Mikeloid

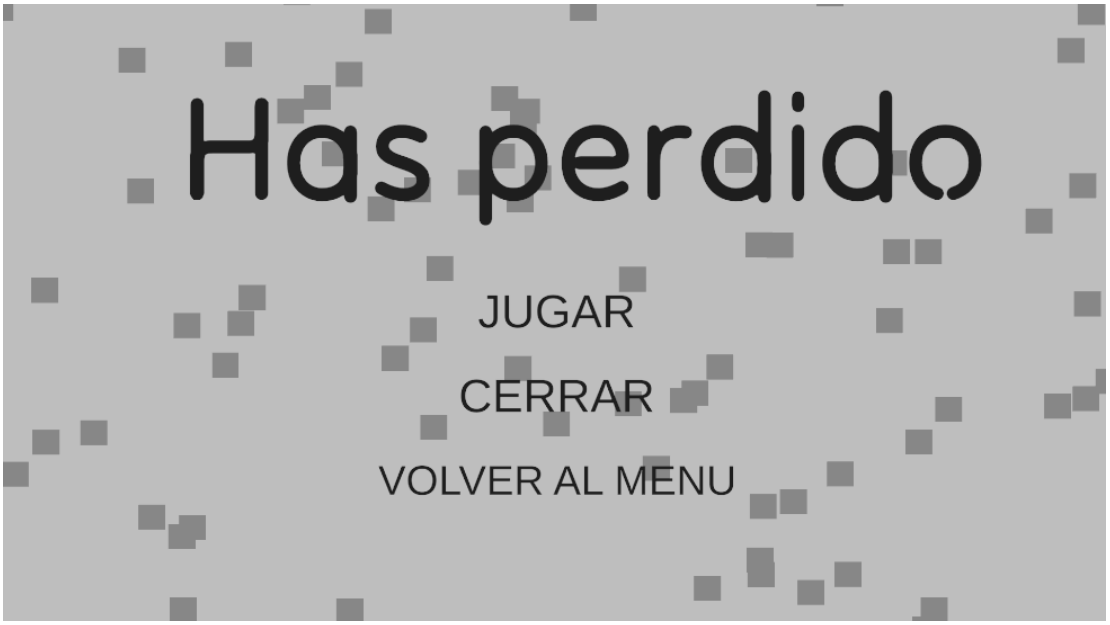
Por Miguel González Cebrián

Reglas:

Mi juego comparte las mismas reglas que el juego “Arkanoid” clásico. Para la estética se ha decidido un aspecto simple de colores planos acompañados de un material que le hace verse como si los objetos desprendieran luces de neón para darle un toque futurista, con partículas de fondo para no dejar el fondo del juego tan vacío.
El juego cuenta con un único nivel, aunque sería fácilmente expandible a más niveles, pero al ser esto una versión “beta” solo hace falta un nivel para probar que todas las mecánicas funcionan de forma correcta.

Durante la partida podemos ver bloques de diferentes colores. Estos bloques cambiarán de color dependiendo de los golpes que sean necesarios para hacerlos desaparecer, siendo así:
- Rojo: 3 golpes
- Amarillo: 2 golpes
- Verde: 1 golpe

Además de esto podemos ver en pantalla un texto que nos indica que debemos pulsar espacio para poder lanzar la pelota. Este texto desaparecerá nada más pulsar espacio y reaparecerá cuando se pierda una vida.
En las esquinas podemos ver los puntos a la izquierda y las vidas a la derecha. Cada vez que un bloque sea golpeado se sumarán 200 puntos al contador y cada vez que se pierda una vida al caer la pelota más abajo del paddle, el contador de vidas a la derecha bajará y se quitará un sprite de la pelota.
Además de esto el juego contará con una pantalla de derrota y otra de victoria, desde las cuales podrás cerrar el juego, volver al menú o comenzar de juego



Código y Scripts:

El juego cuenta con diferentes scripts que realizan diferentes funciones. Se hará un pequeño resumen de cada uno:

GameManager:

Es el script más completo y tiene varias bibliotecas para gestionar diferentes partes del juego, como el cambio de escena o los textos que aparecen en pantalla.

```
using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

Script de Unity (1 referencia de recurso) | 10 referencias
public class GameManager : MonoBehaviour
{
    public static GameManager Instance; //Codigo para instanciar el GameManager
    public int lives;                   //Vidas restantes del jugador
    public int maxLives = 3;            //El maximo de vidas a las que puede aspirar el jugador
    public int bloquesEnPantalla;       //El numero de bloques que hay en pantalla.
    public int puntos = 0;              //Los puntos que tiene el jugador

    public Image[] corazones;           //Un array que almacena los spirtes de los "corazones" par el recuento de vidas
    public Sprite vidaLlena;            //El sprite de la vida
    public Sprite vidaVacia;            //El spirte de una vida perdida

    0 referencias
    public int Lives {  get { return lives; } }
```

Aquí podemos ver las bibliotecas y todas las variables declaradas

```
Mensaje de Unity | 0 referencias
private void Awake()    //Metodo para que el GameManager no se destruya
{
    if (Instance == null)
    {
        Instance = this;
        DontDestroyOnLoad(this.gameObject);
    }
    else
    {
        Destroy(this.gameObject);
    }


    this.lives = 3; //Mantener las vidas a 3
}
```

En el método awake indicamos que el GameManager no se destruya entre escenas y que si existiera otro GameManager que el más recientemente creado fuera destruido. Además de esto indicamos que las vidas iniciales del jugador serán 3.

```
Mensaje de Unity | 0 referencias
public void Update()
{

    PantallaLimpia();

    ManejarCorazonesPantalla();
}
```

En el método update tenemos 2 métodos que serán llamados en cada frame, PantallaLimpia que nos indicará si todos los bloques han sido destruidos y puede saltarse a la siguiente escena y ManejarCorazones en pantalla que comprueba la cantidad del int “lives” y muestra los sprites de corazones correspondientes.

```
1 referencia
public void PantallaLimpia() //Aqui comprobamos que el numero de bloques en pantalla sea 0 menos para acabar el juego
{
    //Asignamos a "bloquesEnPantalla" los bloques que hay en la escena
    bloquesEnPantalla = GameObject.FindGameObjectsWithTag("Bloque").Length;

    //Comprobamos que escena esta cargada y hacemos que avance a la siguiente
    if (this.bloquesEnPantalla <= 0 && SceneManager.GetSceneByName("Nivel1").isLoaded)
    {
        Debug.Log("Nivel 1 superado");
        SceneManager.LoadScene("Victoria");
    }
}

1 referencia
public void ManejarCorazonesPantalla() //Con este for comprobamos cuantas vidas tiene el jugador y cuantos sprites deben mostrarse en pantalla
{
        for (int i = 0; i < corazones.Length; i++)
        {
            if (i < lives)
            {
                corazones[i].sprite = vidaLlena;
            }
            else
            {
                corazones[i].sprite = vidaVacia;
            }
        }
}
```

En el método PantallaLimpia podemos ver como lo primero que hace es darle un valor numérico a “bloquesEnPantalla” dependiendo de cuantos objetos con el tag “Bloque” encuentre en la escena. Previamente se le ha dado a todos los bloques de la escena el tag “Bloque”. Después de esto detecta que no haya ningún bloque y que nos encontremos el Nivel1 (ya que, por ejemplo, en el menú principal tampoco hay bloques) y salta a la escena **Victoria**.

El método ManejarCorazonesPantalla es un for que coloca en el array los sprites de vidas llenas o vidas vacías dependiendo de las vidas del jugador.

```
//Metodo para añadir puntos globales
1 referencia
public void AñadirPuntos()
{
    puntos += 200;

}


//Metodo para quitar vidas a la pelota
1 referencia
public void SubstractLive()
{
    this.lives--;

    if(this.lives <= 0)
    {
        Debug.Log("Game Over");
        SceneManager.LoadScene("Muerte");
        lives = 3;
    }
}

//Metodo para añadir vidas a la pelota
2 referencias
public void AddLive()
{
    if (this.lives >= maxLives) //Solo permite hasta un numero maximo de vidas
    {
        Debug.Log("Vidas maximas alcanzadas");
    }
    else
    {
        this.lives++;
        Debug.Log("Sumas una vida, ahora tienes " + lives);
    }
}
```

Por último en el Game Manager tenemos estos 3 métodos, que a pesar de ser declarados aquí, se usan fuera de este script.

El método AñadirPuntos simplemente suma 200 al número total de puntos.

