Motores de Videojuegos

Práctica 1: Flappy Halloween

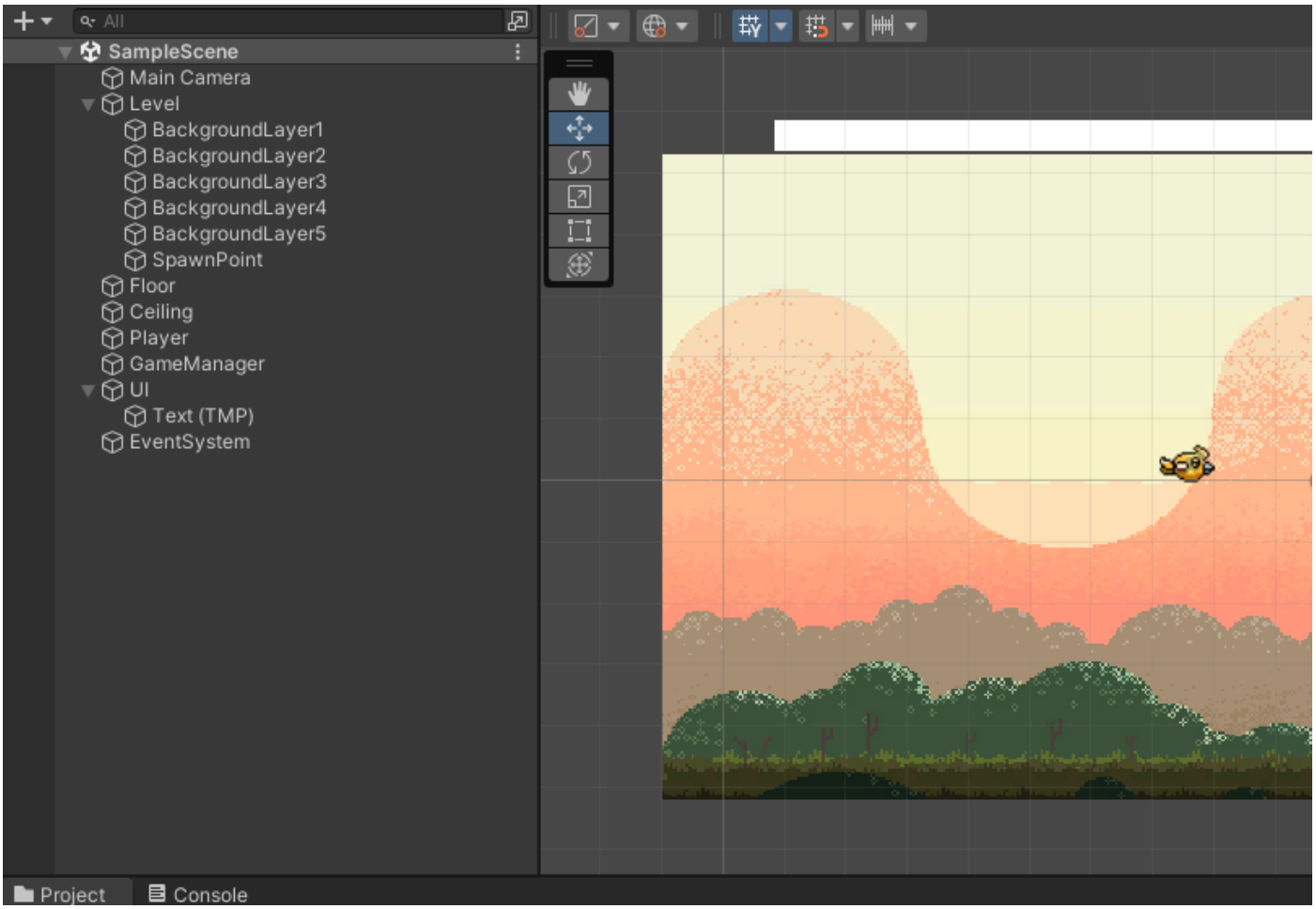
El objetivo es implementar un Flappy Birds sencillo.

