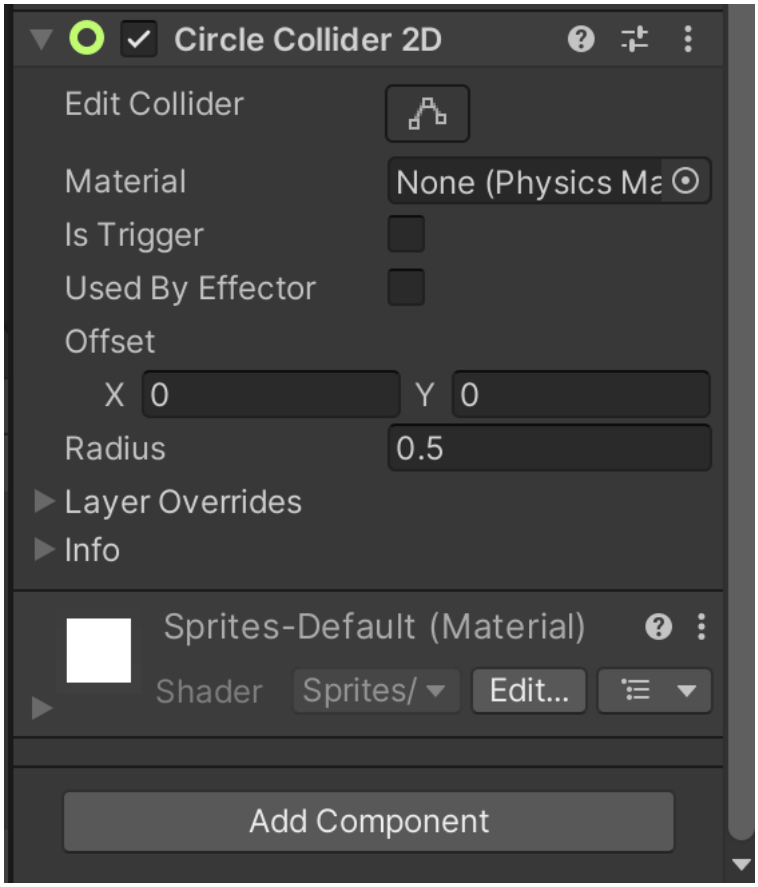


Le daremos el nombre “Pelota” y ahora necesitamos que esta pelota se mueva. A diferencia de las palas, que se desplazan a través de las teclas, queremos que la pelota se mueva libremente. Para lograr esto, aprovecharemos el motor de físicas integrado en Unity y utilizaremos un componente llamado RigidBody2D.



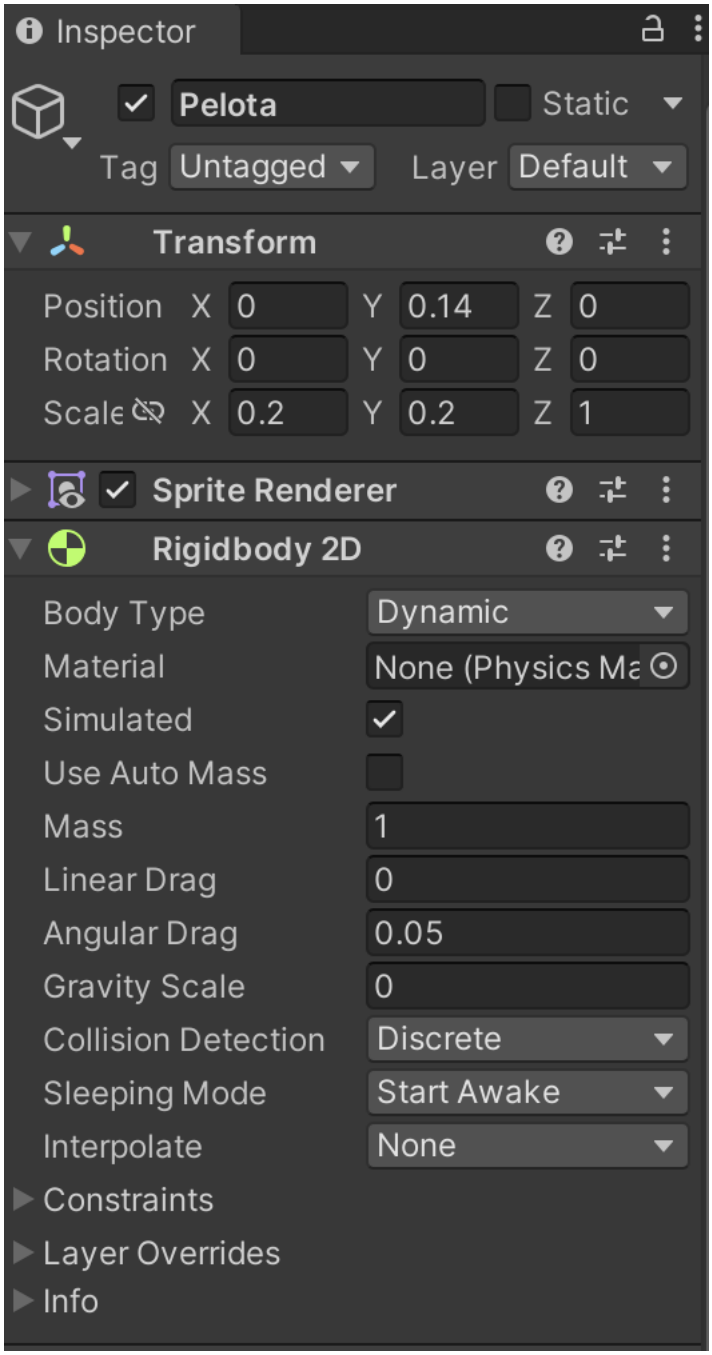
Al asociar este componente, nuestro objeto adquiere masa, gravedad, y otras propiedades físicas.

Como hicimos con las palas, le añadiremos un Collider; podemos utilizar un BoxCollider2D o un CircleCollider2D, dependiendo de la forma que prefiramos para nuestra pelota.



Si ejecutamos el juego ahora, notaremos que la pelota cae hacia abajo debido a que le hemos añadido el componente RigidBody2D, lo que simula la influencia de la gravedad. La pelota se detendrá en el límite inferior gracias al Collider.

Para evitar que la pelota caiga, necesitamos ajustar la influencia de la gravedad. Para ello, estableceremos la propiedad `gravityScale` en 0:



Ahora, al ejecutar el juego, la pelota permanecerá en su posición sin caer.

PelotaController

Para que la pelota se desplace, necesitaremos aplicarle una fuerza. Para lograrlo, generaremos un nuevo script llamado “PelotaController” y lo asignaremos al objeto de la pelota.

