

Space Shmup

Pedro Antonio González Calero (pedro@fdi.ucm.es)



Contents

| | |
|---|-----------|
| El juego | 2 |
| Importación de un paquete de recursos de Unity | 2 |
| Configuración de la escena | 3 |
| La nave | 3 |
| Los enemigos | 13 |
| Despliegue de enemigos aleatoriamente | 16 |
| Configuración de etiquetas, capas y física | 17 |
| Los enemigos destruyen la nave del jugador cuando chocan con ella | 17 |
| Reinicio del juego cuando muere la nave del jugador | 22 |
| A disparar | 22 |
| Power-Ups | 33 |
| “Condiciones de carrera” | 39 |
| Los enemigos sueltan power-ups | 40 |
| Otros tipos de enemigos | 42 |
| Añadir un fondo de estrellas | 52 |
| Referencias | 53 |

El juego

Los juegos tipo SHMUP (*shoot 'em up*) aparecen con algunos clásicos de los 80s como Galaga y Galaxian. En la figura 1 se muestra una captura del juego que vamos a desarrollar. Nuestra nave se mueve por la pantalla libremente, enfrentándose a oleadas de enemigos de distinto tipo que siguen diferentes trayectorias y que al ser destruidos sueltan *power_ups* que pueden mejorar nuestra nave.

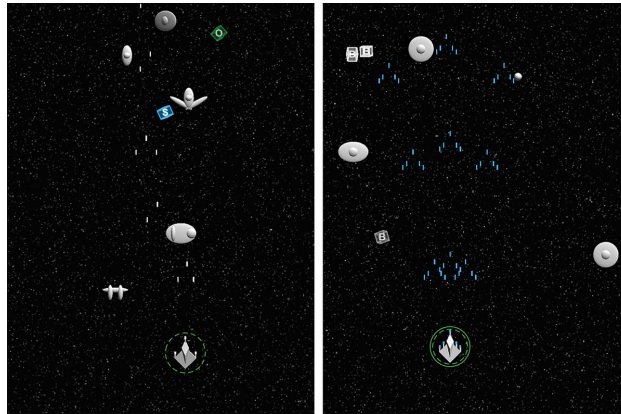


Figure 1: Space Shmup

Importación de un paquete de recursos de Unity

Creamos un nuevo proyecto, creamos las carpetas `__Scripts`, `__Materials` y `__Prefabs` donde organizaremos los recursos del proyecto, renombramos la cámara principal a `__Main Camera`, e importamos el paquete de recursos `space_shmup.unitypackage`, descargado de campus virtual, utilizando para ello la orden `Assets > Import Package > Custom Package`. Una vez seleccionado el fichero, aparecerá un cuadro de diálogo como el de la figura 2

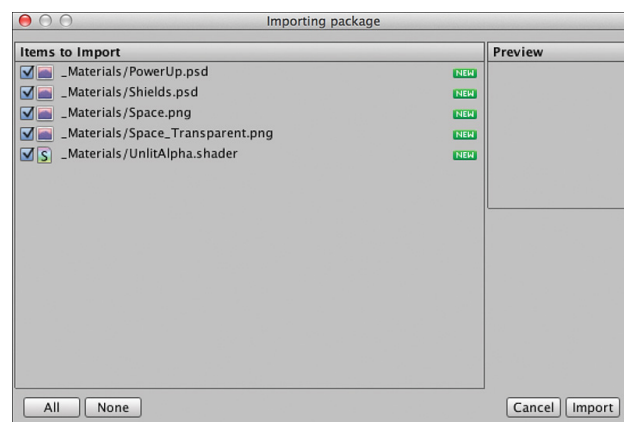


Figure 2: Importación de un paquete de recursos

Se seleccionan todos los archivos y se pulsa el botón *Import*. El paquete contiene algunas texturas creadas con Photoshop y un shader sencillo que permite hacer efectos de transparencia. Más información sobre los shaders de Unity en <http://docs.unity3d.com/Documentation/Components/SL-Reference.html>.

Configuración de la escena

Añade una luz direccional a la escena y ubícala en P: [0,20,0] R: [50,330,0] S: [1,1,1]. Ubica la cámara en P: [0,0,-10] R: [0,0,0] S: [1,1,1] y configúrala con: color de fondo a negro, proyección *Orthographic*; *Size* a 40; y los atributos *Near__* y *Far* de *Clipping Planes* a 0.3 y 100.

Como este juego es un shooter de scroll vertical, vamos a configurar la proporción de aspecto (*aspect ratio*) de la cámara como se suele encontrar en este tipo de juegos. En el panel *Game* pulsa en el desplegable que aparece en la esquina superior izquierda, justo debajo de la pestaña del panel, y en la lista selecciona la opción + que aparece al final, para así añadir un nuevo elemento, *Portrait (3:4)* con los valores que se muestra en la figura 3. Por último asigna esta nueva configuración como el *aspect ratio* de la pantalla de juego.

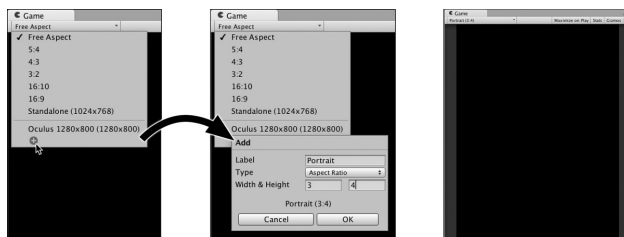


Figure 3: Configuración del aspect ratio de la cámara

La nave

Para crear el modelo de la nave:

1. Crea un objeto vacío de nombre *_Hero* en la posición P: [0,0,0] R: [0,0,0] S: [1,1,1].
2. Crea un cubo de nombre *Wing* y hazlo un hijo de *_Hero*. Ubícalo en P: [0,-1,0] R: [0,0,45] S: [3,3,0.5].
3. Crea un objeto vacío de nombre *Cockpit* y hazlo un hijo de *_Hero*.
4. Crea un cubo y hazlo hijo de *Cockpit*. Ubica el cubo en P: [0,0,0] R: [315,0,45] S: [1,1,1].
5. Ubica el objeto *Cockpit* en P: [0,0,0] R: [0,0,180] S: [1,3,1].
6. Crea un script de nombre *Hero*, colócalo en la carpeta *___Scripts* y añadelo como un componente al objeto *_Hero*.
7. Añade un componente *Rigidbody* a *_Hero*, y configúralo con: *Use Gravity* a falso, *isKinematic* a cierto y en la zona de *Constraints* marca el eje z de *Freeze Position* y los tres ejes xyz de *Freeze Rotation*.

Guarda la escena con el nombre *_Scene_0*.

Movimiento de la nave

Para controlar la nave usaremos el teclado, leyéndolo a través del *InputManager* de Unity. El *InputManager* que se configura a través del cuadro de diálogo que se muestra en la figura 4 accesible con la orden *Edit > Project Settings > Input*, permite tratar la entrada a través de atributos lógicos, o “ejes” como “horizontal”, “vertical” y “fire1”, donde además puede aparecer la misma etiqueta más de una vez, asociada en un caso al teclado y en otra al joystick, como se ve en la configuración de la figura para la etiqueta “horizontal”.

Para acceder al *InputManager* desde programa se utiliza la función *Input.GetAxis()* a la que se le pasa el nombre del “eje” de entrada que se quiere consultar y que devuelve un número entre -1 y 1, con un valor por defecto de 0. Cada eje del *InputManager* además de especificar la entrada que lo modifica, permite definir los valores de sensibilidad (*sensitivity*) y gravedad (*gravity*). La sensibilidad y la gravedad controlan la transición del valor devuelto por *Input.GetAxis()* cuando la entrada cambia de estado, interpolando, por ejemplo, entre

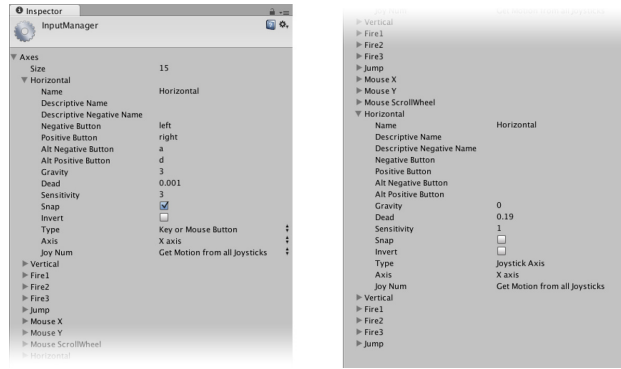


Figure 4: Configuración del InputManager

1 y 0 cuando una tecla que estaba pulsada deja de estarlo. Por ejemplo, en la figura se muestra para el eje horizontal una sensibilidad de 3, lo que quiere decir que cuando se pulsa la flecha derecha, se tarda $1/3$ de segundo en interpolarse entre 0 y 1. Por su parte una gravedad de 3, indica que cuando se libere la flecha izquierda, el valor del eje tardará $1/3$ de segundo en interpolarse entre 1 y 0. Mientras mayor sea el valor de la gravedad o la sensibilidad, más rápida será la interpolación.

Además, si generamos un ejecutable del juego desde Unity, en el cuadro de diálogo que aparece al ejecutarlo es posible configurar las teclas asociadas con los ejes de entrada, como se muestra en la figura 5.

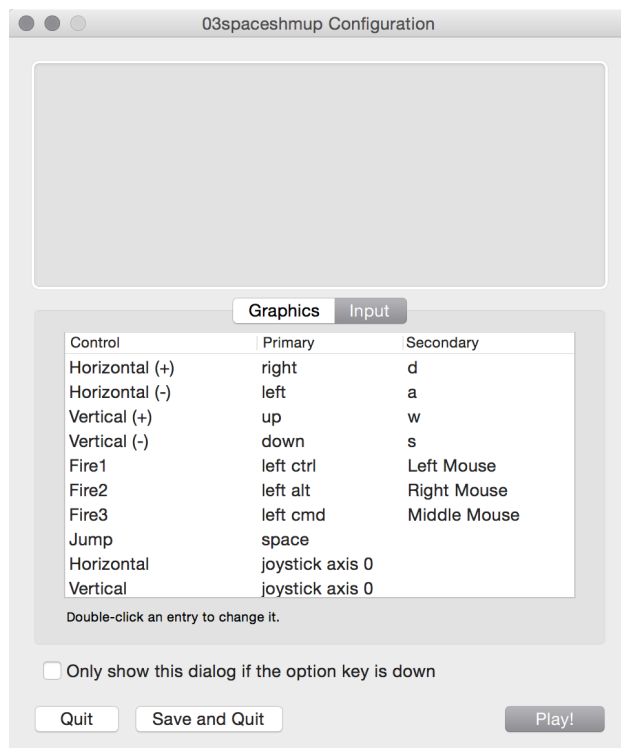


Figure 5: Configuración de input manager

A continuación añadimos el siguiente código a la clase *Hero* para controlar el movimiento de la nave.

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

```

public class Hero : MonoBehaviour {
    static public Hero S; // Singleton

    // These fields control the movement of the ship
    public float speed = 30;
    public float rollMult = -45;
    public float pitchMult = 30;

    // Ship status information
    public float shieldLevel = 1;

    public bool _____;

    void Awake() {
        S = this; // Set the Singleton
    }

    void Update () {
        // Pull in information from the Input class
        float xAxis = Input.GetAxis("Horizontal"); // 1
        float yAxis = Input.GetAxis("Vertical"); // 1

        // Change transform.position based on the axes
        Vector3 pos = transform.position;
        pos.x += xAxis * speed * Time.deltaTime;
        pos.y += yAxis * speed * Time.deltaTime;
        transform.position = pos;

        // Rotate the ship to make it feel more dynamic // 2
        transform.rotation = Quaternion.Euler(yAxis*pitchMult,xAxis*rollMult,0);
    }
}

```

La última línea rota la nave en el sentido del movimiento, utilizando el valor leído del *InputManager*. Nótese que aunque hemos congelado la rotación de la nave en el componente *RigidBody*, eso sólo afecta a los cálculos de la física, porque como al mismo tiempo la hemos configurado como cinemática, podemos modificar su rotación por código

Cuando ejecutamos el juego observamos el efecto de la gravedad y la sensibilidad de los ejes horizontal y vertical del *InputManager* que al interpolar la reacción al teclado dan una sensación de inercia.

El escudo de la nave

El escudo será un objeto de tipo *quad* (un cuadrado plano) con un componente de colisión de tipo *Sphere collider*. Empezamos creando un objeto de tipo *Quad* al que le damos el nombre de *Shield* y lo hacemos un hijo de *__Hero*, ubicándolo en P: [0,0,0] R: [0,0,0], S: [8,8,8]. A continuación le añadimos un componente de tipo *Sphere Collider* (*Component > Physics > Sphere Collider*) y le eliminamos el componente *Mesh Collider* que tenía previamente.

Creamos un material nuevo al que llamamos *Mat Shield* y lo ubicamos en la carpeta *__Materials* del panel del proyecto. Aplicamos este material al objeto *Shield*.

Seleccionamos el objeto *Shield* en la jerarquía y en el inspector configuramos el material *Mat Shield*, asignándole el *shader* de nombre *Custom > UnlitAlpha*. Por debajo del atributo *shader* del material hay una zona donde

se puede configurar su color (si no se ve, hay que pulsar sobre *Mat Shield* en el inspector) y además se puede elegir una textura pulsando en el botón *Select* que aparece en el vértice inferior derecho. Lo hacemos y seleccionamos la textura *Shields* que importamos al principio. Establece el color, *Main Color*, a verde (RGBA:[0,255,0,255]) y a continuación configura los valores *Tiling.x* a 0.2 y *Offset.x* a 0.4. *Tiling.y* se debe mantener a 1, y *Offset.y* a 0.

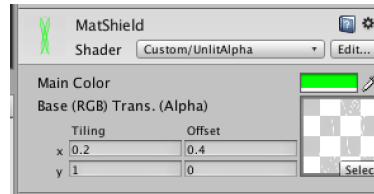


Figure 6: Configuración del material

Lo que estamos haciendo es mostrar una de las imágenes del spritesheet de la figura 7. *Tiling.x* a 0.2 hace que sólo mostremos 1/5 del ancho de la figura y *Offset.x* a 0.4 hace que mostremos el tercer fragmento. Para esta imagen podemos usar los valores 0, 0.2, 0.4, 0.6, y 0.8 como *Offset.x*.

