Práctica 2

Creación de un juego con físicas

Índice

1. Importar el proyecto y configurar la escena y los modelos

En esta práctica vamos a crear un videojuego de lógica muy sencilla. La temática es espacial y jugaremos con una nave, tratando de encontrar **4 puntos en el mapa** para ganar el juego. Por contra, habrá "torretas" que nos dispararán cuando no acerquemos a ellas para evitar que consigamos nuestro objetivo.

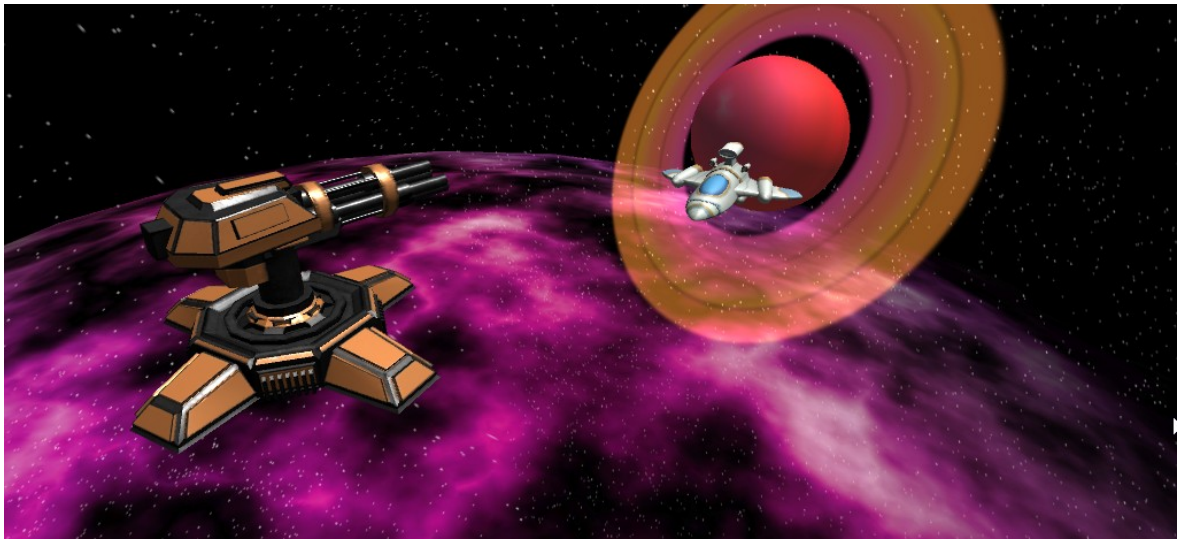

Figura 1. Imagen del escenario.

1.1. Importar un nuevo proyecto y configurar la escena

Para ello primero vamos a importar el proyecto, que utilizaremos como base para nuestro videojuego, desde el siguiente enlace:
https://github.com/oscarviu/Practica-2.git

El siguiente paso será configurar el **fondo del escenario** cuyos elementos encontrarás en la carpeta "Background". Puedes utilizar la escena de prueba **`test`** para obtener una referencia del escenario requerido (fondo estrellado, nebulosas, galaxia y un par de planetas).

Así pues crea una nueva escena llamada **`main`** con el escenario base formado por las nebulosas y galaxia de la escena **`test`** con, al menos, 6 planetas distintos que crearás utilizando los materiales y texturas que encontrarás en **`Background`**. Ubica todos los planetas sobre el **plano XZ** por debajo del **`grid`** pues el resto de elementos (nave, torretas y bases) se ubicarán por encima del escenario (`y=0`). Intenta separarlos un poco para dar variedad al entorno (no como en la figura siguiente). También puedes inclinar algunos planetas (sobre todo los que tienen anillos) para aumentar la originalidad.

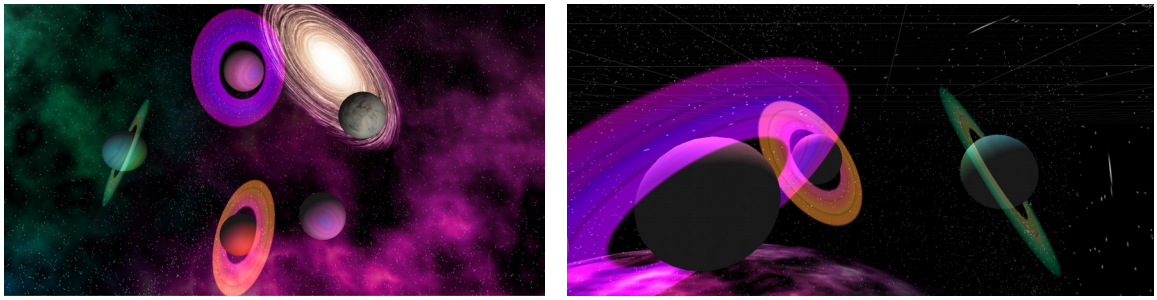

Figura 2. Escenario base.