En la documentación de Unity sobre Rigidbody (https://docs.unity3d.com/ScriptReference/Rigidbody.html), podemos ver que existe un método llamado `AddForce` que nos permite añadir comportamiento a nuestra pelota. Al igual que con las palas, donde accedimos al componente `Transform` para el desplazamiento, en este caso accederemos al componente `Rigidbody` para aplicarle una fuerza. Aunque no tenemos acceso directo, podemos obtener una referencia a este componente cuando se crea el objeto:

```
Rigidbody2D rb;

void Start()
{
    rb = GetComponent<Rigidbody2D>();
}
```

De esta manera, ya tenemos una referencia al componente `Rigidbody2D`. Ahora, vamos a realizar una prueba y aplicaremos una fuerza al crear el objeto:

```
void Start()
{
    rb = GetComponent<Rigidbody2D>();
    rb.AddForce(new Vector2(1, 1), ForceMode2D.Impulse);
}
```

El primer parámetro indica la dirección en forma de vector (x, y), siendo (1, 1) equivalente a 45 grados. El segundo parámetro especifica el modo de desplazamiento; en este caso, estamos utilizando `Impulse`, que sería como dar un impulso a la pelota.

Al ejecutar, observamos que la pelota se desplaza pero se encuentra con una superficie (el collider) y no puede avanzar:

Modificaremos algunos componentes para cambiar este comportamiento.

1. **Fuerza**

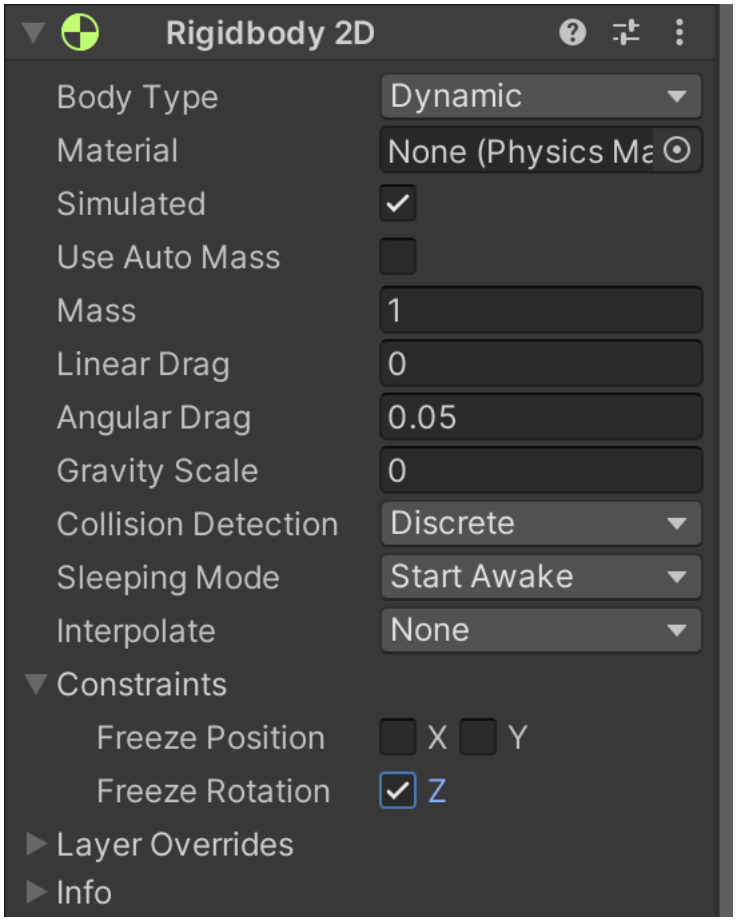
Aplicaremos una fuerza sobre la pelota:

```
[SerializeField] float force;

void Start()
{
    rb = GetComponent<Rigidbody2D>();
    rb.AddForce(new Vector2(1, 1) * force, ForceMode2D.Impulse);
}
```

De esta manera, tendremos un atributo que nos permitirá regular la velocidad. Posteriormente, lo cambiaremos para que sea aleatorio.

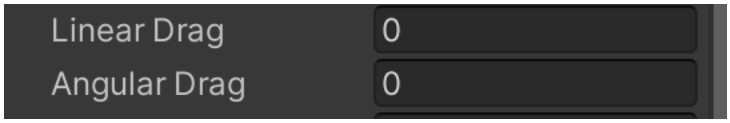
2. **Rotación**

Evitaremos que la pelota rote sobre el eje z para que no quede bloqueada en los laterales. Bloquearemos la rotación en el eje z:



3. **Rozamiento**

Eliminaremos todo tipo de rozamiento:

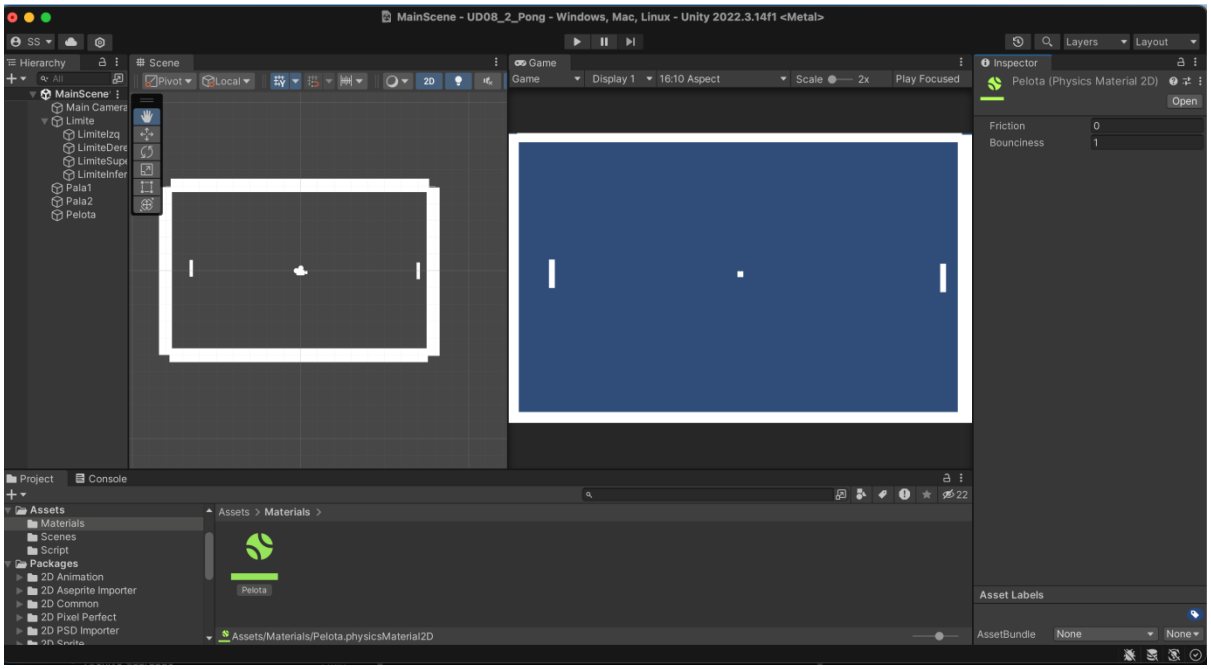


4. **Masa**

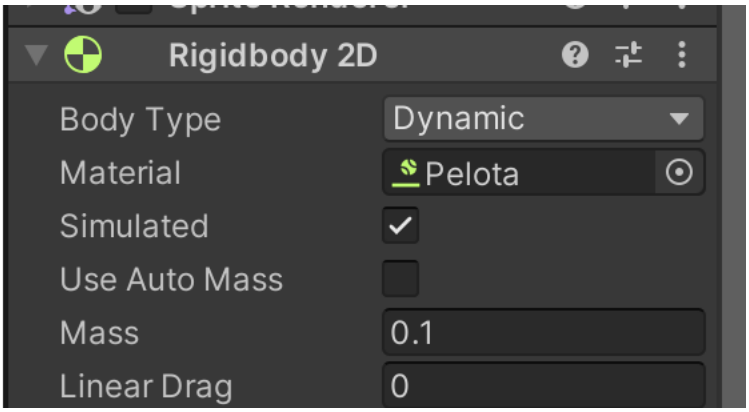
Reduciremos la masa con un valor mucho más pequeño:

5. **Rebote de la pelota**

Para controlar el rebote de la pelota, crearemos un material físico especial y se lo asignaremos a nuestra pelota. Dentro de la carpeta “Assets”, crearemos una carpeta llamada “Materials” y dentro de ella, otra llamada “2D” > “PhysicsMaterial2D”. Le daremos el nombre “Pelota”. Dentro de este componente, ajustaremos las propiedades de fricción y rebote a 1 y 0, respectivamente:



Volviendo a nuestro objeto y a la propiedad Rigidbody, encontramos un campo para el material que actualmente está vacío. Simplemente, arrastraremos nuestro componente al campo correspondiente:



Con esto, el material quedará asignado correctamente. Si ejecutamos el juego ahora, veremos cómo la pelota rebota por nuestra zona de juego:

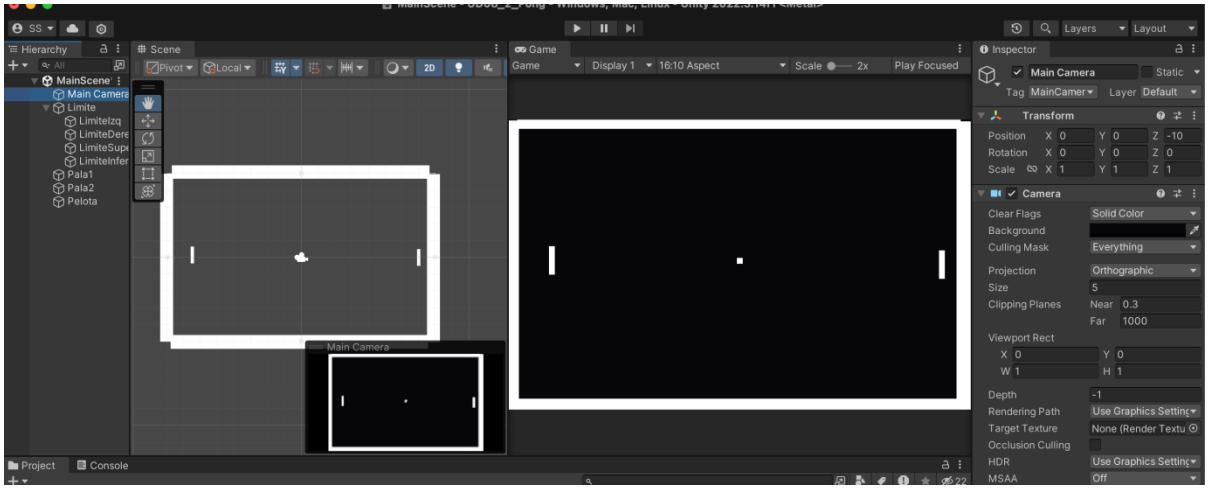
Not found

OTROS AJUSTES

Vamos a realizar algunos ajustes para que el juego se asemeje más al original:

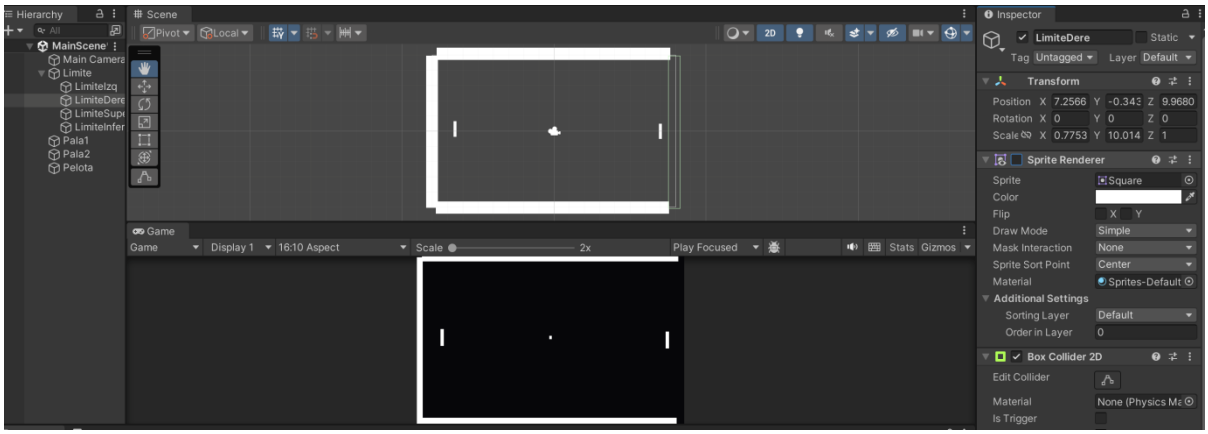
1. **Color de fondo**

Cambiamos el color a negro, que es el color original. Para hacerlo, nos dirigimos a la cámara y modificamos la propiedad “Background”:



2. **Ocultar los límites**

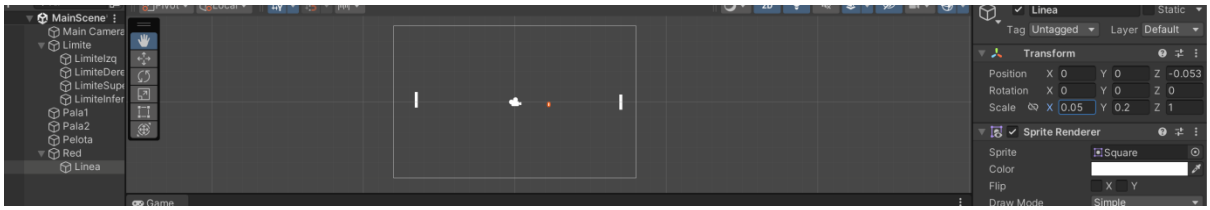
En el juego original, las paredes no existen (son blancas), por lo que las ocultaremos. Cada límite tiene un componente de renderizado encargado de pintarlo. Para ocultarlos, simplemente desactivamos ese componente:



Este ajuste se realiza en todos los límites. Cabe destacar que estamos dejando de mostrarlos, pero los colliders siguen existiendo.

3. **Red**

Añadimos una red utilizando una serie de rectángulos. Creamos un nuevo objeto llamado “Red” y dentro de él generamos rectángulos con tamaños aproximados de 0.05 y 0.2:



A partir de este componente, creamos diferentes elementos para construir la red.

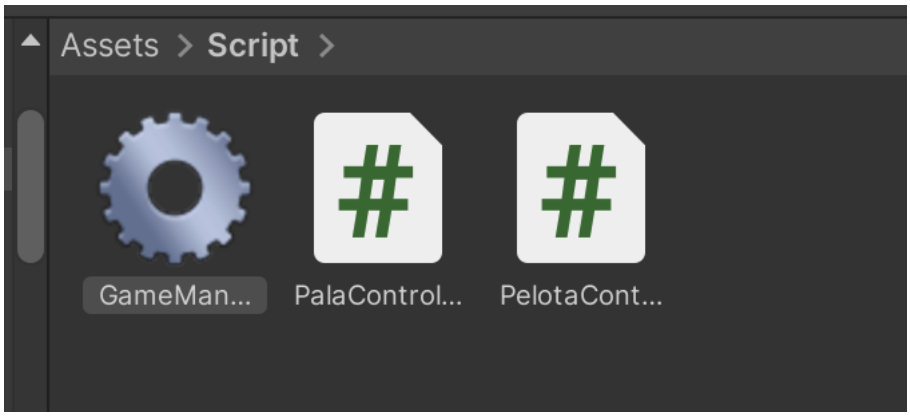
Autor/a: Sabela Sobrino Última actualización: 08.02.2024

Capitulo 3

GAME MANAGER

GAME MANAGER

Vamos a implementar la lógica del juego y las puntuaciones. Aunque la solución que vamos a presentar no es la única posible, crearemos un script que actúe como un controlador general del juego, encargado de calcular y mostrar las puntuaciones.



Al crear el script, notamos que tiene un icono diferente debido al nombre “Manager”. Para que el script se ejecute, debe estar asociado a algún objeto de la escena. En este caso, crearemos un objeto vacío llamado “GameManager” y asignaremos el script a este objeto. La posición del objeto no importa, ya que no tendrá ninguna visualización. Aunque podríamos haberlo asignado a un objeto existente, como la cámara, optamos por generar un nuevo objeto para mantener la coherencia. La clave es que el objeto permanezca siempre en la escena.

Ahora, editaremos nuestro script agregando dos atributos que nos permitan llevar el control de las puntuaciones de los jugadores:

```
public class GameManager : MonoBehaviour
{
    int p1Score;
    int p2Score;

    public void AddPointP1() { p1Score++; }
    public void AddPointP2() { p2Score++; }
}
```

Además, hemos añadido dos métodos (`AddPointP1` y `AddPointP2`) que nos permitirán incrementar las puntuaciones de los jugadores, encapsulando las propiedades.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SUBSECCIONES DE GAME MANAGER

COLISIONES

Cuando creamos los diversos objetos en nuestra escena, como paredes y pelotas, les asignamos a cada una de estas geometrías un ***collider***, una especie de superficie que rodea el objeto y se utiliza para simular las **colisiones**. Además, a nuestra pelota le hemos asignado un **RigidBody** para que, al chocar, actúe en consecuencia, aplicando fuerzas y rebotes según sea necesario.

Para que desde el script podamos controlar situaciones en las que un objeto choca con otro, el motor de Unity lanza una serie de eventos que capturan esas colisiones. Cuando la pelota choca contra una pared, se desencadena un evento `CollisionEnter`, y cuando sale de la colisión, un evento `CollisionExit`.

En la documentación de la API de la clase MonoBehaviour, que es la clase padre de todos estos objetos que estamos creando, podemos observar que tiene definidos una serie de métodos que se invocan en asociación con ciertos eventos:

Aquí podemos ver diferentes eventos asociados a las colisiones. Todos estos eventos tienen un parámetro que proporciona información sobre el objeto con el que estamos colisionando.

Detección de colisiones

Vamos a abordar la detección de colisiones entre la pelota y los demás objetos del entorno. Para ello, accederemos al script y sobrescribiremos el método `OnCollisionEnter2D`:

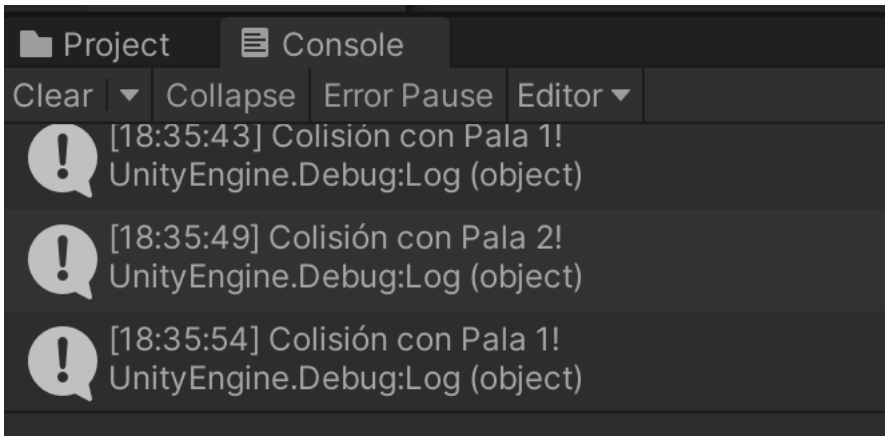
```
public class PelotaController : MonoBehaviour
{
    Rigidbody2D rb;
    [SerializeField] float force;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        rb.AddForce(new Vector2(1, 1) * force, ForceMode2D.Impulse);
    }

    void Update()
    {

    }

    private void OnCollisionEnter2D(Collision2D collision)
    {
        Debug.Log("Colisión!");
    }
}
```