El método SubstractLive te resta vidas y además de esto si detecta que tienes 0 vidas cargará la escena **Muerte**.

El método AddLive suma vidas, pero únicamente si tienes menos vidas del máximo. El máximo se declara con la variable “maxLives”, que es 3. Si tienes más de 3 vidas saltara un mensaje en consola avisando de que no se te sumarán vidas, y si tienes menos se te sumará una vida.

Paddle:

El script paddle se dedica a manejar el paddle y los elementos que influyen en él.

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

Script de Unity (1 referencia de recurso) | 5 referencias
public class Paddle : MonoBehaviour
{
    [SerializeField]
    public float velocidad = 5.0f;              //Velocidad de la pala

    [SerializeField]
    private GameObject paredIzquierda = null;   //Game Object de la pared izquierda

    [SerializeField]
    private GameObject paredDerecha = null;     //Game Object de la pared izquierda

    private float limiteIzquierda;              //Lite hasta el que puede moverse el paddle por la izquierda
    private float limiteDerecha;                //Lite hasta el que puede moverse el paddle por la derecha
    private GameObject paddle;                  //GameObject del paddle
    public float tiempoDePowerUpGrande = 6f;    //Tiempo que durará el PowerUp que hace grande el paddle
    public float tiempoDePowerUpRapido = 6f;    //Tiempo que durará el PowerUp que hace más veloz el paddle
```

Aquí se puede ver todas las variables que se declaran en él.

```
Mensaje de Unity | 0 referencias
private void Start()
{
    CalcularLimites();

    paddle = GameObject.FindGameObjectWithTag("Paddle");
}
```

El método Start se encarga de calcular los límites hasta los que se podrá mover la pala lateralmente hasta chocar con la pared. Además de esto también encuentra el Gameobject del paddle para poder usarlo más adelante.

```
3 referencias
public void CalcularLimites() //Método para calcular los limites hasta los que se puede mover el paddle
{
    var anchoPala = this.GetComponent<SpriteRenderer>().bounds.size.x;
    var anchoParedIzquierda = this.paredIzquierda.GetComponent<SpriteRenderer>().bounds.size.x;
    var anchoParedDerecha = this.paredDerecha.GetComponent<SpriteRenderer>().bounds.size.x;

    limiteIzquierda = this.paredIzquierda.transform.position.x + anchoParedIzquierda / 2 + anchoPala / 2;
    limiteDerecha = this.paredDerecha.transform.position.x - anchoParedDerecha / 2 - anchoPala / 2;
}
```

El método CalcularLimites se usa para darle al Paddle un límite de movimiento mediante la búsqueda de todos los elementos y tras encontrarlos y declararlos calculando su posición y permitiendo al paddle moverse solo hasta ellos.

```
Mensaje de Unity | 0 referencias
void Update()
{
    MovimientoPala();

    AcabarPowerUpGrande();

    AcabarPowerUpRapido();
}
```

En el método Update tenemos los 3 últimos métodos del script; el que calcula el movimiento de la pala y los que que terminan los PowerUps.

```
1 referencia
private void MovimientoPala() //Logica del movimiento de la pala
{
    float h = (Input.GetAxisRaw("Horizontal"));

    float x = Mathf.Clamp(transform.position.x + (h * Time.deltaTime * velocidad), limiteIzquierda, limiteDerecha);

    Vector3 vector2 = new Vector2(x, transform.position.y);

    transform.position = vector2;
}
```

El método del movimiento de la pala detecta si se están pulsando las teclas horizontales y añade una fuerza al paddle para moverlo en la dirección deseada, hasta llegar a los límites calculados en el método CalcularLimites.

```
1 referencia
private void AcabarPowerUpGrande() //Si la escala es igual a la que otorga el power up, tras 5 segundos vuelve a la escala normal
{
    if (paddle != null && paddle.transform.localScale == new Vector3(4, paddle.transform.localScale.y, paddle.transform.localScale.z))
    {
        tiempoDePowerUpGrande -= Time.deltaTime;
        if (tiempoDePowerUpGrande <= 0f)
        {
            Debug.Log("Restaurando escala");
            tiempoDePowerUpGrande = 5f;
            if (paddle != null)
            {
                paddle.transform.localScale = new Vector3(3, paddle.transform.localScale.y, paddle.transform.localScale.z);
                CalcularLimites();
            }
        }
    }
}