1.2. Configurar la nave principal

Ahora tenemos que configurar la nave que representará al jugador principal. Lo primero será crear un material llamada **`matPlayer`**, cuyo color *albedo* lo definirá la textura pertinente, y aplicalo al objeto de la escena.

Figura 3. Nave configurada con el material requerido.

A continuación tenemos que corregir el pivote de la nave porque el modelo que utilizamos no lo tiene centrado. Para ello crearemos en la escena un `gameObject` vacío llamado **`Player`** y en su interior alojaremos el modelo.

Como vemos en la *figura 4* el pivote se encuentra fuera del modelo y su eje Z tiene el sentido contrario deseado.

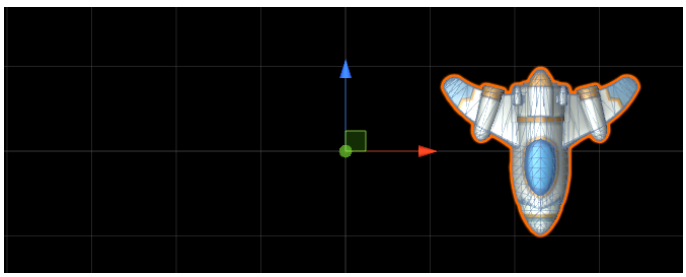

Figura 4. Pivote de la nave desplazado.

Para ello deberemos aplicar una serie de transformaciones (rotación de 180 grados y desplazamiento en el eje X) para centrar el modelo. Estas transformaciones las realizaremos sobre el objeto hijo, no sobre **`Player`**.

Figura 5. Transformaciones para centrar el pivote. Rotación (izquierda), traslación (derecha).

Con lo que el resultado final será el mostrado en la *figura 6*.

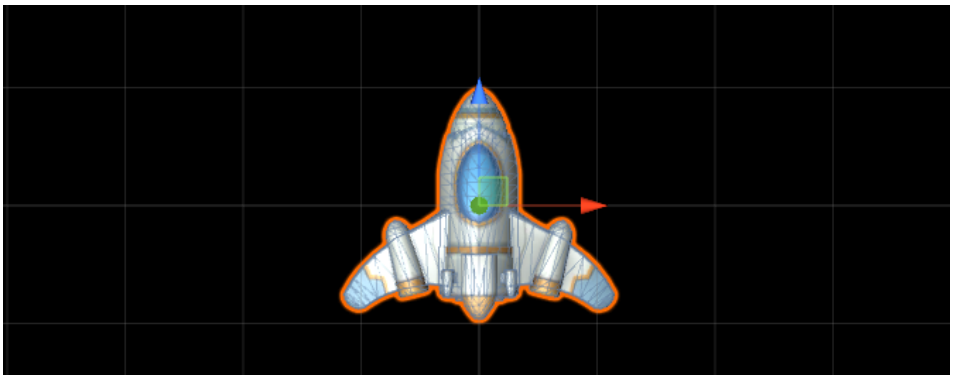

Figura 6. Resultado final de las transformaciones.

Si tienes dificultados en este paso puede utilizar este video como referencia:
https://goo.gl/6ans9x

El siguiente y último paso será añadir al objeto el componente **`RigidBody`** y **`BoxCollider`** que serán necesarios para programar el juego más adelante. Estos componentes los añadiremos al objeto padre, **`Player`**.

En cuanto a **`RigidBody`** deberemos indicar que **no le afecte la gravedad** y configurar las *constraints* para que **no le afecten fuerzas** que lo empujen hacia **fuera del plano XY**, además de forzar las **rotaciones sólo en el eje Y**. El **`BoxCollider`** se utilizará para las colisiones (no configurar como `trigger`).