En este momento, simplemente imprimiremos un **mensaje** en la consola para verificar su funcionamiento. Cada vez que se produce un rebote, se mostrará un mensaje de depuración:

A partir del objeto que recibimos en el método `Collision2D collision`, podríamos determinar qué objeto es con el que colisionamos mediante sus etiquetas. Por ejemplo, si las dos palas tienen definidas etiquetas, podemos utilizar esas etiquetas para identificar con cuál de las palas estamos colisionando:

```
private void OnCollisionEnter2D(Collision2D collision)
{
    string tag = collision.gameObject.tag;

    if (tag.Equals("Pala1"))
        Debug.Log("Colisión con Pala 1!");
    else if (tag.Equals("Pala2"))
        Debug.Log("Colisión con Pala 2!");
}
```

Al ejecutar nuestro juego, observaremos que en un primer momento no se imprime nada, pero cuando la pelota colisiona con las palas, se mostrarán mensajes en la consola:



ATRAVESANDO OBJETOS

En algunas situaciones, puede ser necesario que no se produzca una colisión física sino que simplemente queremos que un objeto pase a través de otro sin interactuar. Por ejemplo, supongamos que queremos que la pelota atraviese las paredes laterales sin rebotar. Esta acción se realiza fácilmente ajustando la configuración de los colliders asociados a los objetos. Al activar la opción **“Is Trigger”** en los *colliders* de las paredes, permitimos que la pelota los atraviese sin rebotar. Esto se hace solo para los límites derecho e izquierdo, como se muestra a continuación:

Ahora, la pelota atravesará las paredes en lugar de rebotar. Sin embargo, aún podemos detectar la ‘colisión’ y tomar medidas en el script. Para manejar estas situaciones, utilizaremos otros métodos, como `OnTriggerEnter2D(Collider2D)`.

Vamos a realizar una prueba en el script “PelotaController” para comprobar que funciona:

```
private void OnTriggerEnter2D(Collider2D other)
{
    Debug.Log("¡Gol!!");
}
```

Cuando la pelota atraviese la pared derecha, se imprimirá un mensaje de “Gol”.

En este caso, también podemos determinar qué objeto ha sido atravesado al invocar el método `OnTriggerEnter2D`. Para lograr esto, asignaremos etiquetas a las dos paredes (derecha e izquierda), que llamaremos “PorteriaDerecha” y “PorteriaIzquierda”. En este método, el objeto `Collider2D` tiene un atributo con la etiqueta asociada:

```
private void OnTriggerEnter2D(Collider2D other)
{
    Debug.Log("Gol en " + other.tag + "!!");
}
```

De esta manera, obtendremos mensajes distintos dependiendo de la pared que atraviese la pelota.

Autor/a: Sabela Sobrino Última actualización: 20.01.2025

MOVIMIENTO ALEATORIO DE LA PELOTA

Cuando se ha producido un gol, es necesario lanzar nuevamente la pelota desde el centro del campo, incorporando un nivel de aleatoriedad para que su trayectoria varíe en ángulos y direcciones diferentes cada vez que se reinicia el juego. Para implementar esto, necesitamos un vector con dos coordenadas (x e y) y valores aleatorios para modificar la trayectoria de la pelota:

La longitud de nuestro vector será 1, y calcularemos de manera aleatoria el ángulo beta dentro de un rango específico, entre 30º y 50º, utilizando el método `Random`:

Luego, emplearemos el método `AddForce` y le proporcionaremos dos coordenadas x y (un `Vector2`), utilizando los valores del ángulo, su coseno y su seno:

Es posible cambiar el signo de cualquiera de las coordenadas para que la pelota se dirija en una dirección específica. Todo esto se realizará en el script `PelotaController`, donde declararemos dos constantes para los ángulos máximos:

```
public class PelotaController : MonoBehaviour
{
    Rigidbody2D rb;
    [SerializeField] float force;
draft: true

    const float MIN_ANG = 25.0f;
    const float MAX_ANG = 40.0f;
    //...
```

Además, crearemos una función para determinar el lanzamiento de la pelota:

```
public void LanzarPelota(int direccionX)
{
    // Definimos el ángulo beta en radianes dentro del rango establecido
    float angulo = Random.Range(MIN_ANG, MAX_ANG) * Mathf.Deg2Rad;

    // Calculamos las coordenadas x e y utilizando el ángulo
    float x = Mathf.Cos(angulo);
    float y = Mathf.Sin(angulo);

    // Creamos un Vector2 con las coordenadas
    Vector2 impulso = new Vector2(x, y);

}
```

Con el parámetro direccion determinamos si queremos ir a la derecha o a la izquierda. Este valor será 1 o -1 dependiendo a donde queramos ir que será la dirección contraria a donde se haya metido gol. La dirección en Y nos da igual por lo que podemos utilizar un número aleatorio:

```
    public void LanzarPelota(int direccionX) {
    // Definimos el ángulo en radianes utilizando valores aleatorios dentro de un rango específico
    float angulo = Random.Range(MIN_ANG, MAX_ANG) * Mathf.Deg2Rad;

    // Calculamos la coordenada x utilizando el ángulo y la dirección X
    float x = Mathf.Cos(angulo) * direccionX;

    // Determinamos si nos movemos hacia arriba o hacia abajo en la coordenada y
    // Si el valor que devuelve Random.Range es igual a 0, la dirección en y será negativa; de lo contrario, será
positiva
    int direccionY = Random.Range(0, 2) == 0 ? -1 : 1; // El límite superior es exclusivo (el 2 quedaría fuera).

    // Calculamos la coordenada y utilizando el ángulo y la dirección en y
    float y = Mathf.Sin(angulo) * direccionY;

    // Creamos un Vector2 con las coordenadas calculadas
    Vector2 impulso = new Vector2(x, y);

}
```

