

Post Mortem: IOC Invaders 2

autor: Xavier Garcia Rodríguez
versió: 1.0
data: 18 de febrer de 2022

Introducció	2
Game Design Document: IOC Invaders 2	3
Resumen	3
El joc	3
Interfície	4
Controls	4
Inspiració	4
USP	5
Historia	5
Graphics	5
Projecte Unity	6
Motor de render	6
Packages	6
Estructura del projecte	6
Escenes	6
Interficies	8
Controls	9
Prefabs	9
Background	10
Jugador (Player)	11
Enemies	12
Projectils	14
Collisions del jugador i els enemics	14
Paths	14
Scriptable Objects (Configuració dels nivells)	15
Codi C#	17
Ús de Singletons	17
Ús de Coroutines	18
Ús de Delegates	21
Scriptable Objects	24
GameLevelSO	24
LevelConfigSO	24
WaveConfigSO	25
Resum de classes	26
Enllaç a Recursos	28

Introducció

En aquest document trobareu la descomposició del projecte IOC Invaders 2, desenvolupar per Unity 2020.3. El joc és un *vertical shooter* inspirat en els *shoot'em up* dels anys 90 amb gràfics en 2D de tipus *pixel art*.

S'ha dividit el text en tres seccions: el game design document on tractem el joc com si fos una proposta amb perspectiva comercial (exagerant el nombre de nivells i enemics previstos), seguidament descrivim l'estructura del projecte, els aspectes més rellevants i els prefabs més destacables i les seves relacions. Finalment, es detallen els aspectes més rellevants del codi, mostrant la relació que hi ha entre els components.

Game Design Document: IOC Invaders 2

Resumen

- **Títol:** IOC Invaders 2
- **Jugadors:** 1
- **Gènere:** *shoot'em up* (vertical)
- **Plataformes:** navegadors, dispositius mòbils.
- **Duració:** il·limitada, els nivells es repeteixen indefinidament fins que el jugador és destruït.
- **Aspectes clau:**
 - Centenars de nivells amb diferents entorns.
 - Desenes de tipus d'enemics.
 - Gran varietat de *pickups* que deixen caure els enemics en ser destruïts.



Imatges de mostra del prototip mostrant el gameplay

El joc

Es tracta d'un *shooter* de l'estil de *Space Invaders*. La dificultat del joc anirà en increment, però el jugador aconseguirà diferents pickups que el faran més poderós a mesura que derrota a les naus enemigues. No es podrà continuar la partida, un cop la nau del jugador és destruïda ha de tornar a començar des del principi i totes les millores adquirides es perden.

Interfície

En la versió web la interfície del joc es mostrarà a la part inferior de la pantalla. El moviment del jugador estarà limitat als $\frac{2}{3}$ inferiors de la pantalla i la majoria dels enemics apareixeran a la

part superior, encara que ocasionalment poden aparèixer des de sota.

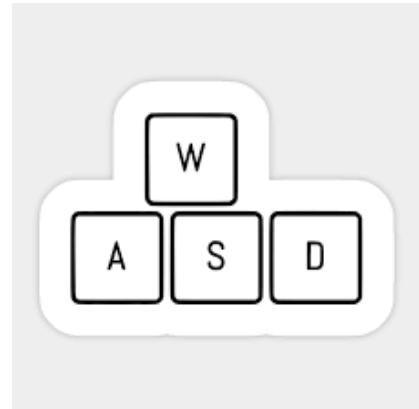
En la versió per dispositius mòbils la interfície es mostrarà a la part superior de la pantalla per deixar espai a la part inferior pels botons controladors de la nau.

Controls

La nau del jugador es pot moure en 8 direccions utilitzant combinant les tecles WASD o les fletxes de direcció.

El jugador només compta amb una altra acció addicional, disparar, que s'executarà prement la barra espaiadora o el botó esquerre del ratolí.

En la versió per dispositius mòbils s'inclourà a la part inferior els controladors de la nau. A l'esquerra els botons de moviment i a la dreta al botó per disparar.



Inspiració

El joc està inspirat en els arcades classics dels 80-90 on l'acció ràpida, les explosions i desenes de projectils omplien les pantalles. Amb aquest joc es vol recuperar l'esperit d'aquest però millorant l'aspecte visual amb nous tipus d'efectes com són el tremolor de la càmera en rebre un impacte i la utilització de partícules i efectes de postprocessament.



Aerofighters. Video System
(1992)



DoDonPachi. Cave (1997)



1942. Capcom (1984)

USP

Els punts diferenciadors d'aquest joc són:

- La gran quantitat d'entorns diferents on es porten a terme les batalles aèries.
- Els efectes visuals (VFX).
- La varietat d'enemics.
- La varietat de *pickups*.

Historia

Estem a l'any 2079 i el sistema solar IOC està sent envaït per una raça alienígena. Han ocupat tots els planetes i les forces de defensa planetàries han sigut desactivades.

L'única nau en actiu és la de la nostre heroïna, que retornava d'una missió d'exploració a l'espai profund quan es va produir l'atac.

Començant el joc a l'espai, haurà d'enfrontar-se als enemics que orbiten cadascun dels planetes per després alliberar-los utilitzant tota la potència de foc que aconsegueix reunir, abans d'enfrontar-se al cap dels alienígenes al planeta Terra-II.

Gràfics

Els gràfics del joc seran tipus pixel art, encara que per alguns dels VFX s'utilitzaran altres tipus d'efectes. **No s'intentarà aconseguir l'efecte de Pixel Perfect** (<https://juanbarcia.com/pixel-perfect/>), ja que es prefereix sacrificar la fidelitat dels gràfics per obtenir una millor qualitat.



Projecte Unity

Motor de render

El projecte utilitza el motor de render estàndard d'Unity. No és necessari utilitzar cap il·luminació ni il·luminació 2D, per tant, no ha sigut necessari canviar-lo.

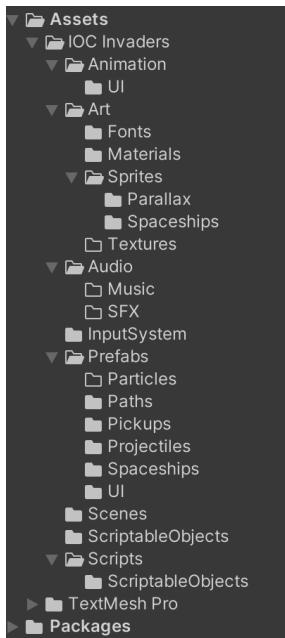
Packages

Aquest projecte usa els següents *packages*:

- **2D Sprite**: per poder treballar amb *sprites*.
- **Input System**: nou sistema de gestió d'entrades de Unity.
- **Post Processing**: permet utilitzar efectes de postprocessament.
- **TextMeshPro**: texts i components d'UI millorats

La resta de paquets són afegits automàticament per Unity i no s'han usat directament.