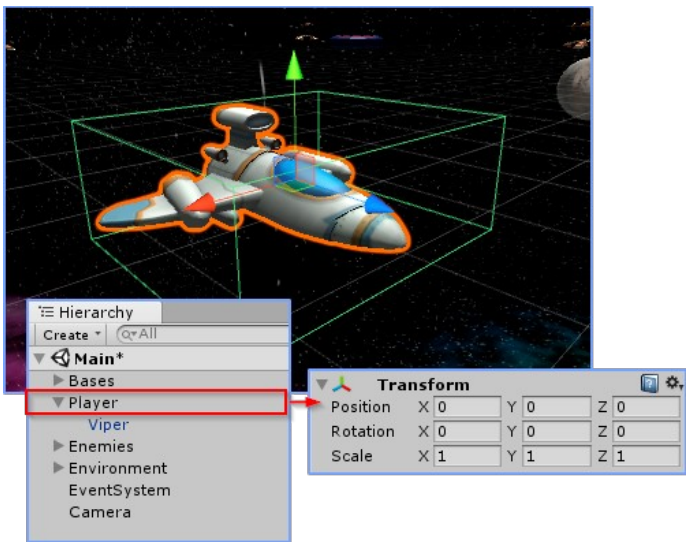

Figura 7. Nave con los componentes requeridos para la práctica.

También indicaremos que el ***tag* de este objeto será *Player*** (si no existe la etiqueta la crearemos).

Una vez configurado el objeto **`Player`** crearemos un *prefab* llamado **`player.prefab`** que nos facilitará la tarea a la hora de realizar modificaciones futuras.

1.3. Configurar las torretas

Ahora toca preparar las torretas. Por suerte ya tenemos un *prefab* con el modelo configurado en cuanto a algunos parámetros se refiere (escala, ajuste de pivote, agrupación de objetos, etc.). Para ello crearemos un nuevo material llamado **`matTurret`** con las 3 texturas correctamente configuradas y se lo **aplicaremos a todas las partes** del *prefab* (figura 8).

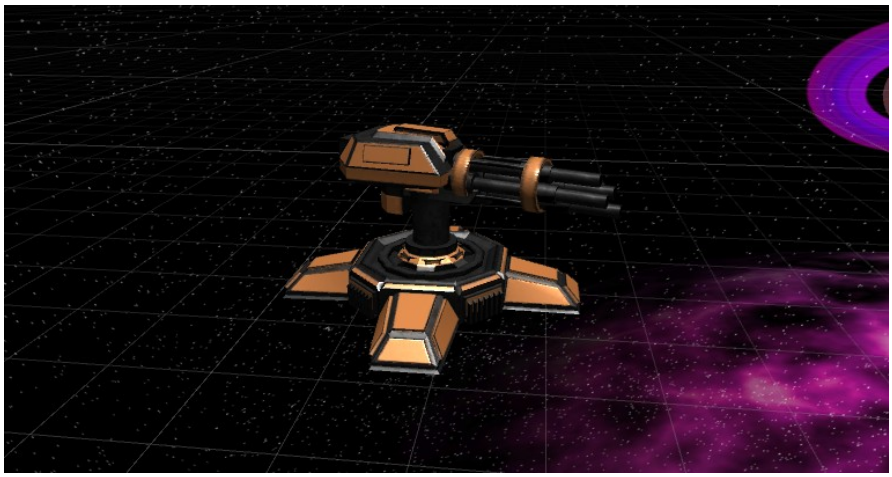

Figura 8. Torreta con el material aplicado.

Añadiremos un **`BoxCollider`** para las colisiones y un **`RigidBody`** que configuraremos para que no le afecte **ninguna fuerza** en absoluto (congelado en todos los ejes).

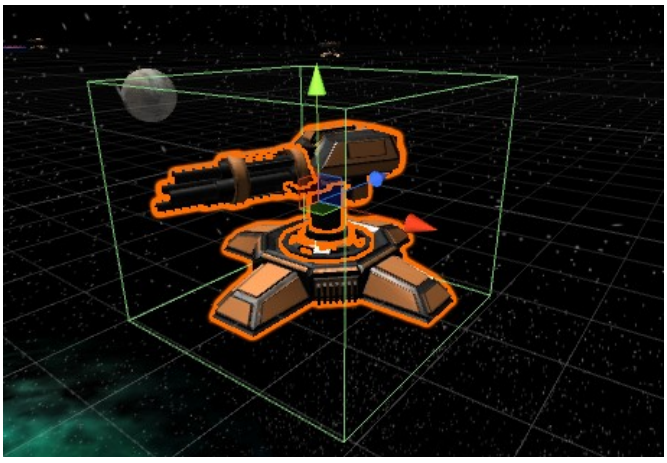

Figura 9. Collider de la torreta.

