BREAKOUT

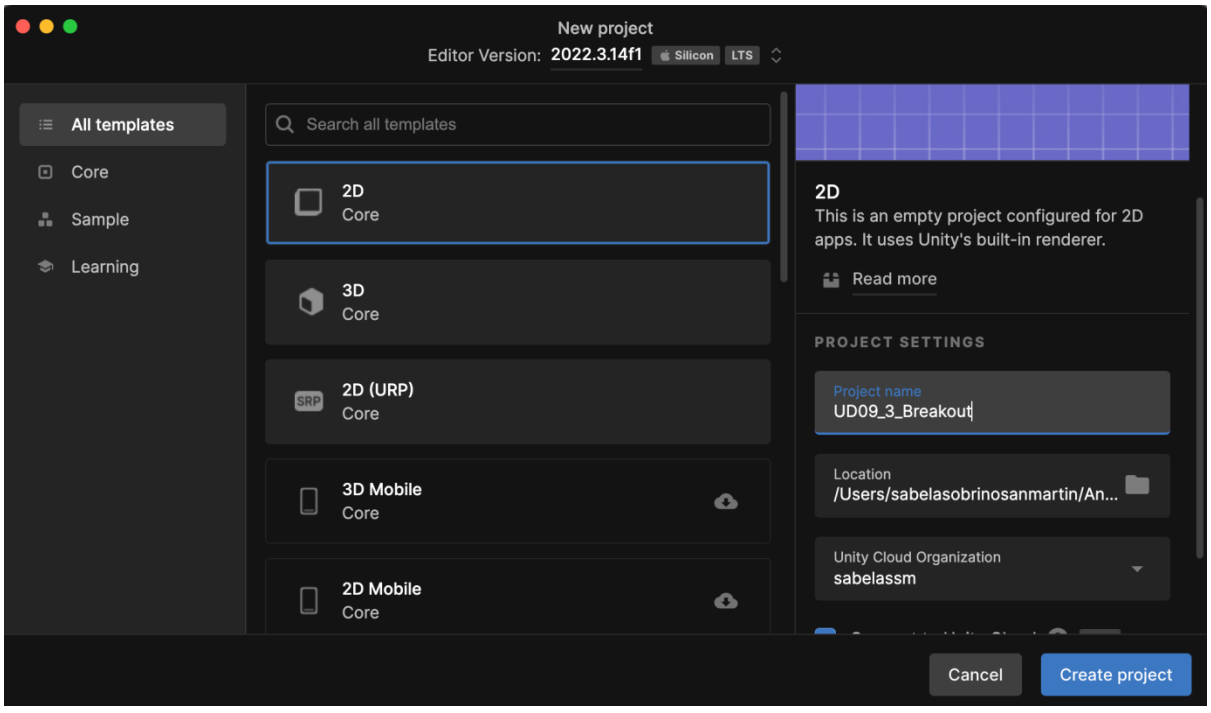
El juego Breakout es un clásico juego de arcade lanzado por Atari en 1976. Fue diseñado por Nolan Bushnell y Steve Bristow, y fue conceptualmente similar a un juego anterior llamado “Pong”, también creado por Atari.

En Breakout, el jugador controla una paleta que se mueve horizontalmente en la parte inferior de la pantalla. El objetivo es usar una pelota para romper una serie de ladrillos dispuestos en la parte superior de la pantalla. Cada vez que la pelota golpea un ladrillo, éste se destruye y el jugador gana puntos. El juego continúa hasta que todos los ladrillos hayan sido eliminados o la pelota caiga fuera de la zona de juego.

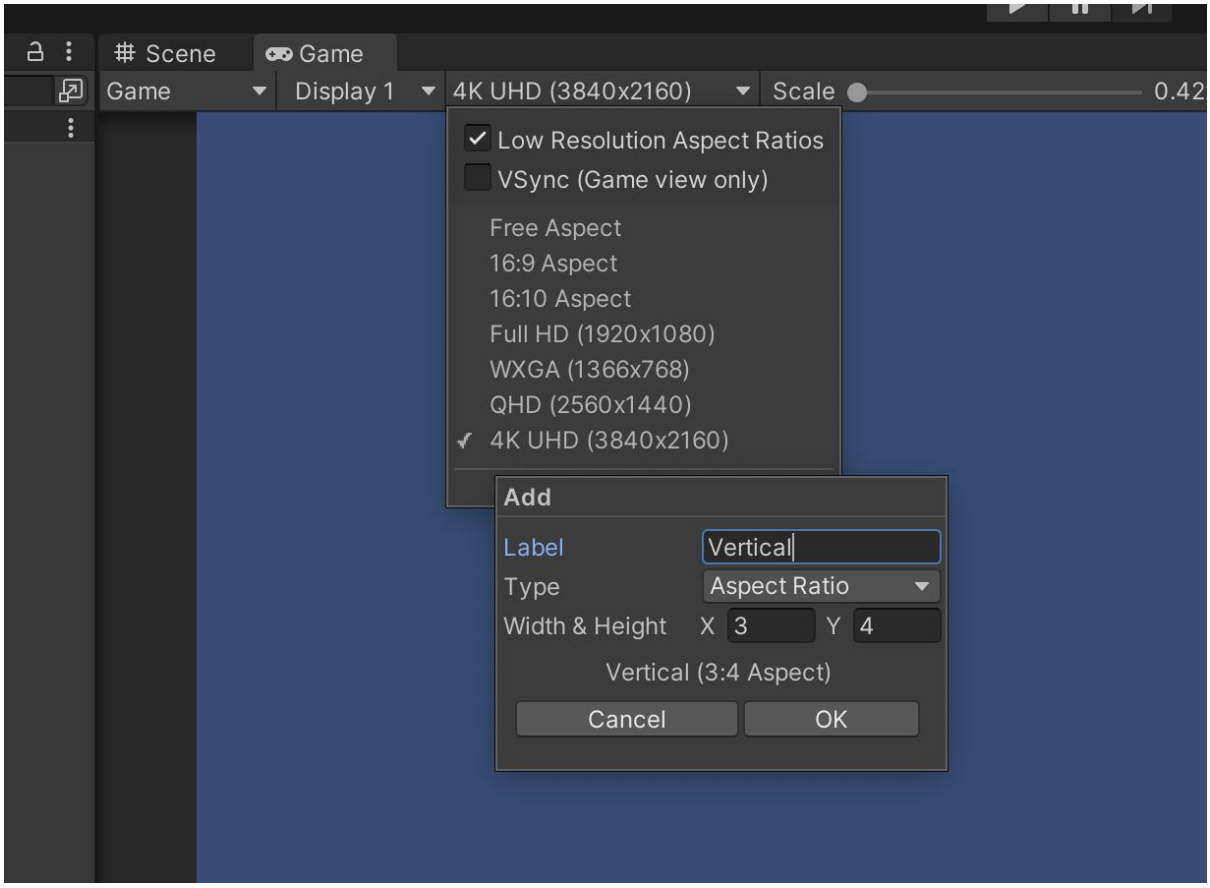
Breakout es conocido por su jugabilidad simple pero adictiva, así como por su diseño visual distintivo. Ha sido adaptado y recreado en numerosas plataformas y sigue siendo popular entre los entusiastas de los videojuegos retro. Además, ha inspirado una serie de juegos similares y variaciones a lo largo de los años

PROYECTO

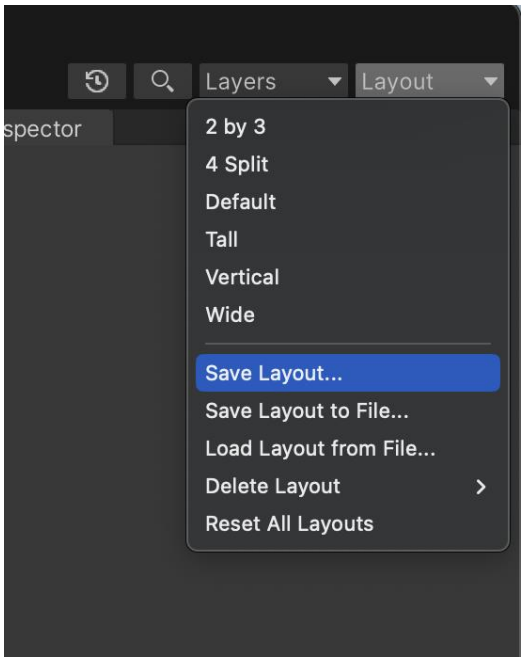
Para comenzar nuestro proyecto en Unity Hub, seleccionaremos la plantilla “2D Core” y le daremos el nombre “UD08_3_Breakout”, además de elegir la ubicación donde deseamos almacenarlo.



Una vez que el proyecto esté en marcha, ajustaremos las distintas pantallas según nuestras preferencias. Dado que este proyecto se ejecutará en vertical, cambiaremos el formato del proyecto creando uno nuevo con una relación de aspecto de 3:4.

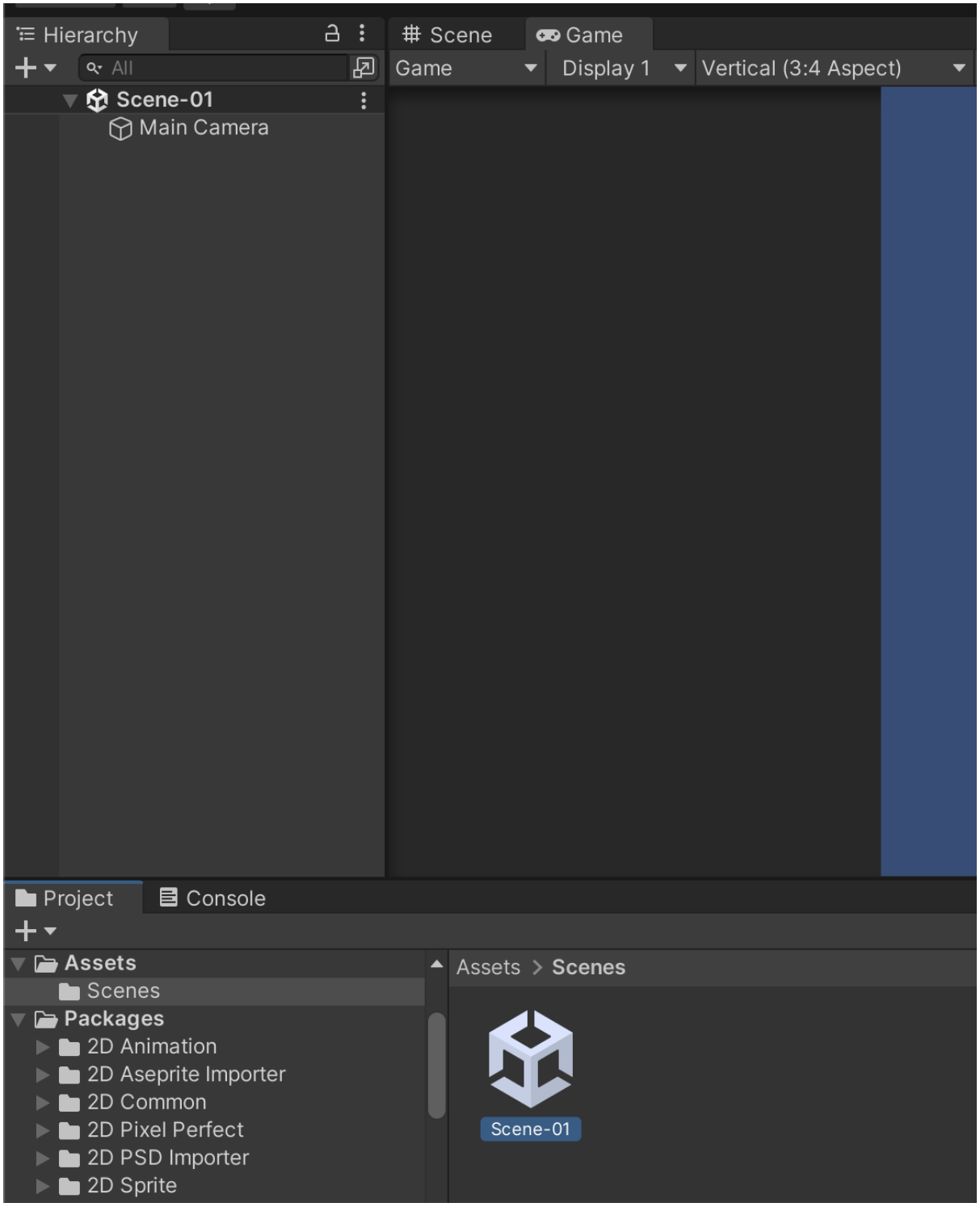


Podemos guardar esta disposición del juego utilizando la función “save layout”:



