SESSIÓ 2. (2)

Ampliar les capacitats del Player.

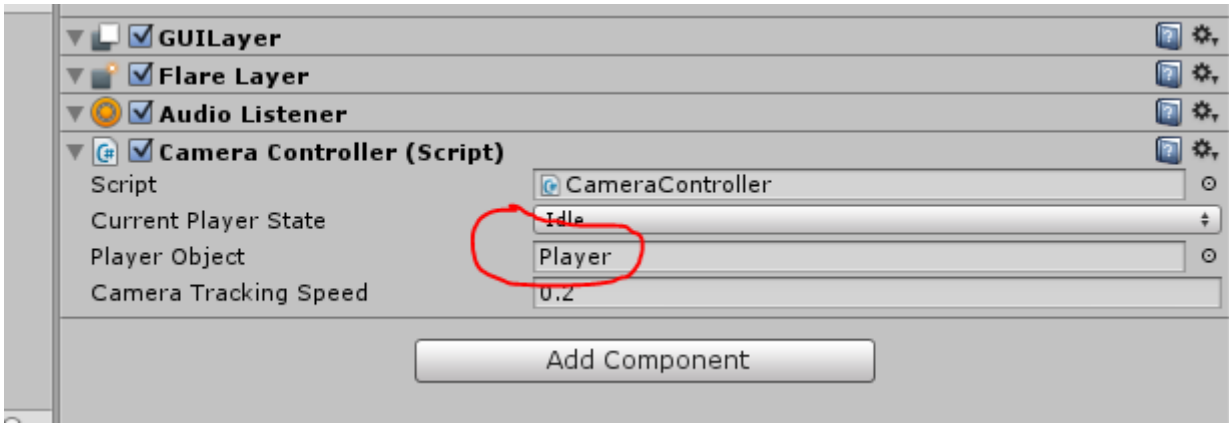
2. Centrar l'acció en el protagonista: Control de la càmara.

Des del primer moment en què creem una escena hi ha present un objecte: la càmara (Main Camera en la pestanya Hierarchy). Fins ara, amb la configuració per defecte, la càmara ens ha mostrat l'escena fixa i el protagonista movent-s'hi dins.

Una altra manera molt habitual d'utilitzar la càmara és fer que segueixi en tot moment a un objecte, normalment el protagonista. Els moviments de càmara es poden gestionar de diferents maneres. Per tal de ser coherents en l'estil, en aquesta introducció a Unity 2D gestionarem el moviment de càmera utilitzant també els events i delegates de C#.

Crearem un nou script al qual anomenarem **CameraController**, que tindrà una estructura semblant a l'script PlayerStateListener, que hem creat abans.

Una vegada creat, **l'script s'ha d'associar a la Main Camera**. Quan l'escript CameraController aparegui a l'Inspector com un component més de Main Camera, ens hem d'assegurar de **posar l'objecte Player al camp Player Object**:



El codi de l'script serà el següent:

```
using UnityEngine;
using System.Collections;
public class CameraController : MonoBehaviour
{
	public	PlayerStateController.playerStates	currentPlayerState	=
PlayerStateController.playerStates.idle;
	public GameObject playerObject = null;
	public float cameraTrackingSpeed = 0.2f;
	private Vector3 lastTargetPosition = Vector3.zero;
	private Vector3 currTargetPosition = Vector3.zero;
	private float currLerpDistance = 0.0f;
```

```
	void Start()
	{
		//Establir la posicio inicial de la camara com la del Player, per evitar un
salt inicial.
		Vector3 playerPos = playerObject.transform.position;
		Vector3 cameraPos = transform.position;
		Vector3 startTargPos = playerPos;
		//Igualar coordenada z del Player  a la de la camara, i evitar moviments
fora del pla.
		startTargPos.z = cameraPos.z;
		lastTargetPosition = startTargPos;
		currTargetPosition = startTargPos;
		currLerpDistance = 1.0f;
	}

	//afegir listener de l'event canvi d'estat del Player
	void OnEnable()
	{
		PlayerStateController.onStateChange +=
			onPlayerStateChange;
	}

	//eliminar listener de l'event canvi d'estat del Player
	void OnDisable()
	{
		PlayerStateController.onStateChange -=
			onPlayerStateChange;
	}

	//Tractament de l'event canvi d'estat del Player: l'estat actual passa a ser el
nou estat que indica l'event
	//El tractament concret es fara a onStateCycle()
	void onPlayerStateChange(PlayerStateController.playerStates newState)
	{
		currentPlayerState = newState;
	}

	//Tractament realitzat en cada frame
	void LateUpdate()
	{
		// Tractaments segons el nou estat del Player
		onStateCycle();
		// Moure camera a la nova posicio del Player
		currLerpDistance += cameraTrackingSpeed;
		transform.position = Vector3.Lerp(lastTargetPosition,
					currTargetPosition, currLerpDistance);
	}


	// Tractament PEL QUE FA A LA CAMERA! que correspon a cada estat del Player
	void onStateCycle()
	{
		switch(currentPlayerState)
		{
		case PlayerStateController.playerStates.idle:
			trackPlayer();
			break;
		case PlayerStateController.playerStates.left:
			trackPlayer();
			break;
		case PlayerStateController.playerStates.right:
			trackPlayer();
			break;
		}
	}
```

```
	//Fer que la camera segueixi al Player
	void trackPlayer()
	{
		// Obtenir i guardar posicio actual de la camera i del Player
		Vector3 currCamPos = transform.position;
		Vector3 currPlayerPos = playerObject.transform.position;

		//Si les posicions son iguals, no cal fer res
		if(currCamPos.x == currPlayerPos.x && currCamPos.y == currPlayerPos.y)
		{
			// Positions are the same - tell the camera not to move, then abort.
			currLerpDistance = 1f;
			lastTargetPosition = currCamPos;
			currTargetPosition = currCamPos;
			return;
		}
		// Actualitzar variables (posicio anterior i posicio actual ...)  per fer
que
		// la camara vagi a la posicio del Player
		currLerpDistance = 0f;
		lastTargetPosition = currCamPos;
		currTargetPosition = currPlayerPos;
		//No volem que canvii la component z, estem en 2D!
		currTargetPosition.z = currCamPos.z;
	}

	//Fer que la camera NO segueixi al Player
	void stopTrackingPlayer()
	{
		// Fer que la posicio desti de la camara sigui la posicio actual.
		Vector3 currCamPos = transform.position;
		currTargetPosition = currCamPos;
		lastTargetPosition = currCamPos;
		currLerpDistance = 1.0f;
	}
}
```

Comentaris al codi:

L'estructura de l'script és semblant a la que s'ha utilitzat ja a PlayerStateListener.

En el mètode **onEnable()** (mètode estàndar, de MonoBehaviour) afegim la funció onPlayerStateChange() a la llista de listeners de l'event PlayerStateController.onStateChange. Gràcies a això, quan el Player canviï d'estat, el sistema passarà el control a onPlayerStateChange.

En el mètode **onDisable()** (mètode estàndar, de MonoBehaviour) deixem d'escoltar l'event PlayerStateController.onStateChange.

A la funció **onPlayerStateChange()** (funció programada per nosaltres, que hem establert com a "escoltador" de l'event "canvi d'estat del Player") simplement guardem a la variable currentPlayerState el nou estat del player (que ens ve indicat a l'event). Aquesta variable s'utilitzarà per decidir el tractament de la càmara.

El mètode **LateUpdate()** (mètode estàndar, de MonoBehaviour) s'executa a cada canvi de frame. El que es fa primer de tot és tractar l'estat actual cridant a la funció que anomenem onStateCycle(), seguint l'estil de l'script PlayerStateListener.

A **onStateCycle()** (funció programada per nosaltres), realitzem el tractament que correspon a l'estat del player, que hem guardat a currentPlayerState. De moment, els tres estats que considerem (idle, left i right) tenen el mateix tractament: seguir el Player amb la càmara. Això ho fem a trackPlayer().

A la funció **trackPlayer()** programem el moviment de la càmara de manera que segueixi al player. De fet, en aquesta funció simplement s'actualitzen les variables lastTargetPosition i currTargetPosition. Si es veu que El moviment de la càmara es realitza al final del mètode **LateUpdate()** utilitzant el valor acabat de calcular d'aquestes variables (es passa la càmara des de la posició anterior del player fins la posició actual del player):

```
. . .

currLerpDistance += cameraTrackingSpeed;
transform.position = Vector3.Lerp(lastTargetPosition,
                                  currTargetPosition, currLerpDistance);
```

En tot el codi, Target és l'objectiu que ha de seguir la càmara, en aquest cas el Player.
Les posicions es guarden en objectes Vector3, de tres components, per això de vegades cal gestionar la component z. Recordem que Unity és un motor 3D...