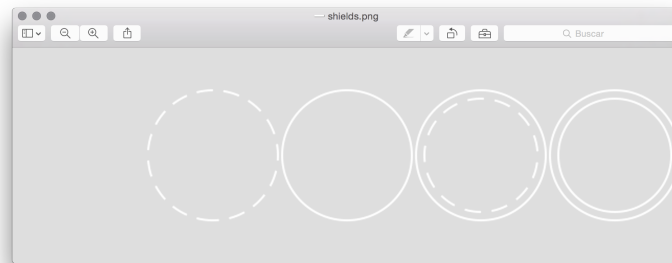


Figure 7: Configuración de input manager

A continuación creamos un script de nombre *Shield* que asociamos al objeto Shield y donde incluimos el siguiente código:

```
using UnityEngine;
using System.Collections;

public class Shield : MonoBehaviour {
    public float    rotationsPerSecond = 0.1f;
    public bool     -----;
    public int      levelShown = 0;

    void Update () {
        // Read the current shield level from the Hero Singleton
        int currLevel = Mathf.FloorToInt( Hero.S.shieldLevel );           // 1
        // If this is different from levelShown...
        if (levelShown != currLevel) {
            levelShown = currLevel;
            Material mat = this.GetComponent<Renderer>().material;
            // Adjust the texture offset to show different shield level
            mat.mainTextureOffset = new Vector2( 0.2f*levelShown, 0 );    // 2
        }
        // Rotate the shield a bit every second
    }
}
```

```

float rZ = (rotationsPerSecond*Time.time*360) % 360f;           // 3
transform.rotation = Quaternion.Euler( 0, 0, rZ );
}
}

```

Evitar que la nave se salga de la pantalla

Para evitar que la nave se salga de la pantalla tenemos que obtener un volumen que la contenga y a partir de las coordenadas de la nave en cada momento comprobar que ese volumen no se sale del campo de visión de la cámara.

La nave como tal es un objeto vacío que no tiene ni representación visual (*renderer*) ni componente de colisiones (*collider*). Lo que haremos será calcular un volumen delimitador que incluya las áreas de los *renderer* y *collider* de sus objetos hijo.

Tanto *renderers* como *colliders* tienen un atributo *bounds* de tipo *Bounds* que define un paralelepípedo a partir de dos valores de tipo *Vector3* que representan el centro y las dimensiones del paralelepípedo, como se muestra en la figura 8, donde se ha creado un objeto de tipo *Bounds* con la sentencia `Bounds bnd = new Bounds(new Vector3(3,4,0), new Vector3(16,16,0));`

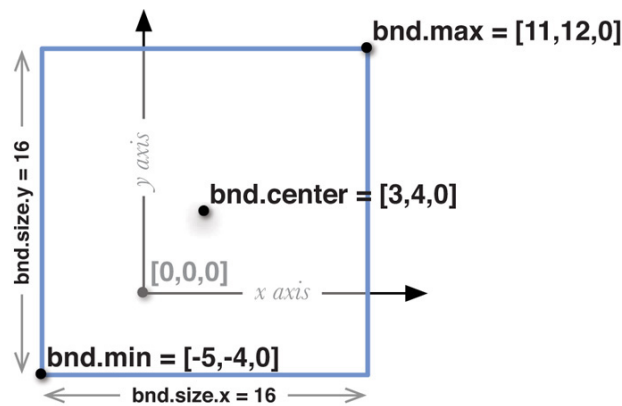


Figure 8: `Bounds bnd = new Bounds(new Vector3(3,4,0), new Vector3(16,16,0));`

Vamos a escribir código que permite obtener un objeto *Bounds* que contenga a dos *Bounds* dados y otro que construya el objeto *Bounds* que contienen a todos los objetos hijos de uno dado. Como este código nos puede servir en otros juegos, lo escribiremos en un script de utilidades al que llamaremos *Utils*. Creamos este script, que ubicamos en la carpeta `__Scripts` y le añadimos el siguiente código, donde usamos el método `Bounds.Encapsulate(Vector3)` para extender un objeto *Bounds* hasta que incluya un *Vector3* dado:

```

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

public class Utils : MonoBehaviour {

//===== Bounds Functions =====\\

    // Creates bounds that encapsulate the two Bounds passed in.
    public static Bounds BoundsUnion( Bounds b0, Bounds b1 ) {
        // If the size of one of the bounds is Vector3.zero, ignore that one

```

```

        if ( b0.size == Vector3.zero && b1.size != Vector3.zero ) {           // 1
            return( b1 );
        } else if ( b0.size != Vector3.zero && b1.size == Vector3.zero ) {
            return( b0 );
        } else if ( b0.size == Vector3.zero && b1.size == Vector3.zero ) {
            return( b0 );
        }
        // Stretch b0 to include the b1.min and b1.max
        b0.Encapsulate(b1.min);           // 2
        b0.Encapsulate(b1.max);
        return( b0 );
    }
}

```

A continuación añadimos el código que calcula el recuadro delimitador de un objeto dado combinando recursivamente los recuadros delimitadores de sus objetos hijo:

```

public class Utils : MonoBehaviour {

    //===== Bounds Functions =====\

    // Creates bounds that encapsulate of the two Bounds passed in.
    public static Bounds BoundsUnion( Bounds b0, Bounds b1 ) {
        ...
    }

    public static Bounds CombineBoundsOfChildren(GameObject go) {
        // Create an empty Bounds b
        Bounds b = new Bounds(Vector3.zero, Vector3.zero);
        // If this GameObject has a Renderer Component...
        if (go.GetComponent<Renderer>() != null) {
            // Expand b to contain the Renderer's Bounds
            b = BoundsUnion(b, go.GetComponent<Renderer>().bounds);
        }
        // If this GameObject has a Collider Component...
        if (go.GetComponent<Collider>() != null) {
            // Expand b to contain the Collider's Bounds
            b = BoundsUnion(b, go.GetComponent<Collider>().bounds);
        }
        // Recursively iterate through each child of this gameObject.transform
        foreach( Transform t in go.transform ) {
            // Expand b to contain their Bounds as well
            b = BoundsUnion( b, CombineBoundsOfChildren( t.gameObject ) );
        }

        return( b );
    }
}

```

Ahora podemos ir a la clase `__Hero` calcular el recuadro delimitador de la nave:

```

public bool _____;

```



```

public Bounds          bounds;

void Awake() {
    S = this; // Set the Singleton
    bounds = Utils.CombineBoundsOfChildren(this.gameObject);
}

```

como este cálculo es costoso, lo realizamos una sola vez, y luego iremos actualizando sólo la posición del centro, a medida que la nave se desplace.

Los límites de la cámara

Para controlar que la nave no se salga del campo de visión, tenemos que calcular la zona del mundo 3D que se proyecta en la pantalla 2D. Esto es sencillo si la proyección es isométrica, como se puede apreciar en la figura

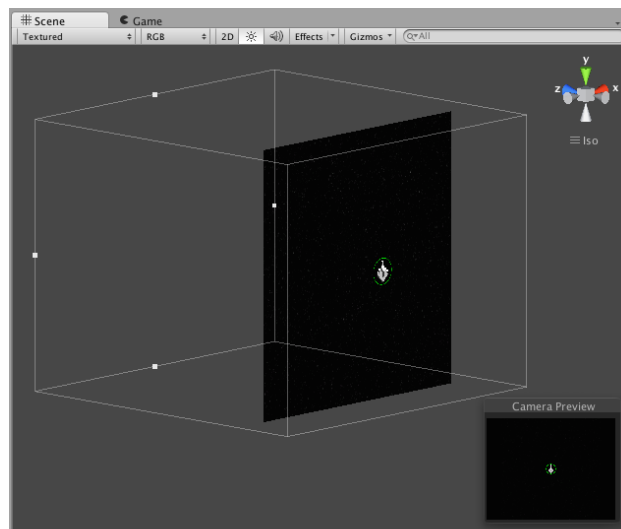


Figure 9: Cámara isométrica

Tomamos las coordenadas del vértice superior izquierdo y el inferior derecho de la pantalla y las transformamos a coordenadas del mundo, *boundTLN* (*top-left near*) y *boundBRF* (*bottom-right far*) y desplazando la coordenada z de *boundTLN* a la coordenada z del plano cercano de la cámara (*Camera.nearClipPlane*) y la coordenada z de *boundBRF* a la coordenada z del plano lejano de la cámara (*Camera.farClipPlane*), como se muestra en el código que debemos añadir al script *Utils*:

```

public class Utils : MonoBehaviour {

//===== Bounds Functions =====\\

    // Creates bounds that encapsulate of the two Bounds passed in.
    public static Bounds BoundsUnion( Bounds b0, Bounds b1 ) {
        ...
    }

    public static Bounds CombineBoundsOfChildren(GameObject go) {
        ...
    }
}

```

```

// Make a static read-only public property camBounds
static public Bounds camBounds {
    get {
        // if _camBounds hasn't been set yet
        if (_camBounds.size == Vector3.zero) {
            // SetCameraBounds using the default Camera
            SetCameraBounds();
        }
        return( _camBounds );
    }
}
// This is the private static field that camBounds uses
static private Bounds _camBounds;

// This function is used by camBounds to set _camBounds and can also be
// called directly.
public static void SetCameraBounds(Camera cam=null) {
    // If no Camera was passed in, use the main Camera
    if (cam == null) cam = Camera.main;
    // This makes a couple of important assumptions about the camera!:
    // 1. The camera is Orthographic
    // 2. The camera is at a rotation of R:[0,0,0]

    // Make Vector3s at the topLeft and bottomRight of the Screen coords
    Vector3 topLeft = new Vector3( 0, 0, 0 );
    Vector3 bottomRight = new Vector3( Screen.width, Screen.height, 0 );

    // Convert these to world coordinates
    Vector3 boundTLN = cam.ScreenToWorldPoint( topLeft );
    Vector3 boundBRF = cam.ScreenToWorldPoint( bottomRight );

    // Adjust their zs to be at the near and far Camera clipping planes
    boundTLN.z += cam.nearClipPlane;
    boundBRF.z += cam.farClipPlane;

    // Find the center of the Bounds
    Vector3 center = (boundTLN + boundBRF)/2f;
    _camBounds = new Bounds( center, Vector3.zero );
    // Expand _camBounds to encapsulate the extents.
    _camBounds.Encapsulate( boundTLN );
    _camBounds.Encapsulate( boundBRF );
}
}

```

Comprobación de los límites

A continuación implementamos un método *BoundsInBoundsCheck()* que determina si un objeto *Bounds* está o no dentro de otro. En realidad implementa tres tipos de comprobaciones distintas, controladas por un enumerado que nos permitirá decidir si queremos que la nave esté siempre dentro de la pantalla, el centro esté dentro de la pantalla, o la nave pueda salir de la pantalla pero no alejarse del borde exterior:

```

using UnityEngine;
using System.Collections;

```

```

using System.Collections.Generic;

// This is actually OUTSIDE of the Utils Class
public enum BoundsTest {
    center,        // Is the center of the GameObject on screen?
    onScreen,       // Are the bounds entirely on screen?
    offScreen       // Are the bounds entirely off screen?
}

public class Utils : MonoBehaviour {
    ...
}

```

Además de ejecutar la comprobación solicitada, *BoundsInBoundsCheck()* devuelve un *Vector3* con el desplazamiento necesario para que la comprobación sea cierta, devolviendo (0, 0, 0) si ya lo era.

```

public class Utils : MonoBehaviour {

//===== Bounds Functions =====\\

    ...

    public static void SetCameraBounds(Camera cam=null) {
        ...
    }

    // Checks to see whether the Bounds bnd are within the camBounds
    public static Vector3 ScreenBoundsCheck(Bounds bnd, BoundsTest test = BoundsTest.center) {
        return( BoundsInBoundsCheck( camBounds, bnd, test ) );
    }

    // Checks to see whether Bounds lilB are within Bounds bigB
    public static Vector3 BoundsInBoundsCheck( Bounds bigB, Bounds lilB,
        BoundsTest test = BoundsTest.onScreen ) {

        // The behavior of this function is different based on the BoundsTest
        // that has been selected.

        // Get the center of lilB
        Vector3 pos = lilB.center;

        // Initialize the offset at [0,0,0]
        Vector3 off = Vector3.zero;

        switch (test) {
            // The center test determines what off (offset) would have to be applied
            // to lilB to move its center back inside bigB
            case BoundsTest.center:
                if ( bigB.Contains( pos ) ) {
                    return( Vector3.zero );
                }

                if (pos.x > bigB.max.x) {
                    off.x = pos.x - bigB.max.x;

```

```

    } else if (pos.x < bigB.min.x) {
        off.x = pos.x - bigB.min.x;
    }
    if (pos.y > bigB.max.y) {
        off.y = pos.y - bigB.max.y;
    } else if (pos.y < bigB.min.y) {
        off.y = pos.y - bigB.min.y;
    }
    if (pos.z > bigB.max.z) {
        off.z = pos.z - bigB.max.z;
    } else if (pos.z < bigB.min.z) {
        off.z = pos.z - bigB.min.z;
    }
    return( off );

// The onScreen test determines what off would have to be applied to
// keep all of lilB inside bigB
case BoundsTest.onScreen:
    if ( bigB.Contains( lilB.min ) && bigB.Contains( lilB.max ) ) {
        return( Vector3.zero );
    }

    if (lilB.max.x > bigB.max.x) {
        off.x = lilB.max.x - bigB.max.x;
    } else if (lilB.min.x < bigB.min.x) {
        off.x = lilB.min.x - bigB.min.x;
    }
    if (lilB.max.y > bigB.max.y) {
        off.y = lilB.max.y - bigB.max.y;
    } else if (lilB.min.y < bigB.min.y) {
        off.y = lilB.min.y - bigB.min.y;
    }
    if (lilB.max.z > bigB.max.z) {
        off.z = lilB.max.z - bigB.max.z;
    } else if (lilB.min.z < bigB.min.z) {
        off.z = lilB.min.z - bigB.min.z;
    }
    return( off );

// The offScreen test determines what off would need to be applied to
// move any tiny part of lilB inside of bigB
case BoundsTest.offScreen:
    bool cMin = bigB.Contains( lilB.min );
    bool cMax = bigB.Contains( lilB.max );
    if ( cMin || cMax ) {
        return( Vector3.zero );
    }

    if (lilB.min.x > bigB.max.x) {
        off.x = lilB.min.x - bigB.max.x;
    } else if (lilB.max.x < bigB.min.x) {
        off.x = lilB.max.x - bigB.min.x;
    }
    if (lilB.min.y > bigB.max.y) {

```

```

        off.y = lilB.min.y - bigB.max.y;
    } else if (lilB.max.y < bigB.min.y) {
        off.y = lilB.max.y - bigB.min.y;
    }
    if (lilB.min.z > bigB.max.z) {
        off.z = lilB.min.z - bigB.max.z;
    } else if (lilB.max.z < bigB.min.z) {
        off.z = lilB.max.z - bigB.min.z;
    }
    return( off );
}

return( Vector3.zero );
}
}

```

Por último modificamos la clase *Hero* para volver a meter la nave en la pantalla cada vez que se intente salir:

```

public class Hero : MonoBehaviour {
    ...

    void Update () {
        ...
        transform.position = pos;

        bounds.center = transform.position;

        // Keep the ship constrained to the screen bounds
        Vector3 off = Utils.ScreenBoundsCheck(bounds, BoundsTest.onScreen);
        if ( off != Vector3.zero ) {
            pos -= off;
            transform.position = pos;
        }

        // Rotate the ship to make it feel more dynamic
        transform.rotation = Quaternion.Euler(yAxis*pitchMult,xAxis*rollMult,0);
    }
}

```

Puedes comprobar cuál es el efecto de probar los otros valores del enumerado *BoundsTest*.