También indicaremos que el ***tag* de este objeto será *Turret*** (si no existe la etiqueta la crearemos).

Una vez lo tengamos configurado la torreta crearemos un *prefab* llamado **`Turret.prefab`**.

1.4. Configurar la base

La base es otro elemento del escenario que deberemos preparar. Lo primero es crear dos materiales distintos llamadas **`matBaseColor`** y **`matBaseNeon`** con los siguientes parámetros:

- **`matBaseColor`**: Standard (Opaque) cuyo color albedo será azul (051347FF).
- **`matBaseNeon`**: Standard (Opaque) cuyo color albedo será negro y con color de emisión rojo (FF0000).

A continuación, configuraremos el **`MeshRenderer`** para utilizar **dos materiales** que serán los anteriormente definidos (por ese orden). También deberemos modificar **`transform`** para orientar correctamente el modelo (con la imagen de la cruz hacia arriba, eje Y) y escalarlo a un valor de 60 en todas las dimensiones.

Ademas de eso añadiremos al objeto una **luz de tipo *point*** (componente **`Light`**) que permanecerá **desactivada por defecto** con los siguientes parámetros:

- *Luz verde.*
- *Rango 1.*
- *Intensidad: 16.*

También le añadiremos **2 `BoxCollider`**. El primero de ellos envolverá al modelo de forma ajustada y se encargará de las colisiones. El segundo, sin embargo, será más grande (de ancho aproximadamente 0.044 unidades) y se utilizará para programar un **`Trigger`**. El aspecto final con los dos colliders los puedes ver en la siguiente figura:

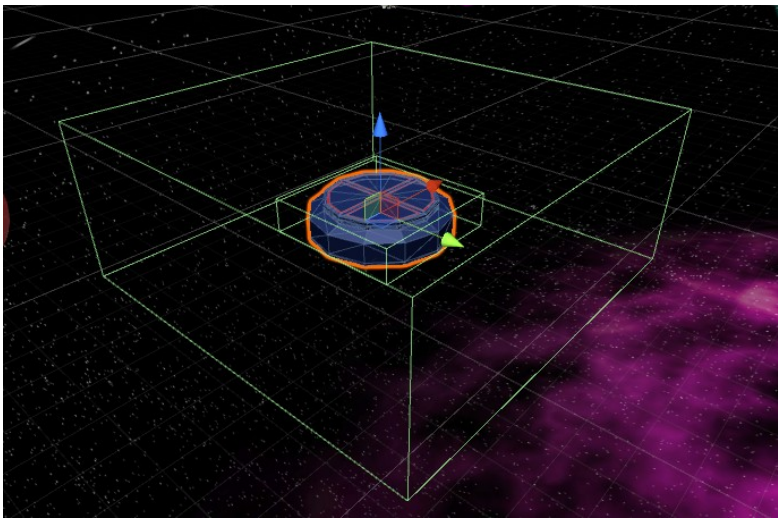

Figura 10. Base Collider de la torreta.

Una vez configurado el objeto crearemos un *prefab* llamado **`Base.prefab`**.

1.5. Configurar el escenario final

Ahora que ya tenemos todos los elementos preparados solamente nos queda ubicarlos por la escena y ya tendremos el nivel sobre el cual jugaremos con nuestra nave. Para ello vamos a

instanciar **4 bases** que posicionaremos en lugares lejanos, y entre ellas **12 torretas** por el resto de zonas vacías. La nave del jugador la colocaremos en el centro de coordenadas (0,0,0). Tenemos un ejemplo en la siguiente figura.