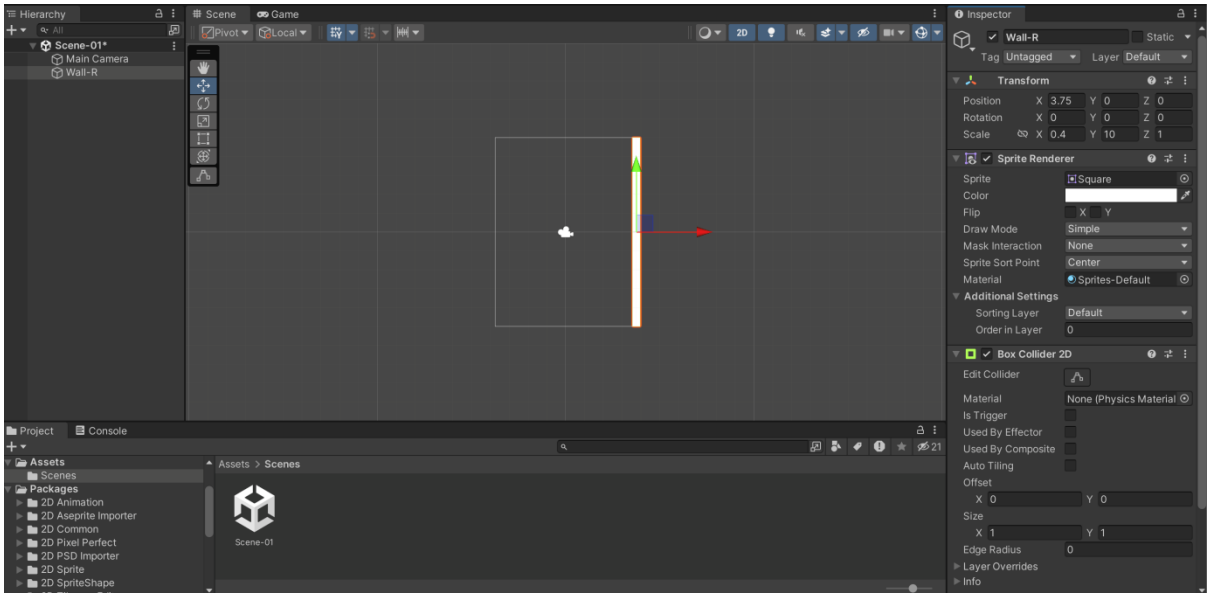
ZONA DE JUEGO

1. Renombrar la escena como “Scene-01”:

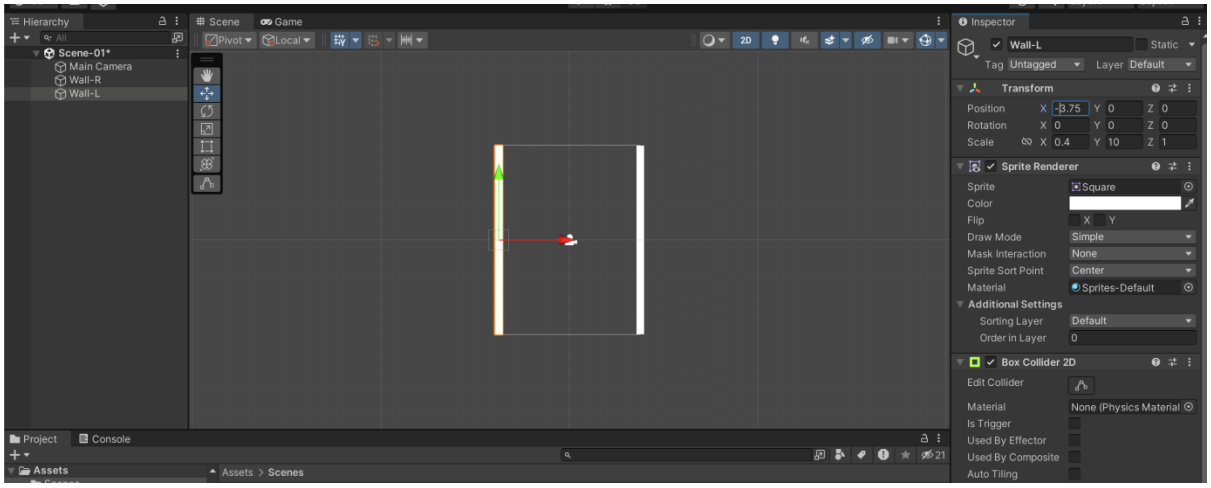


2. Crear la zona de juego:

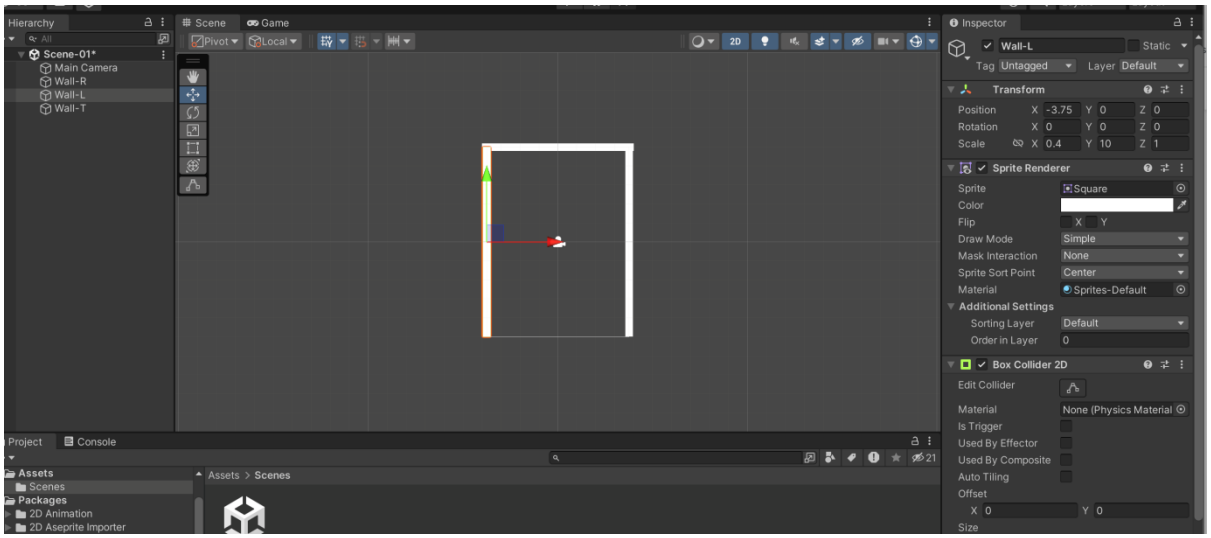
Comenzaremos generando los bordes verticales. Ajustaremos su tamaño y posición, y les daremos el nombre “wall-r”, añadiendo un box collider 2D:



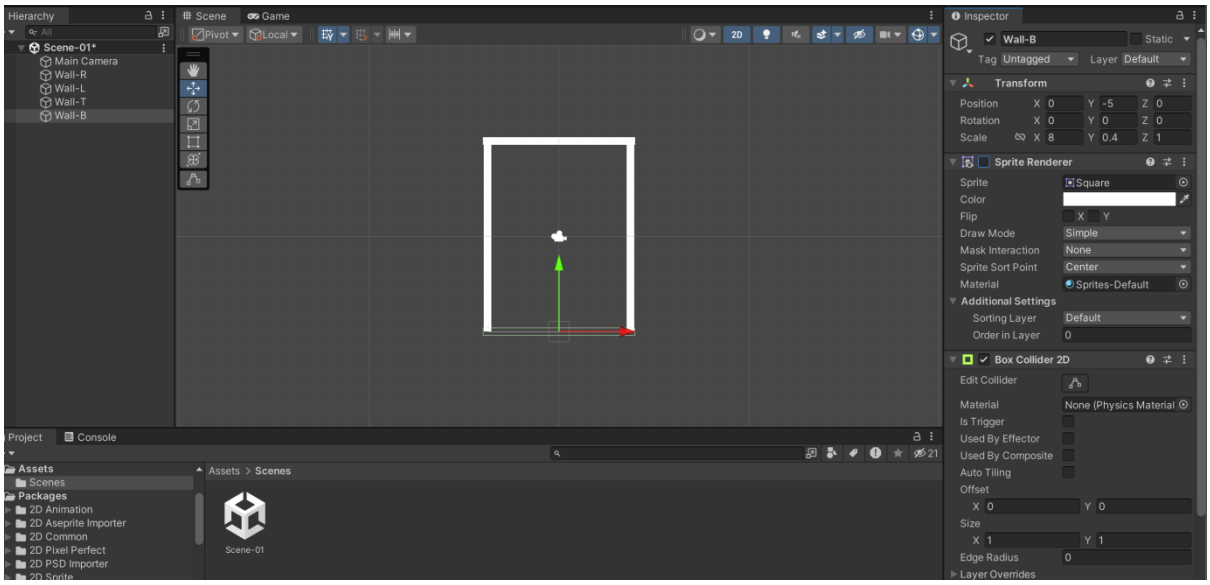
A partir de esta pared, duplicaremos para generar la pared izquierda, cambiando simplemente el signo de la posición X:



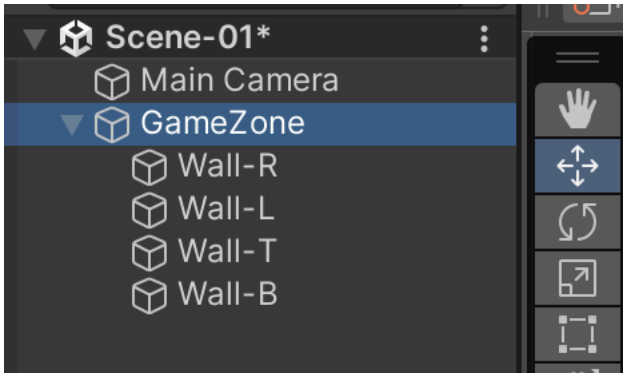
Utilizaremos estas paredes para generar la pared superior, ajustando las dimensiones y la posición:



Por último, desde la pared superior generaremos la pared inferior cambiando el signo de la posición en Y. Sin embargo, en este caso no queremos que se muestre, por lo que deshabilitaremos la opción de Sprite Renderer. También marcaremos el isTrigger del boxCollider, ya que en esta pared no rebotará, sino que la atravesará:

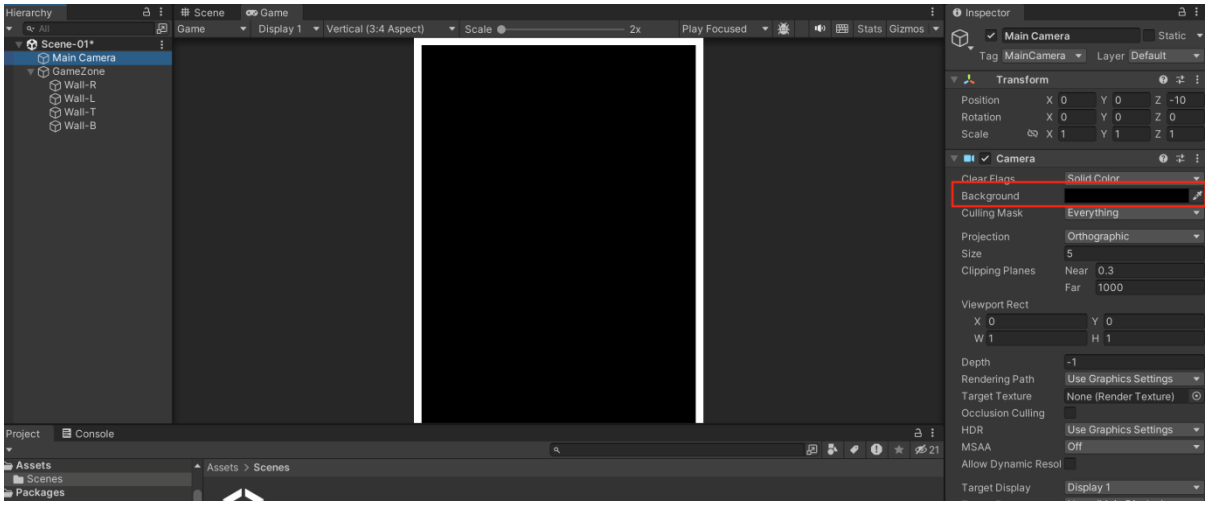


Crearemos un nuevo objeto vacío llamado “GameZone” donde agruparemos todos los objetos de pared:



3. Cambiar el fondo:

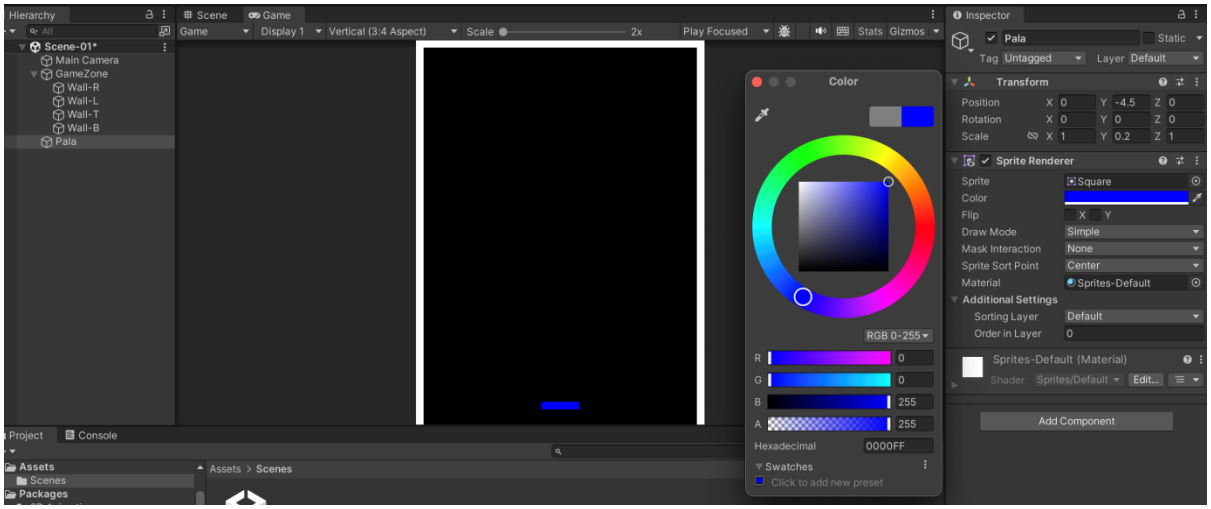
Modificaremos el fondo de nuestro juego a negro. Para ello, seleccionaremos la cámara y estableceremos el color negro:



PALA

Añadir componente

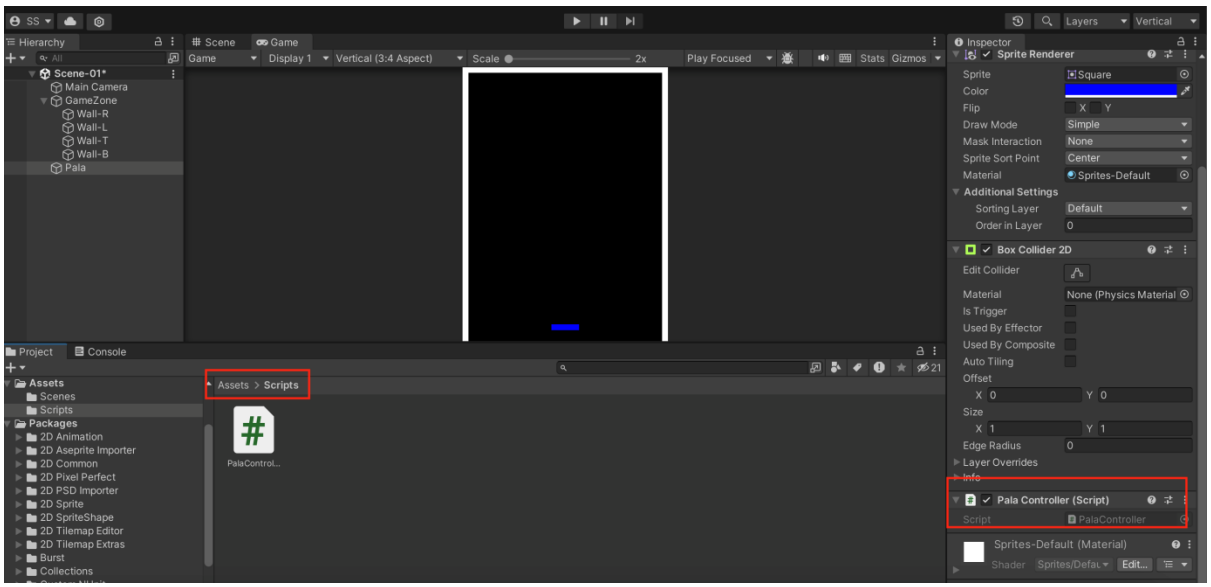
Vamos a incorporar un nuevo cuadrado a nuestra zona de juego para representar la pala. Ajustaremos su tamaño, posición y color a azul:



Dado que nuestra pala debe generar colisiones, necesitaremos añadir un nuevo componente **BoxCollider2D**.

Comportamiento

Para el comportamiento de nuestra pala, crearemos un nuevo script. Antes de eso, crearemos la carpeta “Scripts” dentro de la carpeta “Assets”:



Una vez creada, asignaremos este script al objeto Pala.

En el script, primero definiremos tres constantes, dos para delimitar el eje X y otra más, serializable, para la velocidad:

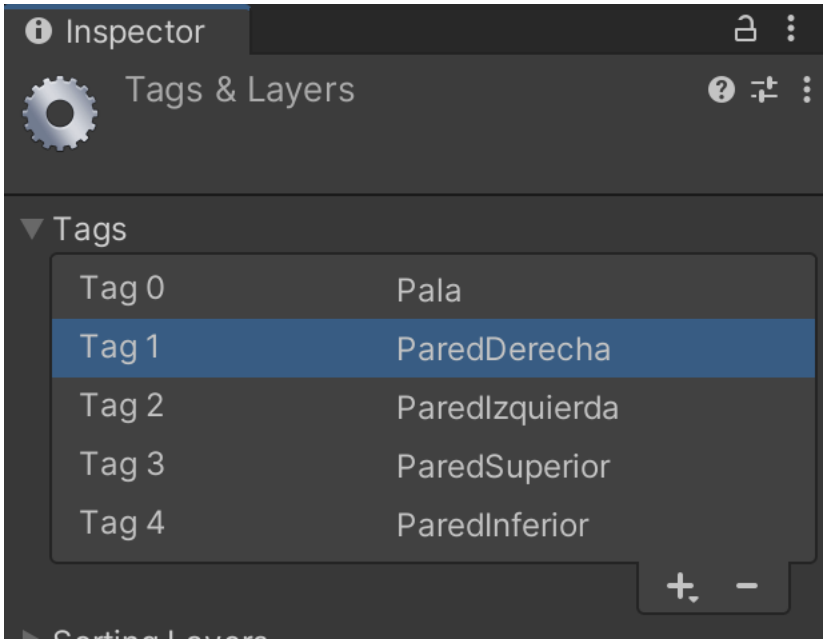
```
const float MAX_X = 3.1f;
const float MIN_X = -3.1f;
[SerializeField] float speed;
```

Luego, trasladaremos la pala dependiendo de la tecla pulsada, en una dirección u otra. Para ello, obtendremos la posición actual y la modificaremos:

```
void Update()
{
    float x = transform.position.x; // Obtener la posición actual de x de la pala
    if (x > MIN_X && Input.GetKey("left")) {
        // Desplazamiento hacia la izquierda con un valor negativo
        // Utilizamos deltaTime para obtener una referencia de la velocidad independiente del hardware
        transform.Translate(-speed * Time.deltaTime, 0, 0);
    } else if (x < MAX_X && Input.GetKey("right")) {
        transform.Translate(speed * Time.deltaTime, 0, 0); // Desplazamiento hacia la derecha
    }
}
```

ETIQUETAS

Vamos a asignar una etiqueta a todos los objetos que tenemos en la escena:

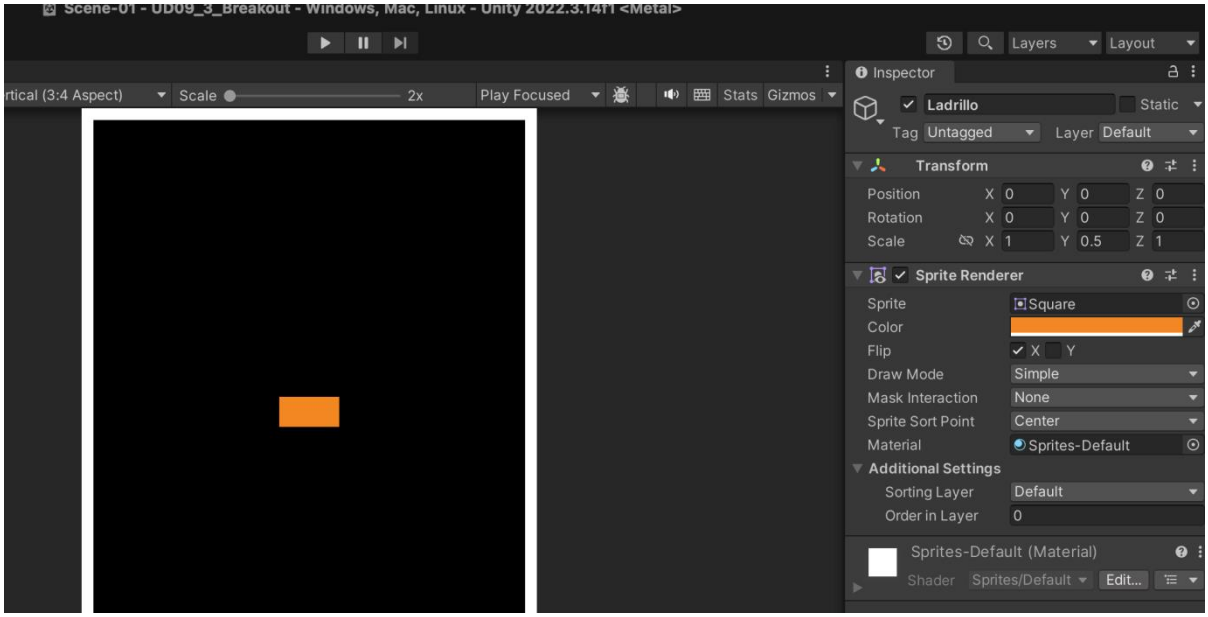


Una vez creadas las etiquetas, solo necesitamos asignarlas a cada objeto.

PREFABS

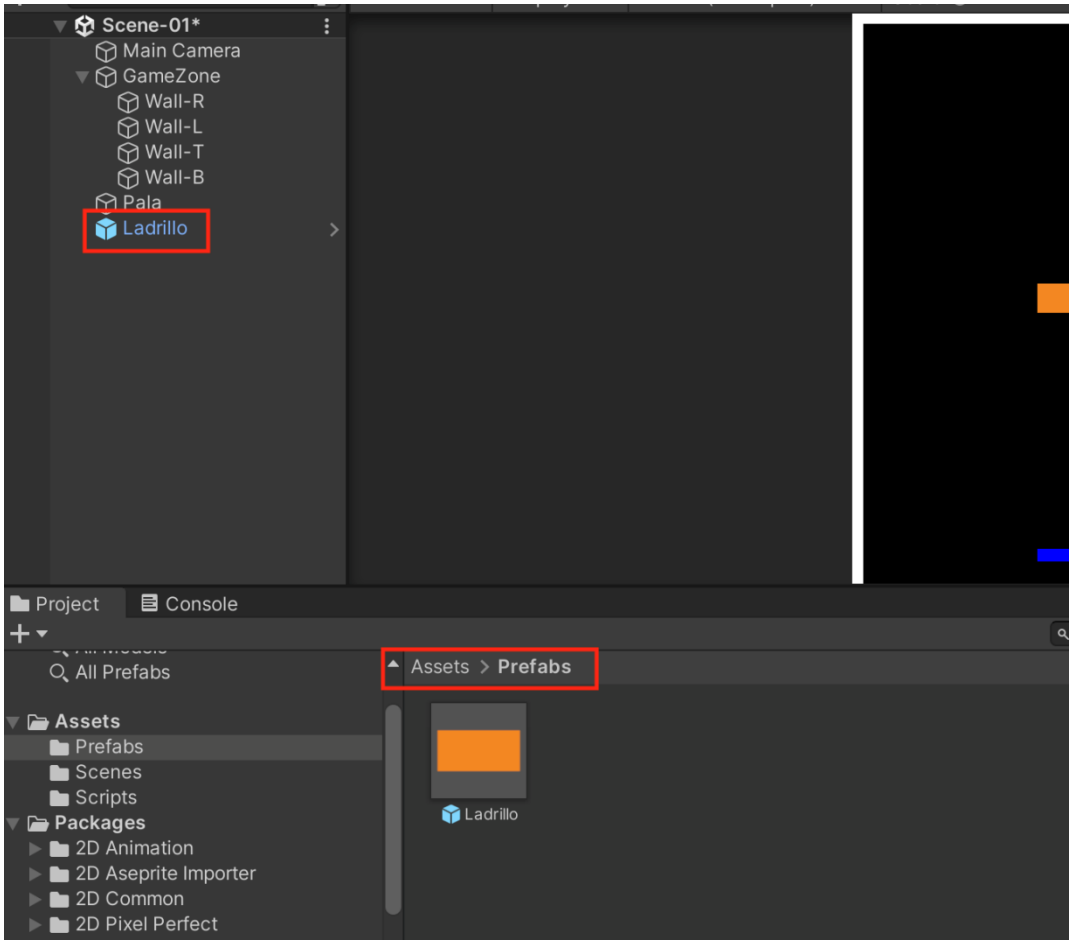
Ahora vamos a crear los ladrillos utilizando un nuevo componente llamado “prefabs” o “prefabricado”. Esto nos permitirá crear copias de un objeto de manera que si modificamos el objeto base, se modificarán todas las copias.