1 referencia
private void AcabarPowerUpRapido() //Si la velocidad es igual a la que otorga el power up, tras 5 segundos vuelve a la velocidad normal
{
    if (paddle != null && velocidad==8f)
    {
        tiempoDePowerUpRapido -= Time.deltaTime;
        if (tiempoDePowerUpRapido <= 0f)
        {
            Debug.Log("Restaurando velocidad");
            tiempoDePowerUpRapido = 5f;
            if (paddle != null)
            {
                velocidad = 5f;
            }
        }
    }
}
```

Los últimos métodos son muy parecidos. Se encargan de detectar si los valores del paddle han sido modificados para concordar con los que le otorga el PowerUp correspondiente y comienza un contador declarado en tiempoDePowerUpRapido y tiempoDePowerUpGrande. Al llegar a 0 el contador el paddle vuelve a sus valores predeterminados.

Ball:

El script Ball se dedica a la pelota enteramente y a calcular cosas como su velocidad, su pérdidas de vida o sumar puntos.

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


[RequireComponent(typeof(Rigidbody2D))]
Script de Unity (1 referencia de recurso) | 0 referencias
public class Ball : MonoBehaviour
{
    [SerializeField]
    private float launchSpeed = 10f;        //Velocidad de salida de la pelota
    private bool isLaunched = false;        //Detecta si la pelota ha sido lanzada
    private Rigidbody2D rigidbodyPelota;    //El Righidbody de la pelota

    [SerializeField]
    private Transform paddle;               //Los valores del paddle

    // Posición de la bola respecto de la pala
    private Vector2 initialPosition = new Vector2 (0.35f, 3.25f);

    public GameObject pulsaText;            //El texto "Pulsa SPACE para lanzar la pelota"

    Mensaje de Unity | 0 referencias
    private void Awake()
    {
        //Obtenemos el componente Rigidbody2D
        this.rigidbodyPelota = GetComponent<Rigidbody2D>();
    }
```

Aquí se ven todas las variables declaradas del script y el método Awake, que simplemente encuentra el componente Rigidbody de la pelota.

```
Mensaje de Unity | 0 referencias
void Update()
{
    LaunchBall();

    EvitarMovimientoHorizontal();
}
```

En el método Update se encuentran dos métodos, el que se encarga de lanzar la bola al pulsar espacio y el que evita que la bola se quede atascada dando botes horizontalmente.

```
1 referencia
private void EvitarMovimientoHorizontal()
{
    if (isLaunched && Mathf.Abs(this.rigidbodyPelota.velocity.y) < 4f)
    {
        this.rigidbodyPelota.velocity = new Vector2(rigidbodyPelota.velocity.x, 10f).normalized * launchSpeed;
    }
}
```

El metodo EvitarMovimientoHorizontal detecta si la pelota va a quedarse atascada dando botes horizontalmente y le calcula un nuevo bote en otra dirección.

```
1 referencia
private void LaunchBall() //Logica para lanzar la bola
{
    // Si pulsamos la tecla Space y la pelota no ha sido lanzada, la lanzamos
    if(Input.GetKeyDown (KeyCode.Space) && isLaunched == false)
    {
        //Establecemos la pelota como lanzada
        this.isLaunched = true;

        //Sacamos a la pelota de la pala
        this.transform.parent = null;

        //Establecemos una direecion aleatoria de lanzamiento
        float randomDirecion = Random.Range (-0.3f, 0.3f);

        //Calculamos la direccion de lanzamiento de la bola
        Vector3 launchDirecion = new Vector3(randomDirecion, 1, 0).normalized;

        //Aplicamos una velocidad a la bola en la direccion calculada
        this.rigidbodyPelota.velocity = launchDirecion * launchSpeed;

        pulsaText.SetActive(false); //Apagamos el texto "Pulsa SPACE"

    }
}
```