Estructura del projecte



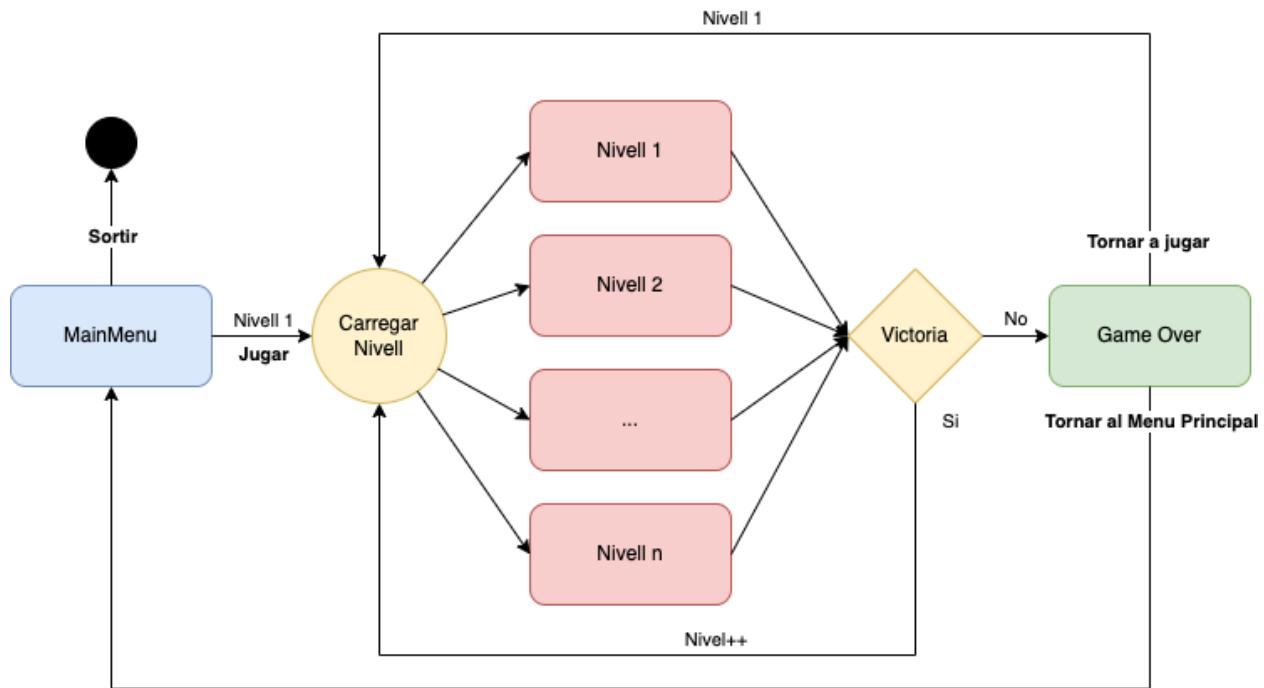
- L'estructura dels fitxers del projecte és la que es veu a la figura de l'esquerra.
- Tots els fitxers del projecte es troben dintre de la carpeta IOC Invaders.
- El codi es troba dintre de la carpeta /Scripts
 - El codi dels scriptable objects es troba dins de la carpeta Scripts/ScriptableObjects.
- Tots els prefabs es troben dintre de la carpeta /Prefabs
- Les escenes es troben dins de /Scenes
- Els scriptable objects instanciats (configuració dels nivells i onades) es troben dins de la carpeta /ScriptableObjects

La carpeta TextMeshPro l'afegeix Unity automàticament quan s'importa el paquet i s'utilitza per primera vegada.

Escenes

L'escena inicial del joc és **MainMenu**, des d'aquesta es pot sortir del joc o iniciar la partida, el que carregarà el primer nivell. En el projecte buit només existeix una escena de joc anomenada

Space. El joc transcorre en aquesta escena i finalitza quan el jugador completa el nivell amb èxit (carrega el següent nivell) o quan és derrotat (carrega l'escena de GameOver) que permet reiniciar el joc o tornar al menú principal.



Totes les escenes compten amb els següents objectes a la jerarquia:

- **MainCamera:** càmera única del joc.
- **Background:** fons de pantalla, múltiples nivells amb efecte *parallax*.
- **AudioManager** (Singleton): proporciona els sons pels trets, les explosions, els efectes de la interfície i permet la reproducció ininterrompuda entre nivells que comparteixen la mateixa pista d'àudio.
- **GameState** (Singleton): informació del joc (puntuacions, nivell, boosts, etc.)
- **GameManager** (Singleton): responsable de carregar les escenes.
- **Fader** (**no inclosa en el projecte base**): aplica una transició Fadeln quan carrega el nivell i FadeOut quan es carrega el següent.
- **EventSystem:** afegit automàticament quan s'afegeix un Canvas.
- **PostprocessingVolume** (**no inclòs en el projecte base**): aplica els efectes de postprocessament de l'escena.

MainMenu

- **MainMenuUI** (**no inclosa en el projecte base**): UI del menú principal .
- **UnmanagedMusic** (**no inclòs en el projecte base**): prefab amb un audio source que reproduceix automàticament la pista d'àudio del menú principal en *loop*.

Space

- **LevelManager**: responsable d'iniciar i finalitzar el nivell.
- **GameUI**: interfície del joc, on es mostra el nom del nivell a l'inici, els punts de vida, puntuació i boosts.

GameOver

- **UnmanagedMusic**(no inclòs en el projecte base): prefab amb un audio source que reproduceix automàticament la pista d'àudio de Game Over en loop.
- **GameOverUI**: interfície del menú de Game Over.

Interfícies

Les interfícies no són incloses en el projecte base.

El joc compta només amb 3 interfícies:

- **UI Menú principal**: mostra el títol del joc, el subtítol i dos botons, un per començar el joc i un altre per sortir del joc.
- **UI Joc**: mostra al principi del nivell el títol, i durant la partida mostra els punts de vida del jugador amb un *slider*, la puntuació, el poder d'atac i el nombre de bales per minut.
- **UI Game Over**: mostra el rètol "Game Over" i un sub-missatge, la puntuació del jugador, i dos botons, un per començar el joc i un altre per tornar al menú principal.



UI Menú principal



UI Joc



UI Game Over

Tant als botons del menú principal com als de fi del joc els botons canvien de color en passar el

cursor per sobre (hover). A més es reproduueix un so quan es detecta el *hover* i un altre quan es prem el botó (configurats al AudioManager).

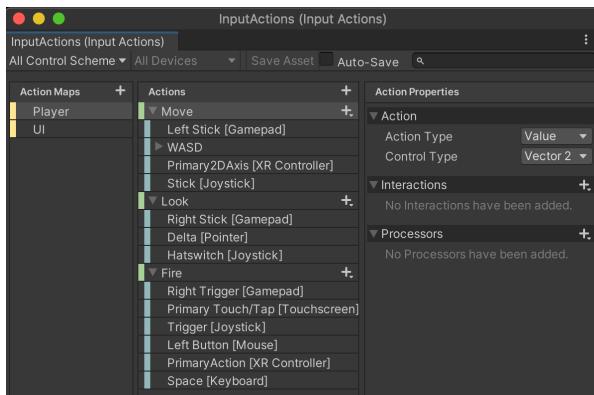
Controls

Per aquest projecte s'ha optat per **utilitzar el nou InputSystem d'Unity**. Cal tenir en compte que aquests dos sistemes poden conviure, però cada vegada que s'obre el projecte demanarà si volem descartar l'anterior.

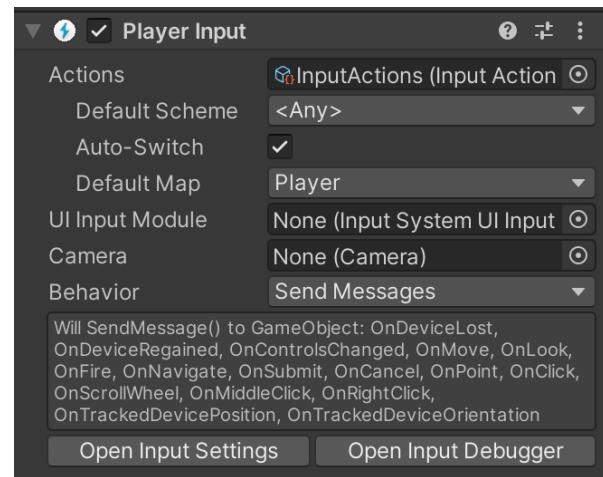
Tingueu en compte que en alguns projectes cal utilitzar tots dos sistemes perquè algunes característiques de l'anterior no s'han implementat en el nou, però això no és necessari en aquest projecte.