Comenzaremos creando un cuadrado al que daremos unas dimensiones para que simule un ladrillo y un color amarillo:



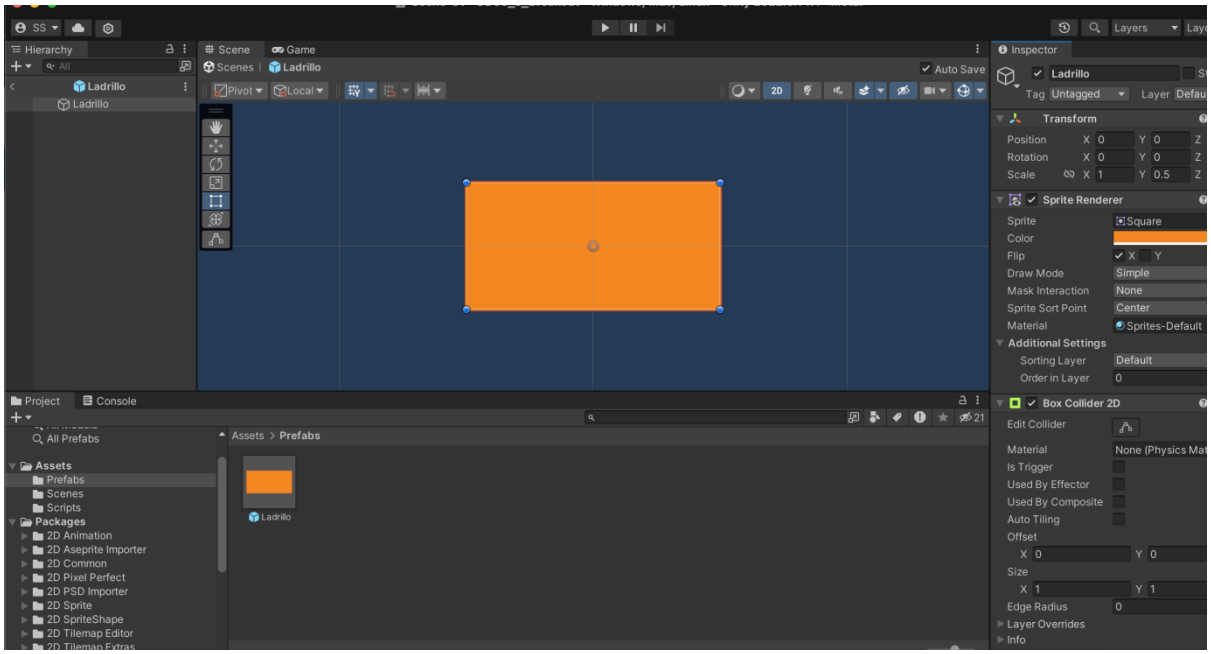
A partir de este ladrillo, podemos copiarlo y desplazarlo para generar una fila de ladrillos. Sin embargo, si queremos cambiar alguna de sus propiedades, tendríamos que ir uno por uno seleccionándolos y modificándolos. Esto se resuelve con los prefabs.

Para crear un prefab, crearemos una carpeta llamada “Prefabs” dentro de la carpeta “Assets” y arrastraremos nuestro ladrillo base a esa carpeta:



Podemos observar cómo el elemento cambia de apariencia y se resalta en azul. Ahora podemos arrastrar de nuevo el objeto prefab a nuestra escena como si fuera otro elemento predefinido de Unity.

La ventaja de este componente es que si lo editamos, se editarán todas las copias que tengamos dentro de nuestro proyecto. Por ejemplo, vamos a añadirle un box-collider:



Ahora, cualquier elemento “ladrillo” que tengamos en nuestra escena tendrá ese componente añadido, al igual que cualquier otro que añadamos a partir de ahora.

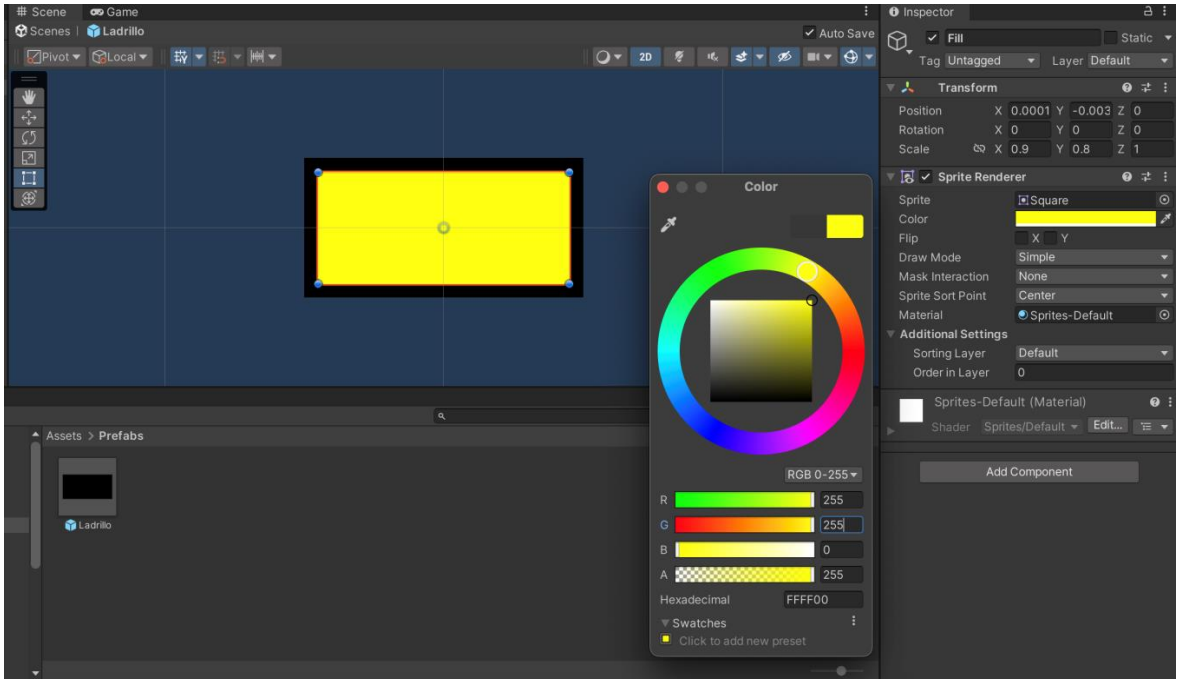
Esto nos permite cambiar cualquier propiedad o incluso añadir nuevos objetos de manera eficiente.

PARED DE LADRILOS

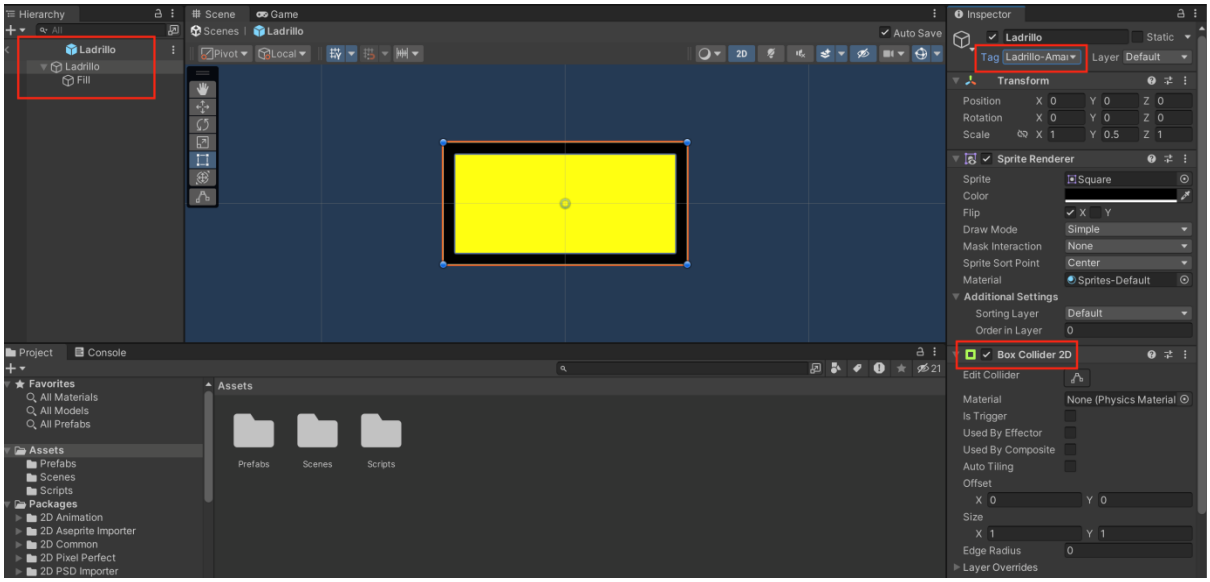
En el juego original, tenemos una pared de 8 filas x 14 columnas de ladrillos de distintos colores. Sin embargo, vamos a reducir la cantidad de ladrillos a 8 por fila.

Para cada una de estas filas, crearemos un prefab de ladrillo distinto para identificar con qué ladrillo estamos colisionando, simplemente asignando una etiqueta diferente a cada uno de esos prefabs.

Para cada ladrillo, crearemos un cuadrado negro (que no se verá pero tendrá el collider) y un cuadrado superpuesto con diferentes colores:



Al cuadrado negro, le añadiremos un collider y una etiqueta:

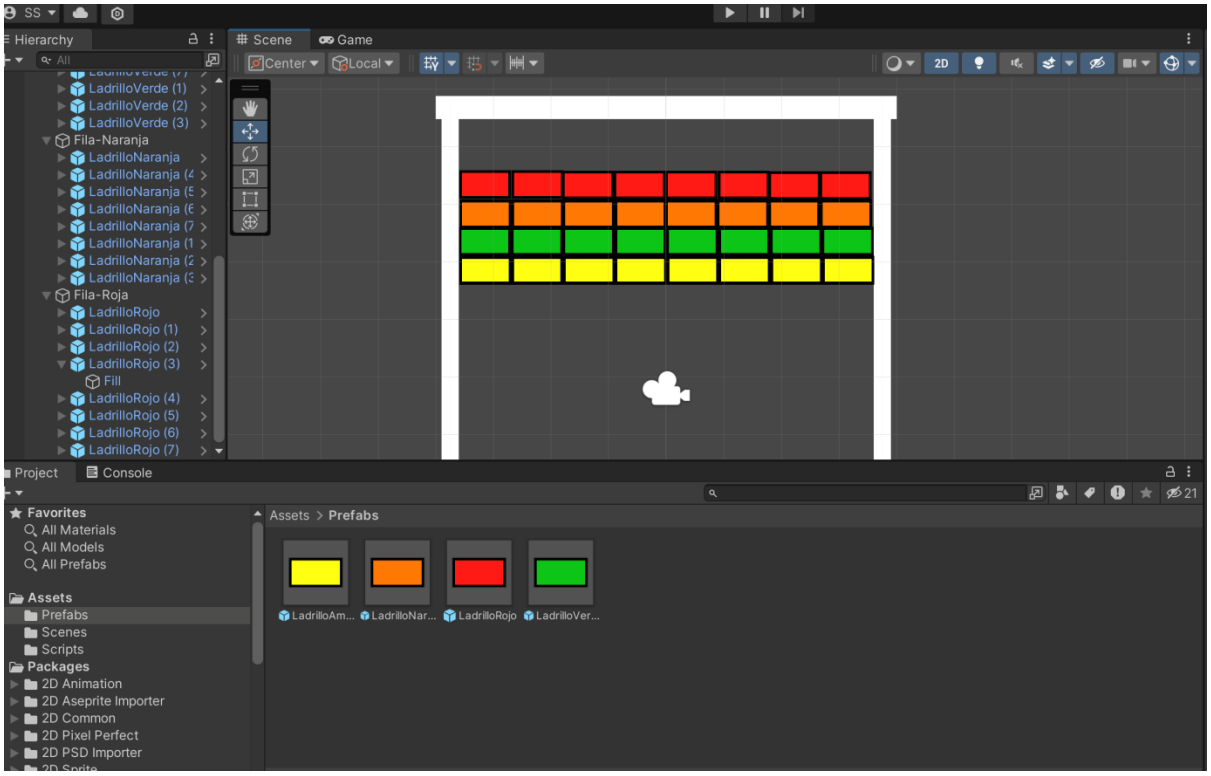


Luego, lo arrastramos a la carpeta “Prefabs” para crear un prefab.

Es posible que dentro del panel de Scene no veamos el relleno del ladrillo porque Unity usa un sistema de capas y el relleno se quedó en la capa de atrás. Para llevar el relleno a primer plano, podemos editar el prefab y modificar la propiedad “orden en Layer” con un valor positivo.