Los enemigos

El arte

Los enemigos que vamos a construir son los que se muestran en la figura 10

Construye los modelos de los enemigos creando jerarquías de objetos del tipo y atributos que se muestran en la figura 11

Añade un componente *Rigidbody* a cada uno de los modelos de los enemigos. Configura cada *Rigidbody* con: *Use Gravity* a false; *isKinematic* a cierto; y marca el eje z de *Freeze Position* y los 3 ejes x,y,z de



Figure 10: Tipos de enemigos

| | | | |
|-------------------|--------------|-------------|-----------------|
| Enemy_0 (Empty) | P:[0,0,0] | R:[0,0,0] | S:[1,1,1] |
| Cockpit (Sphere) | P:[0,0,0] | R:[0,0,0] | S:[2,2,1] |
| Wing (Sphere) | P:[0,0,0] | R:[0,0,0] | S:[5,5,0.5] |
| Enemy_1 (Empty) | P:[0,0,0] | R:[0,0,0] | S:[1,1,1] |
| Cockpit (Sphere) | P:[0,0,0] | R:[0,0,0] | S:[2,2,1] |
| Wing (Sphere) | P:[0,0,0] | R:[0,0,0] | S:[6,4,0.5] |
| Enemy_2 (Empty) | P:[0,0,0] | R:[0,0,0] | S:[1,1,1] |
| Cockpit (Sphere) | P:[-1.5,0,0] | R:[0,0,0] | S:[1,3,1] |
| Sphere | P:[2,0,0] | R:[0,0,0] | S:[2,2,1] |
| Wing (Sphere) | P:[0,0,0] | R:[0,0,0] | S:[6,4,0.5] |
| Enemy_3 (Empty) | P:[0,0,0] | R:[0,0,0] | S:[1,1,1] |
| CockpitL (Sphere) | P:[-1,0,0] | R:[0,0,0] | S:[1,3,1] |
| CockpitR (Sphere) | P:[1,0,0] | R:[0,0,0] | S:[1,3,1] |
| Wing (Sphere) | P:[0,0.5,0] | R:[0,0,0] | S:[5,1,0.5] |
| Enemy_4 (Empty) | P:[0,0,0] | R:[0,0,0] | S:[1,1,1] |
| Cockpit (Sphere) | P:[0,1,0] | R:[0,0,0] | S:[1.5,1.5,1.5] |
| Fuselage (Sphere) | P:[0,1,0] | R:[0,0,0] | S:[2,4,1] |
| Wing_L (Sphere) | P:[-1.5,0,0] | R:[0,0,-30] | S:[5,1,0.5] |
| Wing_R (Sphere) | P:[1.5,0,0] | R:[0,0,30] | S:[5,1,0.5] |

Figure 11: Configuración de los enemigos

Freeze Rotation. Es importante que los enemigos tengan *Rigidbody* porque de lo contrario sus *colliders* no se desplazarían con ellos.

Por último, convierte los 5 enemigos en prefabs y borra todos los enemigos de la jerarquía excepto *Enemy_0*.

Control de los enemigos

Creamos un script *Enemy* que asociamos a *Enemy_0*. Este script se encarga de desplazar cada nave enemiga verticalmente hacia abajo y destruirla una vez que desaparezca por la parte inferior de la pantalla. Esta clase servirá de clase padre para las clases que asociemos con otros tipos de enemigos y por eso es necesario incluir un mecanismo más sofisticado para detectar si el enemigo está en la pantalla. En particular, el centro del cuadro delimitador de la nave puede que no coincida con la posición de la nave (*transform.position*), por lo que si calculamos la distancia inicial entre ambos puntos, y esa distancia es la que guardamos en *boundsCenterOffset*, cada vez que movamos la posición podremos recalculer fácilmente el nuevo centro. Además, las instancias de *Enemy_4* se pueden destruir parte por parte, con lo que el centro de la nave cambiará dependiendo de las piezas que le queden y habrá que recalculer el *boundsCenterOffset*.

Para comprobar si el atributo *bounds* ha sido o no inicializado lo tenemos que comparar con su valor por

defecto `center:[0,0,0]` `size:[0,0,0]` ya que *Bounds* es un tipo estático y no se puede inicializar a *null* como una referencia.

Para comprobar si el enemigo ha salido de la pantalla utilizamos *InvokeRepeating* que ejecuta *CheckOffScreen* cada 2 segundos, ya que esto es más eficiente que hacerlo en cada frame y no es necesario ser tan preciso en la destrucción de los enemigos que salen de la pantalla. Además, como las naves pueden cambiar de forma al destruirse alguna de sus partes, es necesario recalcular cada vez su recuadro delimitador.

```
using UnityEngine;                                // Required for Unity
using System.Collections;                         // Required for Arrays & other Collections

public class Enemy : MonoBehaviour {
    public float    speed = 10f;    // The speed in m/s
    public float    fireRate = 0.3f; // Seconds/shot (Unused)
    public float    health = 10;
    public int      score = 100;    // Points earned for destroying this

    public bool     _____;

    public Bounds    bounds; // The Bounds of this and its children
    public Vector3   boundsCenterOffset; // Dist of bounds.center from position

    void Awake() {
        InvokeRepeating( "CheckOffscreen", 0f, 2f );
    }

    // Update is called once per frame
    void Update() {
        Move();
    }

    public virtual void Move() {
        Vector3 tempPos = pos;
        tempPos.y -= speed * Time.deltaTime;
        pos = tempPos;
    }

    // This is a Property: A method that acts like a field
    public Vector3 pos {
        get {
            return( this.transform.position );
        }
        set {
            this.transform.position = value;
        }
    }

    void CheckOffscreen() {
        // If bounds are still their default value...
        if (bounds.size == Vector3.zero) {
            // then set them
            bounds = Utils.CombineBoundsOfChildren(this.gameObject);
            // Also find the diff between bounds.center & transform.position
            boundsCenterOffset = bounds.center - transform.position;
        }
    }
}
```

```

    }

    // Every time, update the bounds to the current position
    bounds.center = transform.position + boundsCenterOffset;
    // Check to see whether the bounds are completely offscreen
    Vector3 off = Utils.ScreenBoundsCheck( bounds, BoundsTest.offScreen );
    if ( off != Vector3.zero ) {
        // If this enemy has gone off the bottom edge of the screen
        if (off.y < 0) {
            // then destroy it
            Destroy( this.gameObject );
        }
    }
}
}
}

```

Despliegue de enemigos aleatoriamente

Para instanciar a los enemigos, creamos un nuevo script *Main* que asignamos a *__MainCamera*. Para las invocaciones consecutivas utilizamos *Invoke()* que sólo hace una llamada una vez transcurrido el tiempo especificado, de forma que al final de cada invocación programamos la siguiente y podemos variar el intervalo de tiempo entre llamadas:

```

using UnityEngine;                                // Required for Unity
using System.Collections;                          // Required for Arrays & other Collections
using System.Collections.Generic;                  // Required to use Lists or Dictionaries

public class Main : MonoBehaviour {
    static public Main S;

    public GameObject[]    prefabEnemies;
    public float           enemySpawnPerSecond = 0.5f; // # Enemies/second
    public float           enemySpawnPadding = 1.5f; // Padding for position

    public bool _____;

    public float           enemySpawnRate; // Delay between Enemy spawns

    void Awake() {
        S = this;
        // Set Utils.camBounds
        Utils.SetCameraBounds(this.GetComponent<Camera>());
        // 0.5 enemies/second = enemySpawnRate of 2
        enemySpawnRate = 1f/enemySpawnPerSecond;
        // Invoke call SpawnEnemy() once after a 2 second delay
        Invoke( "SpawnEnemy", enemySpawnRate );
    }

    public void SpawnEnemy() {
        // Pick a random Enemy prefab to instantiate
        int ndx = Random.Range(0, prefabEnemies.Length);
        GameObject go = Instantiate( prefabEnemies[ ndx ] ) as GameObject;
    }
}

```



```

    // Position the Enemy above the screen with a random x position
    Vector3 pos = Vector3.zero;
    float xMin = Utils.camBounds.min.x+enemySpawnPadding;
    float xMax = Utils.camBounds.max.x-enemySpawnPadding;
    pos.x = Random.Range( xMin, xMax );
    pos.y = Utils.camBounds.max.y + enemySpawnPadding;
    go.transform.position = pos;
    // Call SpawnEnemy() again in a couple of seconds
    Invoke( "SpawnEnemy", enemySpawnRate );
}
}

```

A continuación borramos *Enemy_0* de la jerarquía (ya era un prefab) y en el componente *Main* de *__Main-Camera* asignamos el valor 1 al atributo *Size* de *prefabEnemies* y en *Element 0* asignamos el prefab *Enemy_0* (en la figura 12 se muestra la configuración tras añadir todos los tipos de enemigos). Ahora generamos un enemigo de tipo *Enemy_0* cada 2 segundos, aunque si choca con la nave del jugador no ocurre nada.

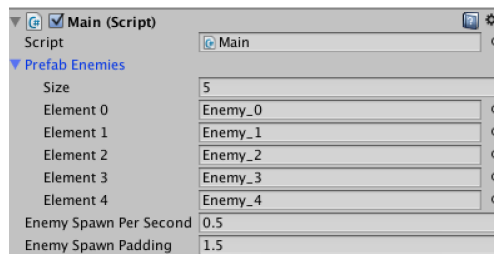


Figure 12: Configuración de los enemigos

Configuración de etiquetas, capas y física

Según la especificación del juego, las colisiones deberían operar así:

- la nave del jugador debe colisionar con los enemigos, los proyectiles de los enemigos y los power-ups pero no con sus propios proyectiles
- los proyectiles de la nave del jugador sólo deben colisionar con los enemigos
- los enemigos deben colisionar con la nave del jugador y sus proyectiles pero no con los power-ups
- los proyectiles de los enemigos sólo deben colisionar con la nave del jugador
- los power-ups sólo deben colisionar con la nave del jugador

Para conseguirlo seleccionamos la orden *Edit > Project Settings > Tags and Layers* y definimos las etiquetas y capas que se muestran en la figura 13

A continuación abre el *PhysicsManager* seleccionado la orden *Edit > Project Settings > Physics* y establece la configuración que corresponde a la especificación de colisiones anterior, según se muestra en la figura ??

Por último, a la nave del jugador *__Hero* le asignamos la capa *Hero* y la etiqueta *Hero* y a cada uno de los prefabs de los enemigos les asignamos la capa *Enemy* y la etiqueta *Enemy*.

Los enemigos destruyen la nave del jugador cuando chocan con ella

Ahora vamos a hacer que la nave del jugador reaccione a las colisiones con los enemigos. Seleccionamos el objeto *Shield* (hijo de *__Hero*) y en su componente *Sphere Collider* marcamos el atributo *Is Trigger* (no queremos que los enemigos reboten con el escudo, sólo queremos ser notificados de ello)

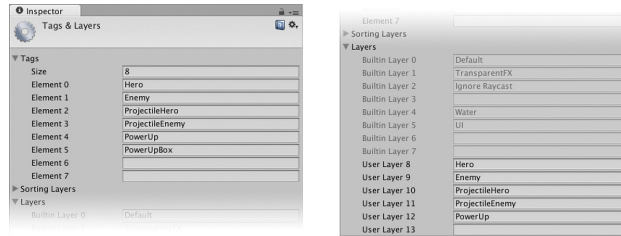


Figure 13: Configuración de etiquetas y capas

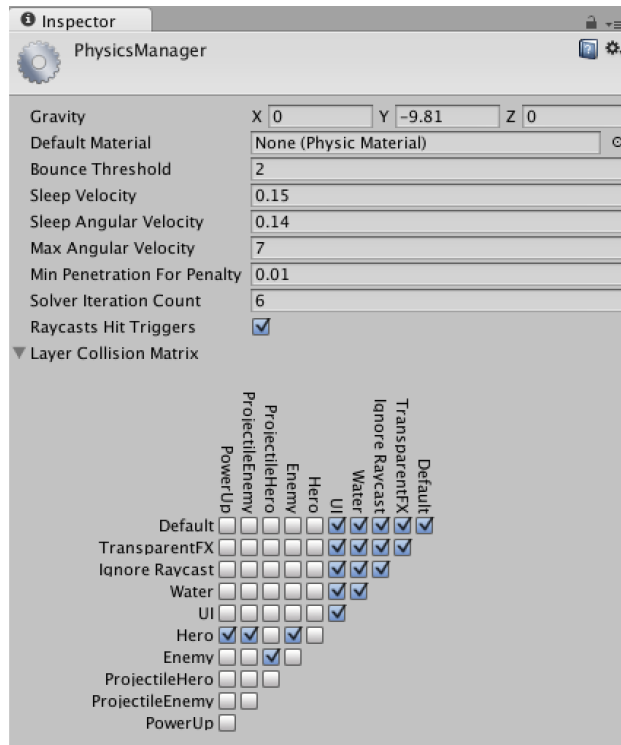


Figure 14: Configuración del PhysicsManager

Añadimos a *Hero* el método *OnTriggerEnter* para ver cuál es el efecto de los choques:

```
public class Hero : MonoBehaviour {
    ...
    void Update() {
        ...
    }

    void OnTriggerEnter(Collider other) {
        print("Triggered: "+other.gameObject.name);
    }
}
```

Observamos que las colisiones son con las partes de los enemigos (*Wing* y *Cockpit*) que son las que tienen collider, pero no así con el objeto padre. Como queremos identificar el tipo de objeto con el que hemos chocado y para eso utilizamos las etiquetas, lo que necesitamos es una forma de obtener la etiqueta del objeto padre de uno dado, y para ello añadimos el siguiente método a la clase *Utils*:

```

public class Utils : MonoBehaviour {

//===== Bounds Functions =====\\

    ...

    // Checks to see whether Bounds lilB are within Bounds bigB
    public static Vector3 BoundsInBoundsCheck( Bounds bigB, Bounds lilB,
        BoundsTest test = BoundsTest.onScreen ) {
        ...
    }

//===== Transform Functions =====\\

    // This function will iteratively climb up the transform.parent tree
    // until it either finds a parent with a tag != "Untagged" or no parent
    public static GameObject FindTaggedParent(GameObject go) {
        // If this gameObject has a tag
        if (go.tag != "Untagged") {
            // then return this gameObject
            return(go);
        }
        // If there is no parent of this Transform
        if (go.transform.parent == null) {
            // We've reached the top of the hierarchy with no interesting tag
            // So return null
            return( null );
        }
        // Otherwise, recursively climb up the tree
        return( FindTaggedParent( go.transform.parent.gameObject ) );
    }
    // This version of the function handles things if a Transform is passed in
    public static GameObject FindTaggedParent(Transform t) {
        return( FindTaggedParent( t.gameObject ) );
    }
}

```

Si ahora en *Hero* escribimos la etiqueta del objeto con el que hemos chocado:

```

public class Hero : MonoBehaviour {
    ...
    void Update() {
        ...
    }

    void OnTriggerEnter(Collider other) {
        // Find the tag of other.gameObject or its parent GameObjects
        GameObject go = Utils.FindTaggedParent(other.gameObject);
        // If there is a parent with a tag
        if (go != null) {
            // Announce it
            print("Triggered: "+go.name);
        }
    }
}

```

```

    } else {
        // Otherwise announce the original other.gameObject
        print("Triggered: "+other.gameObject.name);
    }
}
}

```

veremos que chocamos con *Enemy_0(Clone)*, una instancia de *Enemy_0*.