Els fitxers de configuració de l'InputSystem es troba a IOC Invaders/InputSystem. Dins d'aquesta carpeta es troba el fitxer InputActions on es configuren les accions. En aquest cas s'ha fet servir la configuració per defecte dels InputActions, no s'ha modificat res.

Al jugador s'ha configurat un component de tipus PlayerInput on s'han assignat aquestes accions. El tipus de comportament és *Send Messages*, per tant, per implementar aquestes accions es faran servir les funcions **OnMove** i **OnFire** (ignorem els missatges de tipus Look, no són aplicables en aquest joc).



InputActions a IOC Invaders/InputActions



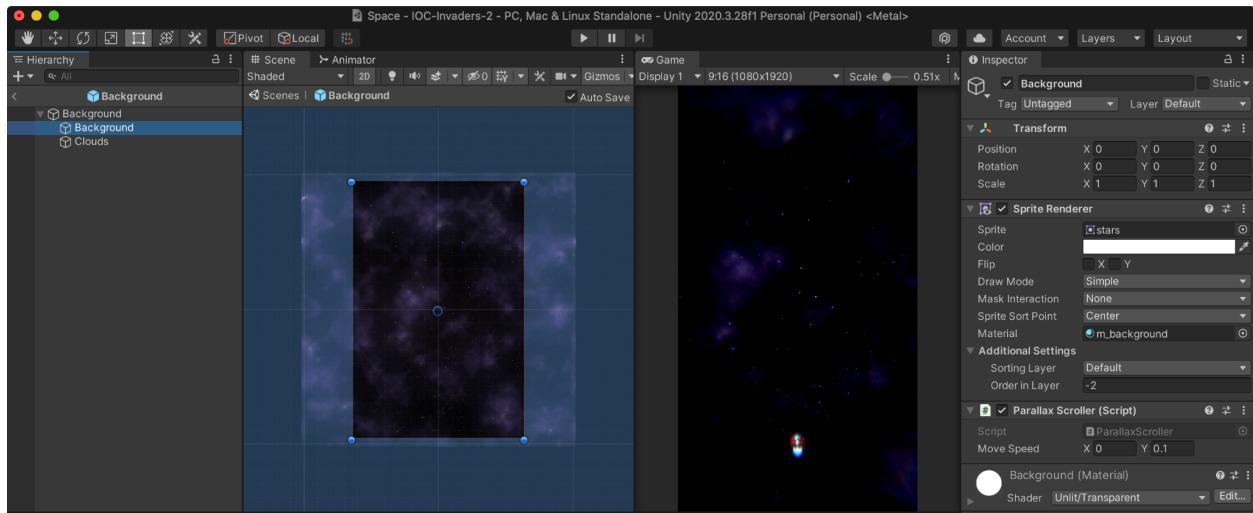
Component PlayerInput al Prefab Player.

Prefabs

A continuació es detallen els prefabs més destacables.

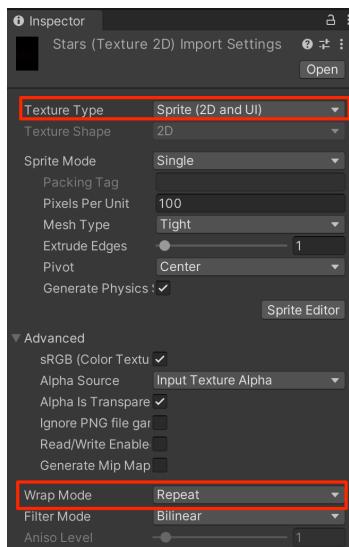
Background

Aquest prefab inclou un o més objectes fills que representen les capes del fons. A cada capa s'ha d'incloure un component de tipus sprite (imatge que es mostrerà), amb un material de tipus **m_background** i un component **ParallaxScroller** amb una velocitat assignada a l'eix Y.



L'ordre de les capes es pot configurar modificant al component Sprite Renderer el valor de **AdditionalSettings->Order in Layer**. Fixeu-vos que s'espera que estiguin per sota de 0, ja que si no es dibuixaran per sobre de les naus i els projectils.

La capa del fons pot ser sòlida, però qualsevol altre capa per sobre haurà de tenir transparències, en cas contrari no es mostraran les capes inferiors.



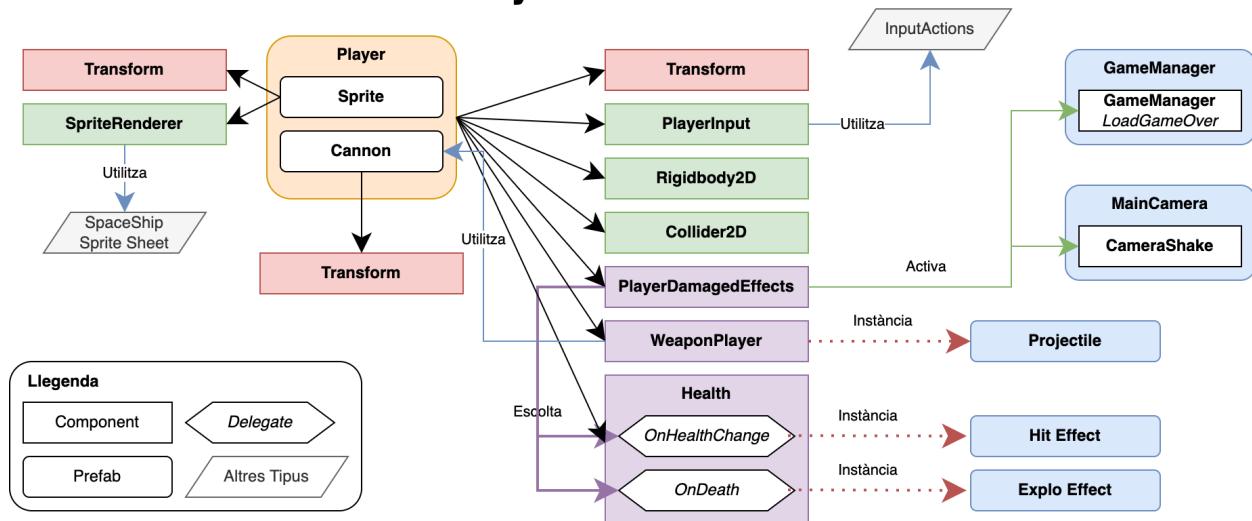
Per importar un nou sprite s'haurà d'escol·lir el **Texture Type: Sprite (2D and UI)**, i el **Wrap Mode: Repeat**, ja que això és necessari per que es vegi correctament quan s'utilitza el script **ParallaxScroller**.

En cas que la imatge contingui més d'un *sprite* haureu d'escol·lir **Sprite Mode: Multiple** i usar el *Sprite Editor* per dividir-la correctament.