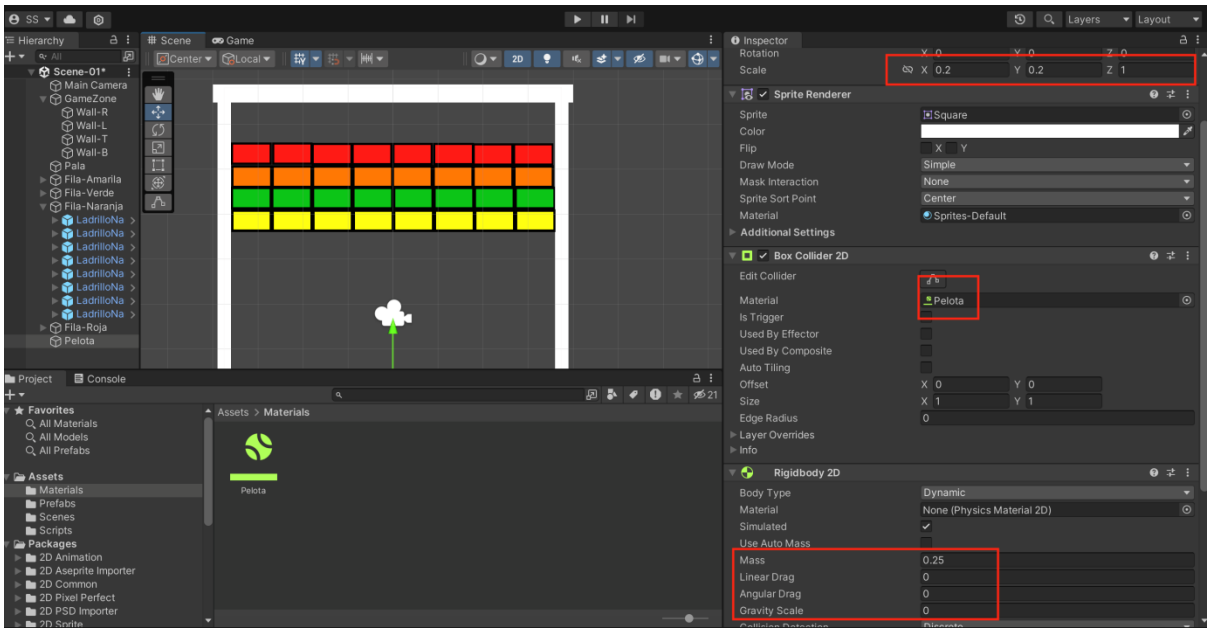
Ahora podremos generar nuestra fila de objetos amarillos, que agruparemos en un objeto vacío llamado “Fila-amarilla”. Para posicionar los ladrillos de forma precisa, podemos seleccionar uno de ellos, pulsar la tecla “V” y moverlo. Esto nos permitirá posicionar el ladrillo en el mismo vértice que el anterior, pegándose a cualquier vértice vecino que le indiquemos.

Generaremos el resto de ladrillos (prefabs), cambiando las etiquetas y el color de cada uno:



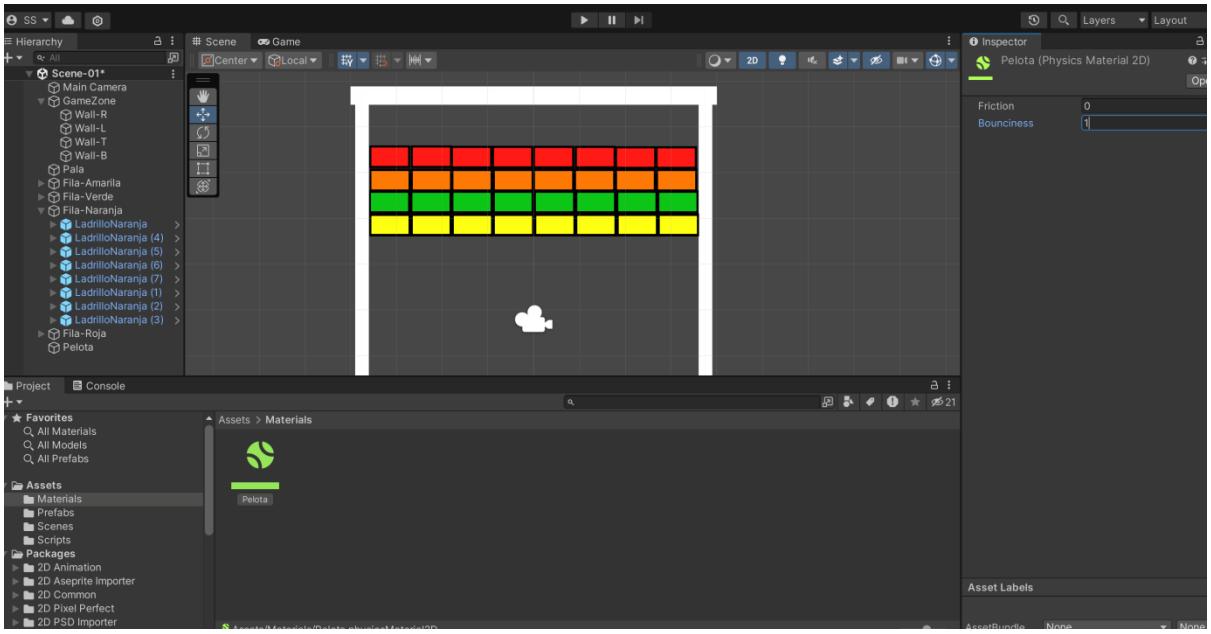
PELOTA

Vamos a crear una pelota agregando un cuadrado a nuestra escena con las propiedades de collider y rigidbody, y realizando algunos ajustes:



- Congelaremos la rotación en el eje z.
- Reduciremos la masa.
- Eliminaremos la gravedad.
- Eliminaremos los rozamientos.

- Asignaremos un material (Create->2D->Physic Material 2D) para mejorar el rebote de la pelota, eliminando el rozamiento y configurando el rebote a 1.



Comportamiento

Vamos a generar un script llamado “PelotaController” y se lo asignaremos a la pelota, añadiendo los siguientes componentes:

- La pelota se lanzará desde el centro de la pantalla. La posición y velocidad se resetearán (a 0) antes del lanzamiento.
- Siempre se lanzará hacia abajo con un ángulo fijo, por ejemplo, 45º.
- Estableceremos un retardo antes del lanzamiento, por ejemplo, 1.5 segundos.

Para lograr esto, definiremos un método que nos permita relanzar la pelota en determinadas situaciones. Primero, resetearemos la posición y la velocidad de la pelota:

```
public class PelotaController : MonoBehaviour
{
    Rigidbody2D rb;

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

    void Update()
    {

    }

    private void LanzarPelota(){
        transform.position = Vector3.zero;
        rb.linearVelocity = Vector2.zero;
```

Luego, calcularemos la velocidad y dirección para que tenga un ángulo de 45º y la pelota vaya hacia abajo:

```
        float dirX, dirY = -1;
```

```
        dirX = Random.Range(0, 2) == 0 ? -1 : 1;
        Vector2 dir = new Vector2(dirX, dirY);
        dir.Normalize();

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

Finalmente, invocaremos al método con un retardo (serializable):

```
    [SerializeField] float delay;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        Invoke("LanzarPelota", delay);
    }
}
```

RELANZAR PELOTA

Para relanzar la pelota cada vez que salga por el límite inferior, implementaremos el método OnTriggerEnter2D(). En este caso, la pelota atravesará la pared en lugar de producir una colisión, por lo que utilizaremos este método.

```
private void OnTriggerEnter2D(Collider2D other) {
    // Comprobamos si el objeto que estamos atravesando es la pared inferior
    if(other.tag == "ParedInferior") {
        // Volvemos a lanzar la pelota
        Invoke("LanzarPelota", delay);
    }
}
```

Luego, invocaremos al método de lanzar pelota.

DESTRUCCIÓN DE LADRILLOS

Para destruir los ladrillos, implementaremos el comportamiento en el método OnCollisionEnter2D, que se invocará cada vez que la pelota colisione con algún objeto de la escena. Lo primero que debemos hacer es determinar con qué objeto estamos colisionando:

```
private void OnCollisionEnter2D(Collision2D other) {
    // Almacenamos la etiqueta del objeto con el que estamos colisionando
    string tag = other.gameObject.tag;
}
```

Para identificar en qué tipo de ladrillo estamos colisionando, utilizaremos un diccionario donde almacenaremos las etiquetas y las puntuaciones de cada uno de ellos:

```
// Estructura donde almacenaremos las etiquetas y la puntuación de cada ladrillo
Dictionary<string,int> ladrillos = new Dictionary<string, int>(){
    {"ladrillo-Amarillo", 10},
    {"ladrillo-Verde", 15},
    {"ladrillo-Naranja", 20},
    {"ladrillo-Rojo", 25},
};
```

Con esta estructura de datos, podemos utilizar un método de la propia colección para consultar si una etiqueta está presente y destruir el ladrillo correspondiente:

```
private void OnCollisionEnter2D(Collision2D other) {
    // Almacenamos la etiqueta del objeto con el que estamos colisionando
    string tag = other.gameObject.tag;

    // Comprobamos si la etiqueta es un ladrillo
    if (ladrillos.ContainsKey(tag)) {
        // Destruimos el objeto
        Destroy(other.gameObject);
    }
}
```

Al ejecutar, veremos cómo los ladrillos desaparecen de la jerarquía. Probaremos con todos los ladrillos para asegurarnos de que desaparecen todos.

GOLPEO DIRIGIDO

Al ejecutar, observamos que no tenemos ningún control horizontal sobre la pelota.

En el caso de que la pelota esté colisionando con la pala, vamos a obtener la posición de la pala:

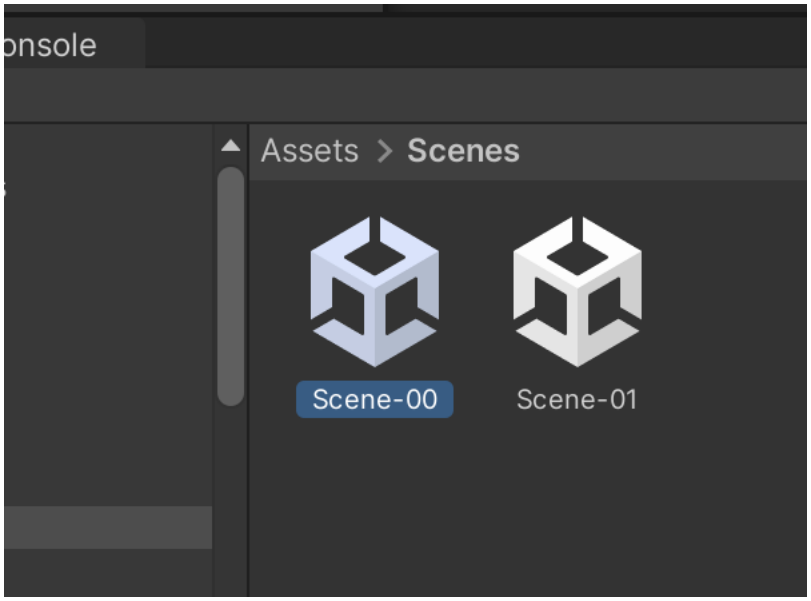
```
private void OnCollisionEnter2D(Collision2D other) {
    // Almacenamos la etiqueta del objeto con el que estamos colisionando
    string tag = other.gameObject.tag;

    // Comprobamos si la etiqueta es un ladrillo
    if (ladrillos.ContainsKey(tag)) {
        // Destruimos el objeto
        Destroy(other.gameObject);
    }

    if (tag == "Pala") {
        // Obtenemos la posición de la pala
        Vector3 pala = other.gameObject.transform.position;
        // Obtenemos el punto de contacto. Cuando colisionan dos objetos, colisionan en una superficie, y devolvería todos los puntos donde colisionan. Nos quedamos con el primero
        Vector2 contact = other.GetContact(0).point;