El tercer paràmetre de Vector3.Lerp serveix per fer interpol·lacions entre la posició inicial i la final. Té valors entre 0 i 1. El valor 0 referencia la posició inicial i el valor 1, la final. Els valors entre 0 i 1 corresponen a posicions intermitges. En aquest programa, els valors entre 0 i 1 s'obtenen incrementant amb el valor de cameraTrackingSpeed.

El nom del paràmetre Lerp ve de Linear intERPolation (interpol·lació lineal). La interpol·lació és un mètode matemàtic per obtenir aproximacions dels punts (desconeguts) que hi podria haver entre dos punts coneguts. Hi ha diferents variants (lineal, polinòmica ...), segons com es realitzen els càlculs. Amb els càlculs "Lerp" Unity pot fer que la transició entre dues imatges sigui més suau.
Veure **http://docs.unity3d.com/ScriptReference/Vector3.Lerp.html**

S'inclou la funció **stopTrackingPlayer()**, encara que de moment no s'utilitzi.

Si ara polsem el botó Play, veurem que tot continua igual però el Player sempre és al centre de l'escena.

3. Permetre que el Player caigui: gravetat + col·lisions.

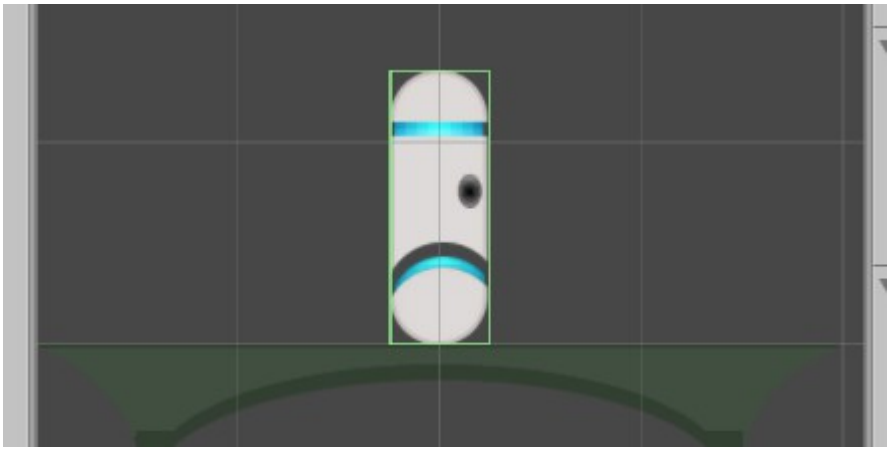
Fins ara, quan movem el Player fora de la plataforma continua el seu camí com si no existís la gravetat. Anem a millorar això.

Els motors de jocs tenen normalment el que s'anomena "motor de físiques" que s'encarrega de gestionar els fenòmens físics (típicament col·lisions (contacte entre dos objectes), inèrcia, simulació de la gravetat ...) que afecten als objectes del joc. En Unity hi ha dos conceptes principals relacionats amb el motor de físiques: el "collider" i el "rigid body".

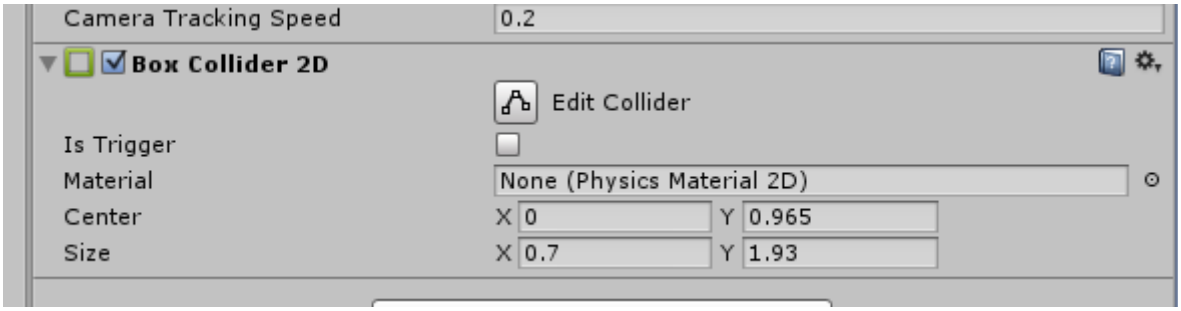
Un **collider** actua com un envolcall que permet a un objecte detectar que ha col·lisionat amb un altre objecte, sempre i quan també estigui "dins" d'un altre collider.