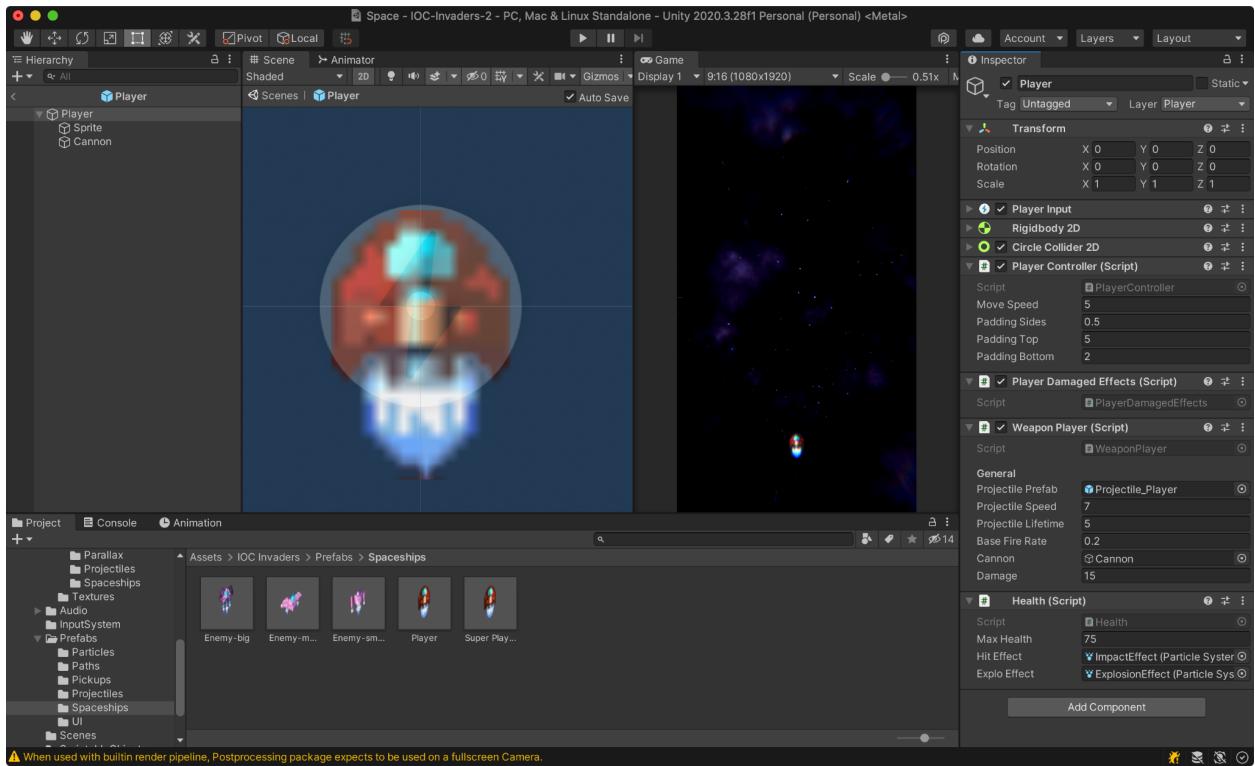
Jugador (Player)

Aquest prefab representa al jugador. Inclou els components d'Unity CircleCollider2D, **RigidBody2D** i **PlayerInput**, per altra banda, també inclou els components propis. L'objecte **Cannon** serveix per configurar la posició i rotació en la que s'ha de disparar i **Sprite** s'usa per configurar el sprite que representarà al jugador.

Relacions del Prefab: Player



- **PlayerDamagedEffects**: component que enllaça amb els esdeveniments de dany sobre el jugador per reproduir la tremolor de la càmera i finalitzar el joc si el jugador és destruït.
- **WeaponPlayer**: component que permet disparar al jugador, permet configurar:
 - Projectile Prefab: ell prefab del projectil utilitzat.
 - **Projectile Speed** i **Projectile Lifetime**: velocitat i temps de vida del projectil un cop disparat.
 - **Base Fire Rate**: temps entre trets.
 - **Cannon**: gameobject des del que es dispararà el projectil, es té en compte la posició i la rotació. En aquesta implementació no s'utilitza, però permetria disparar en diferents direccions (per exemple en diagonal).
- **Health**: aquest component controla els punts de vida que té el jugador, dispara esdeveniments en rebre dany i permet configurar les partícules que s'instancien en rebre mal o ser destruït.