Una vez que hemos comprobado que las colisiones funcionan como nos interesa, actualizamos el código de *Hero* para que cuando choquemos con un enemigo lo destruyamos y le restemos 1 al nivel del escudo.

```

public class Hero : MonoBehaviour {
    ...
    void Update() {
        ...
    }

    // This variable holds a reference to the last triggering GameObject
    public GameObject lastTriggerGo = null;

    void OnTriggerEnter(Collider other) {
        ...

        if (go != null) {
            // Make sure it's not the same triggering go as last time
            if (go == lastTriggerGo) {
                return;
            }
            lastTriggerGo = go;

            if (go.tag == "Enemy") {
                // If the shield was triggered by an enemy
                // Decrease the level of the shield by 1
                shieldLevel--;
                // Destroy the enemy
                Destroy(go);
            } else {
                print("Triggered: "+go.name);
            }
        } else {
            ...
        }
    }
}

```

Además debemos comprobar que no procesamos dos veces las colisiones con distintas partes del mismo enemigo, para lo cual comprobamos que el objeto con el que hemos chocado es distinto de aquel con el que chocamos la última vez que procesamos un evento de colisión en este frame.

El escudo no funciona correctamente

Si ejecutamos el juego veremos que después de tener el escudo a nivel 0, pasamos al máximo nivel y seguimos decreciendo indefinidamente. Si observamos el estado de *_Hero* en el inspector mientras ejecutamos el juego veremos que *shieldLevel* sigue disminuyendo con cada colisión, tomando valores negativos, que el script *Shield* traduce a desplazamientos negativos sobre la textura del escudo en el método *Update* de *Shield*:

```

void Update () {
    // Read the current shield level from the Hero Singleton
    int currLevel = Mathf.FloorToInt( Hero.S.shieldLevel );
    // If this is different from levelShown...
    if (levelShown != currLevel) {
        levelShown = currLevel;
        Material mat = this.GetComponent<Renderer>().material;
        // Adjust the texture offset to show different shield level
        mat.mainTextureOffset = new Vector2( 0.2f*levelShown, 0 );
    }
    // Rotate the shield a bit every second
    float rZ = (rotationsPerSecond * Time.time * 360) % 360f;
    transform.rotation = Quaternion.Euler( 0, 0, rZ );
}

```

que están recorriendo la textura del escudo de forma cíclica.

Para corregir esto, vamos a convertir *shieldLevel* en una propiedad que encapsula el atributo privado `__shieldLevel` impidiendo que valga más de 4 y destruyendo la nave si baja de 0. Los atributos privados no se muestran en el inspector de Unity, si aún así queremos que se visualicen podemos marcarlos con la directiva `[SerializeField]`.

Añadimos el nuevo atributo `__shieldLevel` que sustituye al anterior *shieldLevel* en *Hero*:

```

// Ship status information
[SerializeField]
private float __shieldLevel = 1; // Add the underscore!

```

Y añadimos la propiedad que recubre al atributo privado:

```

public class Hero : MonoBehaviour {

    ...

    void OnTriggerEnter(Collider other) {
        ...
    }

    public float shieldLevel {
        get {
            return( __shieldLevel );
        }
        set {
            __shieldLevel = Mathf.Min( value, 4 );
            // If the shield is going to be set to less than zero
            if (value < 0) {
                Destroy(this.gameObject);
            }
        }
    }
}

```

Reinicio del juego cuando muere la nave del jugador

Finalmente añadimos código para que el juego se reinicie cada vez que sea destruida la nave del jugador, una vez transcurridos dos segundos.

En la clase de *Hero* añadimos un atributo con el tiempo que se debe esperar antes de reiniciar:

```
public float          gameRestartDelay = 2f;
```

y también en *Hero* en el *set* de la propiedad *shieldLevel* añadimos la llamada al método estático *DelayedRestart* de la clase *Main*:

```
if (value < 0) {
    Destroy(this.gameObject);
    // Tell Main.S to restart the game after a delay
    Main.S.DelayedRestart( gameRestartDelay );
}
```

cuya implementación añadimos a la clase *Main*:

```
public class Main : MonoBehaviour {
    ...

    public void SpawnEnemy() {
        ...
    }

    public void DelayedRestart( float delay ) {
        // Invoke the Restart() method in delay seconds
        Invoke("Restart", delay);
    }

    public void Restart() {
        // Reload _Scene_0 to restart the game
        Application.LoadLevel("_Scene_0");
    }
}
```

A disparar

Arte

Crea un objeto vacío de nombre *Weapon* y dale la estructura de la figura 15

| | | | |
|----------------|-------------|-----------|-------------------|
| Weapon (Empty) | P:[0,0,0] | R:[0,0,0] | S:[1,1,1] |
| Barrel (Cube) | P:[0,0.5,0] | R:[0,0,0] | S:[0.25,1,0.1] |
| Collar (Cube) | P:[0,1,0] | R:[0,0,0] | S:[0.375,0.5,0.2] |

Figure 15: configuración del arma

Elimina los componentes *Collider* de los dos objetos que forman parte de *Weapon*, *Barrel* y *Collar*. Crea un nuevo material de nombre *Mat Collar* y asígnalo al objeto *Collar*. En el inspector de *Collar* en el desplegable

Shader del material selecciona *Custom > UnlitAlpha*. Crea un nuevo script de nombre *Weapon* y asígnalo al objeto *Weapon*. A continuación convierte el objeto *Weapon* en un prefab, arrastrándolo a la carpeta *__Prefabs*. Coloca la instancia de *Weapon* como un hijo de *__Hero* en la jerarquía y ubícalo en la posición $[0,2,0]$. El resultado debería ser el que se muestra en la figura 16

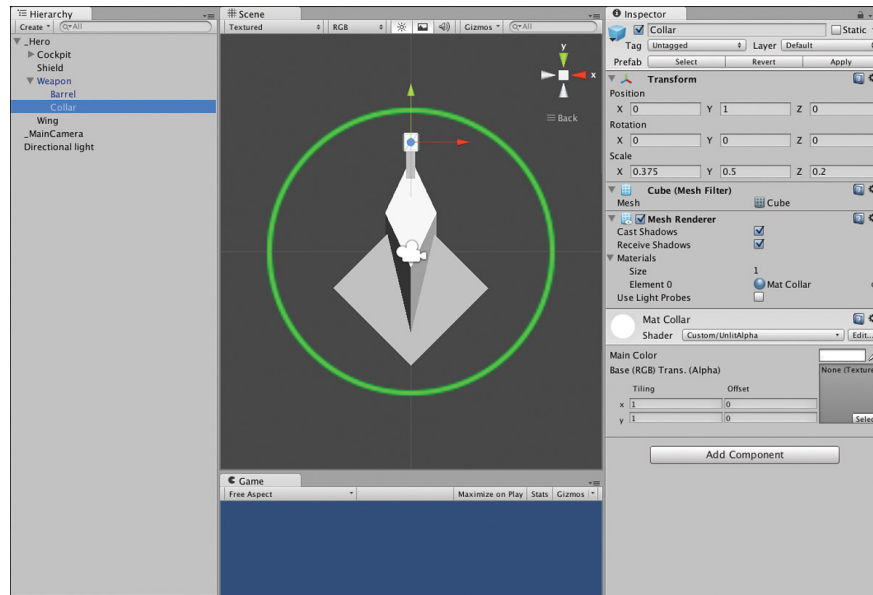


Figure 16: La nave con cañón

A continuación, creamos los proyectiles. Crea un cubo de nombre *ProjectileHero* ubicado en: P: $[10,0,0]$ R: $[0,0,0]$ S: $[0.25,1,0.25]$. Asígnale la etiqueta *ProjectileHero* y la capa *ProjectileHero*. Crea un nuevo material *Mat Projectile*, con el shader *Custom > UnlitAlpha* y asígnalo a *ProjectileHero*. Añádele un componente *Rigidbody* con la configuración que se muestra en la figura 17: Use *Gravity* a falso, y seleccionados *Freeze Position* z, y *Freeze Rotation* x, y, z. En el componente *Box Collider* dale un valor de 10 al atributo *Size.z* para asegurarnos que acertamos a los objetos aunque estén un poco separados del plano $z=0$. Crea un script de nombre *Projectile* y asígnalo a *ProjectileHero*. Por último, convierte el proyectil en un prefab y borra la instancia de la jerarquía.

Definición de los tipos de armas

La configuración de las armas en un juego es un elemento fundamental para ajustar la jugabilidad y es responsabilidad de los diseñadores. Por ello, lo haremos de forma que se puedan configurar desde el inspector. Esto lo haremos definiendo una clase *WeaponDefinition* marcada como *System.Serializable* lo que permite definir instancias suyas en el inspector.

En el script *Weapon* añadimos el siguiente código:

```
using UnityEngine;
using System.Collections;

// This is an enum of the various possible weapon types
// It also includes a "shield" type to allow a shield power-up
public enum WeaponType {
    none,           // The default / no weapon
    blaster,        // A simple blaster
    spread,         // Two shots simultaneously
```

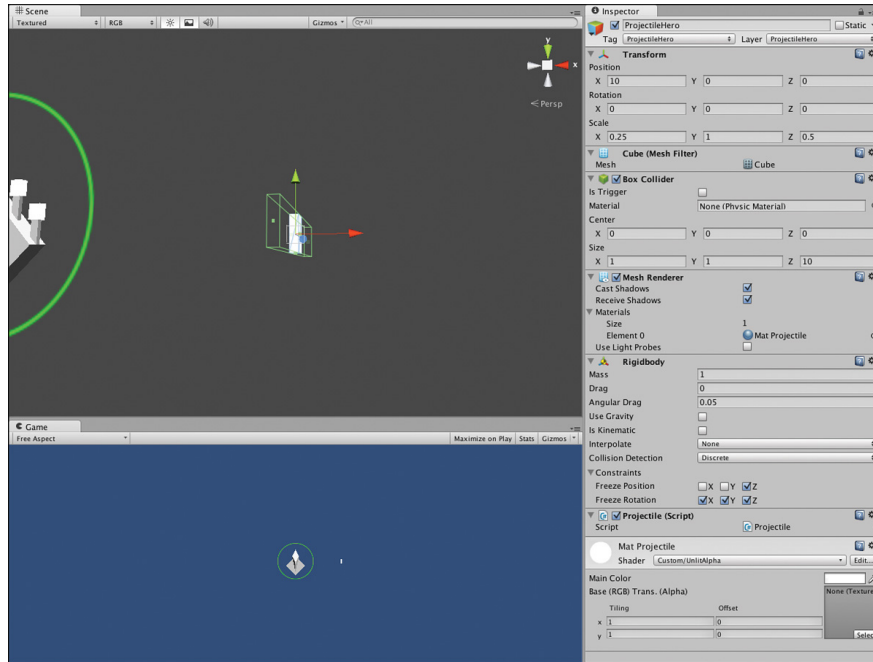


Figure 17: Proyectoil con un collider estirado en el eje z

```

phaser,      // Shots that move in waves
missile,     // Homing missiles
laser,       // Damage over time
shield       // Raise shieldLevel
}

// The WeaponDefinition class allows you to set the properties
// of a specific weapon in the Inspector. Main has an array
// of WeaponDefinitions that makes this possible.
// [System.Serializable] tells Unity to try to view WeaponDefinition
// in the Inspector pane. It doesn't work for everything, but it
// will work for simple classes like this!
[System.Serializable]
public class WeaponDefinition {
    public WeaponType type = WeaponType.none;
    public string letter; // The letter to show on the power-up
    public Color color = Color.white; // Color of Collar & power-up
    public GameObject projectilePrefab; // Prefab for projectiles
    public Color projectileColor = Color.white;
    public float damageOnHit = 0; // Amount of damage caused
    public float continuousDamage = 0; // Damage per second (Laser)
    public float delayBetweenShots = 0;
    public float velocity = 20; // Speed of projectiles
}

// Note: Weapon prefabs, colors, and so on. are set in the class Main.

public class Weapon : MonoBehaviour {
    // The Weapon class will be filled in later.
}

```


A continuación añadimos a la clase *Main* un array de *WeaponDefinition* que configuraremos en el inspector y otro con los tipos de armas que hay en él, inicializando el segundo a partir del primero:

```
public class Main : MonoBehaviour {
    ...
    public float          enemySpawnPadding = 1.5f; // Padding for position
    public WeaponDefinition[] weaponDefinitions;

    public bool _____;

    public WeaponType[]    activeWeaponTypes;
    public float          enemySpawnRate; // Delay between Enemies

    void Awake() {...}

    void Start() {
        activeWeaponTypes = new WeaponType[weaponDefinitions.Length];
        for ( int i=0; i<weaponDefinitions.Length; i++ ) {
            activeWeaponTypes[i] = weaponDefinitions[i].type;
        }
    }
    ...
}
```

Si ahora seleccionamos *__MainCamera* en la jerarquía veremos el array *weaponDefinitions* dentro del componente *Main (Script)*. Le asignamos un tamaño de 3 e introducimos las definiciones de armas que se muestran en la figura 18. No es importante que los colores coincidan exactamente pero sí que el valor de alpha sea totalmente opaco, que se indica por una barra blanca, y no negra, debajo del color en el inspector.