El primer pas per millorar el nostre joc d'exemple és **assignar al Player un collider**, tal com vem fer amb la plataforma. En aquest cas, **en tindrem prou amb un box collider 2D**. Com que té forma rectangular, ens proporciona una base plana que anirà molt bé per detectar col·lisions amb les superfícies.

En l'escena apareixerà el Player envoltat pel collider rectangular:



En l'Inspector veurem que el Player té un nou component, el collider:



Ara anem a assignar característiques físiques al Player. Per fer això, li hem d'assignar un component **Rigid Body 2D**.

En el Rigid Body 2D canviem algunes propietats, si cal:

- **Gravity Scale** a 2, per fer-lo caure d'una manera més realista. (Experimentar l'efecte de diferents valors)
- **Sleeping Mode** ha de ser **Start Awake** perquè comprovi des del principi.
- **Collision Detection** ha de ser **Continuous**. Com que el Player està sempre en moviment, la detecció de col·lisions ha de fer-se contínuament (amb altre valor podria ser que no es detectés alguna col·lisió i el Player passés a través d'altres objectes).

Finalment, anem al component Animator del Player. Hi desmarquem la check box Apply Root Motion i com a Update Mode seleccionem Animate Physics. Això dona al motor de físiques una mica més de control sobre les animacions i evita que Unity faci ajustos inesperats.

Si provem ara el joc, veurem que el Player ja es veu afectat per la gravetat i, si surt de la plataforma, cau.

EXPERIMENT

Elimina el component collider de la plataforma i executa el joc. Explica el que passa.

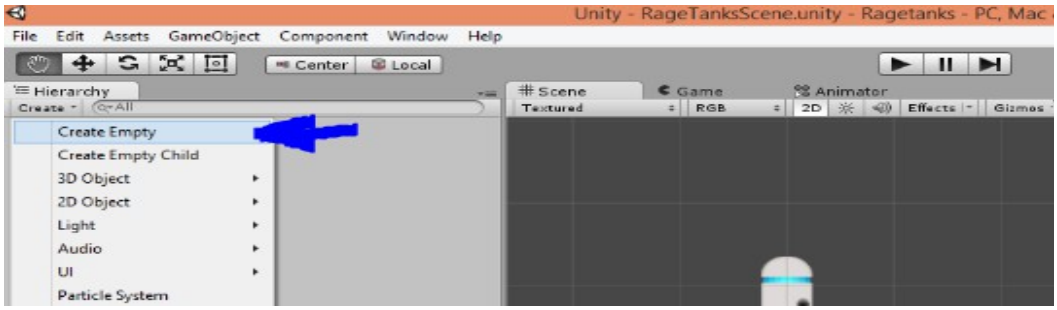
4. Gestió de les vides del protagonista

Habitualment, el protagonista d'un joc té un cert nombre de vides per consumir. La mort definitiva només arriba després d'haber passat per un nombre determinat de morts i renaixements. Anem ara a gestionar la mort i renaixement del Player.

Quan caigui de la plataforma, en comptes de deixar-lo caure indefinidament com ara, implementarem la seva mort i reaparició en un lloc fixat.