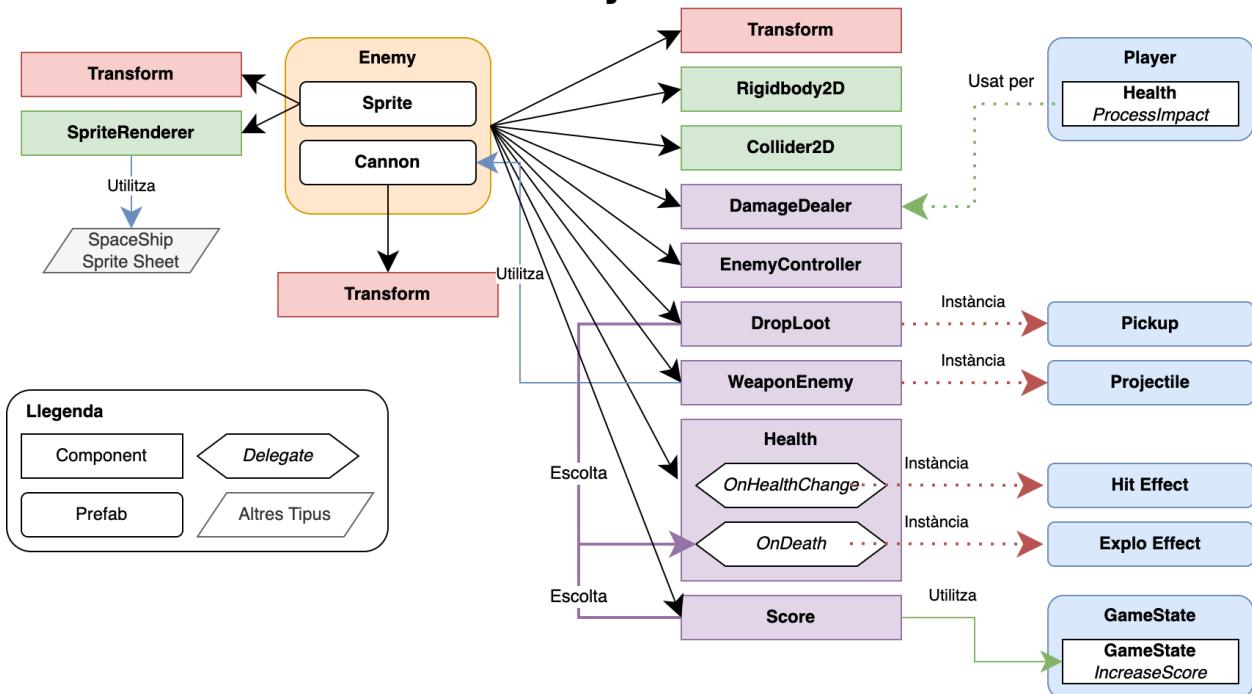


Enemies

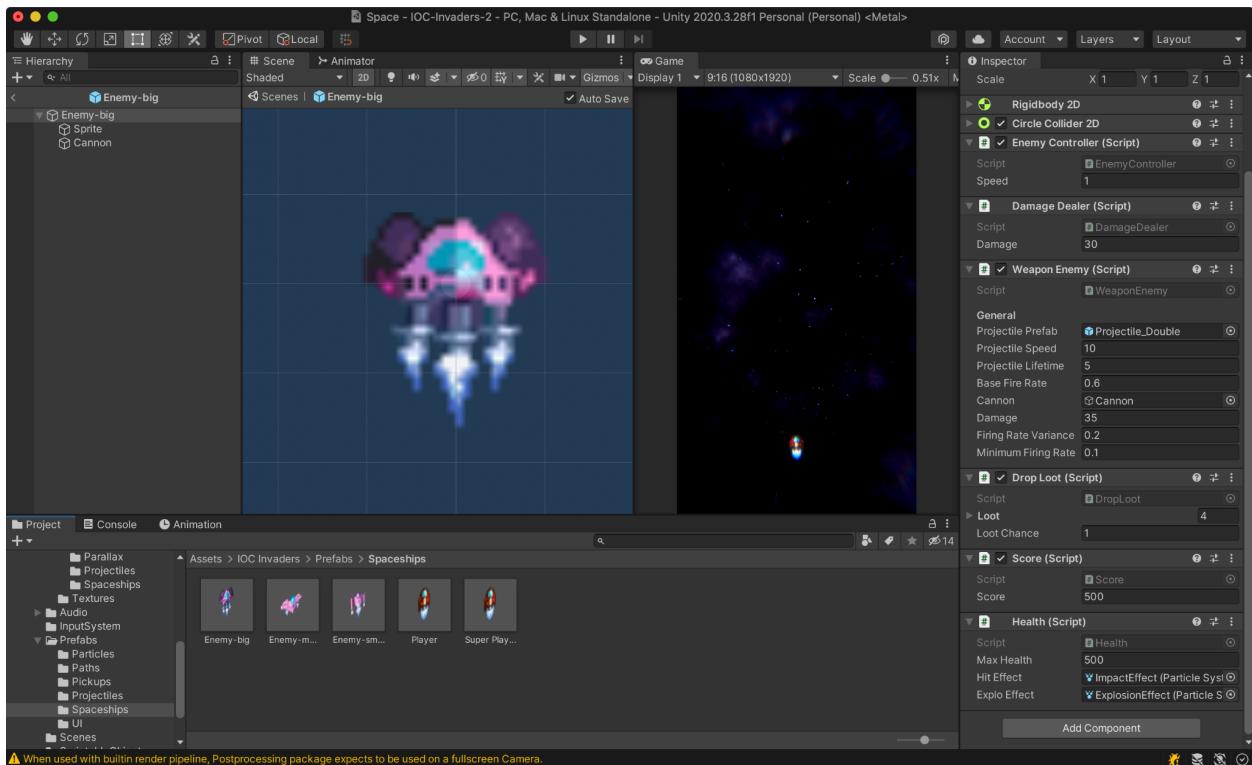
La configuració dels enemics inclou els components Rigidbody2D i CircleCollider2D per detectar els impactes i els components propis DamageDealer, WeaponEnemy, DropLoot, Score i Health.

- **DamageDealer**: script que provoca que la nau causi dany si xoca amb el jugador.
- **WeaponEnemy**: similar a l'script WeaponPlayer, però afegeix:
 - **Firing Rate Variance**: modificador aleatori de temps entre trets.
 - **Minimum Firing Rate**: temps mínim entre trets.
- **Drop Loot (no inclòs en el projecte base)**: permet configurar una llista de *pickups* i la probabilitat que deixi anar un d'ells en ser destruït.
- **Score**: script que augmenta la puntuació del jugador quan la nau és destruïda.
- **Health**: és el mateix component que al jugador, gestiona la salut i els efectes d'impacte i destrucció.

Relacions del Prefab: Enemy



Com en el cas del jugador, Cannon només serveix per indicar la posició i rotació en la qual s'han de disparar els projectils, i Sprite conté el sprite que s'ha de mostrar.



Projectils

Els projectils inclouen un component **Rigidbody2D**, **CapsuleCollider2D** i el component propi **Damage Dealer** que fa que s'apliqui dany quan el projectil col·lisiona amb un enemic o jugador, segons que hagi disparat.

Fixeu-vos que la diferència principal entre un tipus i altre és el layer que tenen assignat, Player o Enemy. En aquesta implementació s'ha optat per crear prefabs diferents per cadascun i assignar el layer adequat, però no costaria gaire canviar el layer quan s'instància el projectil, de manera que tant el jugador com els enemics podrien usar els mateixos projectils.

Collisions del jugador i els enemics

Cal destacar que la nau del jugador i els seus projectils s'han assignat al **layer Player**, mentre que els dels enemics es troben al **layer Enemy** i s'ha ajustat la matriu de col·lisions per fer que només els projectils enemics i els propis enemics impactin contra el jugador i viceversa:

- Els pickups només poden col·lidir amb el jugador.
- El jugador no pot col·lidir amb si mateix ni amb els seus projectils.
- Els enemics no poden col·lidir entre ells ni amb els projectils enemics.

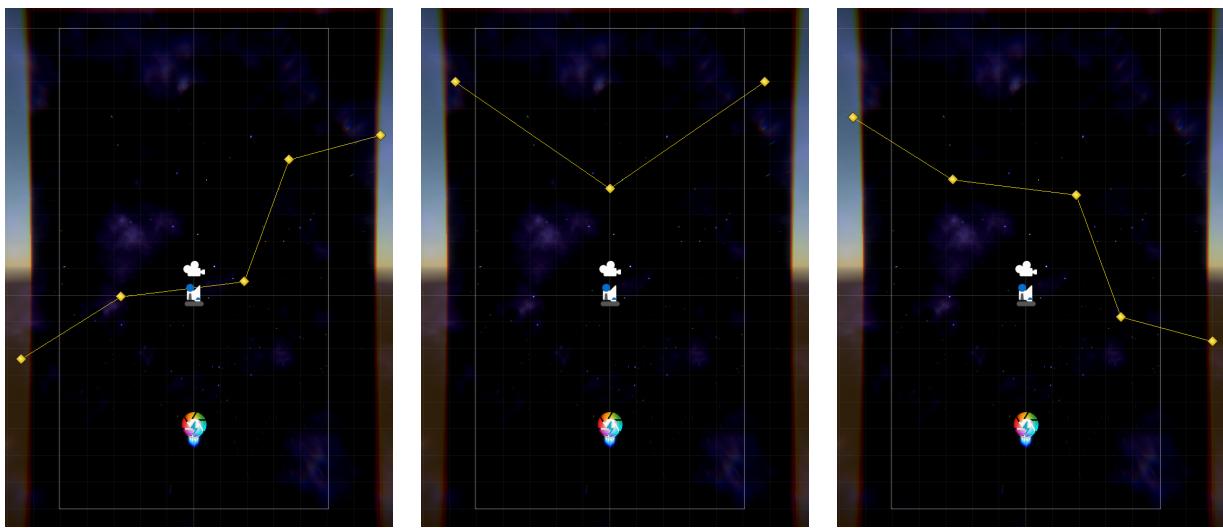
	Default	TransparentFX	Ignore Raycast	Water	UI	Player	Enemy	Postprocessing	Pickup
Default	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
TransparentFX	<input checked="" type="checkbox"/>								
Ignore Raycast	<input checked="" type="checkbox"/>								
Water	<input checked="" type="checkbox"/>								
UI	<input checked="" type="checkbox"/>								
Player	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>						
Enemy		<input checked="" type="checkbox"/>	<input type="checkbox"/>						
Postprocessing		<input checked="" type="checkbox"/>							
Pickup									<input type="checkbox"/>

Important: la matriu que modificada és la corresponent a **Physics 2D** perquè estem utilitzant el **Rigidbody2D** i **Collider2D**, la configuració de *Physics* s'aplica només als components utilitzats a 3D com el **Rigidbody** i els **Colliders** que no són 2D.

Paths

Aquests prefabs són una col·lecció de punts que seran recorreguts pels enemics. Només s'inclou un script que mostra a l'editor una línia entre els punts consecutius, de manera que es

pot visualitzar la ruta correctament.