A continuación añadimos el código que carga las definiciones de las armas en una tabla hash a la que se accede por el tipo de arma como clave. En el diccionario *W_DEFS* almacenaremos la información sobre los tipos de armas que acabamos de escribir en el inspector. Hemos usado esta inicialización en dos pasos porque los datos de tipos *Dictionary* no aparecen en el inspector. El diccionario se inicializa en el método *Awake* y se accede a su contenido a través del método estático *GetWeaponDefinition()*:

```
public class Main : MonoBehaviour {
    static public Main S;
    static public Dictionary<WeaponType, WeaponDefinition> W_DEFS;
    ...
    void Awake() {
        ...
        Invoke( "SpawnEnemy", enemySpawnRate );

        // A generic Dictionary with WeaponType as the key
        W_DEFS = new Dictionary<WeaponType, WeaponDefinition>();
        foreach( WeaponDefinition def in weaponDefinitions ) {
            W_DEFS[def.type] = def;
        }
    }

    static public WeaponDefinition GetWeaponDefinition( WeaponType wt ) {
        // Check to make sure that the key exists in the Dictionary
        // Attempting to retrieve a key that didn't exist, would throw an error,
        // so the following if statement is important.
    }
}
```

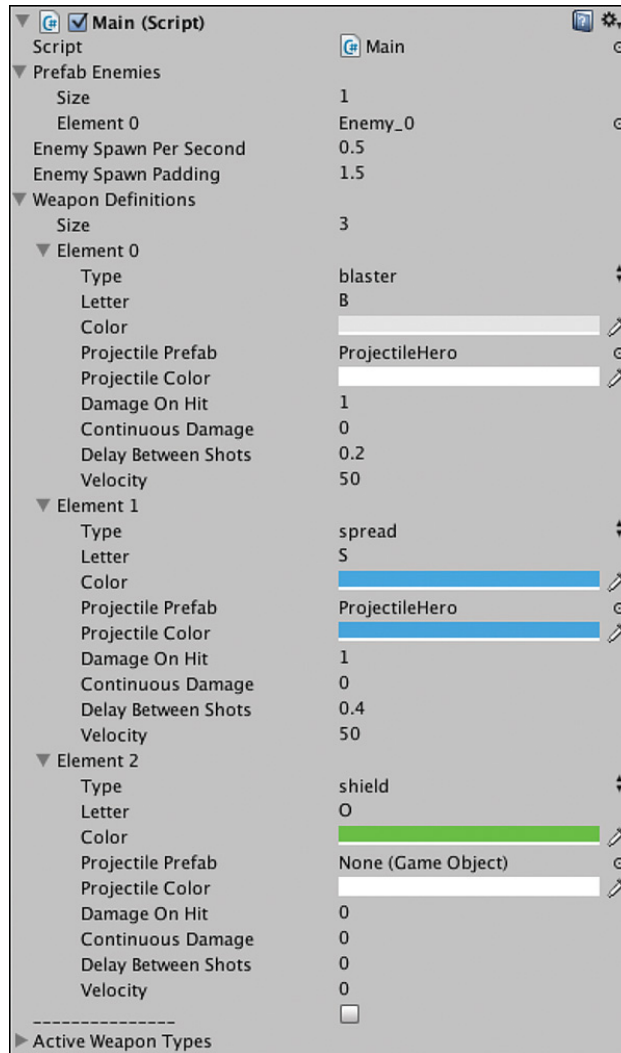


Figure 18: Definición de las armas de tipo blaster y spread, y del escudo

```

if (W_DEFS.ContainsKey(wt)) {
    return( W_DEFS[wt]);
}
// This will return a definition for WeaponType.none,
// which means it has failed to find the WeaponDefinition
return( new WeaponDefinition() );
}

void Start() {...}
}

```

Ahora empezamos a usar el mecanismo de configuración de los tipos de armas, añadiendo código a la clase *Projectile* que cambiará el color del proyectil de acuerdo con su tipo según el color que se haya establecido en *WeaponDefinitions*:

```

using UnityEngine;
using System.Collections;

```

```

public class Projectile : MonoBehaviour {
    [SerializeField]
    private WeaponType    _type;

    // This public property masks the field _type & takes action when it is set
    public WeaponType    type {
        get {
            return( _type );
        }
        set {
            SetType( value );
        }
    }

    void Awake() {
        // Test to see whether this has passed off screen every 2 seconds
        InvokeRepeating( "CheckOffscreen", 2f, 2f );
    }

    public void SetType( WeaponType eType ) {
        // Set the _type
        _type = eType;
        WeaponDefinition def = Main.GetWeaponDefinition( _type );
        renderer.material.color = def.projectileColor;
    }

    void CheckOffscreen() {
        if ( Utils.ScreenBoundsCheck( GetComponent<Collider>().bounds, BoundsTest.offScreen ) != Vector3.zero )
            Destroy( this.gameObject );
    }
}

```

Disparo con funciones delegadas

A continuación crearemos el código que instancia los proyectiles y que está en la clase *Weapon*. Recordemos que una nave tiene un array de objetos *Weapon* que pueden estar activos o no y cada uno de los cuales puede ser de uno u otro tipo. De esta forma cuando pulsamos la tecla de disparo (la barra espaciadora) tendremos que disparar una o más armas con proyectiles de uno entre varios tipos posibles. Para conseguirlo usaremos el mecanismo de C# de las funciones delegadas, que permite asignar varias funciones a la misma variable y así hacer una invocación múltiple con una sola llamada.

```

using UnityEngine;
using System.Collections;

public class DelegateExample : MonoBehaviour {
    // Create a delegate definition named FloatOperationDelegate
    // This defines the parameter and return types for target functions
    public delegate float FloatOperationDelegate( float f0, float f1 );

    // FloatAdd must have the same parameter and return types as

```

```

// FloatOperationDelegate
public float FloatAdd( float f0, float f1 ) {
    float result = f0+f1;
    print("The sum of "+f0+" & "+f1+" is "+result+".");
    return( result );
}

// FloatMultiply must have the same parameter and return types as well
public float FloatMultiply( float f0, float f1 ) {
    float result = f0 * f1;
    print("The product of "+f0+" & "+f1+" is "+result+".");
    return( result );
}

// Declare a field "fod" of the type FloatOperationDelegate
public FloatOperationDelegate fod; // A delegate field

void Awake() {
    // Assign the method FloatAdd() to fod
    fod = FloatAdd;

    // Add the method FloatMultiply(), now BOTH are called by fod
    fod += FloatMultiply;

    // Check to see whether fod is null before calling
    if (fod != null) {
        // Call fod(3,4); it calls FloatAdd(3,4) & then FloatMultiply(3,4)
        float result = fod( 3, 4 );
        // Prints: The sum of 3 & 4 is 7.
        // then Prints: The product of 3 & 4 is 12.

        print( result );
        // Prints: 12
        // This result is 12 because the last target method to be called
        // is the one that returns a value via the delegate.
    }
}

```

Con este mecanismo, cada arma que se activa añade su propio método *Fire* al delegado *fireDelegate* que será quien se invoque con cada pulsación de la barra espaciadora.

Añadimos este código a la clase *Hero*:

```

public class Hero : MonoBehaviour {
    ...
    public Bounds          bounds;

    // Declare a new delegate type WeaponFireDelegate
    public delegate void WeaponFireDelegate();
    // Create a WeaponFireDelegate field named fireDelegate.
    public WeaponFireDelegate fireDelegate;

    void Awake() {
        ...
    }
}

```

```

}

void Update () {
    ...
    // Rotate the ship to make it feel more dynamic
    transform.rotation = Quaternion.Euler(yAxis*pitchMult,xAxis*rollMult,0);

    // Use the fireDelegate to fire Weapons
    // First, make sure the Axis("Jump") button is pressed
    // Then ensure that fireDelegate isn't null to avoid an error
    if (Input.GetAxis("Jump") == 1 && fireDelegate != null) {           // 1
        fireDelegate();
    }
}
    ...
}

```

Ahora añadimos este código en la clase Weapon:

```

public class Weapon : MonoBehaviour {
    static public Transform    PROJECTILE_ANCHOR;

    public bool _____;
    [SerializeField]
    private WeaponType        _type = WeaponType.none;
    public WeaponDefinition    def;
    public GameObject          collar;
    public float               lastShot; // Time last shot was fired

    void Start() {
        collar = transform.Find("Collar").gameObject;
        // Call SetType() properly for the default _type
        SetType( _type );

        if (PROJECTILE_ANCHOR == null) {
            GameObject go = new GameObject("_Projectile_Anchor");
            PROJECTILE_ANCHOR = go.transform;
        }
        // Find the fireDelegate of the parent
        GameObject parentGO = transform.parent.gameObject;
        if (parentGO.tag == "Hero") {
            Hero.S.fireDelegate += Fire;
        }
    }

    public WeaponType type {
        get {    return( _type );    }
        set {    SetType( value );    }
    }

    public void SetType( WeaponType wt ) {
        _type = wt;
        if (type == WeaponType.none) {
            this.gameObject.SetActive(false);
        }
    }
}

```

```

        return;
    } else {
        this.gameObject.SetActive(true);
    }
    def = Main.GetWeaponDefinition(_type);
    collar.GetComponent<Renderer>().material.color = def.color;
    lastShot = 0; // You can always fire immediately after _type is set.
}

public void Fire() {
    // If this.gameObject is inactive, return
    if (!gameObject.activeInHierarchy) return;
    // If it hasn't been enough time between shots, return
    if (Time.time - lastShot < def.delayBetweenShots) {
        return;
    }
    Projectile p;
    switch (type) {
    case WeaponType.blaster:
        p = MakeProjectile();
        p.GetComponent<Rigidbody>().velocity = Vector3.up * def.velocity;
        break;

    case WeaponType.spread:
        p = MakeProjectile();
        p.GetComponent<Rigidbody>().velocity = Vector3.up * def.velocity;
        p = MakeProjectile();
        p.GetComponent<Rigidbody>().velocity = new Vector3( -.2f, 0.9f, 0 ) * def.velocity;
        p = MakeProjectile();
        p.GetComponent<Rigidbody>().velocity = new Vector3( .2f, 0.9f, 0 ) * def.velocity;
        break;

    }
}

public Projectile MakeProjectile() {
    GameObject go = Instantiate( def.projectilePrefab ) as GameObject;
    if ( transform.parent.gameObject.tag == "Hero" ) {
        go.tag = "ProjectileHero";
        go.layer = LayerMask.NameToLayer("ProjectileHero");
    } else {
        go.tag = "ProjectileEnemy";
        go.layer = LayerMask.NameToLayer("ProjectileEnemy");
    }
    go.transform.position = collar.transform.position;
    go.transform.parent = PROJECTILE_ANCHOR;
    Projectile p = go.GetComponent<Projectile>();
    p.type = type;
    lastShot = Time.time;
    return( p );
}
}

```

Aunque en esta versión del juego no lo incluimos, este código está preparado para que los enemigos también disparen. Se puede observar como la parte variable de las armas se saca de *W_DEFS* a través del atributo *def* de la clase *Weapon*. Se crea un *GameObject* de nombre *__Projectile_Anchor* que sirve de padre de todos los proyectiles e impide que el panel de la jerarquía de Unity se llene objetos de tipo proyectil.

Como el valor de *__type* es inicialmente *WeaponType.none* no disparamos. Es necesario asignar otro valor a este atributo por ejemplo en el inspector para el *GameObject Weapon*.

Daño a los enemigos

A continuación añadimos un método *OnCollisionEnter()* a la clase *Enemy*:

```
public class Enemy : MonoBehaviour {
    ...
    void CheckOffscreen() {
        ...
    }

    void OnCollisionEnter( Collision coll ) {
        GameObject other = coll.gameObject;
        switch (other.tag) {
            case "ProjectileHero":
                Projectile p = other.GetComponent<Projectile>();
                // Enemies don't take damage unless they're onscreen
                // This stops the player from shooting them before they are visible
                bounds.center = transform.position + boundsCenterOffset;
                if (bounds.extents == Vector3.zero ||
                    Utils.ScreenBoundsCheck(bounds, BoundsTest.offScreen) !=
                        Vector3.zero) {
                    Destroy(other);
                    break;
                }
                // Hurt this Enemy
                // Get the damage amount from the Projectile.type & Main.W_DEFS
                health -= Main.W_DEFS[p.type].damageOnHit;
                if (health <= 0) {
                    // Destroy this Enemy
                    Destroy(this.gameObject);
                }
                Destroy(other);
                break;
            }
        }
    }
}
```

Es bueno dar un feedback visual cada vez que se acierta con un disparo a un enemigo. Para ello, haremos que las naves enemigas parpadeen en rojo cada vez que sean acertadas. Eso requiere modificar por un cierto tiempo el material de todos los objetos que componen una nave enemiga. Acceder a los materiales de todos los hijos de un objeto dado puede ser una utilidad general y por ello lo incluimos en la clase *Utils*

```
public class Utils : MonoBehaviour {

    //===== Bounds Functions =====\\
```

```

...

//===== Transform Functions =====\\

...
public static GameObject FindTaggedParent(Transform t) {
    return( FindTaggedParent( t.gameObject ) );
}

}

//===== Materials Functions =====\\

// Returns a list of all Materials on this GameObject or its children
static public Material[] GetAllMaterials( GameObject go ) {
    List<Material> mats = new List<Material>();
    if (go.GetComponent<Renderer>() != null) {
        mats.Add(go.GetComponent<Renderer>().material);
    }
    foreach( Transform t in go.transform ) {
        mats.AddRange( GetAllMaterials( t.gameObject ) );
    }
    return( mats.ToArray() );
}

}

```

Ahora modificamos la clase *Enemy* para que los enemigos parpadeen un par de frames en rojo cada vez que les alcance un proyectil. Al crearse el enemigo guardamos los colores de sus materiales en un array para que después de ponerlos en rojo podamos restaurarlos. Cada vez que un enemigo es alcanzado invoca *ShowDamage* para cambiar los colores de los materiales a rojo, y una vez transcurridos los frames que establece *showDamageForFrames* invoca a *UnShowDamage* para devolverles sus colores originales.

```

public class Enemy : MonoBehaviour {
    ...
    public int          score = 100; // Points earned for destroying this

    public int          showDamageForFrames = 2; // # frames to show damage

    public bool _____;

    public Color[]      originalColors;
    public Material[]   materials; // All the Materials of this & its children
    public int          remainingDamageFrames = 0; // Damage frames left

    public Bounds       bounds; // The Bounds of this and its children

    void Awake() {
        materials = Utils.GetAllMaterials( gameObject );
        originalColors = new Color[materials.Length];
        for (int i=0; i<materials.Length; i++) {
            originalColors[i] = materials[i].color;
        }
    }
}

```



```

        InvokeRepeating( "CheckOffscreen", 0f, 2f );
    }

    // Update is called once per frame
    void Update() {
        Move();
        if (remainingDamageFrames>0) {
            remainingDamageFrames--;
            if (remainingDamageFrames == 0) {
                UnShowDamage();
            }
        }
    }

    void OnCollisionEnter( Collision coll ) {
        GameObject other = coll.gameObject;
        switch (other.tag) {
            case "ProjectileHero":
                ...
                // Hurt this Enemy
                ShowDamage();
                // Get the damage amount from the Projectile.type & Main.W_DEFS
                ...
                break;
        }
    }

    void ShowDamage() {
        foreach (Material m in materials) {
            m.color = Color.red;
        }
        remainingDamageFrames = showDamageForFrames;
    }

    void UnShowDamage() {
        for ( int i=0; i<materials.Length; i++ ) {
            materials[i].color = originalColors[i];
        }
    }
}

```

Power-Ups

Implementamos tres tipos de power-ups:

- blaster [B]: si el arma no es de tipo blaster, se pasa a este tipo inicialmente con un solo cañón. Si el jugador ya tenía blasters, entonces los incrementa en uno.
- spread [S]: si el arma no es de tipo spread, se pasa a este tipo inicialmente con un solo cañón. Si el jugador ya tenía spreads, entonces los incrementa en uno.
- escudo [O]: aumenta en 1 el nivel de escudo del jugador.

Arte de los Power-Ups

Los power-ups se muestran como una letra en 3D con un cubo que da vueltas por detrás

1. Crea un texto 3D (*GameObject > Create Other > 3D Text*) llámalo *PowerUp* y ubícalo en P: [10, 0, 0], R: [0, 0, 0], S: [1, 1, 1] (a partir de la versión 4.6 de Unity ya no se incluye el tipo de objeto *3D Text* y lo que se debe hacer es crear un objeto vacío y añadirle un componente *Text Mesh*). Crea un cubo como un hijo de *PowerUp* y ubícalo en: P: [0, 0, 0], R: [0, 0, 0], S: [2, 2, 2]
2. Selecciona el *PowerUp* y configura el componente *Text Mesh* como se muestra en la figura 19.
3. Añádele un componente *Rigidbody* y configúralo también de acuerdo a la figura 19.
4. Establece la capa *PowerUp* y la etiqueta *PowerUp* para el objeto *PowerUp*.