El método LaunchBall detecta si la pelota se considera lanzada con el bool IsLaunched y la desempareja de la pala y la lanza en un ángulo aleatorio. Además de esto el texto de “Pulsa Start” se apaga.

```
Mensaje de Unity | 0 referencias
private void OnCollisionEnter2D(Collision2D collision)
{
    //Obtenemos el componente IDamagable del objeto con el que hemos colisionado
    var objectDamagable = collision.gameObject.GetComponent<IDamagable>();

    //Si el objeto es distinto de null, llamamos su metodo TakeDamage()
    if (objectDamagable != null)
    {
        objectDamagable.TakeDamage();
    }

    if (collision.gameObject.CompareTag("Bloque"))
    {
        // Añade Puntos
        GameManager.Instance.AñadirPuntos();
    }

    //Si el objeto es la pala
    var objectPaddle = collision.gameObject.GetComponent<Paddle>();
    if(objectPaddle != null)
    {
        Vector3 paddlePosition = collision.transform.position;
        float paddleWidth = collision.collider.bounds.size.x;

        //Calcula el porcentaje de la bola en la pala (0 a 1)
        float hitPercent = (transform.position.x - paddlePosition.x) / paddleWidth + 0.5f;
        hitPercent = Mathf.Clamp01(hitPercent); //Limita el porcentaje entre 0 y 1

        //Mapea el porcentaje al rango de angulos (45 a 135 grados)
        float minAngle = 45;
        float maxAngle = 135;
        float bounceAngle = Mathf.Lerp(maxAngle, minAngle, hitPercent);

        //Calcula la nueva direccion de la bola
        Vector2 newDirection = Quaternion.Euler(0, 0, bounceAngle) * Vector2.right;

        //Ajustar la velocidad manteniendo a la direccion
        this.rigidbodyPelota.velocity = newDirection.normalized * this.launchSpeed;

    }
}
```

El método OnCollisionEnter2D detecta con que está chocando la pelota y realiza diferentes cosas dependiendo si es la pala o un bloque. Si es un bloque, se le aplica al bloque el método Take Damage para que baje un golpe restante y además de esto suma 200 puntos en el GameManager. Si es la pala calcula contra que parte de la pala ha chocado para ser lanzada en ese ángulo, ajustando su trayectoria y su velocidad.

```
Mensaje de Unity | 0 referencias
private void OnTriggerEnter2D(Collider2D other) //Si la pelota toca el Collider de muerte, se activa la función de Muerte()
{
    if (other.CompareTag("Muerte"))
    {
        Muerte();
        Debug.Log("Te quedan " + GameManager.Instance.lives);
    }
}

//Si muere, le quitamos la velocidad, la anclamos al paddle, la contamos como "no lanzada", la colocamos sobre el paddle y le quitamos una vida
1 referencia
void Muerte()
{
    rigidbodyPelota.velocity = Vector2.zero;
    this.transform.parent = paddle;
    this.isLaunched = false;
    transform.localPosition = initialPosition;
    GameManager.Instance.SubstractLive();


    pulsaText.SetActive(true); //Encendemos el texto "Pulsa SPACE"
}
```

Por último el método OnTriggerEnter que detecta si el objeto ha entrado en la hitbox del objeto “Muerte”, un objeto invisible e intangible que tan solo tiene una hitbox a la que se puede entrar, y al entrar en contacto con esta realiza el método Muerte.

El método Muerte le quita la velocidad a la pelota, la vuelve a asociar al paddle, la declara como “no lanzada”, la coloca sobre el paddle y llamando al GameManager quita una vida de la pelota. Finalmente activa el texto de “Pulsa SPACE” de nuevo.

Block:

Es el script encargado de los bloques y de generar los PowerUps.

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

Script de Unity (15 referencias de recurso) | 0 referencias
public class Block : MonoBehaviour, IDamagable
{
    [SerializeField]
    private int hits;                   //Golpes necesarios para romper el bloque
    public int contadorRotos;           //Bloques que se han roto
    private PowerUp powerUp;            //PowerUp
    private GameManager gameManager;    //GameManager
```

Estas son las bibliotecas y variables usadas por este script. Además hereda de la interfaz IDamagable.

```
Mensaje de Unity | 0 referencias
public void Start()
{
    gameManager = GameObject.FindObjectOfType<GameManager>();
    RevisarColor();
}
```

En el método Start se encuentra primero encontrar el Game Manager y después el método RevisarColor, que es el encargado de cambiar el color dependiendo los golpes que le queden a los bloques.