        // Comprobamos la dirección en x (para saber si está viajando hacia la izquierda o a la derecha)
        // Si la pelota está viajando desde la izquierda hacia la derecha y está golpeando con la parte derecha de la pala
        // o si la pelota está viajando desde la derecha hacia la izquierda y está golpeando con la parte izquierda de la pala
        if(rb.linearVelocity.x < 0 && contact.x > positionPaddle.x ||
            rb.linearVelocity.x > 0 && contact.x < positionPaddle.x){
            rb.linearVelocity = new Vector2(-rb.linearVelocityX, rb.linearVelocityY);
        }
    }
}
```

SCEENES

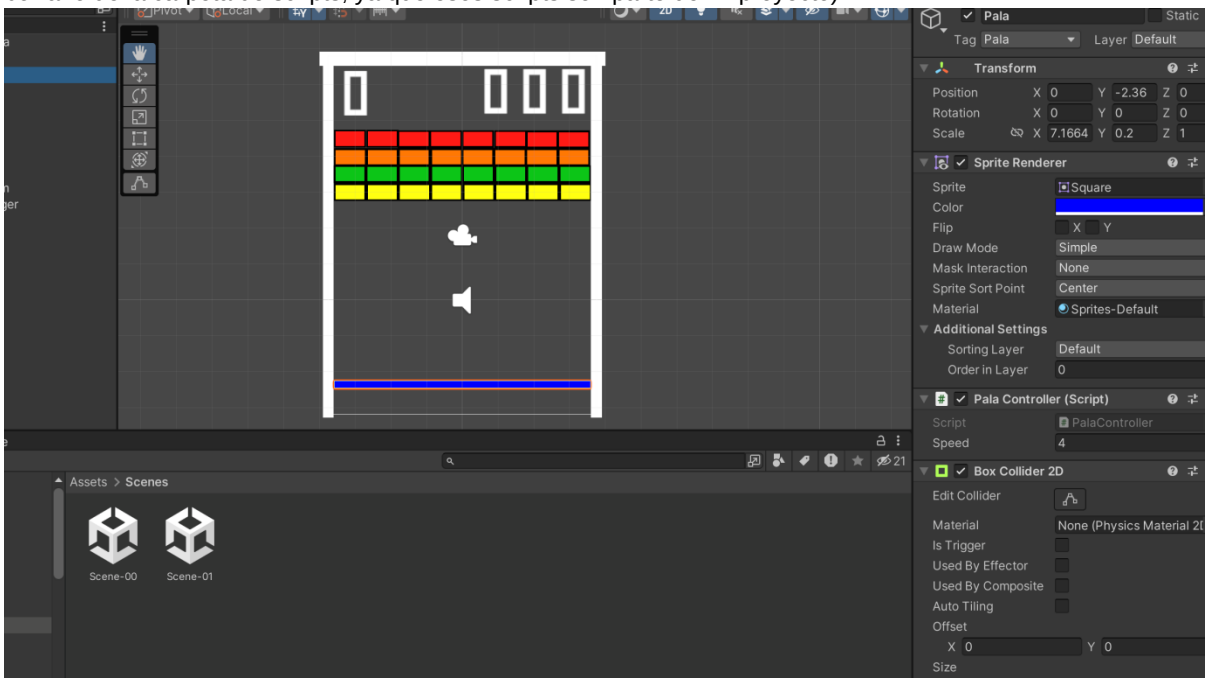
Vamos a introducir múltiples escenas en nuestro juego para permitir la transición entre distintas pantallas.

SCENE-00

Estamos creando una escena de presentación similar al juego GamePlayer, donde la pelota rebota hasta que se inicia el juego. Una vez iniciado, la barra inicial (que ocupa toda la pantalla) se reducirá. Dado que esta escena será muy similar a la actual, vamos a duplicarla:



Nuestro primer paso en esta nueva escena es ajustar el tamaño de la pala para que ocupe todo el ancho de la pantalla. Como la pala no se moverá y no necesitaremos el script del GameManager, podemos eliminarlo (sin borrarlo de la carpeta de scripts, ya que esos scripts son parte de mi proyecto):



Luego, vamos a cambiar el comportamiento de la pelota. Para ello, necesitaremos un nuevo **script** para la pelota. Dado que utilizaremos muchos métodos del script anterior, podemos **duplicarlo** y asignarle un nuevo nombre: `PelotaController0`. Al abrir este nuevo script, lo primero que haremos será cambiar el nombre de la clase para que coincida con el nombre del archivo.

Eliminaremos todo el código que no sea necesario, conservando únicamente el código relacionado con el sonido y el movimiento de la pala:

Aquí tienes el código sin cambios:

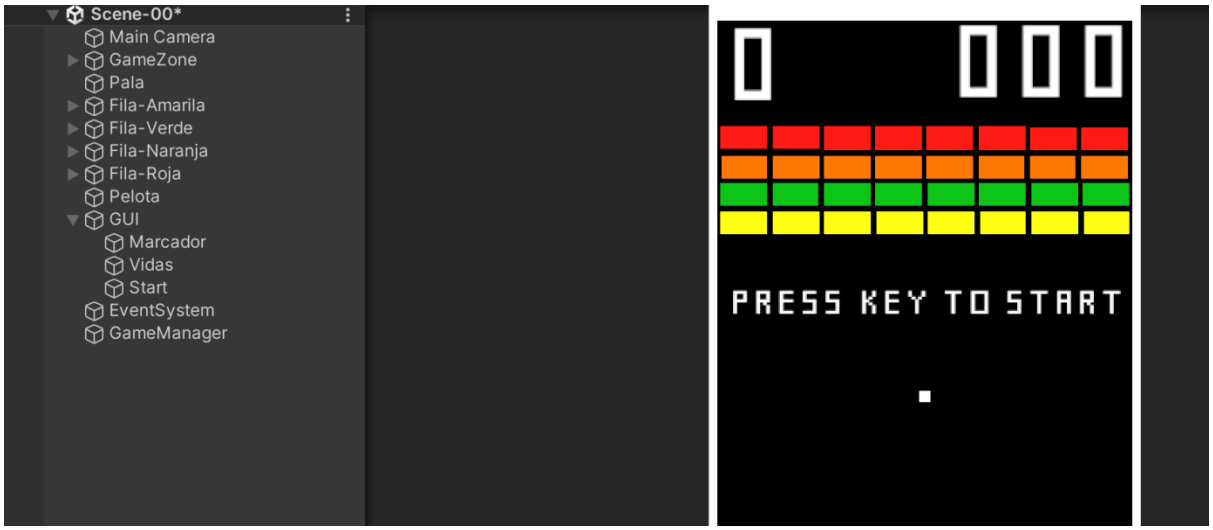
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PelotaController0 : MonoBehaviour
{


    Rigidbody2D rb;

    AudioSource sfx;
    [SerializeField] AudioClip sfxPaddel;
    [SerializeField] AudioClip sfxBrick;
    [SerializeField] AudioClip sfxWall;


    //Será serializable para poder acceder a través de Unity
    [SerializeField] float force;

    [SerializeField] float delay;
    Dictionary<string,int> ladrillos = new Dictionary<string, int>(){
        {"Ladrillo-Amarillo", 10},
        {"Ladrillo-Verde", 15},
        {"Ladrillo-Naranja", 20},
        {"Ladrillo-Rojo", 25},

    };

    void Start()
    {
        sfx = GetComponent<AudioSource>();
        //Inicializamos la variable RigidBody
        rb = GetComponent<Rigidbody2D>();
        Invoke("LanzarPelota", delay);
    }



    private void LanzarPelota(){

        //Reseteamos posición y velocidad
        transform.position = Vector3.zero; //Accedemos directamente a la posición de la pelota y la reseteamos a la posición (0,0,0)
        //Para resetear la velocidad debemos acceder al componente RigidBody
        rb.velocity = Vector2.zero;

        //Obtener una dirección aleatoria en el eje X
        float dirX, dirY = -1; //En el eje y siempre se desplazará hacia abajo
        //Queremos un ángulo 45º por lo que tiene que ser el mismo valor en eje X y en eje Y. Como en Y es -1
        dirX = Random.Range(0,2) == 0 ? -1 : 1;  //Como son enteros son exclusivos
        Vector2 dir = new Vector2(dirX, dirY);
        dir.Normalize(); //Podemos normalizarlo (no es necesario)

        //Aplicamos una fuerza determinando la dirección y una fuerza
        rb.AddForce(dir * force, ForceMode2D.Impulse);
```

```
    }

    private void OnCollisionEnter2D(Collision2D other) {

        //Almacenamos la etiqueta que tiene el objto con el que estamos colisionando
        string tag = other.gameObject.tag;


        //Comprobamos si estamos chocando con una pared
        if(tag == "ParedDerecha" || tag == "ParedIzquierda" || tag == "ParedSuperior" ){

            sfx.clip = sfxWall;
            sfx.Play();
        }

        //Comprobamos si la etiqueta es un ladrillo
        if(ladrillos.ContainsKey(tag)){
            sfx.clip = sfxBrick;
            sfx.Play();
        }

        else if(tag == "Pala"){

            sfx.clip = sfxPaddel;
            sfx.Play();
        }
    }


}
```

Necesitamos asignar todos los sonidos y parámetros como serializables. Sería conveniente crear una única clase para gestionar todos los sonidos. Actualmente, la pelota está rebotando continuamente entre los diferentes objetos de nuestra escena.

Inicio

Ahora vamos a crear un mensaje de texto para indicar al jugador que pulse una tecla para comenzar a jugar.



Color LERP

Vamos a agregar una animación con un cambio gradual de color al texto para hacerlo más atractivo. Para lograr esto, utilizaremos corutinas. Necesitaremos agregar un GameObject y adjuntarle un nuevo script donde implementaremos la lógica para cambiar el color del texto. Para ello, primero necesitaremos una referencia al texto:

```
using UnityEngine.UI;

public class TextColorLerp : MonoBehaviour
{
    [SerializeField] Text msg;
```

Ahora crearemos un método encargado del cambio de color. Este cambio de color tomará un cierto tiempo, el cual podemos definir mediante un campo serializable:

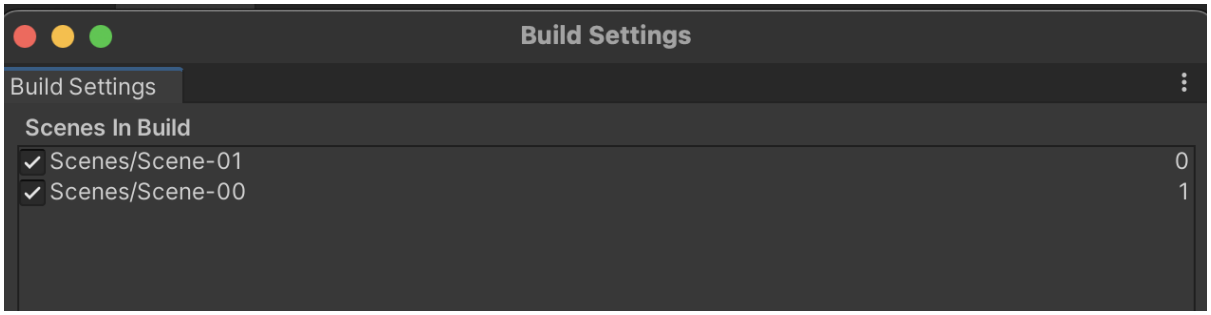
```
    [SerializeField] float duration;
```

Dado que este método se llamará como una corutina, debe devolver un IEnumerator. Las corutinas ejecutan parte del código en cada frame, deteniéndose en los puntos que definimos con `yield`, y continuando desde ese punto en la próxima iteración. La función LERP nos proporcionará el color correspondiente a un instante dado (que obtendremos con `Time.deltaTime`):

```
    void Start()
    {
        StartCoroutine("ChangeColor");
    }

    IEnumerator ChangeColor()
    {
        float t = 0;
        while(t < duration)
        {
            t += Time.deltaTime;
            msg.color = Color.Lerp(Color.black, Color.white, t / duration);
            yield return null;
        }

        StartCoroutine("ChangeColor");
    }
```

CAMBIO DE ESCENA

Para cambiar de escena, disponemos de un método llamado `loadScene`, el cual nos permite pasarle el nombre o el índice de la escena que deseamos cargar. Este índice puede encontrarse dentro del menú “File” -> “Build Settings”, donde se muestra una lista de “Scene in Build” junto con sus respectivos índices:



Para utilizar este método, creamos un nuevo script llamado StartGame y lo asignamos a un objeto vacío. Invocaremos el método `LoadScene` pasándole el índice de la escena a la que deseamos ir:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class StartGame : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        if(Input.anyKeyDown()){
            SceneManagement.LoadScene(0);
        }
    }
}
```

Amimación cambio de escena

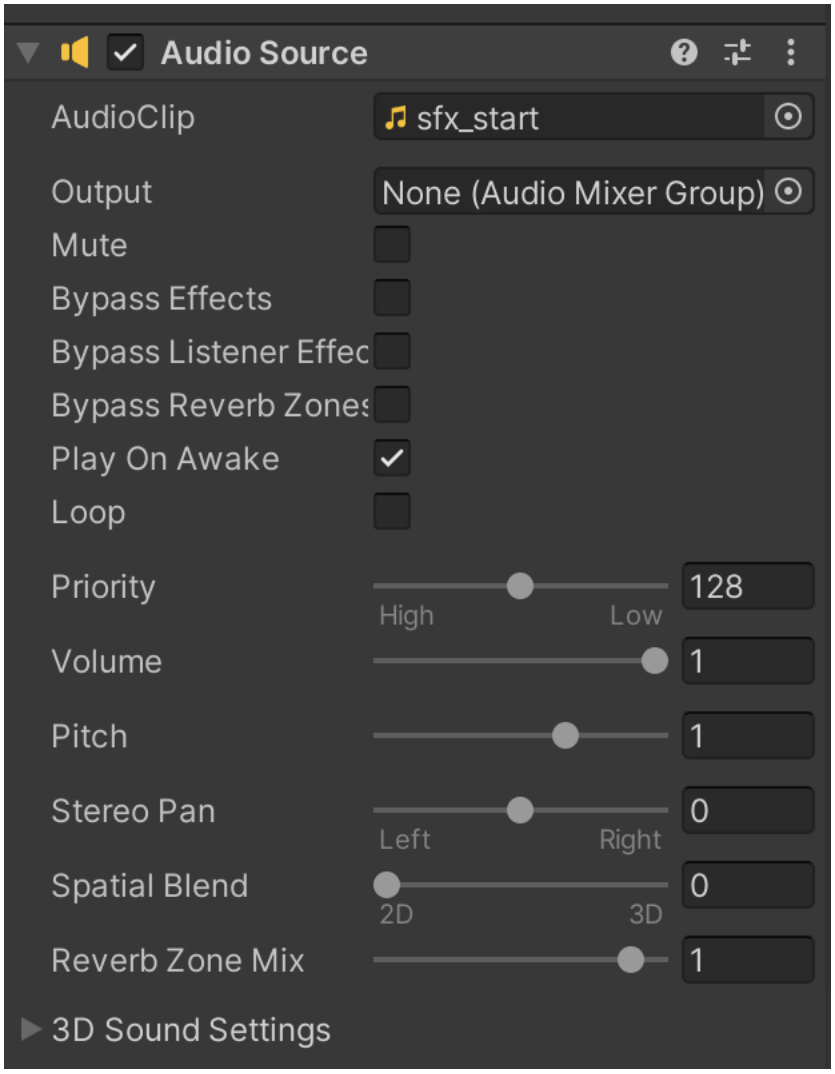
Al presionar cualquier tecla, la barrera se **reducirá progresivamente** en tamaño horizontal hasta desaparecer, avanzando luego a la siguiente escena. En la escena inicial, eliminaremos la barrera y el cambio de escena. Ajustaremos gradualmente la escala (localScale) del componente transform de la barrera mediante interpolación lineal, utilizando la función Lerp de la clase Vector3. No avanzaremos a la siguiente escena hasta que la barrera haya desaparecido y se produzca un sonido inicial. s