Las features a implementar serán:

- Fondo con varias capas en movimiento con parallax scroll.
- Movimiento del pájaro mediante input del jugador y uso de físicas.
- Generación procedural de obstáculos.
- Implementación de condición de derrota.
- Gestión de final de partida mediante GameManager.

Jerarquía de la escena

La siguiente captura muestra la jerarquía de la escena con la práctica completada.



Organización del Proyecto

Un Proyecto debe estar siempre ordenado, cada cosa en su sitio. Asegúrate de crear el menos estas cuatro carpetas: Scenes, Scripts, Prefabs y Materials, y de ir guardando los assets siempre en la carpeta adecuada**.**

No respetar esta regla implica el suspenso inmediato en la práctica sin entrar a valorar el contenido de la misma.

Convención de Nombres

Todos los nombres del Proyecto deben respetar una convención de nombres básica que cualquier programador pueda entender.

- Obligatorio: Todos los nombres de variables, clases y funciones en inglés.
- Obligatorio: Los nombres de propiedades privadas deben empezar con "_" (Guión bajo) seguidos de minúscula.
- Obligatorio: Los nombres de clases empiezan con mayúscula.
- Obligatorio: Los nombres de métodos empiezan con mayúscula. Deben ser descriptivos de lo que hacen.
- Obligatorio: Los nombres de variables y parámetros de métodos deben ser descriptivos de para qué sirven.
- Obligatorio:Si un nombre consta de varias palabras, la segunda y sucesivas palabras del mismo tendrán la inicial en mayúscula.
- Obligatorio: Nada de líneas huérfanas. Siempre bloques de Código con llaves y retorno de carro, siguiendo las recomendaciones de Unity y Unreal.

No respetar estas reglas implica el suspenso inmediato en la práctica sin entrar a valorar el contenido de la misma.

Sesión 1: El fondo se mueve

1.1. Creación del Proyecto

Elegiremos crear un Proyecto 2D vacío.

1.2. Importación de recursos

Para importer los recursos proporcionados, deberás clicker en Assets → Import package y seguir los pasos para importer un Custom Parckage. Antes de preguntar a tus compañeros o al professor, recuerda que lo hemos visto enc lase y lo puedes buscar en internet.

1.3. Creación de la distintas capas de parallax scroll

Deberás agregarlas como sprites a la escena.

Algunos consejos:

- El material Sprite por defecto dará problemas para realizar el scroll. Mejor crea uno por defecto que utilice el shader Unlit/Transparent y asígnalo al sprite.
- Para que las capas se visualicen correctamente, asegúrate de colocarlas correctamente delante de la cámara y asignar el valor **Order in Layer** correctamente si fuera necesario.
- La importación de los sprites también puede darte problemas. En los settings de importación asegúrate de setear lo siguiente:
    - Mesh Type: Full Rect
    - Wrap Mode: Repeat
- Una vez esté todo listo, es el momento de crear el primer script.

1.4. Paralax component

1.4.1. Template

```
2.  using System.Collections;
3.  using System.Collections.Generic;
4.  using UnityEngine;
5.
6.  public class ParalaxScroller : MonoBehaviour
7.  {
8.      #region Parameters
9.      /// <summary>
10.     /// Speed used to move the texture
11.     /// </summary>
12.     [SerializeField]
13.     private float _scrollSpeed;
14.     #endregion
```

```
15.
16.     #region References
17.     /// <summary>
18.     /// Reference to own Sprite Renderer
19.     /// </summary>
20.     private SpriteRenderer _mySpriteRenderer;
21.     /// <summary>
22.     /// Reference to own Material
23.     /// </summary>
24.     private Material _myMaterial;
25.     #endregion
26.
27.     #region methods
28.     /// <summary>
29.     /// Disables the component, so the texture movement stops
30.     /// </summary>
31.     private void Stop()
32.     {
33.         //TODO
34.     }
35.     #endregion
36.
37.     // Start is called before the first frame update
38.     void Start()
39.     {
40.         //TODO
41.     }
42.
43.     // Update is called once per frame
44.     void Update()
45.     {
46.         //TODO
47.
48.     }
49. }
```

1.4.2. Explicación

Este componente debe permitir el desplazamiento de la textura variando el offset de la misma. La velocidad del desplazamiento será parametrizable

Sesión 2: El pájaro vuela

2.1. Creación del GameManager

El GameManager será un objeto que incluirá los components InputComponent, Gamemanager y ObstaclesGenerator.