```
2 referencias
private void RevisarColor() // Logica para elegir el color de los bloques dependiendo de los golpes que les queden
{
    switch (hits) {
        case 1:
            gameObject.GetComponent<Renderer>().material.color = new Color(0.5618103f, 0.9528302f, 0.6449021f, 0.33f); //Verde
            break;
        case 2:
            gameObject.GetComponent<Renderer>().material.color = new Color(0.83f, 0.69f, 0.33f, 0.66f); //Amarillo
            break;
        case 3:
            gameObject.GetComponent<Renderer>().material.color = new Color(0.84f, 0.33f, 0.36f, 1f); //Rojo
            break;

    }
}
```

El método funciona mediante un switch que comprueba los hits que le quedan a los bloques y les asigna un color dependiendo de esto.

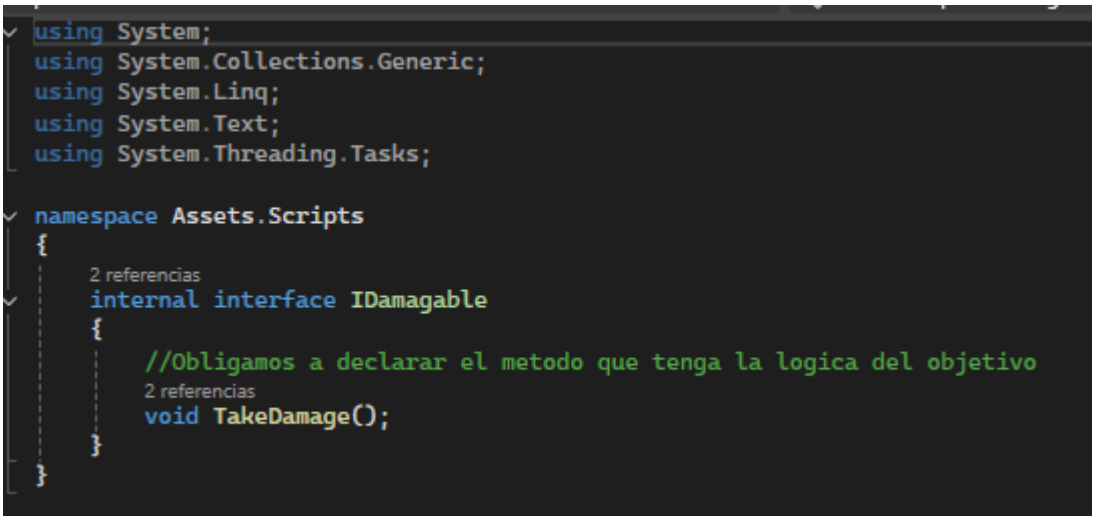
```
2 referencias
public void TakeDamage() //Logica para quitarle golpes a los bloques
{
    //Se le resta un golpe
    this.hits -= 1;
    RevisarColor();
    //Si los bloques son 0 o menos (por si hay bugs) destruimos el objeto
    if(this.hits <= 0)
    {
        contadorRotos += 1; //Suma un bloque mas roto
        this.powerUp = GeneradorPowerUp(); //Crea la posibilidad de generar un PowerUP
        Destroy(this.gameObject); //Destruye el objeto

        if (powerUp != null) //Si existe un PowerUp, ejecutalo
        {
            powerUp.Ejecutar();

        }
    }
}
```

El método TakeDamage es llamado en el script Ball y hace diferentes cosas, como quitarle un golpe al bloque, revisar el color para cambiarlo, si no quedan bloques destruir el objeto y sumar un bloque al total de bloques que se han roto. Por último genera la posibilidad de dar un PowerUp.

```
1 referencia
private PowerUp GeneradorPowerUp() //Logica para generar los PowerUps cuando se rompe un bloque
{
    PowerUp powerUpADevolver = null;

    int random = Random.Range(1, 7); //Crea un numero aleatorio
    switch (random)
    {
        case 1:
            powerUpADevolver = new PaddleGrande();
            break;
        case 2:
            powerUpADevolver = new PaddleGrande();
            break;

        case 3:
            GameManager.Instance.AddLive();
            break;

        case 4:
            GameManager.Instance.AddLive();
            break;

        case 6:
            powerUpADevolver = new PaddleRapido();
            break;
        case 7:
            powerUpADevolver = new PaddleRapido();
            break;
    }