```
public class StartGame : MonoBehaviour
{
    [SerializeField] AudioSource sfx;
    [SerializeField] Transform paddle;
    [SerializeField] GameObject ball;
    [SerializeField] float duration;

    void Update()
    {
        if(Input.anyKeyDown){
            StartCoroutine("StartNextLevel");
        }
    }

    IEnumerator StartNextLevel(){
        Vector3 scaleStart = paddle.localScale;
        Vector3 scaleEnd = new Vector3(0, scaleStart.y, scaleStart.x);

        float t = 0;
        while(t< duration){
            t += Time.deltaTime;
            paddle.localScale = Vector3.Lerp(scaleStart, scaleEnd, t/duration);
            yield return null;
        }
    }
}
```

Opcionalmente, elimina la pelota y reproduce algún sonido de inicio del juego.

Sonido inicial

Reproduciremos un sonido inicial. Dado que siempre será el mismo, podemos asignarlo directamente desde el editor.

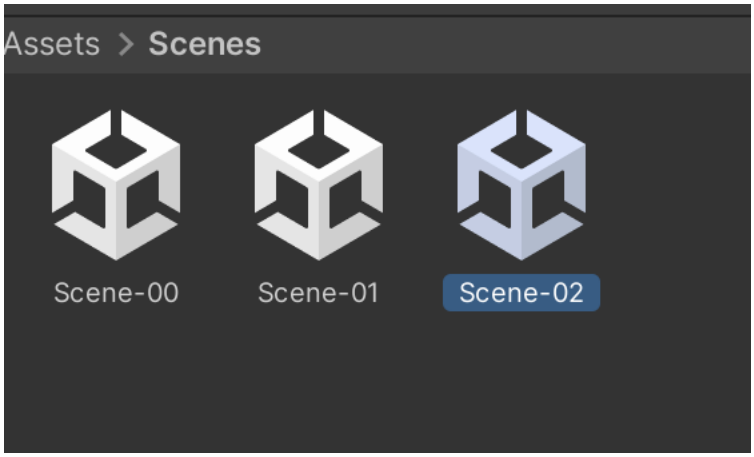


SCENE-02

Para crear un nuevo nivel a partir del primer nivel del juego, diseñaremos una nueva escena (Scene-02):

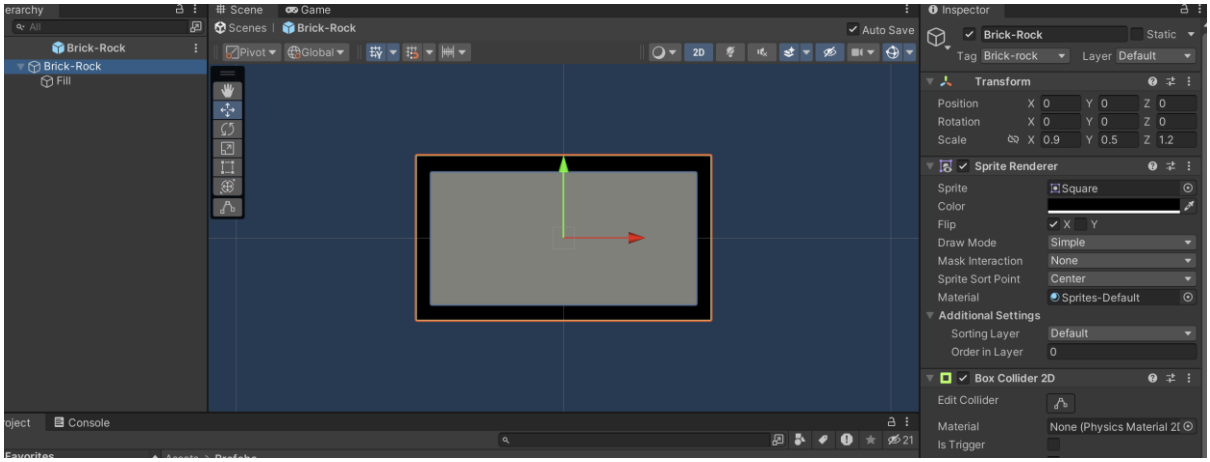
- Recuerda establecerles un Collider2D adecuado y asignarles un tag para identificar las colisiones.
- Puedes crear nuevos prefabs de ladrillos con diferentes colores y tamaños.
- Puedes crear ladrillos “indestructibles” o que no se destruyan.

Para crear la Scene-02, duplicaremos la Scene-01 y la pegaremos:



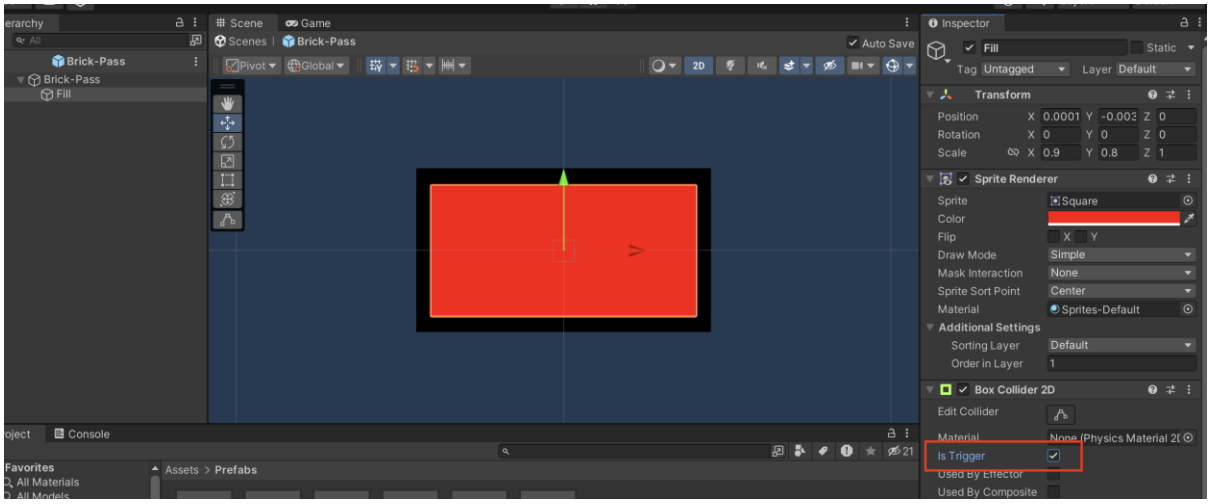
Ladrillo Indestructible

Vamos a crear un ladrillo que no se pueda destruir. En apariencia, será similar a los ladrillos existentes. Para ello, podemos copiar cualquier prefab y cambiarle el nombre, la etiqueta y el color del relleno a gris:



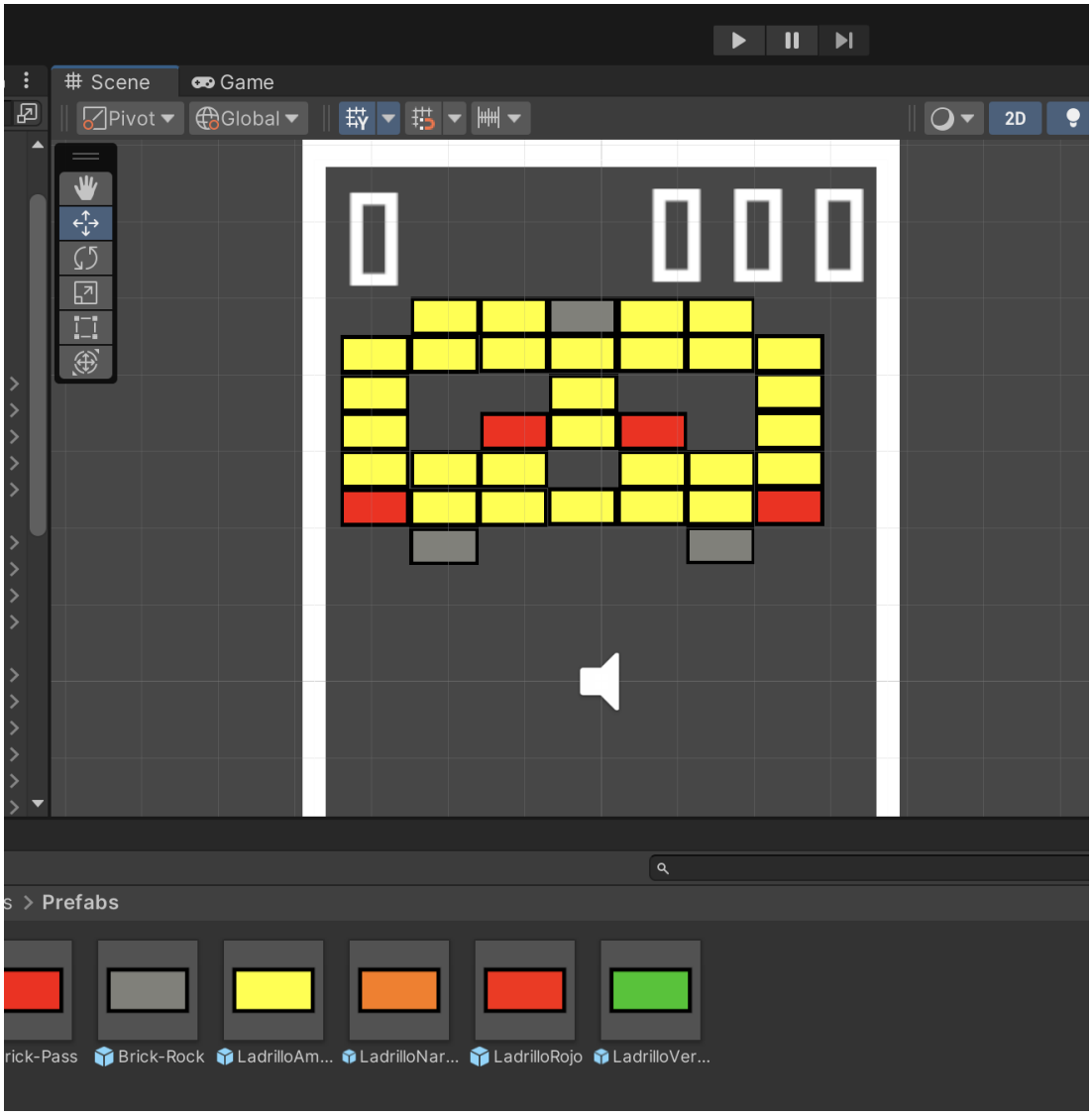
Ladrillo Atravesable

Crearemos un ladrillo que no se pueda atravesar. En apariencia, será similar a los ladrillos existentes. Para ello, podemos copiar cualquier prefab y cambiarle el nombre, la etiqueta y el color del relleno a rojo. Además, activaremos la opción “IsTrigger” en su BoxCollider:



Diseño de la Nueva Escena

Crearemos una nueva escena con una configuración de ladrillos diferente, utilizando los nuevos ladrillos generados:



Todos los comportamientos asociados a la pelota deben funcionar, ya que hemos copiado la escena. Sin embargo, hay algunas cosas que no funcionarán correctamente relacionadas con los nuevos ladrillos. Para darles comportamiento a los nuevos ladrillos, debemos editar “PelotaController”:

Agregaremos un sonido de choque si la pelota colisiona con un ladrillo no atravesable:

```
private void OnCollisionEnter2D(Collision2D other) {
    // Almacenamos la etiqueta del objeto con el que estamos colisionando
    string tag = other.gameObject.tag;

    // Verificamos si estamos chocando con una pared
    if(tag == "ParedDerecha" || tag == "ParedIzquierda" || tag == "ParedSuperior" || tag == "Brick-rock" ){
        sfx.clip = sfxWall;
        sfx.Play();
    }
}
```

Añadiremos al diccionario los ladrillos rojos que atraviesan y les daremos la misma puntuación que a los ladrillos rojos:

```
Dictionary<string,int> ladrillos = new Dictionary<string, int>(){
    {"Ladrillo-Amarillo", 10},
    {"Ladrillo-Verde", 15},
    {"Ladrillo-Naranja", 20},
    {"Ladrillo-Rojo", 25},
    {"Ladrillo-Pass", 25},
    {"Ladrillo-Rojo", 25},
};
```

Crearemos una función para destruir ladrillos:

```
void DestroyBrick(GameObject obj){
    sfx.clip = sfxBrick;
    sfx.Play();
    // Actualizamos la puntuación
    gameManager.UpdateScore(ladrillos[obj.tag]);
    // Se destruye el objeto
    Destroy(obj);
}
```

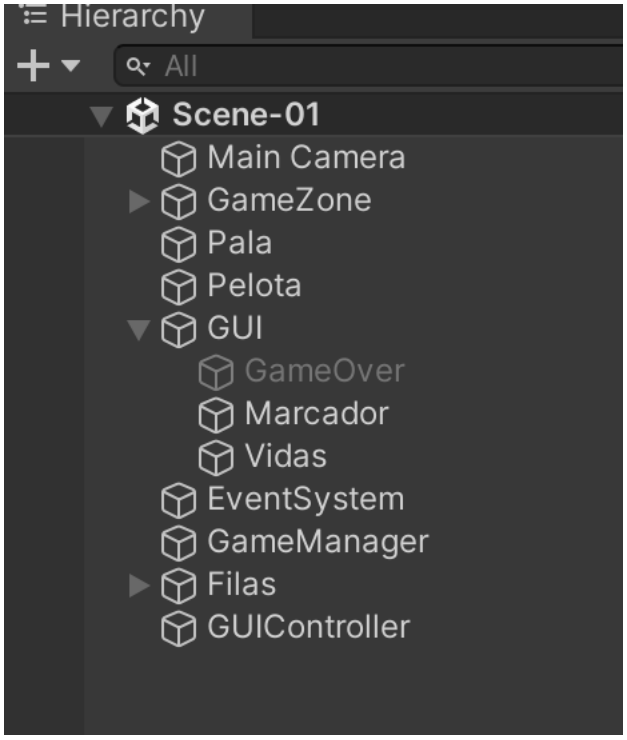
Luego, llamaremos a esta función desde las dos funciones:

```
private void OnCollisionEnter2D(Collision2D other) {
    // Verificamos si la etiqueta es un ladrillo
    if(ladrillos.ContainsKey(tag)){
        DestroyBrick(other.gameObject);
    }
}

private void OnTriggerEnter2D(Collider2D other) {
    // Si atravesamos un ladrillo rojo atravesable
    if(other.tag=="Brick-Pass"){
        DestroyBrick(other.gameObject);
    }
}
```

PROPAGAR DATOS

Entre los diferentes niveles, necesitamos transferir información como el número de vidas, contador, etc. Actualmente, estas escenas son **independientes** y todos los objetos de la escena anterior se destruyen, por lo que no conservamos información anterior. Existen múltiples maneras de pasar información entre diferentes escenas.

Para lograr esto, primero eliminaremos el GameManager en la “Scene-02”, ya que este es responsable de llevar el control de vidas y puntos, y establece la puntuación y el número de vidas en 0 al iniciarse la escena.



Para pasar la información a la Scene-02, debemos volver a la Scene-01 y en el GameManager, definiremos los atributos que deseamos transferir a la Scene-02 como atributos estáticos:

```
public class GameManager : MonoBehaviour
{
    public static int score { get; private set; } = 0;
    public static int lifes { get; private set; } = 3;

    public static void UpdateScore(int points) { score += points; }

    public static void UpdateLifes() { lifes--; }
}
```

Al hacer estos atributos estáticos, son atributos de clase compartidos por todos los objetos. Los hacemos públicos para poder verlos en Unity, pero solo queremos que sean públicos para lectura, no para escritura.

De esta forma, podemos acceder a ellos a través de la clase sin necesidad de instanciarla. Por lo tanto, en el script de Pelota podemos eliminar la instancia de la clase:

```
//[SerializeField] GameManager gameManager;
```

Y modificaremos las llamadas a los métodos de actualización de puntos y vidas:

```
GameManager.UpdateScore(ladrillos[obj.tag]);
GameManager.UpdateLifes();
```

Del GameManager, vamos a eliminar la gestión de los componentes de texto y trasladarla a un nuevo script que se encargará de toda la parte de visualización de texto:

NOTFOUND

En este nuevo script, trasladaremos toda la gestión del texto que actualmente está en el GameController:

NOTFOUND

Para poder hacer referencia a las variables de vidas y puntuación, accederemos al GameController:

```
private void OnGUI() {
    // Actualizar el texto
    txtScore.text = string.Format("{0,3:D3}", GameManager.score); // Queremos formatearlo a 3 dígitos
    // Primer cero, el índice de los valores de la lista que se va a introducir, cuántos caracteres vamos a querer y el formato va a ser dígitos a 3.

    // Actualizamos marcador
    txtLives.text = GameManager.lifes.ToString();
}
```

Comprobaremos que en la primera escena todo funciona correctamente. Ahora, en la **Scene-02**, no tenemos el GameManager para gestionar la actualización de las vidas, por lo que tendremos que actualizar esa visualización de las puntuaciones y vidas. Para ello, crearemos otro objeto llamado GUIController al que asignaremos ese script. Este script se encargará de actualizar la puntuación basándose en la clase estática.

Al ejecutar la Scene-02, veremos que los marcadores se actualizan correctamente. Ahora, lo que nos falta es propagar los datos desde el nivel anterior. Para ello, primero nos aseguraremos de que las tres escenas estén añadidas a BuildSettings y actualizaremos los IDs para que sean los correctos:

NOTFOUND

Ahora, generaremos una nueva variable estática para almacenar la cantidad de ladrillos destructibles en nuestra escena:

```
public class GameManager : MonoBehaviour
{
    // Se asocia el número de ladrillos destructibles al ID de la escena (Scene-0: 0 ladrillos, Scene-1: 32)
    public static List<int> totalBricks = new List<int> {0, 32, 32};
}
```

En el controlador de la Pelota, podemos crear una variable que aumentará cada vez que golpeemos un ladrillo:

```
public class PelotaController : MonoBehaviour
{
    // Acumulamos los ladrillos que destruimos
    int brickCount;
}
```

En el método para destruir ladrillos, actualizamos este valor y comprobamos si excedemos el número de ladrillos:

```
public void DestroyBrick(GameObject obj){
    sfx.clip = sfxBrick;
    sfx.Play();
    // Actualizamos la puntuación
    GameManager.UpdateScore(ladrillos[obj.tag]);
    // Se destruye el objeto
```

```
    Destroy(obj);
    // Actualizamos el contador de ladrillos destruidos
    ++brickCount;
    // Comprobamos si hemos alcanzado el máximo de ladrillos. Necesitamos el índice de la escena en la que nos encontramos para saber cuántos ladrillos tenemos.
    if(brickCount == GameController.totalBricks[SceneManager.GetActiveScene().buildIndex]){
        // Aquí añadiremos el código para avanzar al siguiente nivel
    }
}
```

Para conocer el número de ladrillos, necesitamos el ID de la escena. Para ello, declaramos una variable y la inicializamos en el método `Start`:

```
using UnityEngine.SceneManagement;

public class PelotaController : MonoBehaviour
{
    // ID de la escena
    int sceneId;

    void Start()
    {
        sceneId = SceneManager.GetActiveScene().buildIndex;
    }

    // Resto del código...
}
```

Ahora podemos invocar la carga de la siguiente escena cuando se destruyan todos los ladrillos:

```
++brickCount;
// Comprobamos si hemos alcanzado el máximo de ladrillos. Necesitamos el índice de la escena en la que nos encontramos para saber cuántos ladrillos tenemos.
if(brickCount == GameManager.totalBricks[sceneId]){
    SceneManager.LoadScene(sceneId + 1);
}
```

De esta manera, al quitar todos los ladrillos, avanzaremos a la siguiente escena.

Espera

Como mejora, podemos agregar un retraso antes de cargar la siguiente escena. Para ello, modificaremos el método `DestroyBrick` de la siguiente manera:

```
public void DestroyBrick(GameObject obj){
    sfx.clip = sfxBrick;
    sfx.Play();
    // Actualizamos la puntuación
    GameManager.UpdateScore(ladrillos[obj.tag]);
    // Se destruye el objeto
    Destroy(obj);
    // Actualizamos el contador de ladrillos destruidos
    ++brickCount;
    // Comprobamos si hemos alcanzado el máximo de ladrillos. Necesitamos el índice de la escena en la que nos encontramos para saber cuántos ladrillos tenemos.
    if(brickCount == GameManager.totalBricks[sceneId]){
        Invoke("NextScene", 3);
```

```
    }
}

void NextScene(){
    int nextId = sceneId + 1;
    if(nextId == SceneManager.sceneCountInBuildSettings){
        nextId = 0;
    }
    SceneManager.LoadScene(nextId);
}
```

En este código, hemos añadido un retraso de 3 segundos antes de llamar al método `NextScene`, que carga la siguiente escena. Además, hemos agregado una condición para que, si estamos en la última escena, la próxima escena sea la primera, completando así un ciclo.

Sonido

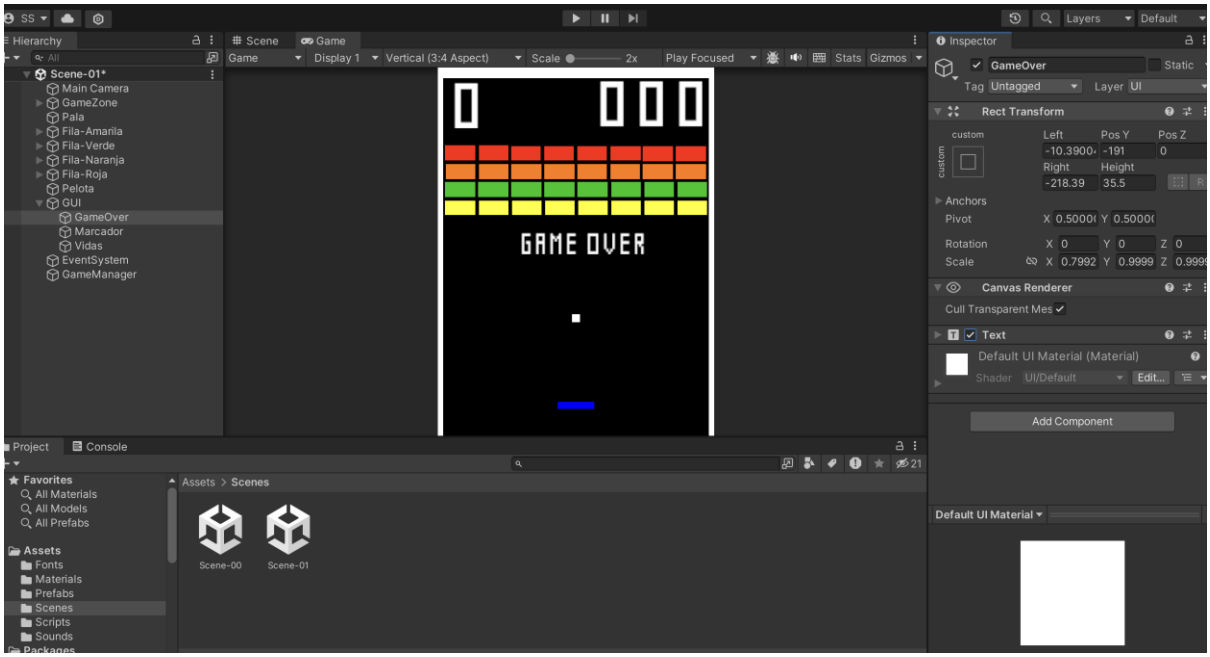
¡Claro! Podemos agregar un nuevo sonido antes de pasar a la siguiente escena y detener el movimiento de la pelota. Modificaremos el método `DestroyBrick` para incluir estas nuevas funcionalidades:

```
public void DestroyBrick(GameObject obj){
    ...
    // Comprobamos si hemos alcanzado el máximo de ladrillos. Necesitamos el índice de la escena en la que nos encontramos para saber cuántos ladrillos tenemos.
    if(brickCount == GameManager.totalBricks[sceneId]){
        // Reproducimos el sonido de transición
        sfx.clip = sfxNextLevel;
        sfx.Play();
        // Detenemos el movimiento de la pelota
        rb.velocity = Vector2.zero;
        // Invocamos el método para pasar a la siguiente escena después de 3 segundos
        Invoke("NextScene", 3);
    }
}
```

Con estos cambios, al destruir todos los ladrillos en la escena, se reproducirá un nuevo sonido antes de pasar a la siguiente escena, y la pelota dejará de moverse para evitar ejecutarse antes de pasar a la siguiente escena.

Game Over

Para añadir un nuevo componente de texto en la Escena-01 que se muestre al finalizar el juego, simplemente copiaremos el texto de “Start” y cambiaremos el texto por “Game Over”. Por defecto, estará deshabilitado:



Luego, necesitamos generar la lógica para que este texto se muestre.

Mejora
Crear una escena final con los créditos.