A continuación crearemos un material para los power-ups:

1. Crea un material de nombre *Mat PowerUp* y arrástralo sobre el cubo que es un hijo del objeto *PowerUp*. Selecciona ese cubo y establece la propiedad *Shader* de *Mat PowerUp* al valor *Custom > UnlitAlpha*.
2. Pulsa el botón *Select* que aparece en la parte inferior derecha de la textura de *Mat PowerUp* y selecciona la textura de nombre *PowerUp* que está entre los recursos del proyecto.
3. Establece el atributo *Main Color* de *Mat PowerUp* a un azul claro (RGBA: [0, 255, 255, 255]).
4. Haz que el cubo sea un trigger, marcando la propiedad *Is Trigger* en su componente *Box Collider*.

Código de los power-ups

Ahora creamos un script de nombre *PowerUp* que asignamos el objeto *PowerUp* y en el que incluimos el siguiente código:

```
using UnityEngine;
using System.Collections;

public class PowerUp : MonoBehaviour {
    // This is an unusual but handy use of Vector2s. x holds a min value
    // and y a max value for a Random.Range() that will be called later
    public Vector2      rotMinMax = new Vector2(15,90);
    public Vector2      driftMinMax = new Vector2(.25f,2);
    public float         lifeTime = 6f; // Seconds the PowerUp exists
    public float         fadeTime = 4f; // Seconds it will then fade
    public bool          _____;
    public WeaponType    type; // The type of the PowerUp
    public GameObject    cube; // Reference to the Cube child
    public TextMesh       letter; // Reference to the TextMesh
    public Vector3        rotPerSecond; // Euler rotation speed
    public float          birthTime;

    void Awake() {
        // Find the Cube reference
        cube = transform.Find("Cube").gameObject;
        // Find the TextMesh
        letter = GetComponent<TextMesh>();

        // Set a random velocity
        Vector3 vel = Random.onUnitSphere; // Get Random XYZ velocity
    }
}
```

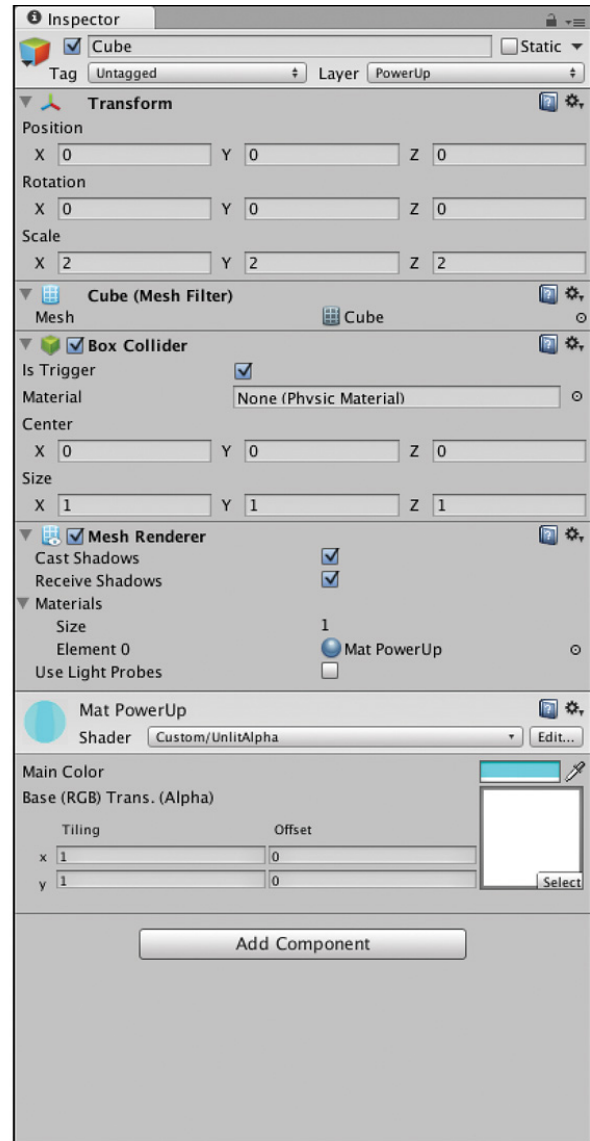
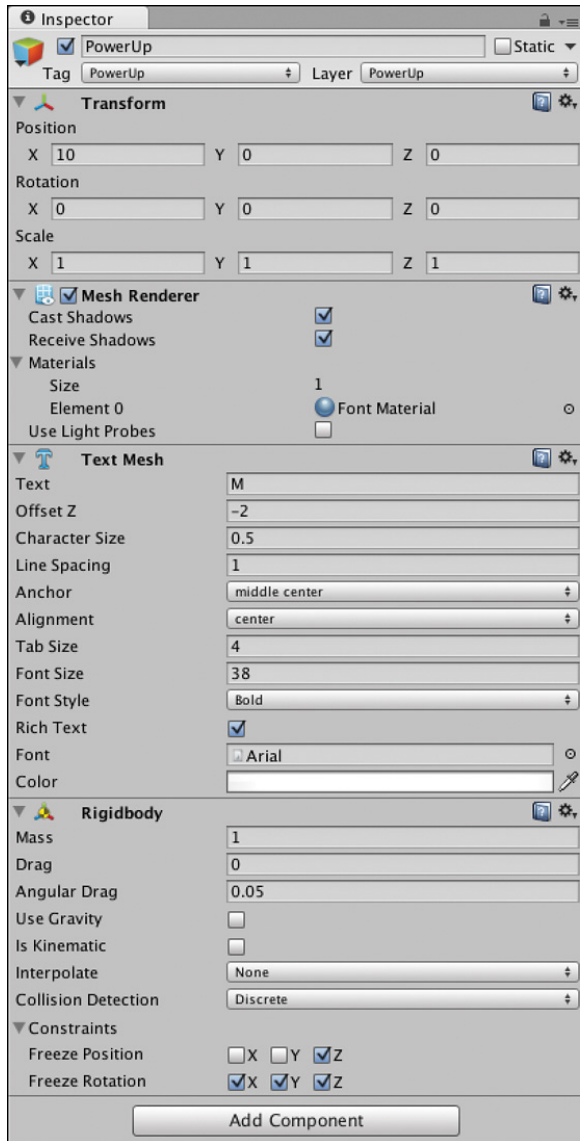


Figure 19: Configuración de los power-ups

```
// Random.onUnitSphere gives you a vector point that is somewhere on
// the surface of the sphere with a radius of 1m around the origin
vel.z = 0; // Flatten the vel to the XY plane
vel.Normalize(); // Make the length of the vel 1
// Normalizing a Vector3 makes it length 1m
vel *= Random.Range(driftMinMax.x, driftMinMax.y);
// Above sets the velocity length to something between the x and y
// values of driftMinMax
GetComponent<Rigidbody>().velocity = vel;

// Set the rotation of this GameObject to R:[0,0,0]
transform.rotation = Quaternion.identity;
// Quaternion.identity is equal to no rotation.
```

```

    // Set up the rotPerSecond for the Cube child using rotMinMax x & y
    rotPerSecond = new Vector3( Random.Range(rotMinMax.x,rotMinMax.y),
                                Random.Range(rotMinMax.x,rotMinMax.y),
                                Random.Range(rotMinMax.x,rotMinMax.y) );

    // CheckOffscreen() every 2 seconds
    InvokeRepeating( "CheckOffscreen", 2f, 2f );

    birthTime = Time.time;
}

void Update () {
    // Manually rotate the Cube child every Update()
    // Multiplying it by Time.time causes the rotation to be time-based
    cube.transform.rotation = Quaternion.Euler( rotPerSecond*Time.time );

    // Fade out the PowerUp over time
    // Given the default values, a PowerUp will exist for 10 seconds
    // and then fade out over 4 seconds.
    float u = (Time.time - (birthTime+lifeTime)) / fadeTime;
    // For lifeTime seconds, u will be <= 0. Then it will transition to 1
    // over fadeTime seconds.
    // If u >= 1, destroy this PowerUp
    if (u >= 1) {
        Destroy( this.gameObject );
        return;
    }
    // Use u to determine the alpha value of the Cube & Letter
    if (u>0) {
        Color c = cube.GetComponent<Renderer>().material.color;
        c.a = 1f-u;
        cube.GetComponent<Renderer>().material.color = c;
        // Fade the Letter too, just not as much
        c = letter.color;
        c.a = 1f - (u*0.5f);
        letter.color = c;
    }
}

// This SetType() differs from those on Weapon and Projectile
public void SetType( WeaponType wt ) {
    // Grab the WeaponDefinition from Main
    WeaponDefinition def = Main.GetWeaponDefinition( wt );
    // Set the color of the Cube child
    cube.GetComponent<Renderer>().material.color = def.color;
    //letter.color = def.color; // We could colorize the letter too
    letter.text = def.letter; // Set the letter that is shown
    type = wt; // Finally actually set the type
}

public void AbsorbedBy( GameObject target ) {
    // This function is called by the Hero class when a PowerUp is collected
    // We could tween into the target and shrink in size,
    // but for now, just destroy this.gameObject
}

```

```

        Destroy( this.gameObject );
    }

    void CheckOffscreen() {
        // If the PowerUp has drifted entirely off screen...
        if ( Utils.ScreenBoundsCheck( cube.GetComponent<Collider>().bounds, BoundsTest.offScreen) !=
            Vector3.zero ) {
            // ...then destroy this GameObject
            Destroy( this.gameObject );
        }
    }
}

```

Ahora tenemos que hacer que la nave del jugador reaccione ante la colisión con un power-up, más allá del mensaje “Triggered: Cube” que está mostrando ahora.

Convertimos el objeto *PowerUp* en un prefab arrastrándolo al panel de proyecto y eliminándolo de la jerarquía.

Ahora añadimos a *Hero* el código que reacciona ante las colisiones con los power-ups y modifica la nave adecuadamente:

```

public class Hero : MonoBehaviour {
    ...
    private float                _shieldLevel = 1;

    // Weapon fields
    public Weapon[]               weapons;

    public bool                   _____;

    void Awake() {
        S = this; // Set the Singleton
        bounds = Utils.CombineBoundsOfChildren(this.gameObject);

        // Reset the weapons to start _Hero with 1 blaster
        ClearWeapons();
        weapons[0].SetType(WeaponType.blaster);
    }

    void OnTriggerEnter(Collider other) {
        ...
        if (go != null) {
            ...

            if (go.tag == "Enemy") {
                // If the shield was triggered by an enemy
                // Decrease the level of the shield by 1
                shieldLevel--;
                // Destroy the enemy
                Destroy(go);
            } else if (go.tag == "PowerUp") {
                // If the shield was triggered by a PowerUp
                AbsorbPowerUp(go);
            } else {
                print("Triggered: "+go.name);           // Move this line here!
            }
        }
    }
}

```

```

    }
    }
    ...
}

public void AbsorbPowerUp( GameObject go ) {
    PowerUp pu = go.GetComponent<PowerUp>();
    switch (pu.type) {
        case WeaponType.shield: // If it's the shield
            shieldLevel++;
            break;

        default: // If it's any Weapon PowerUp
            // Check the current weapon type
            if (pu.type == weapons[0].type) {
                // then increase the number of weapons of this type
                Weapon w = GetEmptyWeaponSlot(); // Find an available weapon
                if (w != null) {
                    // Set it to pu.type
                    w.SetType(pu.type);
                }
            } else {
                // If this is a different weapon
                ClearWeapons();
                weapons[0].SetType(pu.type);
            }
            break;
    }
    pu.AbsorbedBy( this.gameObject );
}

Weapon GetEmptyWeaponSlot() {
    for (int i=0; i<weapons.Length; i++) {
        if ( weapons[i].type == WeaponType.none ) {
            return( weapons[i] );
        }
    }
    return( null );
}

void ClearWeapons() {
    foreach (Weapon w in weapons) {
        w.SetType(WeaponType.none);
    }
}
}

```

A continuación debemos modificar la nave para que tenga 5 cañones. Para ello, seleccionamos el objeto *Weapon* que es hijo de *__Hero* y lo duplicamos 4 veces (pulsando Ctrl+D). Las cinco armas resultantes serán hijos de *__Hero* y las configuramos como se muestra en la figura 21

Por último, seleccionamos el objeto *__Hero* y desplegamos el atributo *Weapons* del componente *Hero (Script)* en el inspector. Fijamos a 5 el tamaño de *Weapons* y asignamos, en orden, *Weapon_0* a *Weapon_4* a los 5 slots del array *Weapons*. En la figura 21 se muestra la configuración final

| | | | |
|----------|-------------------|-----------|-----------|
| _Hero | P:[0,0,0] | R:[0,0,0] | S:[1,1,1] |
| Weapon_0 | P:[0,2,0] | R:[0,0,0] | S:[1,1,1] |
| Weapon_1 | P:[-2,-1,0] | R:[0,0,0] | S:[1,1,1] |
| Weapon_2 | P:[2,-1,0] | R:[0,0,0] | S:[1,1,1] |
| Weapon_3 | P:[-1.25,-0.25,0] | R:[0,0,0] | S:[1,1,1] |
| Weapon_4 | P:[1.25,-0.25,0] | R:[0,0,0] | S:[1,1,1] |

Figure 20: Configuración de las armas

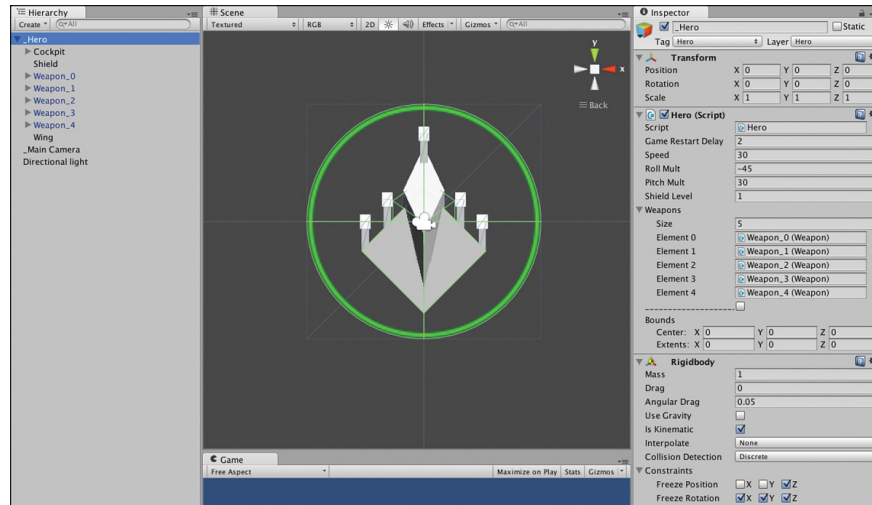


Figure 21: Configuración final de la nave

“Condiciones de carrera”

Si ejecutamos el juego en este momento, es posible que nos encontremos con un error de ejecución debido a una “condición de carrera” (*race condition*). Las condiciones de carrera se producen cuando la correcta ejecución de un programa depende de que distintas partes del programa se ejecuten en un orden determinado, y no esté garantizado que el orden vaya a ser el esperado, de ahí su carácter indeterminista.