    return powerUpADevolver;
}
```

Por último el método GeneradorPowerUp, que mediante un switch que detecta un número random, da un Power Up u otro dependiendo del número aleatorio asignado a cada uno.

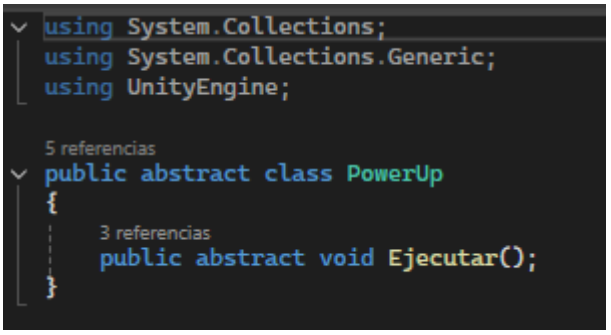
IDamagable:



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Assets.Scripts
{
    2 referencias
    internal interface IDamagable
    {
        //Obligamos a declarar el metodo que tenga la logica del objetivo
        2 referencias
        void TakeDamage();
    }
}
```

Este script se trata de una interfaz que obliga a los objetos a realizar el método TakeDamage.

PowerUp:



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

5 referencias
public abstract class PowerUp
{
    3 referencias
    public abstract void Ejecutar();
}
```

Clase que se encarga de ejecutar todo el código que herede de esta y se encuentre en métodos “Ejecutar”

PaddleGrande:

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

2 referencias
public class PaddleGrande : PowerUp
{
    private GameObject paddle;
    private Paddle paddleScript;


    2 referencias
    public override void Ejecutar()
    {

        var paddle = GameObject.FindGameObjectWithTag("Paddle"); //Encontramos el paddle
        var codigo = GameObject.FindObjectOfType<Paddle>();      //Encontramos el codigo del paddle
        if (paddle != null) //Si el paddle existe, hazlo mas largo y calcula los limites de nuevo
        {
            Transform escala = paddle.transform;

            //Declaramos la nueva escala de la pala
            escala.localScale = new Vector3(4, escala.localScale.y, escala.localScale.z);

            codigo.CalcularLimites(); //Recalculamos los limites
            Debug.Log("Alargando Pala");
        }
    }
}
```

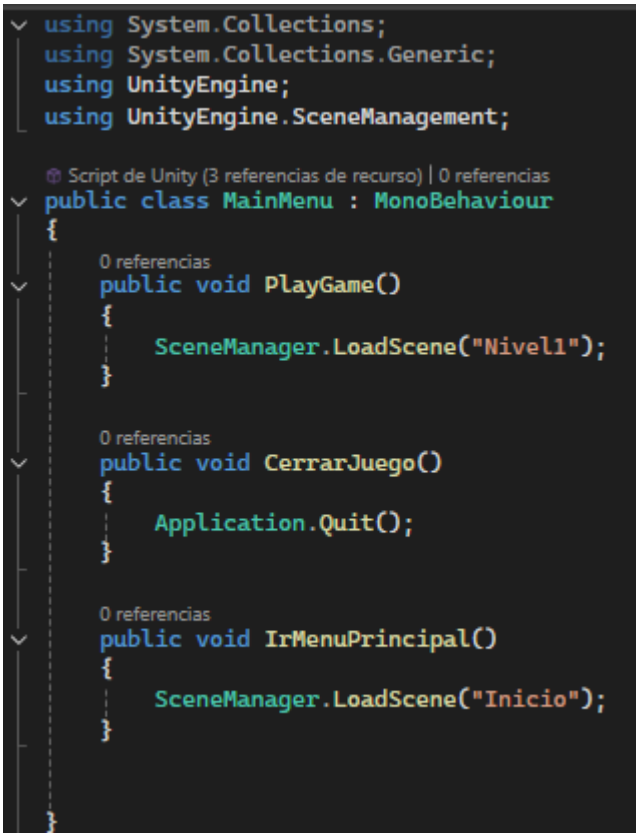
Hereda de la clase PowerUp. Se encarga de hacer grande el Paddle mediante el método Ejecutar. Esto lo hace encontrando tanto el Gameobject como el código del paddle y cambiando la escala. Tras esto vuelve a calcular los límites de movimiento del paddle.