Path S L Up-Down

Path V Left-Right

Path S Left-Right

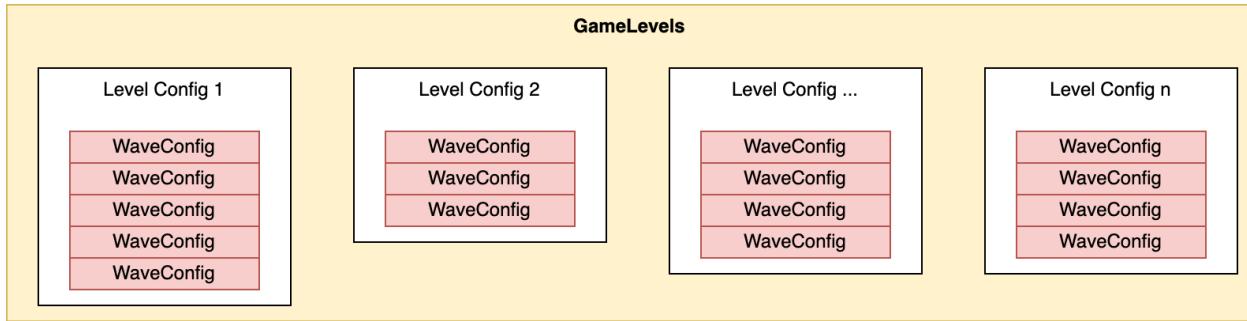
Per modificar una ruta només cal moure els punts, eliminar-los o afegir-ne de nous. Es recomanable duplicar la ruta, arrosseggar-la a la escena, resetear la posició (ha d'estar a 0,0,0) i llavors modificar-la.

Per crear variants ràpidament podeu canviar l'escala dels eixos X (reflex horitzontal) i/o Y (reflex vertical).

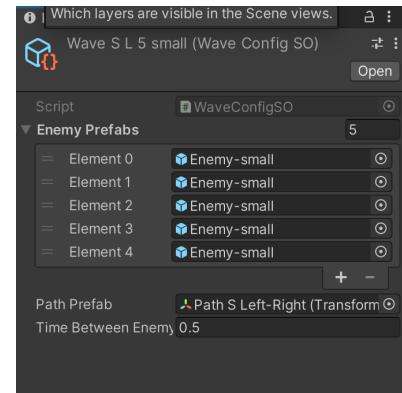
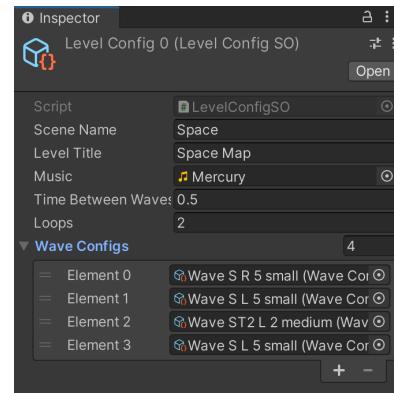
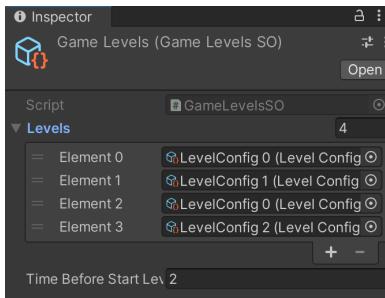
El nom de les rutes és indiferent, però s'ha optat per crear-los a partir de la forma (S, St, V) i la direcció: Up-Down, Down-Up, Left-Right o Right-Left.

Scriptable Objects (Configuració dels nivells)

Per facilitar la creació de nivells s'ha utilitzat un sistema de Scriptable Objects, on el **GameLevel** conté un llistat de **LevelConfigs** (amb la configuració del nivell) i cada **LevelConfig** conté una llista de **WaveConfigs** que contenen la informació sobre la ruta que recorrerà l'onada i les naus que la componen.



Des del punt de vista del dissenyador de nivells, aquesta seria la seva eina per dissenyar els diferents nivells, ja que li permet crear-los (s'assigna l'escena que s'utilitzarà: l'entorn, la música de fons, i la composició dels atacs: les onades).



ScriptableObject: GameLevels

ScriptableObject: LevelConfig

ScriptableObject: WaveConfig

S'ha optat per aquest sistema perquè permet reutilitzar la feina ja feta. Per exemple, un cop configurat un WaveConfig aquest podrà forma part de qualsevol nivell i inclús repetir el mateix tipus d'onada dins del mateix LevelConfig.

De la mateixa manera, es poden reutilitzar els mateixos nivells intercalant-los en la seqüència de GameLevels. Per exemple, al vídeo de demostració s'ha fet servir el nivell Space com a lligam entre els nivells Mars i Venus, de manera que simulem el desplaçament entre els planetes.

Codi C#

Tot el codi del joc es troba dintre de la carpeta IOC Invaders/Scripts. S'ha decidit no separar-los en carpetes, a excepció dels scriptable objects perquè la seva utilització és molt diferent.

Ús de Singletons

Hem optat per utilitzar una implementació simple dels Singletons, sense crear una jerarquia, evitant la creació automàtica d'objectes i deixant la responsabilitat d'afegir-los a les escenes al desenvolupador.

La base de tots els singletons és la següent::

```
private static Singleton _instance;

public static Singleton Instance
{
    get { return _instance; }
}

private void Awake()
{
    if (_instance != null)
    {
        gameObject.SetActive(false);
        Destroy(gameObject);
    }
    else
    {
        _instance = this;
        DontDestroyOnLoad(gameObject);
    }
}
```

Com es pot apreciar, s'accedeix a la instància del singleton mitjançant la propietat **Instance** que només compta amb un getter. El mètode **Awake** es dispara quan arranca l'escena, abans de cridar al mètode Start), aquí comprovem si ja existeix una instància i si és així es descarta aquesta. En cas contrari s'assigna aquesta com a instància i s'indica al sistema que no s'ha de destruir quan canvia l'escena (DontDestroyOnLoad).

Així ens assegurem que només existeix una única instància de la classe, que aquesta és accessible mitjançant la propietat **Instance**, i que no es destrueix en carregar altres escenes.

S'ha considerat adient implementar aquest patró per les següents classes:

- **AudioManager**: aquesta classe gestiona el so del joc i indica quins sons han de ser reproduïts pels trets, les explosions i la interfície, a més és la responsable de reproduir

la música del nivell (que s'obté del LevelConfig) evitant tallar-la si és igual que la música que s'està reproduint.

Com es pot apreciar, és necessària en tots els nivells i és indispensable que no es destrueixi entre nivells, perquè en cas contrari la música de fons s'interrompria. A més, facilita accedir als sons utilitzant la forma **AudioManager.Instance.PlayClip(sound)**.

- **GameState**: en aquesta classe es controla l'estat actual del joc, desant la puntuació, els boost acumulats (poder d'atac i rati de tir) i el nivell actual. Aquesta informació ha de persistir entre nivells, s'ha de modificar i consultar només en un únic punt i és necessari utilitzar-la en totes les escenes. En tractar-se d'un singleton es pot utilitzar la forma **GameState.Instance.IncreaseScore()** o subscriure's a events rellevants mitjançant amb **GameState.Instance.OnScoreChanged**.
- **GameManager**: En aquest cas no seria estrictament necessari implementar-ho com a Singleton, però s'ha fet així perquè no és necessari tenir més d'un component d'aquest tipus en cap moment i cal utilitzar-lo a cada escena.

Una classe candidata a tractar-la com a singleton seria LevelManager, però com que aquesta classe ha de executar tasques al Start (iniciar el nivell) no es pot utilitzar d'aquesta manera, ja que només es dispararia un cop perquè no es destruiria entre nivells.