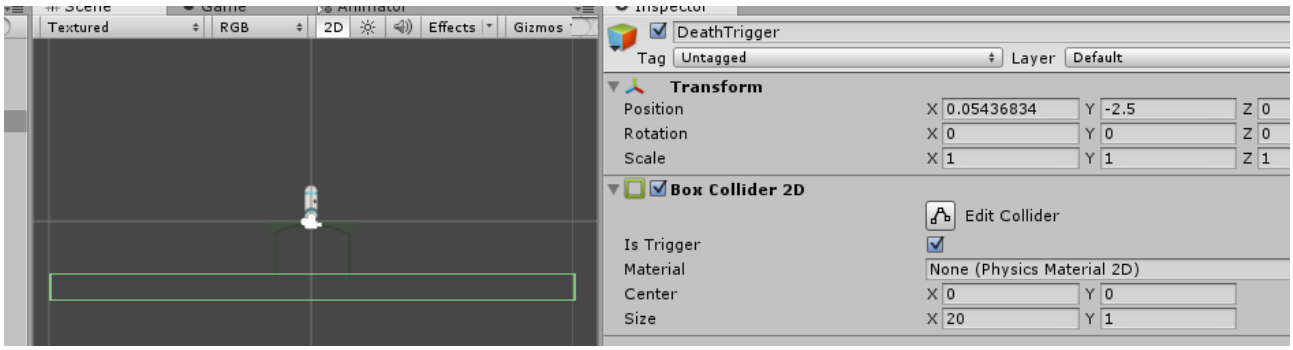
4.1 Gestió de les vides: crear els objectes necessaris

Per fer desaparèixer el player crearem un **nou objecte** del joc, al qual anomenarem **DeathTrigger**. Aquest objecte serà un **objecte buit**, sense cap imatge associada.

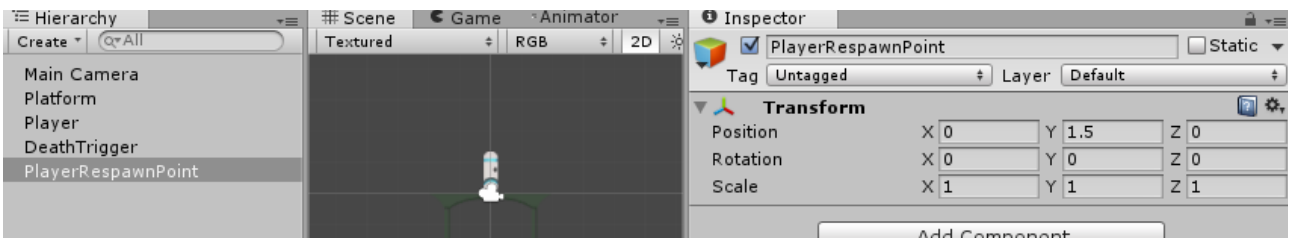


El situarem a sota de la plataforma i quan el Player caigui i hi col·lisioni desapareixerà per reaparèixer a la posició inicial, a sobre de la plataforma.

Per poder detectar la col·lisió amb DeathTrigger li hem d'associar un component **Box Collider 2D.** Li assignem **Size** X:20 i Y:1 i el situem a la posició Y:-2.5, sota la plataforma. Marquem la check box **IsTrigger**.



Per fer reviure al Player quan acabi de "morir" en col·lisionar amb **DeathTrigger**, crearem un altre objecte buit que anomenarem **PlayerRespawnPoint** (punt de reaparició del Player). Li assignem la posició X:0 Y:1,5 una mica per sobre de la posició inicial del Player.

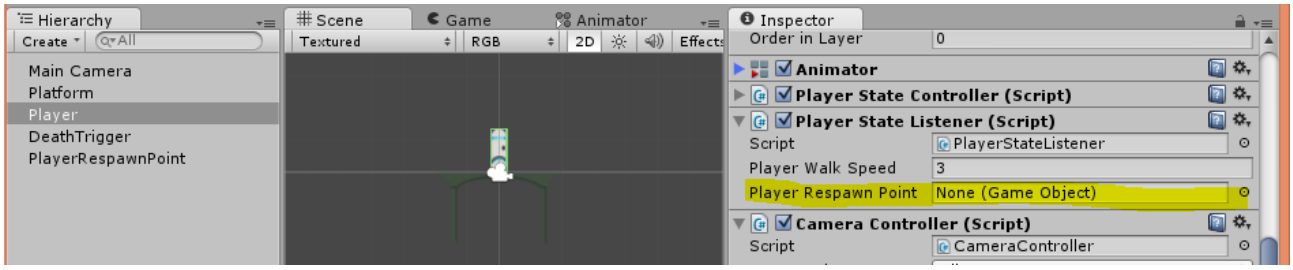


Per poder utilitzar l'**objecte PlayerRespawnPoint** afegim a l'script PlayerStateListener la **variable** playerRespawnPoint.