El error con el que nos podemos encontrar es:

```
NullReferenceException: Object reference not set to an instance of an object Main.GetWeaponDefinition
(WeaponType wt) (at Assets/Scripts/Main.cs:38) Weapon.SetType (WeaponType wt)
(at Assets/Scripts/Weapon.cs:77) Hero.Awake () (at Assets/___Scripts/Hero.cs:35)
```

y si hacemos doble click sobre el error llegaremos a esta línea del fichero *Main.cs* en el método *Main.GetWeaponDefinition()*:

```
if (W_DEFS.ContainsKey(wt)) {
```

Si utilizamos el depurador y establecemos un punto de ruptura justo antes de esta línea, al ejecutar el programa, cuando se detenga aquí podremos observar que la variable *W_DEFS* tiene valor *null*. Sin embargo, este atributo es inicializado en el método *Main.Awake()* que se supone que se ejecuta antes que cualquier otro de la clase, pero si *W_DEFS* no está inicializada entonces quiere decir que aún no se ha ejecutado ...

El problema es que *Main.GetWeaponDefinition()* es invocado desde *Hero.Awake()* por lo que para que se ejecute correctamente es necesario que se ejecute primero *Main.Awake()*, pero no tenemos manera de garantizar

esto. Entre ejecuciones puede que el orden cambie y unas veces funcione y otras no, y si se ha producido el error es porque *Hero.Awake()* invocó a *Weapon.SetType()* que a su vez llamó a *Main.GetWeaponDefinition()* como podemos comprobar consultando la pila de llamadas con la orden *View > Debug Windows > Call Stack* en MonoDevelop.

Esta es una de las razones por las que Unity tiene los métodos *Awake()* y *Start()*. Mientras que *Awake()* se llama al instanciar el objeto, *Start()* se llama justo antes del primer *Update()*, y entre ambos pueden pasar varios milisegundos. Podemos estar seguro que todos los *Awake()* se llaman antes que cualquiera de los *Start()*. Para resolver el problema lo que haremos será mover el código conflictivo de *Hero.Awake()* a *Hero.Start()*:

```
public class Hero : MonoBehaviour {
    ...

    void Awake() {
        S = this; // Set the Singleton
        bounds = Utils.CombineBoundsOfChildren(this.gameObject);
    }

    void Start() {
        // Reset the weapons to start _Hero with 1 blaster
        ClearWeapons();
        weapons[0].SetType(WeaponType.blaster);
    }

    ...
}
```

Y es posible que aún nos encontremos con otro error producido por una causa similar:

UnassignedReferenceException: The variable collar of Weapon has not been assigned.
 You probably need to assign the collar variable of the Weapon script in the Inspector.
 Weapon.SetType (WeaponType wt) (at Assets/Scripts/Weapon.cs:78) Hero.Start () (at
 Assets/Scripts/Hero.cs:38)

El problema ahora es que tanto *Hero.Start()* como *Weapon.Start()* llaman a *Weapon.SetType()*. Si el que llama primero es *Weapon.Start()* entonces no hay problema, pero si es al revés entonces se produce el error porque todas las armas tienen que haber definido su atributo *Weapon.collar* antes de que se ejecute *Hero.Start()*. La solución es mover la inicialización de *Weapon.collar* al método *Awake()*:

```
void Awake() {
    collar = transform.Find("Collar").gameObject;
}

void Start() {
    // Call SetType() properly for the default _type
    SetType( _type );
}

...
```

Los enemigos sueltan power-ups

Hagamos que los enemigos suelten un power-up elegido al azar cuando sean destruidos. En la clase *Enemy* añadimos código uan llamada a *Main.ShipDestroyed()*


```

public class Enemy : MonoBehaviour {
    ...
    public int          showDamageForFrames = 2; // # frames to show damage
    public float         powerUpDropChance = 1f; // Chance to drop a power-up

    public bool _____;
    ...
    void OnCollisionEnter( Collision coll ) {
        ...
        case "ProjectileHero":
            ...
            if (health <= 0) {
                // Tell the Main singleton that this ship has been destroyed
                Main.S.ShipDestroyed( this );
                // Destroy this Enemy
                Destroy(this.gameObject);
            }
            ...
        }
    }
    ...
}

```

que implementamos en la clase *Main* donde se encarga de instanciar un power-up:

```

public class Main : MonoBehaviour {
    ...
    public WeaponDefinition[] weaponDefinitions;
    public GameObject         prefabPowerUp;
    public WeaponType[]       powerUpFrequency = new WeaponType[] {
        WeaponType.blaster, WeaponType.blaster,
        WeaponType.spread,
        WeaponType.shield
    };

    public bool _____;
    ...

    public void ShipDestroyed( Enemy e ) {
        // Potentially generate a PowerUp
        if (Random.value <= e.powerUpDropChance) {
            // Random.value generates a value between 0 & 1 (though never == 1)
            // If the e.powerUpDropChance is 0.50f, a PowerUp will be generated
            // 50% of the time. For testing, it's now set to 1f.

            // Choose which PowerUp to pick
            // Pick one from the possibilities in powerUpFrequency
            int ndx = Random.Range(0,powerUpFrequency.Length);
            WeaponType puType = powerUpFrequency[ndx];

            // Spawn a PowerUp
            GameObject go = Instantiate( prefabPowerUp ) as GameObject;
            PowerUp pu = go.GetComponent<PowerUp>();
            // Set it to the proper WeaponType
            pu.SetType( puType );
        }
    }
}

```

```

        // Set it to the position of the destroyed ship
        pu.transform.position = e.transform.position;
    }
}
}

```

Por último, para que esto funcione, seleccionamos el objeto *__MainCamera* y en el inspector asignamos al atributo *prefabPowerUp* del componente *Main (Script)* el prefab *PowerUp* que hemos creado antes. El atributo *powerUpFrequency* ya debería contener los valores que hemos establecido en la declaración en el código y que se muestra en la figura 22

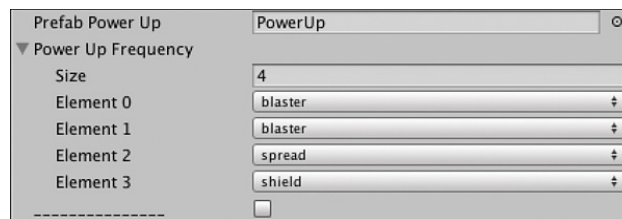


Figure 22: Configuración de los power-ups

Para controlar la frecuencia de cada tipo de power-up lo que hemos hecho es repetir en el array *powerUpFrequency* los que queremos que aparezcan con más frecuencia. Un truco que podemos usar en otros caso.

Otros tipos de enemigos

A continuación vamos a crear un script diferente para cada tipo de enemigo. Crea los scripts *Enemy_1*, *Enemy_2*, *Enemy_3* y *Enemy_4* y asignaselos a los correspondientes prefabs *Enemy_1* a *Enemy_4*.

Enemy_1

Este es el código para *Enemy_1* que hace que se mueva siguiendo una trayectoria sinusoidal:

```

using UnityEngine;
using System.Collections;

// Enemy_1 extends the Enemy class
public class Enemy_1 : Enemy {
    // Because Enemy_1 extends Enemy, the _____ bool won't work // 1
    // the same way in the Inspector pane. :/

    // # seconds for a full sine wave
    public float waveFrequency = 2;
    // sine wave width in meters
    public float waveWidth = 4;
    public float waveRotY = 45;

    private float x0 = -12345; // The initial x value of pos
    private float birthTime;
}

```

```

void Start() {
    // Set x0 to the initial x position of Enemy_1
    // This works fine because the position will have already
    // been set by Main.SpawnEnemy() before Start() runs
    // (though Awake() would have been too early!).
    // This is also good because there is no Start() method
    // on Enemy.
    x0 = pos.x;

    birthTime = Time.time;
}

// Override the Move function on Enemy
public override void Move() { // 2
    // Because pos is a property, you can't directly set pos.x
    // so get the pos as an editable Vector3
    Vector3 tempPos = pos;
    // theta adjusts based on time
    float age = Time.time - birthTime;
    float theta = Mathf.PI * 2 * age / waveFrequency;
    float sin = Mathf.Sin(theta);
    tempPos.x = x0 + waveWidth * sin;
    pos = tempPos;

    // rotate a bit about y
    Vector3 rot = new Vector3(0, sin*waveRotY, 0);
    this.transform.rotation = Quaternion.Euler(rot);

    // base.Move() still handles the movement down in y
    base.Move(); // 3
}
}

```

Aunque *waveFrequency*, *waveWidth* y *waveRotY* están pensadas para que sean modificadas por el diseñador en el inspector, aparecerán debajo de la raya que creamos con el atributo booleano `_____` que usamos para dividir la parte pública de la privada, porque los atributos de una clase hija siempre aparecen debajo de los atributos de la clase padre.

Para probar el funcionamiento de este tipo de enemigo, en Unity puedes cambiar el elemento *Element 0* del atributo *prefabEnemies* del componente *Main (Script)* del objeto *__MainCamera*, sustituyendo *Enemy_0* por *Enemy_1*.

Como se muestra en la figura 24 el collider de tipo *Sphere Collider* que lo rodea es más grande que la nave, por lo que los proyectiles harán impacto antes de que visualmente alcancen a la nave. Esto se puede resolver usando otras forma de colliders que se ajusten más a la nave.

Como estos enemigos giran ligeramente sobre el eje y según se mueven, dependiendo de la forma del collider, puede que salgan del plano z y es por eso que establecimos el atributo *Size.z* del *Box Collider* de *ProjectileHero* a 10.

Enemy_2

Enemy_2 aparecerá rápidamente por uno de los lados de la pantalla, irá frenando, cambiará de dirección ligeramente, seguirá lentamente para finalmente desaparecer de la pantalla a su velocidad inicial. Esto se

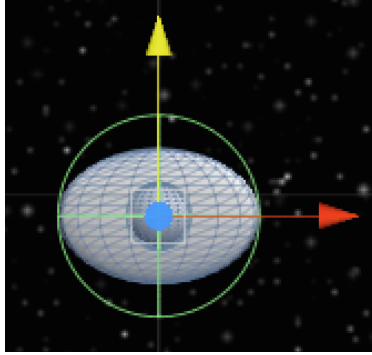


Figure 23: Sphere Collider en el enemigo de tipo 2

consigue con una interpolación lineal entre dos puntos de la forma:

$$p01 = (1-u) * p0 + u * p1$$

donde el parámetro u se calcula usando la función seno:

$$u = u + 0.6 * \sin(2\pi * u)$$

Este es el código de *Enemy_2*

```
using UnityEngine;
using System.Collections;

public class Enemy_2 : Enemy {
    // Enemy_2 uses a Sin wave to modify a 2-point linear interpolation
    public Vector3[]    points;
    public float        birthTime;
    public float        lifeTime = 10;
    // Determines how much the Sine wave will affect movement
    public float        sinEccentricity = 0.6f;

    void Start () {
        // Initialize the points
        points = new Vector3[2];

        // Find Utils.camBounds
        Vector3 cbMin = Utils.camBounds.min;
        Vector3 cbMax = Utils.camBounds.max;

        Vector3 v = Vector3.zero;
        // Pick any point on the left side of the screen
        v.x = cbMin.x - Main.S.enemySpawnPadding;
        v.y = Random.Range( cbMin.y, cbMax.y );
        points[0] = v;

        // Pick any point on the right side of the screen
        v = Vector3.zero;
        v.x = cbMax.x + Main.S.enemySpawnPadding;
```

```

        v.y = Random.Range( cbMin.y, cbMax.y );
        points[1] = v;

        // Possibly swap sides
        if (Random.value < 0.5f) {
            // Setting the .x of each point to its negative will move it to the
            // other side of the screen
            points[0].x *= -1;
            points[1].x *= -1;
        }

        // Set the birthTime to the current time
        birthTime = Time.time;
    }

    public override void Move() {
        // Bézier curves work based on a u value between 0 & 1
        float u = (Time.time - birthTime) / lifeTime;

        // If u>1, then it has been longer than lifeTime since birthTime
        if (u > 1) {
            // This Enemy_2 has finished its life
            Destroy( this.gameObject );
            return;
        }

        // Adjust u by adding an easing curve based on a Sine wave
        u = u + sinEccentricity*(Mathf.Sin(u*Mathf.PI*2));

        // Interpolate the two linear interpolation points
        pos = (1-u)*points[0] + u*points[1];
    }
}

```

Swap the Enemy_2 prefab into the Element 0 slot of Main.S.prefabEnemies using the *MainCamera Inspector* and press Play. As you can see the easing function causes each Enemy2 to have very smooth movement that waves between the points it has selected on either side of the screen.

Enemy_3

Enemy_3 utiliza una curva de Bézier de tres puntos. Esto consiste básicamente en dos interpolaciones lineales consecutivas como se muestra en la figura ??

$$\begin{aligned}
 p01 &= (1-u) * p0 + u * p1 \\
 p12 &= (1-u) * p1 + u * p2 \\
 p012 &= (1-u) * p01 + u * p12
 \end{aligned}$$

Y este es el código de *Enemy_3*

```

using UnityEngine;
using System.Collections;

```

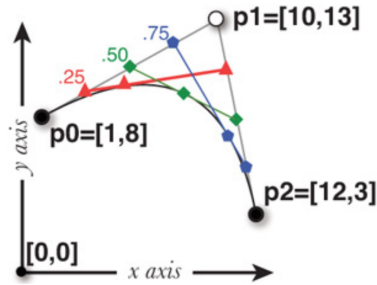


Figure 24: Curva Bézier de 3 puntos

```
// Enemy_3 extends Enemy
public class Enemy_3 : Enemy {

    // Enemy_3 will move following a Bezier curve, which is a linear
    // interpolation between more than two points.
    public Vector3[] points;
    public float birthTime;
    public float lifeTime = 10;

    // Again, Start works well because it is not used by Enemy
    void Start () {
        points = new Vector3[3]; // Initialize points

        // The start position has already been set by Main.SpawnEnemy()
        points[0] = pos;

        // Set xMin and xMax the same way that Main.SpawnEnemy() does
        float xMin = Utils.camBounds.min.x+Main.S.enemySpawnPadding;
        float xMax = Utils.camBounds.max.x-Main.S.enemySpawnPadding;

        Vector3 v;
        // Pick a random middle position in the bottom half of the screen
        v = Vector3.zero;
        v.x = Random.Range( xMin, xMax );
        v.y = Random.Range( Utils.camBounds.min.y, 0 );
        points[1] = v;

        // Pick a random final position above the top of the screen
        v = Vector3.zero;
        v.y = pos.y;
        v.x = Random.Range( xMin, xMax );
        points[2] = v;

        // Set the birthTime to the current time
        birthTime = Time.time;
    }

    public override void Move() {
        // Bezier curves work based on a u value between 0 & 1
        float u = (Time.time - birthTime) / lifeTime;
```

```

    if (u > 1) {
        // This Enemy_3 has finished its life
        Destroy( this.gameObject );
        return;
    }

    // Interpolate the three Bezier curve points
    Vector3 p01, p12;
    p01 = (1-u)*points[0] + u*points[1];
    p12 = (1-u)*points[1] + u*points[2];
    pos = (1-u)*p01 + u*p12;
}
}

```

Si lo ejecutamos observaremos que aunque el punto central está fuera de la pantalla, el enemigo no se sale de la pantalla, ya que el punto medio nunca se alcanza. También se observa que en la parte central de la curva se ralentiza mucho el movimiento. Si se quiere que la velocidad sea más uniforme se puede utilizar un término de interpolación sinusoidal:

```

Vector3 p01, p12;
u = u - 0.2f*Mathf.Sin(u*Mathf.PI*2);
p01 = (1-u)*points[0] + u*points[1];

```

Enemy_4

Estos son los “bosses” del nivel y será más difícil destruirlos. Habrá que destruir parte por parte hasta que lo destruyamos entero.