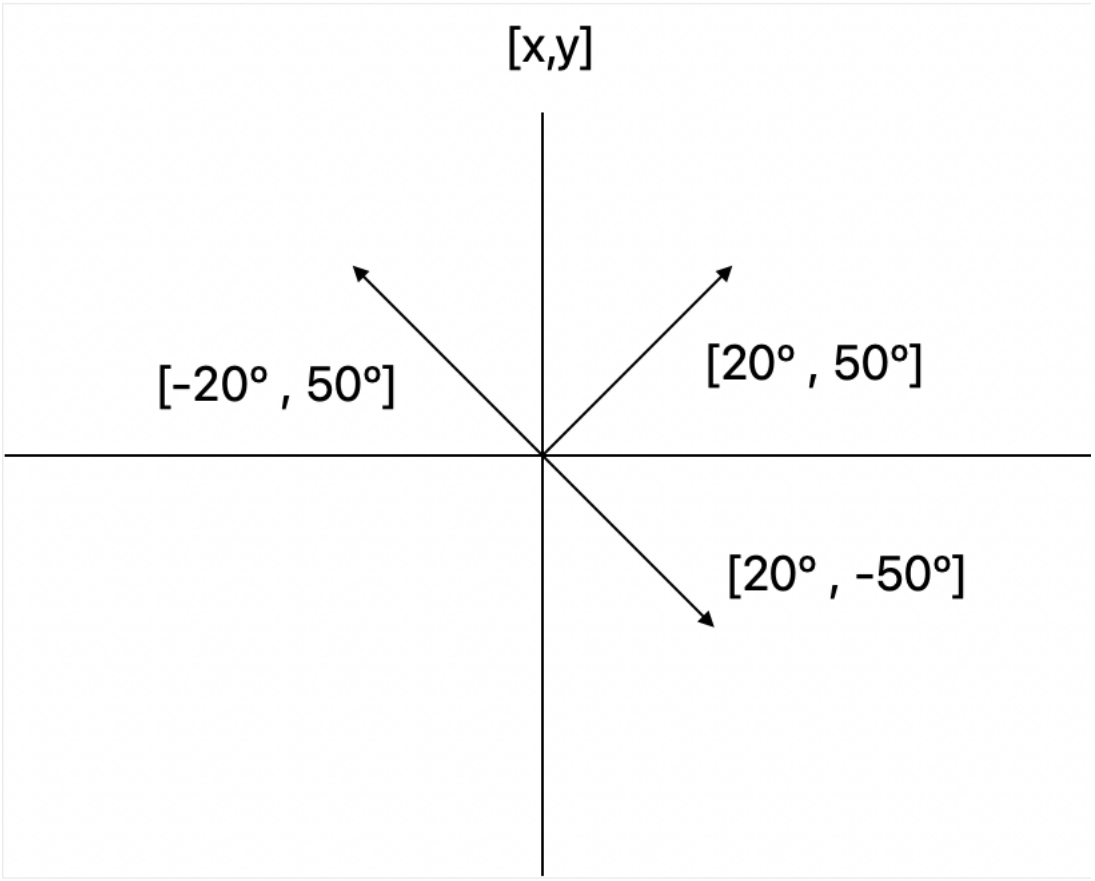
Ahora ya podría llamar a la función de addForce para generar el movimiento aleatorio:

```
public void LanzarPelota(int direccionX)
{
    // Definimos el ángulo en radianes utilizando valores aleatorios dentro de un rango específico
    float angulo = Random.Range(MIN_ANG, MAX_ANG) * Mathf.Deg2Rad;

    // Calculamos la coordenada x utilizando el ángulo y la dirección X
    float x = Mathf.Cos(angulo) * direccionX;

    // Determinamos si nos movemos hacia arriba o hacia abajo en la coordenada y
    // Si el valor que devuelve Random.Range es igual a 0, la dirección en y será negativa; de lo contrario, será positiva
    int direccionY = Random.Range(0, 2) == 0 ? -1 : 1; // El límite superior es exclusivo (el 2 quedaría fuera).
    float y = Mathf.Sin(angulo) * direccionY;

    // Creamos un Vector2 con las coordenadas calculadas
    Vector2 impulso = new Vector2(x, y);

    // Aplicamos la fuerza a la pelota utilizando el impulso y la fuerza definida
    rb.AddForce(impulso * force, ForceMode2D.Impulse);
}
```

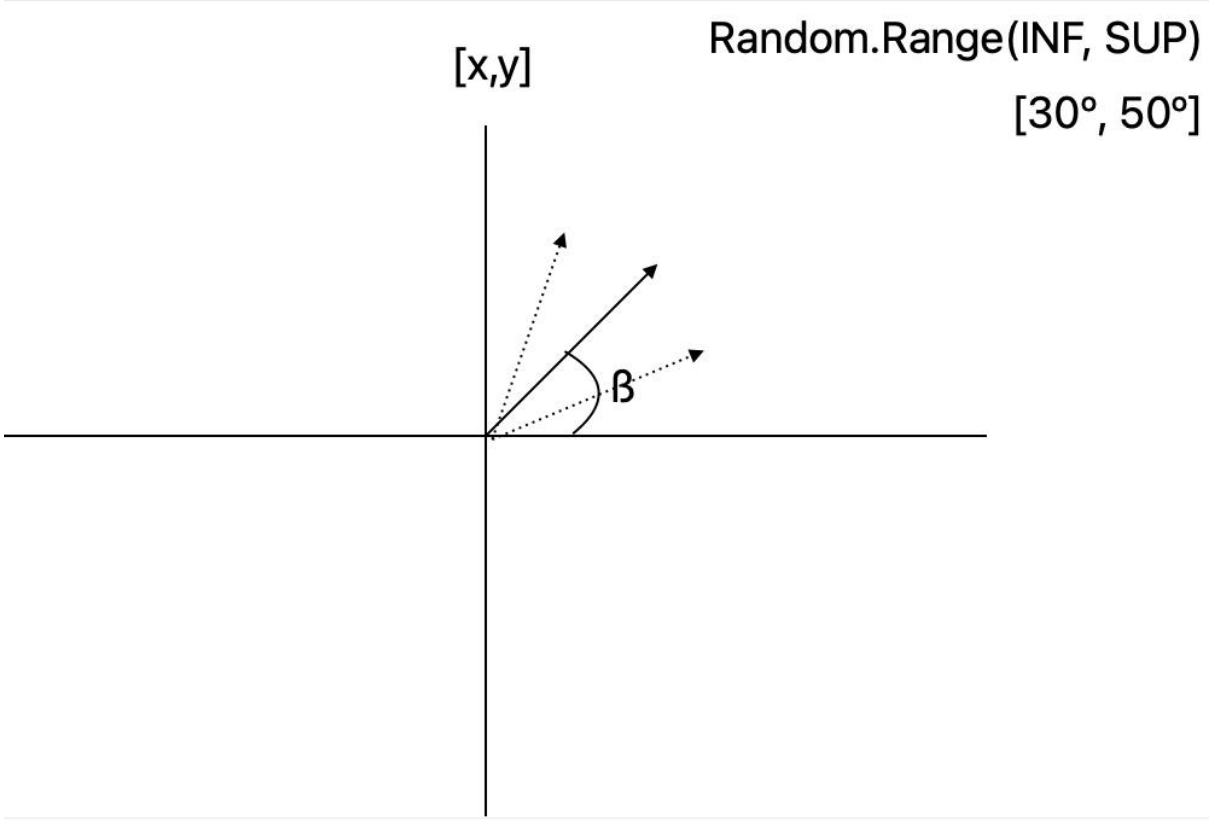
Autor/a: Sabela Sobrino Última actualización: 20.01.2025