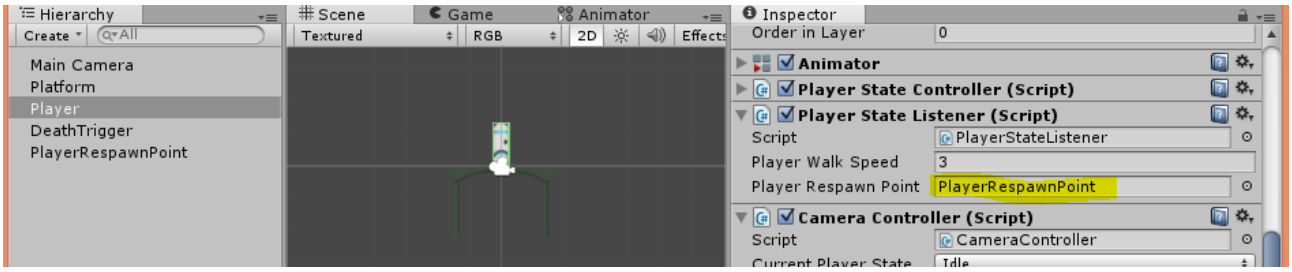
```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Animator))]
public class PlayerStateListener : MonoBehaviour
{
	public float playerWalkSpeed = 3f;
	public GameObject playerRespawnPoint = null;

	private Animator playerAnimator = null;

	.
	.
	.
```

L'assignació de l'objecte a la variable la podem fer des de l'Object Inspector, on es mostren les variables públiques dels scripts.

Seleccionem l'objecte **Player** a la pestanya Hierarchy i, a l'Object Inspector veiem els seus components. En el component corresponent a l'**script PlayerStateListener**, hi apareix la variable pública **playerRespawnPoint** (ho mostra amb les paraules separades i la primera començant en majúscula).



Cliquem al petit cercle a la dreta i seleccionem l'objecte PlayerRespawnPoint com a contingut d'aquesta variable:



4.2 Gestió de les vides: crear i adaptar scripts

A continuació gestionarem per mitjà d'scripts les següents accions:

1. Permetre que l'objecte DeathTrigger detecti la col·lisió i reaccioni enviant-li al Player un missatge que li digui "estàs mort".
2. Permetre que el Player rebi el missatge i ...
3. ... gestioni la seva mort: moure's al punt de reaparició PlayerRespawnPoint.

4.2.1 Implementació del punt 1:

Per realitzar el que es descriu en el **punt 1**, creem un nou script, que anomenem **DeathTriggerScript**, i l'assignem a l'objecte DeathTrigger. El codi és el següent:

```
using UnityEngine;
using System.Collections;

public class DeathTriggerScript : MonoBehaviour
{

      void OnTriggerEnter2D( Collider2D collidedObject )
      {
            collidedObject.SendMessage("hitDeathTrigger");
      }
}
```

Comentaris al codi:

Com que en el collider de DeathTrigger hem marcat el check **IsTrigger**, quan es produeix una col·lisió es dispara un event que és tractat al mètode **OnTriggerEnter2D**. Veiem que aquest mètode rep com a paràmetre l'objecte amb el qual ha col·lisionat (en el nostre cas només pot ser el Player).
Veure http://docs.unity3d.com/ScriptReference/Collider2D.OnTriggerEnter2D.html

Com a tractament de l'event, l'objecte DeathTrigger envia un missatge a l'objecte col·lisionat, utilitzant el mètode SendMessage i passant-li com a paràmetre el nom d'un mètode, de l'objecte que ha col·lisionat (en aquest cas "hitDeathTrigger").
Veure http://docs.unity3d.com/ScriptReference/GameObject.SendMessage.html

4.2.2 Implementació del punt 2:

Per tal que el missatge de DeathTrigger faci reaccionar al Player, li hem d'implementar el mètode hitDeathTrigger. Ho farem a l'script PlayerStateListener.

```
//Metode cridat en caure de la plataforma i col.lisionar amb DeathTrigger
public void hitDeathTrigger()
{
      onStateChange(PlayerStateController.playerStates.kill);
}
```

Comentaris al codi:
Aquest mètode s'executa com a conseqüència del missatge de DeathTrigger. La única

cosa que fa és cridar directament a onStateChange (com si s'hagués produit un event de canvi d'estat) per canviar l'estat del Player a **kill**.

Amb aquest canvi d'estat n'hi ha prou per a iniciar els tractaments adequats, que comentarem a continuació.