Com podeu veure, de vegades usar o no un patró és una decisió de disseny. Sempre hi ha pros i contres a l'hora d'implementar un patró o utilitzar una arquitectura. En el nostre cas ens obliga a afegir a totes les escenes els Singleton, perquè si no no funcionaria. Per altra banda, si un mecanisme que instanciï automàticament no funciona en tots els casos, perquè AudioManager i GameManager tenen informació assignada (clips a reproduir i informació dels nivells respectivament). En canvi, es podria implementar al GameState, ja que totes les dades s'estableixen mitjançant codi.

Ús de Coroutines

Podeu veure la utilització

- Crear un retard abans d'executar un bloc de codi, per exemple la càrrega d'un nivell quan ja ha acabat o es prem un botó.
- Limitar la ràtio de tir, evitant que el codi continuï executant-se fins que passa el temps estipulat entre trets.
- Repetir una acció cada *frame* durant un temps determinat.

Com es pot apreciar, sempre es tracta d'aplicar un retard, ja sigui per executar un bloc de codi un sol cop o de forma repetitiva en el cas dels trets i el *spawn* d'enemics i d'onades d'enemics.

- **CameraShake**: en aquest cas la coroutina el que fa és repetir un bloc de codi limitant-lo

a una iteració cada bloc mitjançant `yield return new WaitForSeconds();`.

```
public void Play()
{
    StartCoroutine(Shake());
}

IEnumerator Shake()
{
    float elapsedTime = 0;
    while (elapsedTime < shakeDuration)
    {
        transform.position = _initialPosition +
(Vector3)Random.insideUnitCircle * shakeMagnitude;
        elapsedTime += Time.deltaTime;
        yield return new WaitForSeconds();
    }

    transform.position = _initialPosition;
}
```

- **GameManager:** en aquest cas s'utilitza una única vegada per afegir un delay abans de realitzar la càrrega. Fixeu-vos que s'ha fet servir `yield return new WaitForSeconds(delay);` on el delay és el temps que s'ha d'esperar abans de carregar l'escena.

```
public void DelayedLoadLevel(string sceneName)
{
    if (OnLevelChange != null) OnLevelChange();
    StartCoroutine(WaitAndLoad(sceneName, sceneLoadDelay));
}

IEnumerator WaitAndLoad(string sceneName, float delay)
{
    yield return new WaitForSeconds(delay);
    SceneManager.LoadScene(sceneName);
}
```

- **WeaponAbstract:** quan es dispara uan arma el que fem és limitar el temps entre tirs. En lloc de controlar manualment el temps que ha passat entre trets (desant el moment en què es va fer el darrer el temps i contrastar-lo amb el temps actual) s'han utilitzat coroutines. Com que el comportament del jugador i els enemics és diferents s'ha implementat la coroutina a les subclasses:

```
void Fire()
{
    if (isFiring && firingCoroutine == null)
```

```

    {
        firingCoroutine = StartCoroutine(FireContinuously());
    }
    else if (!isFiring && firingCoroutine != null)
    {
        StopCoroutine(firingCoroutine);
        firingCoroutine = null;
    }
}

protected abstract IEnumerator FireContinuously();

```

A més, quan es detecta que no s'està disparant satura la corutina amb `StopCoroutine(firingCoroutine)`.

Les implementacions de les dues subclasses és molt similar, es tracta d'un bucle infinit que només serà interromput quan finalitzi la corutina amb `StopCoroutine`. Després de cada iteració s'utilitza `yield return new WaitForSeconds(baseFireRate);` per afegir un temps d'espera entre trets.

- **EnemySpawner:** aquesta classe té una peculiaritat en el que respecta a les coroutines, ja que és l'únic dels que trobem al projecte utilitza dos `yields`, un per retardar les iteracions entre enemics a cada onada i un altre per afegir un retard entre el final de l'anterior onada i l'inici de la següent:

```

public void StartSpawn()
{
    StartCoroutine(SpawnEnemyWaves());
}

IEnumerator SpawnEnemyWaves()
{
    _spawning = true;
    for (int i = 0; i < _loops; i++)
    {
        foreach (WaveConfigSO wave in _waveConfigs)
        {
            _currentWave = wave;

            for (int j = 0; j < _currentWave.GetEnemyCount(); j++)
            {
                SpawnEnemy(_currentWave.GetEnemyPrefab(j));
                yield return new WaitForSeconds(_currentWave.GetSpawnTime());
            }

            yield return new WaitForSeconds(_timeBetweenWaves);
        }
    }
}

```

```
_spawning = false;  
TryToEndLevel();  
}
```

Fixeu-vos que la crida a aquesta corutina instància tots els enemics del nivell un per un, no cal fer res més, i finalment comprova si s'ha de finalitzar el nivell (tots els enemics han sigut destruïts i s'ha finalitzat la instació d'onades).

Ús de Delegates

En aquest projecte s'han utilitzat delegates com *events* per comunicar diferents components entre ells. Els emissors d'*events* són GameState, EnemySpawner, GameManager, Health i LevelManager.

Cal tenir en compte que **si no hi ha cap subscriptor a un delegat el valor d'aquest és null**, per tant, si s'intenta invocar **es produirà un error**. Per aquesta raó sempre cal comprovar primer si és null.

- **GameState:** el GameState defineix dos *events*:
 - *OnScoreChanged*: delegat sense arguments que es dispara quan es modifica la puntuació:

```
public void IncreaseScore(int value)  
{  
    _score += value;  
    Mathf.Clamp(_score, 0, int.MaxValue);  
  
    if (OnScoreChanged != null) OnScoreChanged();  
}
```

La UI de l'escena de joc és subscriu a aquest mètode per detectar quan s'ha modificat la puntuació i actualitzar el text a la pantalla.

- *OnStatsChanged*: delegat sense arguments que es dispara quan es modifica el poder d'atac o la ràtio de tir.

```
public void IncreaseDamage(int power)  
{  
    this._power += power;  
    if (OnStatsChanged != null) OnStatsChanged();  
}
```

Com en el cas anterior, l'UI de la escena de joc es subscriu per detectar quan hi

ha canvis per actualitzar-se.

Fixeu-vos que l'alternativa a utilitzar delegates hauria estat comprovar dins del mètode **Update()** si la puntuació o els *stats* han canviat, això implica fer la comprovació 60 vegades per segon. En canvi, amb el sistema de delegats només es comprova quan realment s'ha produït un canvi.

- **EnemySpawner:** en aquesta classe es declara l'*event* **OnAllEnemiesDestroyed**, que es dispara quan es comprova si s'ha de finalitzar el nivell amb èxit:

```
void TryToEndLevel()
{
    if (_spawnedEnemies == 0 && !_spawning && OnAllEnemiesDestroyed != null)
    {
        OnAllEnemiesDestroyed();
    }
}
```

El **LevelManager** se subscriu a aquest *event* per determinar quan s'han de finalitzar el nivell i carregar el següent.