REINICIO DE LA PELOTA

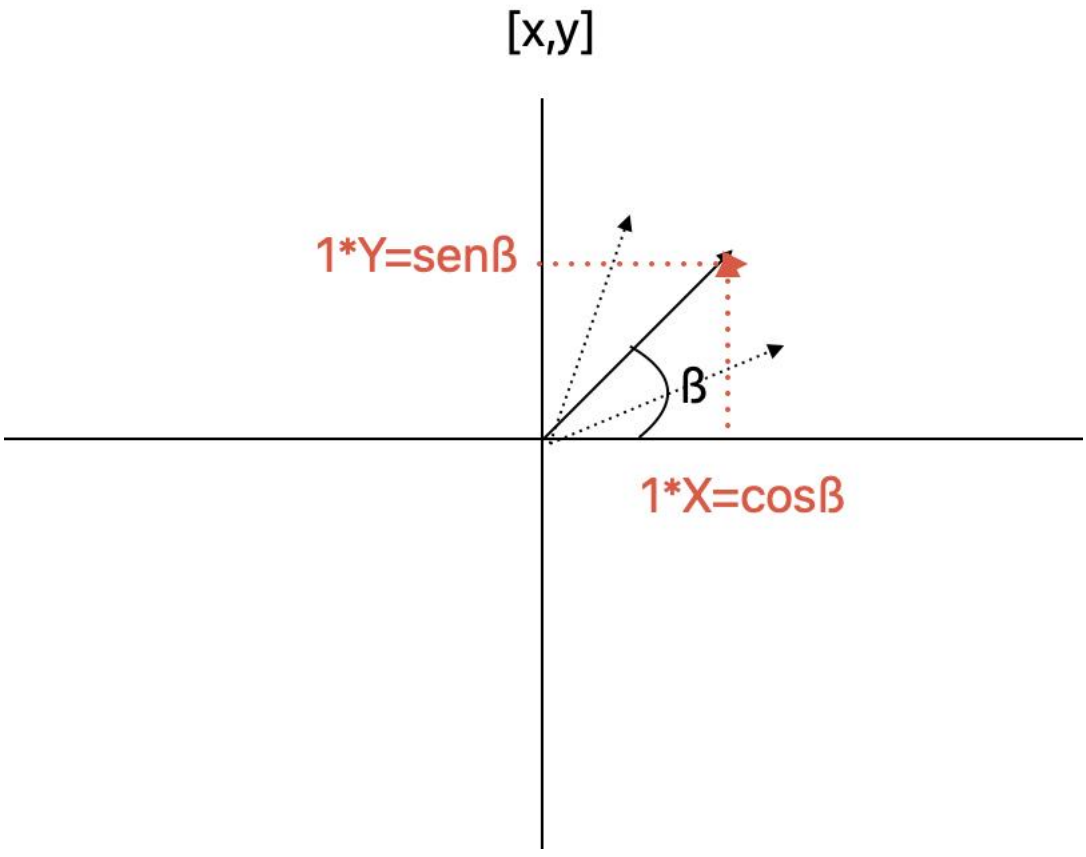
Después de anotar un gol, es esencial **reiniciar** el juego lanzando la **pelota** desde el centro del campo. Este reinicio debe incorporar un grado de aleatoriedad, asegurando que la pelota siga trayectorias únicas en términos de ángulos y direcciones en cada ocasión. Para lograr esto, utilizaremos un vector bidimensional (x, y) y números aleatorios que modificarán la trayectoria de la pelota.



La longitud de nuestro vector será 1, y calcularemos de manera aleatoria el ángulo beta. Para esto, emplearemos la clase `Random` y un vector, asegurándonos de proporcionar valores dentro de un rango específico, limitado entre 30º y 50º.



Utilizaremos el método `AddForce`, al cual le pasaremos las dos coordenadas x e y (un `Vector2`). Para hacerlo, emplearemos el ángulo y sus respectivos coseno y seno.



Vamos a utilizar el signo de cualquiera de las coordenadas para que la pelota pueda salir en direcciones opuestas. Todo esto se llevará a cabo en el script `PelotaController`, donde declararemos dos constantes con los ángulos máximos.

```
public class PelotaController : MonoBehaviour
{
    Rigidbody2D rb;
    [SerializeField] float force;

    const float MIN_ANG = 25.0f;
    const float MAX_ANG = 40.0f;
    // ...
```

Además, crearemos una función para determinar el lanzamiento de la pelota:

```
public void LanzarPelota(int direccionX)
{
    // Definimos el ángulo en radianes usando Range, especificando el mínimo y máximo.
    float angulo = Random.Range(MIN_ANG, MAX_ANG) * Mathf.Deg2Rad;
    float x = Mathf.Cos(angulo) * direccionX;

    // Determinamos si nos movemos hacia la derecha o izquierda.
    // Si el valor devuelto es 0, la dirección en Y será negativa; si es 1, será positiva.
    int direccionY = Random.Range(0, 2) == 0 ? -1 : 1;
    float y = Mathf.Sin(angulo) * direccionY;

    Vector2 impulso = new Vector2(x, y);
    rb.AddForce(impulso * force, ForceMode2D.Impulse);
}
// ...
```

Finalmente, llamaremos a esta función en el método `Start()`. Al inicio del juego, no importa hacia dónde vaya la pelota, así que calcularemos la dirección de forma aleatoria.

```
void Start()
{
    rb = GetComponent<Rigidbody2D>();
    int direccionX = Random.Range(0, 2) == 0 ? -1 : 1;
    LanzarPelota(direccionX);
    // ...
}
```

Con estos ajustes, hemos mejorado la redacción y estructura del código para hacer más claro el proceso de reinicio de la pelota con trayectoria aleatoria.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

POSICIÓN PELOTA

Vamos a introducir variabilidad en la posición inicial de la pelota, evitando que siempre comience en la posición y=0, en el centro de nuestro programa. Para lograr esto, hemos definido dos constantes que indican las posiciones máximas y mínimas permitidas:

```
// Declaramos dos constantes con las posiciones y máximas y mínimas.
```

```
const float MAX_Y = 2.5f;
const float MIN_Y = -2.5f;
```

Ahora, procederemos a calcular la posición y de la pelota de manera aleatoria en la función `LanzarPelota`:

```
public void LanzarPelota(int direccionX){
    // Calculamos la posición vertical de forma aleatoria.
    float posY = Random.Range(MIN_Y, MAX_Y);
    transform.position = new Vector3(0, posY, 0);
}
```