- InputComponent: Recibe el Input del jugador.
- GameManager: Gestiona el input del jugador recibido, así como todos los aspectos de flujo de juego.
- ObstaclesGenerator: Genera los obstáculos que van apareciendo de forma procedimental.

En esta session nos ocuparemos de la gestion del input por parte del GameManager, y dejaremos el resto para más adelante.

2.2. InputComponent

2.2.1. Código

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

public class InputComponent : MonoBehaviour
{

    // Update is called once per frame
    void Update()
    {
        //TODO
    }
}
```

2.2.2. Explicación

En el Update, el componente debe comprobar si el jugador ha pulsado la barra espaciadora, y de ser así, informar de ello mediante SendMessage.

Observa que hemos borrado el Start porque no se va a utilizar, para evitar llamadas innecesarias por parte del motor a métodos vacíos. **La no observancia de esta práctica será causa de suspenso directo, independientemente del contenido de la práctica.**

2.3. GameManager

2.3.1. Código

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

public class GameManager : MonoBehaviour
{
    #region References
    /// <summary>
    /// Reference to player
    /// </summary>
    [SerializeField]
    private GameObject _player;
    /// <summary>
    /// Reference to UI Manager
    /// </summary>
    [SerializeField]
    private GameObject _UIManager;
    /// <summary>
    /// Reference to Level Manager
    /// </summary>
    [SerializeField]
    private GameObject _levelManager;
    #endregion

    #region properties
    /// <summary>
    /// True if the Game is running, False otherwise
    /// </summary>
    private bool _isGameRunning;
    /// <summary>
    /// Accesor to get the state of the game
    /// </summary>
    public bool IsGameRunning
    {
        get
        {
            return _isGameRunning;
        }
    }
    #endregion

    #region methods
    /// <summary>
    /// Method to send message Jump to player
    /// </summary>
    private void Jump()
    {
        //TODO
    }
    /// <summary>
    /// Method to manage finalization of the game
    /// </summary>
    private void OnPlayerDies()
    {
        //TODO
```

```
    }
    #endregion
    // Start is called before the first frame update
    void Start()
    {
        //TODO

    }

}
```

2.3.2. Explicación

- En este punto solo es necesario implementar los métodos Start y Jump.
- No olvides asignar las referencias externas en el editor.

2.4. Creación del Player

Seguramente ya habías creado un objeto Player para poder asignarlo en el GameManager. Si no lo habías hecho, hazlo ahora. Puedes hacerlo a partir de uno de los sprites de pájaro incluídos, y necesitarás añadirle:

- PolygionCollider2D → Nos proporciona colisiones que Podemos adaptar al sprite.
- RigidBody2D → Junto con el collider, nos proporciona las físicas 2D.

Además, deberemos añadir dos componentes creados por nosotros mismos:

- MovementComponent → Gestionará el movimiento del personaje
- LifeComponent → Detectará las colisiones con objetos del scenario que causarán la Muerte del jugador.

En esta session nos ocuparemos solo del MovementComponent.