- **GameManager:** exposa el delegat **OnSceneChange** per comunicar quan comença la transició de nivells. Això permet al prefab **Fader** assabentar-se quan s'ha d'iniciar el FadeOut per fer la transició entre nivells més suau.

```
public void DelayedLoadLevel(string sceneName)
{
    if (OnSceneChange != null) OnSceneChange();
    StartCoroutine(WaitAndLoad(sceneName, sceneLoadDelay));
}
```

Fixeu-vos que es dispara l'*event* **OnSceneChange** i seguidament s'inicia una corutina, és a dir, la transició de nivells no s'efectuarà immediatament, sinó que es produirà una mica després (el temps indicat per *sceneLoadDelay*)

- **Health:** aquest component exposa tres delegats diferents, dos sense arguments i un amb un argument de tipus int:
 - *OnDeath:* es dispara quan el propietari del component es destruït. En el cas del jugador és escoltat pel component PlayerDamageEffects i provoca el fi de la partida. Per altra banda, en el cas dels enemies, provoca l'increment de puntuació i la possible instanciació de drop.

```

protected virtual void Die()
{
    if (OnDeath != null) OnDeath();
}

```

- *OnDestroyEvent*: aquest *event* es dispara quan el component es destrueix. És escoltat pel **EnemySpawner** per detectar quan un enemic ha sigut destruït. Cal diferenciar aquest d'*OnDeath*, ja que un enemic que abandona l'àrea de joc també és eliminat, però no ha sigut destruït pel jugador (cosa que provocaria un increment de puntuació i la possible generació de *loot*).

```

private void OnDestroy()
{
    if (OnDestroyEvent != null) OnDestroyEvent();
}

```

- *OnHealthChange(int amount)*: com es pot apreciar, aquest delegat passa un enter amb el nombre de punts de vida (positius o negatius) en què s'ha modificat la salut. Aquest esdeveniment és escoltat per **PlayDamageEffects** i **UIDisplay**, en el primer cas si la quantitat és negativa s'activa el tremolor de la càmera i en el segon si la quantitat és positiva s'activa l'animació de la barra de salut.

```

private void TakeDamage(int damage)
{
    _currentHealth = Mathf.Clamp(_currentHealth - damage, 0,
int.MaxValue);

    if (OnHealthChanged != null) OnHealthChanged(-damage);

    if (_currentHealth <= 0)
    {
        PlayEffect(exploEffect);
        Die();
        Destroy(gameObject);
    }
    else
    {
        PlayEffect(hitEffect);
    }
}

```

Com podeu veure, els punts de vida es passen amb el signe canviat, perquè la funció el que pren són valors positius que es restan dels punts de vida, per tant un valor positiu implica prendre mal i al contrari.

Scriptable Objects

El codi dels scriptable objects es força simple, es limita a una llista de propietats que són les que poden editar-se en crear un recurs d'aquest tipus.

Tots els scritpable objects icnlouen una capçalera que indica el nom de l'opció de menú que permet crear-los i el nom del fitxer per defecte que s'assigna al recurs creat.

GameLevelsSO

Fixeu-vos que s'han declarat les propietats **levels** i **timeBeforeStartLevel** com a privades, però configurables des de l'editor i s'han proporcionat els *getters* per evitar que es puguin modificar els valors des del codi.

```
[CreateAssetMenu(menuName = "Game Levels Config", fileName = "New GameLevels")]
public class GameLevelSSO : ScriptableObject
{
    [SerializeField] private List<LevelConfigSO> levels;

    public List<LevelConfigSO> Levels
    {
        get { return levels; }
    }

    [SerializeField] private float timeBeforeStartLevel = 3f;

    public float TimeBeforeStartLevel
    {
        get { return timeBeforeStartLevel; }
    }
}
```

Aquest scriptable object conté una llista de configuracions de nivell (LevelConfigSO) i un temporitzador amb el temps d'espera un cop es carrega un nivell abans de començar a instanciar enemics.

LevelConfigSO

En aquest cas hem simplificat i hem declarat totes les propietats com a públiques. En un projecte petit com aquest, en el que només treballa un desenvolupador, aquesta solució és acceptable perquè se simplifica el codi (compareu-lo amb l'anterior):

```
[CreateAssetMenu(menuName = "Level Config", fileName = "New LevelConfig")]
public class LevelConfigSO : ScriptableObject
{
    [SerializeField] public string sceneName = "Game";
```

```

[SerializeField] public string levelTitle = "Level";
[SerializeField] public AudioClip music = null;
[SerializeField] public float timeBetweenWaves = 1f;
[SerializeField] public int loops = 3;
[SerializeField] public List<WaveConfigSO> waveConfigs;
}

```

WaveConfigSO

Per acabar, en aquest tercer scriptable objects que és utilitzat per configurar les onades d'enemic, malgrat que les propietats són privades i configurables des de l'editor, no hi ha *getters* ni *setters*. S'usen els mètodes **GetStartingWaypoints**, **GetWayPoint**, **GetEnemyCount** i **GetEnemyPrefab** per accedir la informació necessària que es deriva de les propietats.

```

using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "Wave Config", fileName = "New WaveConfig")]
public class WaveConfigSO : ScriptableObject
{
    [SerializeField] private List<GameObject> enemyPrefabs;
    [SerializeField] private Transform pathPrefab;
    [SerializeField] float timeBetweenEnemySpawns = 1f;

    public Transform GetStartingWaypointGetWayPointsGetEnemyCount()
    {
        return enemyPrefabs.Count;
    }

    public GameObject GetEnemyPrefab(int index)
    {
        return enemyPrefabs[index];
    }
}

```

```

public float GetSpawnTime()
{
    return timeBetweenEnemySpawns;
}
}

```

Resum de classes

A continuació es mostra un llistat de les classes que formen part del projecte i quina és la seva funció:

- **AudioManager**: facilita l'accés a efectes de so comuns i permet la reproducció de pistes d'àudio sense interrupció entre nivells.
- **CameraShake**: aplica un efecte de tremolor a la càmera.
- **DamageDealer**: component utilitzat pel component *Health* que indica que la col·lisió amb aquest provoca danys.
- **DropLoot**: component que permet configurar una llista de *drops* i quan l'enemic és destruit hi ha la possibilitat que deixi caure un d'aquests *drops*.
- **EnemyController**: component responsable de fer que l'enemic recorri la ruta corresponent.
- **EnemySpawner**: instància els enemics, onada darrera onada.
- **Fader**: encarregat d'aplicar l'efecte *FadeIn* en carregar el nivell i *FadeOut* abans de carregar el següent.
- **GameManager**: singleton que inclou els mètodes per carregar el menú principal, la pantalla de game over i la càrrega de nivells.
- **GameState**: guarda la informació de la partida actual.
- **GizmoLineHelper**: classe auxiliar que mostra una línia entre els punts d'una ruta.
- **Health**:
- **LevelManager**: component que inicialitza l'EnemySpawner i quan detecta que el nivell ha finalitzat, incrementa el nivell actual i carrega el següent.
- **ParallaxScroller**: component que desplaça vertical i/o horitzontalment un *sprite*.
- **PickupAbstract**: objectes que s'instancien en el nivell i en col·lidir amb el jugador aporten diferents bonus.
 - *PickupFireRateBoost*: incrementa la velocitat d'atac.
 - *PickupHealthBoost*: incrementa la salut.
 - *PickupDamageBoost* (no inclòs en el projecte base): incrementa el poder d'atac.
 - *PickupScoreBoost* (no inclòs en el projecte base): incrementa la puntuació del jugador.
- **PlayerController**: recull les entrades del jugador (Inputs) i és responsable del moviment.

- **PlayerDamagedEffects**:
- **Score**: component que increments la puntuació del jugador quan el propietari del component és destruït.
- **UIDisplay**: component que permet lligar les puntuacions desades al **GameState** amb la interfície.
- **UIGameOver** (no inclòs en el projecte base): component que actualitza la puntuació del jugador.
- **WeaponAbstract**: responsable d'iniciar i aturar l'acció de disparar.
 - *WeaponEnemy*: la forma de disparar és aleatòria.
 - *WeaponPlayer*: la forma de disparar està limitada pels *stats*.

Enllaç a Recursos

- Kenney. <https://kenney.nl/>
- Dafont. <https://www.dafont.com/es/>
- Itch.io. <https://itch.io/>
- Repositori IOC Invaders 2: <https://github.com/XavierGar0/IOC-Invaders2-Base>
- Vídeo gameplay base: <https://www.youtube.com/watch?v=UZRQ5zMIZXY>
- Vídeo gameplay final: <https://www.youtube.com/watch?v=xPCdTvmvyyl>