Con estos ajustes, hemos introducido variabilidad en la posición inicial de la pelota, permitiendo que se genere de manera aleatoria en el eje vertical entre las posiciones máxima y mínima especificadas.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

INVOCACIÓN RETARDADA DE MÉTODOS

Invoke

Además de introducir variabilidad en la posición inicial de la pelota, también implementaremos una **breve pausa** para evitar que la pelota se lance inmediatamente al comenzar el juego. Unity nos ofrece una solución simple para esto mediante el método **Invoke**:

```
// Start is called before the first frame update
void Start()
{
    rb = GetComponent<Rigidbody2D>();
    int direccionX = Random.Range(0, 2) == 0 ? -1 : 1; // El límite superior es exclusivo (el 2 quedaría fuera).
    LanzarPelota(direccionX);

    Invoke("LanzarPelotaConPausa", 2.0f);
    // rb.AddForce(new Vector2(1,1)*force, ForceMode2D.Impulse);
}
```

En este caso, el método Invoke espera el nombre del método que queremos invocar y la duración de la pausa en segundos. Sin embargo, surge un problema al intentar pasarle **parámetros** a este método. Aunque podríamos solucionarlo creando un atributo privado de la clase, para mantener el código lo más intacto posible, optaremos por utilizar otro método para detener la invocación del método específico.

Corutinas

Las corutinas representan métodos que nos permiten verificar una **condición** específica para la ejecución de un método, sin afectar al resto de nuestro programa.

```
void Start()
{
    rb = GetComponent<Rigidbody2D>();
    int direccionX = Random.Range(0, 2) == 0 ? -1 : 1; // El límite superior es exclusivo (el 2 quedaría fuera).
    StartCoroutine(LanzarPelota(direccionX));
}
```

Todos los métodos que deseamos ejecutar como corutinas deben tener como valor de retorno `IEnumerator`. Dentro de estos métodos, podemos establecer el tiempo que queremos pausar nuestro programa:

```
IEnumerator LanzarPelota(int direccionX)
{
    yield return new WaitForSeconds(delay);
}
```

Hemos introducido la variable `delay` como un campo serializable:

```
public class PelotaController : MonoBehaviour
{
    Rigidbody2D rb;
    [SerializeField] float force;
    [SerializeField] float delay;
    // ...
```

Este enfoque nos proporciona un control más flexible al gestionar las pausas en la ejecución de métodos específicos, mejorando así la modularidad de nuestro código.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

REINICIO DE LA PELOTA

El siguiente paso implica que la pelota **regrese** a su posición original una vez que se anota un gol, es decir, cuando atraviesa una de las porterías. Dado que las porterías están **etiquetadas**, podemos identificarlas y manejar este comportamiento dentro del script “PelotaController”:

```
private void OnTriggerEnter2D(Collider2D other) {
    Debug.Log("Gol en " + other.tag + "!!");

    if (other.tag == "PorteriaIzquierda") {
        // Lanzaremos la pelota hacia la derecha
        StartCoroutine(LanzarPelota(1));
    } else if (other.tag == "PorteriaDerecha") {
        // Lanzaremos la pelota hacia la izquierda
        StartCoroutine(LanzarPelota(-1));
    }
}
```

Al ejecutar el juego, podemos observar que el script funciona correctamente: cuando se anota un gol, la pelota se relanza desde el centro hacia la portería contraria. Sin embargo, si dejamos el juego en ejecución durante un tiempo prolongado, notaremos que la pelota se lanza con mayor velocidad en cada ocasión.

Este comportamiento puede deberse a que la velocidad se acumula con cada reinicio de la pelota. Para abordar este problema, podemos asegurarnos de restablecer la velocidad de la pelota a un valor inicial en el

método `LanzarPelota`. Esta modificación puede ayudar a mantener la consistencia en la velocidad de la pelota durante el juego.

RigidBody.velocity

Identificamos que el problema radica en la siguiente línea de código:

```
rb.AddForce(impulso * force, ForceMode2D.Impulse);
```

Actualmente, estamos aplicando una nueva fuerza a un objeto que ya posee velocidad y a la cual ya se le aplicó una fuerza previamente. Para resolver este inconveniente, es necesario reiniciar la fuerza de la pelota antes de lanzarla. A través del `Rigidbody`, podemos restablecer el atributo `velocity`, el cual es de tipo `Vector2`.

```
Vector2 impulso = new Vector2(x, y);
// Resetear la velocidad lineal de la pelota
// rb.velocity = new Vector2(0, 0);
rb.LinearVelocity = Vector2.zero;
// Ahora podemos aplicar el impulso
rb.AddForce(impulso * force, ForceMode2D.Impulse);
```

Hemos utilizado el vector constante cero, equivalente a `new Vector2(0, 0)`, para resetear la velocidad lineal de la pelota. Esta modificación asegura que la fuerza se aplique de manera coherente y evita acumulaciones no deseadas de velocidad durante el juego.

De esta forma, el script nos quedaría de la siguiente forma:

```
using System.Collections;
using UnityEngine;

public class BallController : MonoBehaviour
{

    private Rigidbody2D rb;
    [SerializeField] float force;
    [SerializeField] float delay;

    const float MIN_ANG = 25.0f;
    const float MAX_ANG = 40.0f;

    // Start is called once before the first execution of Update after the MonoBehaviour is created
    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        //throwBall();
        //Invoke("throwBall", delay);
        transform.position = new Vector3(0,0,0); //Vector3.zero;

         int directionX = Random.Range(0, 2) == 0 ? -1 : 1; // El límite superior es exclusivo (el 2 quedaría fuera).
         StartCoroutine(throwBall(directionX));
    }

    IEnumerator throwBall(int directionX){

        transform.position = new Vector3(0,0,0); //Vector3.zero;
        rb.linearVelocity = Vector2.zero;

        yield return new WaitForSeconds(delay);

        float angulo = Random.Range(MIN_ANG, MAX_ANG) * Mathf.Deg2Rad;
```

```
        int directionY = Random.Range(0,2) == 0 ?-1:1;

        float x = Mathf.Cos(angulo) * directionX;
        float y = Mathf.Sin(angulo) * directionY;

        rb.AddForce(new Vector2(x,y) * force, ForceMode2D.Impulse);

    }

    // Update is called once per frame
    void Update()
    {

    }

   private void OnTriggerEnter2D(Collider2D collider){
    Debug.Log("Gol en " +collider.tag + "!!");
    if(collider.tag.Equals("GoalLeft")){
        StartCoroutine(throwBall(1));
    }else if(collider.tag.Equals("GoalRigth")){
        StartCoroutine(throwBall(-1));
    }



    }
}
```