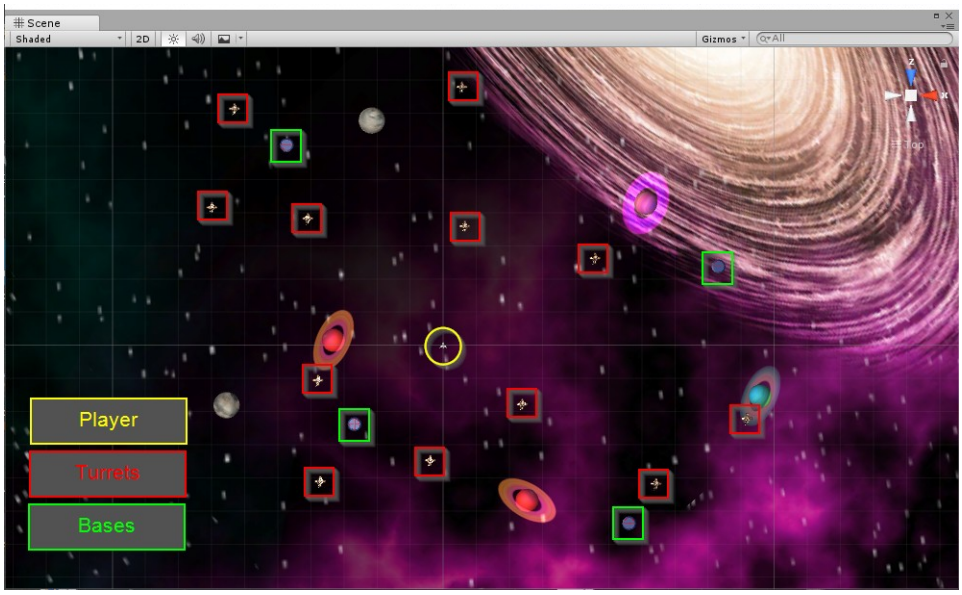

Figura 11. Ejemplo de disposición de elementos del juego.

Además agruparemos cada elementos como se muestra en la figura inferior:

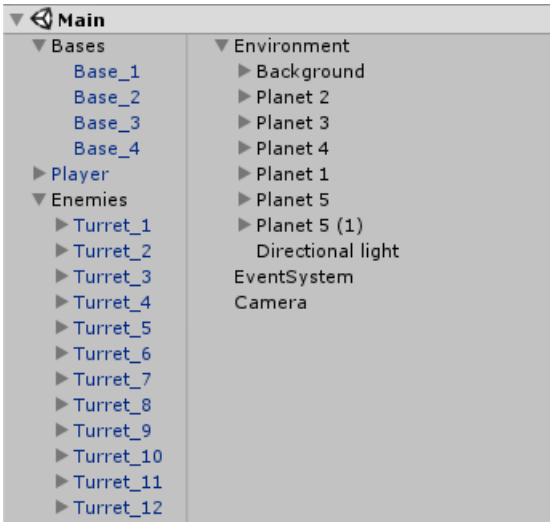

Figura 12. Agrupación de elementos.

Cuando lo tengamos listo guardaremos la escena **`main.scene`**.

2. Programar el juego

A partir de ahora nos vamos a dedicar a programar la lógica del juego. Vamos a empezar configurando la cámara para que siempre mire a la nave y se mueva con ella.

2.1. Configurar la cámara

Primero debemos indicar que el ángulo de apertura (*field of view*) sea **21**. Crearemos un *script* llamada **`CameraBehaviour`** que añadiremos a la cámara principal para seguir al jugador (nave).

Figura 13. La cámara configurada como vista aérea.

El esqueleto del *script* de la cámara será el siguiente:

```
public class CameraBehaviour : MonoBehaviour
{
	/* Variables publicas
	 * - Variable objetivo (player)
	 * - Altura mínima (4f)
	 * - Factor de altura
	*/

	/* Variables privadas
	 * - Altura
	*/

	void LateUpdate ()
	{
		// Mi posición = posicion de objetivo + Altura
		// Altura = [Altura mínima, Altura mínima + velocidad objetivo * factor de altura]
	}
}
```

Figura 13. Esqueleto del *script* de la cámara.

Podemos observar que necesitaremos asignarle el objetivo por el inspector, arrastrando el objeto **`Player`** en el espacio correspondiente. Una configuración válida del *script* es la mostrada en la siguiente figura:

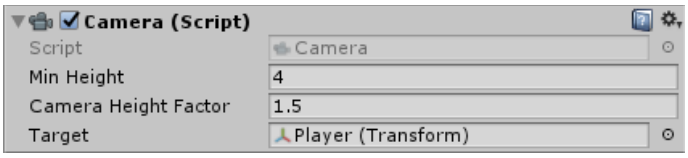

Figura 14. Configuración de *CameraBehaviour*.

La cámara permanecerá a una altura mínima cuando la nave esté parada y subirá cuando ésta se mueva. En la figura superior podemos observar que hemos configurado una **altura mínima de 4** unidades (cuando la nave está parada) pero la altura en movimiento estará definida por la fórmula:

`MinHeight * (1 + velocidad_Target`[1] `/ CameraHeightFactor)`.