2.5. Movement Component

2.5.1. Código

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

public class MovementComponent : MonoBehaviour
{
    #region parameters
    /// <summary>
    /// Force added to the player with each jump
    /// </summary>
    [SerializeField]
    private float _jumpForce;
    #endregion
    #region references
    /// <summary>
    /// Reference to own Rigid Body
    /// </summary>
```

```
    private Rigidbody2D _myRigidBody2D;
    #endregion

    #region methods
    /// <summary>
    /// Executes the jump of the player by:
    /// . Setting speed to zero
    /// . Adding the required force in vertical direction
    /// </summary>
    private void Jump()
    {
        //TODO
    }
    #endregion
    // Start is called before the first frame update
    void Start()
    {
        //TODO

    }

}
```

2.5.2. Explicación

El método Jump deberá ejecutar el salto, añadiendo una fuerza vertical al Rigid Body del objeto. Será llamado por el GameManager cuando se recibe la orden de saltar. Start deberá realizar las oportunas inicializaciones.

Observa que hemos borrado el Update porque no se va a utilizar, para evitar llamadas innecesarias por parte del motor a métodos vacíos. **La no observancia de esta práctica será causa de suspenso directo, independientemente del contenido de la práctica.**

2.6. Rematando la faena

Es buen momento para añadir Floor y Celing, dos colliders 2D que impedirán que el jugador se nos caiga al vacío o se escape por arriba. También para evitar problemas, será bueno restringir la rotación del RigidBody.

Sesión 3: Generación procedimental de obstáculos

3.1. Movimiento de los obstáculos

Deberemos crear prefabs para cada tipo de obstáculo que queramos crear. Instanciaremos al menos Pilas de paja, espantapájaros y árboles.

¡Ojo! Deberemos comprobar en el Sprite Editor que todos ellos tengan el pivot point en la base. De lo contrario, será difícil que podamos instanciarlos donde queramos. Deberemos añadirles a todos ellos un componente PolygonCollider2D, que Podemos ajustar a nuetro gusto. En el caso de los árboles permitiremos que el jugador pase por debajo.

Además, deberemos añadir a todos ellos los siguientes componentes que implementaremos nosotros:

- Lateral Movement Component
- Limited Time Life

3.2. Lateral Movement Component

3.2.1. Código

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

public class LateralMovementComponent : MonoBehaviour
{
    #region parameters
    /// <summary>
    /// Movement speed for the obstacle
    /// </summary>
    [SerializeField]
    private float _speed;
    #endregion

    #region references
    /// <summary>
    /// Reference to own transform
    /// </summary>
    private Transform _myTransform;
    /// <summary>
    /// Reference to Game Manager
    /// </summary>
    private GameManager _gameManager;
    #endregion
    // Start is called before the first frame update
    void Start()
    {
        //TODO

    }
```

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

    }
}
```

3.2.2. Explicación

El método Start deberá setear correctamente las referencias. El método Update se encargará de gestionar el movimiento.

3.3. Limited Time Life

3.3.1. Código

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

public class LimitedTimeLife : MonoBehaviour
{
    #region paramaters
    /// <summary>
    /// Maximum time before the script destroys the object
    /// </summary>
    [SerializeField]
    private float _maxLifetime;
    #endregion
    #region methods
    /// <summary>
    /// Destroys the associated game object
    /// </summary>
    private void SelfDestroy()
    {
        //TODO
    }
    #endregion
    // Start is called before the first frame update
    void Start()
    {
        //TODO

    }

}
```

3.3.2. Explicación

El método Start se encargará de que el método SelfDestroy sea llamado después del tiempo estipulado.

3.4. Obstacles Generator

Ahora sí, es el momento de añadir el componente ObstaclesGenerator al objeto Game Manager.

Este script tomará aleatoriamente objetos de una lista de prefabs y los instanciará en el nivel en la posición adecuada.