Para empezar tenemos que ajustar los collider de *Enemy_4*. Para ello arrastramos el prefab a la jerarquía para modificarlo. En la jerarquía seleccionamos el objeto *Fuselage* que es hijo de *Enemy_4* y reemplazamos el *Sphere Collider* que tiene con un *Capsule Collider* que configuramos como **Center:** [0,0,0]; **Radius:** 0.5; **Height:** 1; **Direction:** Y-Axis. *Capsule Collider* aproxima mucho mejor la forma del fuselaje que el *Sphere Collider* original.

A continuación sustituye también el *Sphere Collider* de *Wing_L* por un *Capsule Collider* configurado como **Center:** [0,0,0]; **Radius:** 0.1; **Height:** 5; **Direction:** X-Axis.

Por último, selecciona *Wing_R* y sustituye su *Sphere Collider* con un *Capsule Collider* configurado de la misma forma que en *Wing_L*.

Ahora, pulsando el botón *Prefab > Apply* en la parte superior del inspector con el objeto *Enemy_4* seleccionado podemos aplicar los cambios al prefab *Enemy_4*. Para confirmar que el prefab se ha actualizado adecuadamente puedes arrastrar una nueva instancia suya a la jerarquía y comprobar que los collider están configurados correctamente. Entonces ya puedes borrar las instancias de *Enemy_4* dejando sólo el prefab.

Enemy_4 se mueve siguiendo una interpolación lineal entre dos puntos, eligiendo un nuevo punto de destino cada vez que llega a uno hasta que es completamente destruido.

```

using UnityEngine;
using System.Collections;

public class Enemy_4 : Enemy {
    // Enemy_4 will start offscreen and then pick a random point on screen to
    // move to. Once it has arrived, it will pick another random point and

```

```

// continue until the player has shot it down.

public Vector3[]      points; // Stores the p0 & p1 for interpolation
public float          timeStart; // Birth time for this Enemy_4
public float          duration = 4; // Duration of movement

void Start () {
    points = new Vector3[2];
    // There is already an initial position chosen by Main.SpawnEnemy()
    // so add it to points as the initial p0 & p1
    points[0] = pos;
    points[1] = pos;

    InitMovement();
}

void InitMovement() {
    // Pick a new point to move to that is on screen
    Vector3 p1 = Vector3.zero;
    float esp = Main.S.enemySpawnPadding;
    Bounds cBounds = Utils.camBounds;
    p1.x = Random.Range(cBounds.min.x + esp, cBounds.max.x - esp);
    p1.y = Random.Range(cBounds.min.y + esp, cBounds.max.y - esp);

    points[0] = points[1]; // Shift points[1] to points[0]
    points[1] = p1;        // Add p1 as points[1]

    // Reset the time
    timeStart = Time.time;
}

public override void Move () {
    // This completely overrides Enemy.Move() with a linear interpolation

    float u = (Time.time-timeStart)/duration;
    if (u>=1) { // if u >=1...
        InitMovement(); // ...then initialize movement to a new point
        u=0;
    }

    u = 1 - Mathf.Pow( 1-u, 2 ); // Apply Ease Out easing to u

    pos = (1-u)*points[0] + u*points[1]; // Simple linear interpolation
}
}

```

Ahora añadimos el código que *Enemy_4* tenga 4 partes que se destruyan por separado, siendo la parte central, el *Cockpit*, la última que se destruya.

Añadimos una nueva clase serializable *Part* al principio del fichero *Enemy_4.cs*, e incluimos un atributo *parts* de tipo *Part[]* en la clase *Enemy_4*:

```

using UnityEngine;
using System.Collections;

```



```

// Part is another serializable data storage class just like WeaponDefinition
[System.Serializable]
public class Part {
    // These three fields need to be defined in the Inspector pane
    public string      name;           // The name of this part
    public float       health;         // The amount of health this part has
    public string[]    protectedBy;    // The other parts that protect this

    // These two fields are set automatically in Start().
    // Caching like this makes it faster and easier to find these later
    public GameObject  go;             // The GameObject of this part
    public Material    mat;            // The Material to show damage
}

public class Enemy_4 : Enemy {
    ...
    public float      duration = 4;    // Duration of movement

    public Part[]      parts;          // The array of ship Parts

    void Start() {
        ...
    }
    ...
}

```

La clase *Part* guarda información sobre cada una de las 4 partes de un *Enemy_4*: *Cockpit*, *Fuselage*, *Wing_L* y *Wing_R*. En Unity seleccionamos el prefab *Enemy_4* y en el componente *Enemy_4 (Script)* incluimos la configuración de la figura 25

Está configurado de forma que *Cockpit* está protegido por *Fuselage*, que a su vez está protegido por *Wing_L* y *Wing_R*.

Por último debemos añadir este código a *Enemy_4* para que funcione la protección:

```

public class Enemy_4 : Enemy {
    ...
    void Start () {
        ...
        InitMovement();

        // Cache GameObject & Material of each Part in parts
        Transform t;
        foreach(Part prt in parts) {
            t = transform.Find(prt.name);
            if (t != null) {
                prt.go = t.gameObject;
                prt.mat = prt.go.GetComponent<Renderer>().material;
            }
        }
    }
    ...
}

```

```

public override void Move() {
    ...
}

// This will override the OnCollisionEnter that is part of Enemy.cs
// Because of the way that MonoBehaviour declares common Unity functions
// like OnCollisionEnter(), the override keyword is not necessary.
void OnCollisionEnter( Collision coll ) {
    GameObject other = coll.gameObject;
    switch (other.tag) {
        case "ProjectileHero":
            Projectile p = other.GetComponent<Projectile>();
            // Enemies don't take damage unless they're on screen
            // This stops the player from shooting them before they are visible
            bounds.center = transform.position + boundsCenterOffset;
            if (bounds.extents == Vector3.zero ||
                Utils.ScreenBoundsCheck(bounds, BoundsTest.offScreen) !=
                Vector3.zero) {
                Destroy(other);
                break;
            }

            // Hurt this Enemy
            // Find the GameObject that was hit
            // The Collision coll has contacts[], an array of ContactPoints
            // Because there was a collision, we're guaranteed that there is at
            // least a contacts[0], and ContactPoints have a reference to
            // thisCollider, which will be the collider for the part of the
            // Enemy_4 that was hit.
            GameObject goHit = coll.contacts[0].thisCollider.gameObject;
            Part prtHit = FindPart(goHit);
            if (prtHit == null) { // If prtHit wasn't found
                // ...then it's usually because, very rarely, thisCollider on
                // contacts[0] will be the ProjectileHero instead of the ship
                // part. If so, just look for otherCollider instead
                goHit = coll.contacts[0].otherCollider.gameObject;
                prtHit = FindPart(goHit);
            }
            // Check whether this part is still protected
            if (prtHit.protectedBy != null) {
                foreach( string s in prtHit.protectedBy ) {
                    // If one of the protecting parts hasn't been destroyed...
                    if (!Destroyed(s)) {
                        // ...then don't damage this part yet
                        Destroy(other); // Destroy the ProjectileHero
                        return; // return before causing damage
                    }
                }
            }
            // It's not protected, so make it take damage
            // Get the damage amount from the Projectile.type & Main.W_DEFS
            prtHit.health -= Main.W_DEFS[p.type].damageOnHit;
            // Show damage on the part
            ShowLocalizedDamage(prtHit.mat);

```

```

        if (prtHit.health <= 0) {
            // Instead of Destroying this enemy, disable the damaged part
            prtHit.go.SetActive(false);
        }
        // Check to see if the whole ship is destroyed
        bool allDestroyed = true; // Assume it is destroyed
        foreach( Part prt in parts ) {
            if (!Destroyed(prt)) { // If a part still exists
                allDestroyed = false; // ...change allDestroyed to false
                break; // and break out of the foreach loop
            }
        }
        if (allDestroyed) { // If it IS completely destroyed
            // Tell the Main singleton that this ship has been destroyed
            Main.S.ShipDestroyed( this );
            // Destroy this Enemy
            Destroy(this.gameObject);
        }
        Destroy(other); // Destroy the ProjectileHero
        break;
    }
}

// These two functions find a Part in this.parts by name or GameObject
Part FindPart(string n) {
    foreach( Part prt in parts ) {
        if (prt.name == n) {
            return( prt );
        }
    }
    return( null );
}

Part FindPart(GameObject go) {
    foreach( Part prt in parts ) {
        if (prt.go == go) {
            return( prt );
        }
    }
    return( null );
}

// These functions return true if the Part has been destroyed
bool Destroyed(GameObject go) {
    return( Destroyed( FindPart(go) ) );
}

bool Destroyed(string n) {
    return( Destroyed( FindPart(n) ) );
}

bool Destroyed(Part prt) {
    if (prt == null) { // If no real Part was passed in
        return(true); // Return true (meaning yes, it was destroyed)
    }
    // Returns the result of the comparison: prt.health <= 0
    // If prt.health is 0 or less, returns true (yes, it was destroyed)
}

```

```

        return (prt.health <= 0);
    }

    // This changes the color of just one Part to red instead of the whole ship
    void ShowLocalizedDamage(Material m) {
        m.color = Color.red;
        remainingDamageFrames = showDamageForFrames;
    }
}

```

Añadir un fondo de estrellas

Para crear un fondo estrellado de dos capas empezamos creando un quad en la jerarquía de nombre *StarfieldBG* y ubícalo en P: [0,0,10], R: [0,0,0], S: [80,80,1]

A continuación creamos un material nuevo de nombre *Mat Starfield* y fija su atributo *shader* a *Custom > UnlitAlpha* y su textura al recurso *Space Texture2D* que se importó al principio. Aplicamos el material *Mat Starfield* sobre *StarfieldBG*.

Selecciona *Mat Starfield* en el panel de proyecto y duplícalo (Control+D), dale el nombre *Mat Starfield Transparent* y asígnale la textura *Space_Transparent*.

Selecciona *StarfieldBG* en la jerarquía y duplícalo. Al duplicado asígnale el nombre *StarfieldFG_0*, asígnale el material *Mat Starfield Transparent* y ubícalo en P: [0,0,5], R: [0,0,0], S: [160,160,1]

Para producir un cierto efecto de paralaje duplicamos *Starfield_FG_0* y lo llamamos duplicado *Starfield_FG_1*.

Creamos un nuevo script de nombre *Parallax* donde incluimos el código:

```

using UnityEngine;
using System.Collections;

public class Parallax : MonoBehaviour {

    public GameObject      poi; // The player ship
    public GameObject[]    panels; // The scrolling foregrounds
    public float           scrollSpeed = -30f;
    // motionMult controls how much panels react to player movement
    public float           motionMult = 0.25f;

    private float panelHt; // Height of each panel
    private float depth;   // Depth of panels (that is, pos.z)

    // Use this for initialization
    void Start () {
        panelHt = panels[0].transform.localScale.y;
        depth = panels[0].transform.position.z;

        // Set initial positions of panels
        panels[0].transform.position = new Vector3(0,0,depth);
        panels[1].transform.position = new Vector3(0,panelHt,depth);
    }

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

```

```

float tY, tX=0;
tY= Time.time * scrollSpeed % panelHt + (panelHt*0.5f);

if (poi != null) {
    tX = -poi.transform.position.x * motionMult;
}

// Position panels[0]
panels[0].transform.position = new Vector3(tX, tY, depth);
// Then position panels[1] where needed to make a continuous starfield
if (tY >= 0) {
    panels[1].transform.position = new Vector3(tX, tY-panelHt, depth);
} else {
    panels[1].transform.position = new Vector3(tX, tY+panelHt, depth);
}
}
}

```

Asigna este script a *__MainCamera*, y configúralo en el inspector asignando *__Hero* al atributo *poi* y añadiendo *StarfieldFG_0* y *StarfieldFG_1* al array *panels*.

Referencias

- [Jeremy Gibson](#). Introduction to Game Design, Prototyping, and Development: From Concept to Playable Game with Unity and C#. Addison-Wesley Professional, 2014

Enemy_4 (Script)

| | | | | | | | |
|------------------------|-------------------------|--------------------|---|---|---|---|---|
| Script | Enemy_4 | | | | | | |
| Speed | 10 | | | | | | |
| Fire Rate | 0.3 | | | | | | |
| Health | 20 | | | | | | |
| Score | 100 | | | | | | |
| Show Damage For Frames | 2 | | | | | | |
| Power Up Drop Chance | 1 | | | | | | |
| ----- | | | | | | | |
| Bounds | | | | | | | |
| | Center: | X | 0 | Y | 0 | Z | 0 |
| | Extents: | X | 0 | Y | 0 | Z | 0 |
| | Bounds Center Offset | X | 0 | Y | 0 | Z | 0 |
| ► Materials | | | | | | | |
| ► Original Colors | | | | | | | |
| | Remaining Damage Frames | 0 | | | | | |
| ► Points | | | | | | | |
| | Time Start | 0 | | | | | |
| | Duration | 4 | | | | | |
| ▼ Parts | | | | | | | |
| | Size | 4 | | | | | |
| ▼ Cockpit | | | | | | | |
| | Name | Cockpit | | | | | |
| | Health | 10 | | | | | |
| | ▼ Protected By | | | | | | |
| | Size | 1 | | | | | |
| | Element 0 | Fuselage | | | | | |
| | Go | None (Game Object) | | | | | |
| | Mat | None (Material) | | | | | |
| ▼ Fuselage | | | | | | | |
| | Name | Fuselage | | | | | |
| | Health | 10 | | | | | |
| | ▼ Protected By | | | | | | |
| | Size | 2 | | | | | |
| | Element 0 | Wing_L | | | | | |
| | Element 1 | Wing_R | | | | | |
| | Go | None (Game Object) | | | | | |
| | Mat | None (Material) | | | | | |
| ▼ Wing_L | | | | | | | |
| | Name | Wing_L | | | | | |
| | Health | 10 | | | | | |
| | ► Protected By | | | | | | |
| | Go | None (Game Object) | | | | | |
| | Mat | None (Material) | | | | | |
| ▼ Wing_R | | | | | | | |
| | Name | Wing_R | | | | | |
| | Health | 10 | | | | | |
| | ► Protected By | | | | | | |
| | Go | None (Game Object) | | | | | |
| | Mat | None (Material) | | | | | |

Figure 25: Configuración de las partes de Enemy4