Consejo: Para suavizar el efecto también podemos utilizar la función **`Lerp`**. *No es obligatorio*.

2.2. Configurar Player

Para el comportamiento de la nave principal crearemos un *script* llamado **`Player`** en el cual programaremos que la nave responda a los eventos de teclado:

- Con las flechas '**arriba**' y '**abajo**' desplazaremos la nave hacia delante y hacia atrás. Para ello capturaremos cada evento en la función **`FixedUpdate()`** y aplicaremos una fuerza en la dirección correspondiente.
- Con las flechas '**izquierda**' y '**derecha**' giraremos la nave sobre su eje Y. Para ello aplicaremos en el momento adecuado una fuerza de giro (**`AddTorque()`**).

Cuando pulsemos la tecla '**espacio**' dispararemos. Para realizar este efecto lo que vamos a hacer será **instanciar un proyectil** y configurarle una dirección (hacia donde miro). El proyectil será una esfera con partículas. Si observamos el proyecto veremos que tenemos un *prefab* llamado **`Bullet`** que ya tiene esta peculiaridad diseñada. Por lo tanto, el siguiente paso será, basándose en este *prefab*, crear uno nuevo llamado **`BulletYellow`** cuyo **color de emisión y de partículas** (`Particle System`) será **amarillo** (crear un material llamado **`matBulletYellow`** para ello).

También crearemos un *collider* esférico de tipo **`Trigger`** que nos permitirá detectar la colisión del mismo con otros elementos y poder, así, decidir si debemos destruirlos o no.

Otro componente que añadiremos será una **luz puntual** (color amarillo, intensidad 12 y rango 1).

1. Para obtener la velocidad de un objeto podemos leerla del atributo `velocity.magnitude` de su componente `RigidBody`.

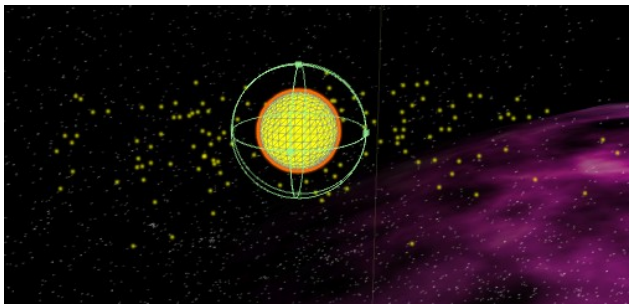

Figura 15. Configuración del *prefab* **Bullet**.

Al *prefab* **BulletYellow** vamos a asignarle un nuevo *script* llamado **BulletBehaviour** con la lógica necesaria para que, al instanciarse, se desplace hacia una dirección dada. Además especificaremos también qué hacer cuando algún objeto entre en su collider. El esqueleto de la clase **BulletBehaviour** será el siguiente:

```
public class BulletBehaviour : MonoBehaviour
{
	/* Variable pública
	 * Velocidad
	*/

	/* Variable privada
	 * Vector dirección
	*/

	public void setDirection(Vector3 dir)
	{
		// dirección = dir
	}

	// Update is called once per frame
	void FixedUpdate ()
	{
		// Actualizo mi posición con respecto a mi velocidad
	}

	void OnTriggerEnter(Collider other)
	{
		// Si mi other.tag != mi_tag
		//		Si tag = Turret
		//			Imprime "Destruida unidad other.nombre"
		//			Desactiva torreta
		//		si tag = player
		//			Llamar al metodo 'hit' de Player.
		//		Me desactivo (porque he impactado)
}
```

Figura 16. Esqueleto de **BulletBehaviour**.

Como podemos observar tenemos un método público para asignar una dirección al proyectil. Vemos también que para saber a quien debe "destruir" el proyectil utilizamos los *tags*. Por ejemplo, cuando **`Player`** dispara (instancia un proyectil) le indicará a **`Bullet`** cual es la dirección hacia la que irá y que su *tag* será ***Player***, así si impacta sobre sí mismo comprobará que su dueño (en este caso representado por la etiqueta) es el mismo y no hará nada. En cambio, cuando impacte con una torreta verá que el *tag* del objeto (*Turret* en este caso) es diferente y lo destruirá. Pasará lo mismo al contrario, pues cuando sea la torreta la que dispare llamará al método **`hit`** de **`Player`** en el momento del impacto (será ese método, en ese caso, quien evaluará si se ha quedado sin vidas y debe destruirse).