PaddleRapido:

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

2 referencias
public class PaddleRapido : PowerUp
{
    private Paddle paddleScript;


    2 referencias
    public override void Ejecutar() //Encontramos el codigo del paddle y le subimos la velocidad
    {
        var codigo = GameObject.FindObjectOfType<Paddle>();
        codigo.velocidad = 8f;
    }


}
```

Muy similar a PaddleGrande. Hereda de la clase PowerUp. Encuentra el código del paddle y sube la variable de su velocidad a 8.

MainMenu:



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

Script de Unity (3 referencias de recurso) | 0 referencias
public class MainMenu : MonoBehaviour
{
    0 referencias
    public void PlayGame()
    {
        SceneManager.LoadScene("Nivel1");
    }

    0 referencias
    public void CerrarJuego()
    {
        Application.Quit();
    }

    0 referencias
    public void IrMenuPrincipal()
    {
        SceneManager.LoadScene("Inicio");
    }


}
```

Script encargado de crear pequeños métodos que cargan diferentes escenas para cargar escenas. Este código es usado por los botones de las escenas de **Inicio**, **Muerte** y **Victoria** para saltar entre escenas en los botones de la interfaz de estos.

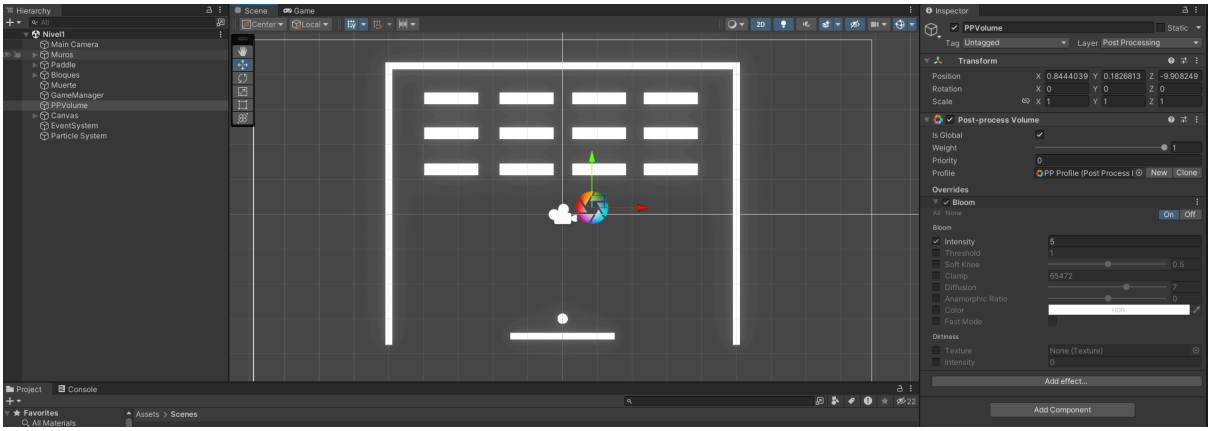
VidasTextManager y ScoreManager:

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

Script de Unity (1 referencia de recurso) | 0 referencias
public class VidasTextManager : MonoBehaviour
{

    public TextMeshProUGUI vidas;
    public GameManager gameManager;

    Mensaje de Unity | 0 referencias
    void Update() //Comprueba constamente los puntos de la variable "vidas" del Game Manager y las enseña en pantalla
    {
        vidas.text = "Vidas: " + gameManager.lives;
    }
}
```

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

Script de Unity (1 referencia de recurso) | 0 referencias
public class ScoreManager : MonoBehaviour
{

    public TextMeshProUGUI score;
    public GameManager gameManager;

    Mensaje de Unity | 0 referencias
    void Update() //Comprueba constamente los puntos de la variable "puntos" del Game Manager y los enseña en pantalla
    {
        score.text = "Puntos: " + gameManager.puntos;
    }
}
```

Códigos muy similares aplicados al texto para que muestre las vidas y los puntos en pantalla.

PPVoume:



Este es un objeto y no un código. Lo resalto aquí para explicar su funcionamiento. Es un objeto de un paquete de Unity que sirve para dar diferentes efectos a la escena. En este caso sirve para dar el efecto de Bloom y brillo a los bloques, el paddle y la pelota y que tengan este aspecto de luces de neón deseado.

Bugs conocidos:

Al terminar el nivel 1 o perder en este y volver a cargarlo desde otra escena, saldrá este error. El juego sigue siendo jugable, pero las vidas y los puntos no se contabilizarán en pantalla, pero si internamente. He probado de todo pero no he conseguido encontrar la causa del problema.