3.4.1. Código

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

public class ObstaclesGenerator : MonoBehaviour
{
    #region references
    /// <summary>
    /// Array of prefabs to instantiate as obstacles
    /// </summary>
    [SerializeField]
    private GameObject[] _prefabs;
    /// <summary>
    /// Reference to transform where obstacles will be instantiated
    /// </summary>
    [SerializeField]
    private Transform _spawnTransform;
    #endregion

    #region paramaters
    /// <summary>
    /// Maximum time between obstacles generation
    /// </summary>
    [SerializeField]
    private float _maxTimeInterval = 5.0f;
    /// <summary>
    /// Minimum time between obstacles generation
    /// </summary>
    [SerializeField]
    private float _minTimeInterval = 1.0f;
    #endregion

    #region properties
    /// <summary>
    /// Elapsed time since last obstacle generation
    /// </summary>
    private float _elapsedTime;
    /// <summary>
    /// Time interval set for next obstacle generation
    /// </summary>
    private float _nextSpawnTime;
    #endregion
```

```
    #region methods
    /// <summary>
    /// Method to disable the script and thus stop obsctacles generation
    /// </summary>
    private void Stop()
    {
        //TODO

    }

    #endregion

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

    }
}
```

3.4.2. Explicación

El método Stop será llamado cuando el juego termina para desactivar el componente.

El método Update se debe encargar de instanciar los obstáculos elegidos aleatoriamente en los intervalos de tiempo establecidos también aleatoriamente.

Sesión 4: Terminando el flujo de juego

4.1. Cartel de Game Over

Deberemos tener un objeto UI de tipo Canvas. Setear el render mode como Screen Space – Overlay nos evitará problemas. Deberemos tener otro objeto hijo de éste que contenga el propio texto “GameOver”.

Nuestro componente UIManager, de implementación propia, se encargará de activar este objeto cuando el GameManager se lo pida.

4.2. UIManager

4.2.1. Código

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

public class UIManager : MonoBehaviour
{
    #region references
    /// <summary>
    /// Reference to object containing GameOver text element
    /// </summary>
    [SerializeField]
    private GameObject _gameOverObject;
    #endregion

    #region methods
    /// <summary>
    /// Metrhod callled to activate the GameOver text element
    /// </summary>
    private void GameOver()
    {
        //TODO
    }
    #endregion
    // Start is called before the first frame update
    void Start()
    {
        //TODO
    }

}
```

4.2.2. Explicación

El método Start deberá desactivar el objeto con el texto Game Over. El método Game Over se encargará de activarlo.

4.3. LevelManager

Este componente se deberá añadir al objeto Level, padre de todos los objetos con Paralax Scroll.

4.3.1. Código

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

public class LevelManager : MonoBehaviour
{

    #region methods
    /// <summary>
    /// Stops the paralax scrollers
    /// </summary>
    private void GameOver()
    {
        //TODO
    }
    #endregion

}
```

4.3.2. Explicación

El método GameOver se encargará de parar el Paralax Scroll de todas las capas de scroll del nivel. En definitive, detiene el nivel cuando es llamado. Esta llamada deberá hacerla el GameManager cuando corresponda.

4.4. GameManager

En su momento dejamos el método OnPlayerDies del Game Manager sin implementar. Ahora por fin es el momento de hacerlo, implementando las llamadas a todos los elementos del juego que deban ser llamados cuando el jugador muere.

4.5. Life Component

Hemos implementado lo necesario para que el juego termine, pero no hemos implementado la condición que dispara el final de la partida. El componente de vida del personaje será el encargado de hacerlo, informando al GameManager de su Muerte.

4.5.1. Código

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

public class LifeComponent : MonoBehaviour
{
    #region references
    /// <summary>
    /// Reference to Game Manager object
    /// </summary>
    [SerializeField]
    private GameObject _gameManager;
    #endregion

    #region methods
    /// <summary>
    /// Executed on collision. The component informs GameManger of its death before
destroying itself.
    /// </summary>
    /// <param name="collision">Colliding element collision</param>
    private void OnCollisionEnter2D(Collision2D collision)
    {
        //TODO
    }
    #endregion

}
```

4.5.2. Explicación

Cuando el jugador colisiona con cualquier elemento, muere. Por lo tanto, debe informar al GameManager de su muerte para que éste lleve a cabo las acciones necesarias y, una vez hecho, autodestruírse.

Sesión 5

Por ultimo, queremos asegurarnos deshabilitar el Input cuando la partida termina.

¿Cómo lo harías?