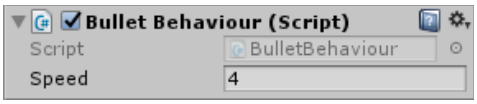

Figura 17. Parámetros públicos de **`BulletBehaviour`**.

Como observamos en la figura anterior el único parámetro que podemos configurar es la **velocidad del proyectil**.

Volviendo a **`Player`**, crearemos un script llamada **`PlayerBehaviour`** similar al *script* de la siguiente figura y se lo asignaremos.

```
public class PlayerBehaviour : MonoBehaviour {

	/* Variables privadas
	* Vidas restantes
	* Bases descubiertas
	*/

	/* Variables públicas
	 * Aceleración
	 * Aceleración de giro
	 * Objeto 'bala'
	 * Componente Rigidbody
	*/


	public void shot(){
		// Instanciar la bala
		// Indicar como tag mi tag
		// Ubicar la bala en mi posición
		// indicar a la bala la dirección a donde miro
		// con el método bala.setDirection('vector a donde miro')
	}

	public void baseFounded(){
		//Incrementar Bases descubiertas
		// Si las bases = 4 imprimir ¡¡Fin del juego!!
	}

	// función llamada cuando he sido alcanzado
	public void hit(GameObject other){
		// Restar 1 vida. Si no me quedan vidas desactivar objeto.
		// En caso contrario imprimir "Player: Daño recibido"
		// y aplicar fuerza explosion (500 unidades)
	}

	void FixedUpdate(){
		// Mover la nave con las teclas
		// Posición Addforce * acceleration
		// Rotación AddTorque * turnAcceleration

		// Si pulso tecla'R' ubicar nave en el centro de coordenadas

		// Si pulso 'space' llamo a 'shot':
	}
}
```

Figura 18. Esqueleto de **PlayerBehaviour**.

Player tendrá 3 vidas al inicio e irá perdiendo una cada vez que reciba un impacto de una torreta. Además, guardaremos en otra variable las bases descubiertas (cuando descubramos las 4 ganamos el juego, **indicándolo con un mensaje por consola**).

Cuando disparamos, internamente llamaremos al método **shot()** que instanciará un proyectil y le configuraremos la dirección correspondiente. Este proyectil lo obtendremos de una variable pública de tipo `GameObject` sobre el cual habremos arrastrado el *prefab* **BulletYellow**, previamente configurado (color amarillo y **velocidad 4**).

En cuanto al movimiento solamente comentar que tanto las fuerzas de traslación como las de rotación van multiplicadas por las variables **Acceleration** y **TurnAcceleration** respectivamente.

Para terminar comentar que cada vez que descubrimos una base se llamará al método **baseFounded()** y que cuando recibamos un impacto llamaremos a **hit()**[2].

En la siguiente figura podemos ver una configuración válida del componente **PlayerBehaviour**.

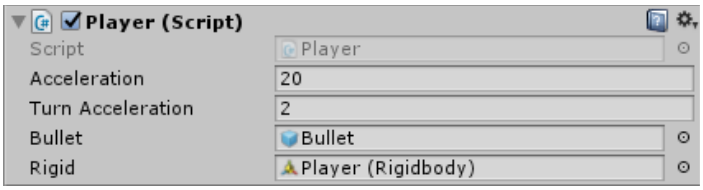

Figura 19. Configuración válida de PlayerBehaviour[3].

2.3. Configurar Turret

Para empezar, vamos a crear el proyectil de las torretas. Para ello crearemos un nuevo *prefab* llamado **BulletRed** cuyo color de emisión y de partículas (`Particle System`) será rojo (crear un material llamado **matBulletRed**). También crearemos el ***collider*** esférico de tipo *trigger*, una **luz puntual** de color rojo (rango 1, intensidad 12) y le asignaremos el script **BulletBehaviour** al igual que hicimos con el proyectil de la nave.

A continuación crearemos un *script* llamada **TurretBehaviour** donde programaremos la lógica para atacar a **Player** cuando se acerque a las torretas.

El funcionamiento básico será el siguiente: Cuando **Player** se encuentre a una distancia mínima de la torreta le apuntará con su arma y disparará con una cadencia determinada.

En la siguiente figura tenemos la implementación mínima del script en cuestión.