4.2.3 Implementació del punt 3:

(Nota: part del codi que es descriu en aquest apartat ja hi és als scripts, encara que fins ara no fos necessari.)

En el procés de matar i fer reviure el Player hi intervenen els estats **kill** (han matat al Player) i **resurrect** (el Player reviu i torna a aparèixer). De moment cap dels dos estats té programat cap tractament. A mida que anem completant el joc hi anirem afegint codi. Ara comencem a fer-ho.

La programació de la mort i reaparició del Player suposa fer el següent:

4.2.3.1 Incloure els estats **kill** i **resurrect** a l'enumeració **playerStates** de l'script PlayerStateController.

4.2.3.2 Tractar els estats **kill** i **resurrect** en el switch de la funció **onStateChange** de l'script PlayerStateListener.

La funció **onStateChange és cridada de manera asíncrona per Unity** quan es genera un event de canvi d'estat del Player. També es pot cridar des del nostre codi per provocar directament un canvi d'estat, sense intervenció de l'usuari.

Aquesta funció realitza els **tractaments inicials corresponents al nou estat** i actualitza les variables on es guarda l'estat actual i l'anterior. De moment no hi ha cap tractament inicial particular per a l'estat **kill** **(B)**. Com a tractament inicial de l'estat **resurrect**, mourem el Player al punt de reaparició (l'objecte **PlayerRespawnPoint**) **(C)**.

Per manegar l'objecte **PlayerRespawnPoint** utilitzem playerRespawnPoint, la variable que hem afegit abans a l'script PlayerStateListener **(A)**.

```
. . .
public GameObject playerRespawnPoint = null;   (A)
. . .
      case PlayerStateController.playerStates.kill:    (B)
           break;

      case PlayerStateController.playerStates.resurrect:    (C)
                                    //posicio: la de PlayerRespawnPoint
           transform.position = playerRespawnPoint.transform.position;
           transform.rotation = Quaternion.identity; //rotacio: cap
           rigidbody2D.velocity = Vector2.zero;      //velocitat lineal: zero
           break;
. . .
```

Veure http://docs.unity3d.com/ScriptReference/Quaternion-identity.html
http://docs.unity3d.com/ScriptReference/Rigidbody2D-velocity.html

4.2.3.3 Tractar els estats **kill** i **resurrect** en el switch de la funció **onStateCycle** de l'script PlayerStateListener.

La funció **onStateCycle es crida a cada frame** des de LateUpdate(). Realitza els **tractaments periòdics corresponents a l'estat actual**. De moment no hi ha cap tractament periòdic particular per a l'estat **kill** ni per a l'estat **resurrect**.

Temporalment escribim el següent codi per passar directament de **kill** a **resurrect** i de **resurrect** a **idle**.

```
. . .

      case PlayerStateController.playerStates.kill:
           onStateChange(PlayerStateController.playerStates.resurrect);
           break;

      case PlayerStateController.playerStates.resurrect:
           onStateChange(PlayerStateController.playerStates.idle);
           break;
. . .
```

4.2.3.4 Tractar els estats **kill** i **resurrect** en el switch de la funció **checkForValidStatePair** de l'script PlayerStateListener.

Aquesta funció es crida des de **onStateChange**, en el moment en que es detecta un canvi d'estat, i comprova que la transició cap al nou estat sigui vàlida. Des de **kill** només podem passar a **resurrect** i des de **resurrect** només podem passar a **idle**:

```
            case PlayerStateController.playerStates.kill:

                  // Des de kill només es pot passar a resurrect

                  if(newState == PlayerStateController.playerStates.resurrect)
                        returnVal = true;
                  else
                        returnVal = false;
                  break;
. . .
            case PlayerStateController.playerStates. Resurrect :

                  // Des de resurrect només es pot passar a Idle

                  if(newState == PlayerStateController.playerStates.idle)
                        returnVal = true;
                  else
                        returnVal = false;
                  break;
```

Ara podem provar d'executar el joc. Si el Player cau de la plataforma, en tocar el DeathTrigger, desapareix i reviu al punt PlayerRespawnPoint.

La càmara es continua comportant de manera adequada perquè el seu moviment està associat als estats del Player.

Activitats

1. Experiment proposat a la pàgina 6.

2. Fer un llistat amb els conceptes de introduits en aquest document, indicar la pàgina i explicar què són i per a què s'utilitzen.