2. Para aplicar una fuerza de explosión podemos utilizar el método `AddExplosionForce` de la clase `RigidBody`.

3. Hay una errata en la figura, en la parte superior izquierda debería poner *PlayerBehaviour*, en lugar de *Player*.

```
public class TurretBehaviour : MonoBehaviour
{
	/* Variables públicas
	 * Cañon (Pylon)
	 * Objetivo (Player)
	 * Objeto 'bala'
	 * Distancia de disparo
	 * Frecuencia de disparo
	*/

	/* Variable privada
	 * Último tiempo de disparo,
	 *   para controlar el tiempo
	 *   que llevo sin disparar.
	*/

	void Update()
	{
		// Si la resta de la posición objetivo menos la mía < Distancia de disparo
		//		Apunto al objetivo (roto Cañon)
		//		Si Último tiempo de disparo >= Frecuencia de disparo
		//			Disparo
		// Actualizo Último tiempo de disparo
	}

	void shot()
	{
		// Último tiempo de disparo = 0
		// Instanciar la bala
		// Indicar como tag mi tag
		// Ubicar la bala en mi posición
		// indicar a la bala la dirección a donde miro
		// con el método bala.setDirection('vector a donde miro')
	}
}
```

Figura 19. Esqueleto de la clase **TurretBehaviour**.

De la misma manera que **Player** llamaremos al método **shot()** cuando la torreta dispare. La diferencia fundamental será que sólo disparará cuando tenga el objetivo cerca (**Player**) y lo hará a una cierta frecuencia de disparo.

Tendremos una variable publica que representará el cañón (la parte superior de la torreta con el arma) que haremos girar para apuntar a **Player** cuando se encuentre cerca. La distancia mínima para detectar al objetivo también será pública, pudiendo modificarla desde el inspector.

Los valores de estas variables los tenemos en la siguiente figura:

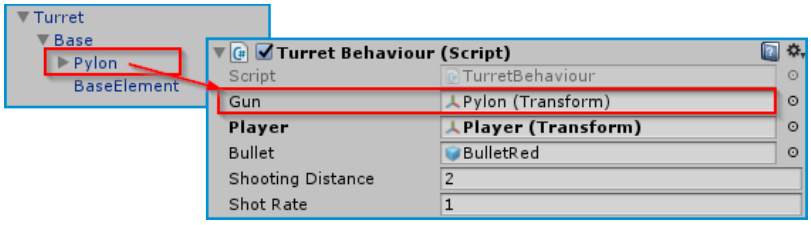

Figura 20. Valores y referencias de **TurretBehaviour**.

2.4. Configurar Base

Para finalizar el juego ya sólo nos queda programar el comportamiento de la base mediante el script **BaseBehaviour**. Tenemos el esqueleto de la clase en la siguiente figura.

```
public class BaseBehaviour : MonoBehaviour
{
	// Variable interna para
	// saber si he sido detectada

	void OnTriggerEnter(Collider other)
	{
		// Si other.tag = Player y no he sido detectada
		// - Cambiar el color de emisión a verde
		// - Encender la luz
		// - Llamar a baseFounded() de Player
		// - Recargar las vidas
	}
}
```

Figura 21. Esqueleto de **BaseBehaviour**.

El mecanismo es muy sencillo. Cuando **Player** se introduzca en el **Trigger**, la base comprobará si el objeto es *Player* y si no había sido antes descubierta. En ese caso cambia el color de emisión a verde y enciende la su luz. A continuación informa a **Player** de que ha sido descubierta (llama al método **baseFounded()**) y luego recarga las vidas del jugador a tres. Esta última acción podemos delegarla a **Player** dentro de su método **baseFounded()**.

Para cambiar el color de emisión de un material tenemos que utilizar el método **SetColor** de la clase **Material**, utilizando como nombre del campo **"_EmissionColor"**. Por ejemplo, para cambiar el color de emisión a verde haremos:

```
material.SetColor("_EmissionColor", Color.green);
```

Calificación de la práctica	
1.1. Importar un nuevo proyecto y configurar la escena	5%
1.2. Configurar la nave principal	10%
1.3. Configurar las torretas	10%
1.4. Configurar la base	10%
1.5. Configurar el escenario final	5%
2.1. Configurar la cámara	10%
2.2. Configurar Player	20%
2.3. Configurar Turret	20%
2.4. Configurar Base	10%

Para realizar la entrega se deberá exportar el proyecto completo indicando, como nombre del archivo, vuestro nombre completo con la siguiente estructura:
`P2_Apellido1_Apellido2_Nombre.unitypackage`