



Laboratorio 5

Motores de Videojuegos © 2020-21 © Eva Ullán



Creación de objetos

Cubo “físico”

- Creamos un cubo y lo convertimos en físico, añadiéndole un componente **RigidBody**
- Guardamos y ejecutamos la escena para ver qué pasa
 - El cubo cae debido a la gravedad
 - *Todas las simulaciones físicas en Unity están basadas en el sistema métrico:*
 - 1 unidad de distancia = 1 metro
(por ejemplo, las unidades para la posición de un Transform)
 - » Un personaje humano medio tendrá una altura de unas 2 unidades
 - 1 unidad de masa = 1 kg
(por ejemplo, las unidades de masa de un RigidBody)
 - La gravedad por defecto es $-9.8 = 9.8 \text{ m/s}^2$ hacia abajo

Quiero más cubos

- Hemos creado un cubo tuneado a nuestro gusto
- Queremos otro cubo igual
 - Lo duplicamos y lo cambiamos de posición para diferenciarlo
- Y ahora queremos otro cubo más
 - Otra vez lo mismo
- Y otro más...
 - Hasta que me canse, algo que ocurrirá seguro
- Entonces, me planteo
 - Si cambio el original, ¿la modificación afectará al resto?
 - Si quisiera 100, ¿haría lo mismo? ¿O buscaría alternativa?

Creación de objetos

- Lo habitual es que los *GameObjects* de una escena se añadan y configuren desde el editor de escenas
 - Esto ya sabemos hacerlo
- Sin embargo, hay ocasiones en las que interesa **crear objetos desde código**
 - Disparar balas
 - Generar (*spawn*) enemigos
 - Crear los ladrillos de un muro (como en Arkanoid)
 - Crear *power-ups* cuando un enemigo es destruido

Creación de *GameObjects* por código

- ¿Cómo puedo tener un objeto pre-construido?
 - Bala, enemigo, ladrillo, *power-up*...
- La manera más sencilla de crear un *GameObject* desde código es **clonar** otro
- Esto nos plantea un problema: ¿de dónde sacamos el objeto que vamos a clonar...?
 - Podemos clonar un *GameObject* que ya esté en la escena
 - Tendría que tener una bala, un enemigo, etc, en la escena, por ejemplo
 - Unity incluye otro mecanismo más adecuado: los *prefab*

Prefabs

- Un ***prefab*** es un objeto “complejo” pre-construido pensado para
 - **Ser reutilizado durante el diseño** de un nivel desde el editor
 - Duplicar objetos no es buena idea ya que cada uno es independiente
 - Si decido cambiar algún detalle, estoy obligada a editarlos todos
 - Un cambio en todos los objetos clonados es más fácil si usamos un *prefab*
 - **Ser *instanciado* desde código** en tiempo de ejecución
 - Podríamos crear desde cero un *GameObject* completo desde código, pero sería más complicado
 - Usando un *prefab* lo consigo con una única instrucción

Prefabs

- Un ***prefab*** es un prototipo/plantilla de *GameObject*
 - Es como un molde de un *GameObject*, a partir del cual podemos crear *GameObjects* que serán copias suyas
 - Lo mismo que una clase es una plantilla de un objeto (componente)
- Cada *GameObject* creado a partir del *prefab* se llama **instancia del *prefab***
 - Cada objeto creado a partir de una clase se llama instancia de la clase
- El *prefab* será un recurso del proyecto
 - Es un elemento reutilizable en un proyecto que puede instanciarse cualquier número de veces

Creación de *prefabs*

- ¿Cómo se crea un *prefab* en Unity?
- Un *prefab* se puede crear a partir de un objeto de la escena/jerarquía
 1. Creamos un *GameObject* en la escena
 2. Lo configuramos como resulte conveniente
 3. Lo arrastramos de la vista Jerarquía a la vista Proyecto, creando así el *prefab*
- Borramos la(s) copia(s) y usamos nuestro primer cubo para crear un *prefab*

Creación de *prefabs*

- Han sucedido un par de cosas
 - Se ha creado un *prefab* llamado *Cube* en la vista Proyecto
 - El icono de cubo azul siempre representa un *prefab* (aunque éste no sea un *prefab* de un cubo)
 - El nombre (y el icono) del *GameObject* *Cube* en la jerarquía se ha vuelto de color azul
 - Cuando un *GameObject* tiene un nombre azulado en la jerarquía es porque ese *GameObject* es una instancia de un *prefab*
- Para no liarnos, renombramos el *prefab* a *CubePrefab*

Creación de copias de un *prefab*

- ¿Cómo creamos copias a partir de un *prefab*?
- Si queremos crearlas desde el editor
 - Basta con arrastrarlo de la vista Proyecto a la vista Jerarquía o Escena
- Creamos varias copias en distintas posiciones

Cambios en el *prefab*

- Los *prefabs* nos permiten **cambiar todas sus instancias de una sola vez**, modificando el valor de la propiedad o el componente desde el *prefab*
 - Si hacemos cambios en el *prefab*, éstos se propagarán a las copias que hayamos hecho
 - Probamos a cambiar la escala
- Si modificamos una instancia del *prefab*, tendremos la opción de mantener esa versión modificada o consolidar los cambios en el *prefab*, afectando así a todas las copias de éste
 - Probamos a cambiar el *prefab* desde una instancia

Clonación de un *prefab* desde código

- Aún nos quedan algunas cuestiones por resolver antes de poder clonar un *prefab* desde un *script* en C#
 - ¿Cómo obtengo una referencia al *prefab* en el código?
 - ¿A qué *GameObject* asocio el *script*?
 - ¿Con qué instrucción se clona un *prefab*?

Clonación de un *prefab* desde código

- Una vez que tenemos nuestro *prefab*, ya no necesitamos ninguna instancia suya en la escena
 - Eliminamos las instancias del cubo
- Instanciaremos los cubos desde código
 1. Creamos un *script* *CubeSpawner*
 2. Declaramos una variable pública para asociarle una referencia a nuestro *prefab* desde el editor
 - ¿De qué tipo?
 - `public GameObject myPrefab;`
 3. Añadimos el *script* como componente ¿de quién?
 - Un *GameObject* vacío es lo más apropiado (*Spawner*)

Instanciación

- La **instanciación** consiste en crear un duplicado de un objeto a partir de otro, generalmente un *prefab*
 - En Unity se hace con los métodos `Instantiate`
 - Son métodos estáticos con varias formas/variantes/signaturas
 - La más general es
 - `Instantiate<Tipo> (T original, Vector3 position, Quaternion rotation, Transform parent)`
 - Clona el objeto original situándolo en la posición y rotación indicadas, haciéndolo hijo del *transform* indicado

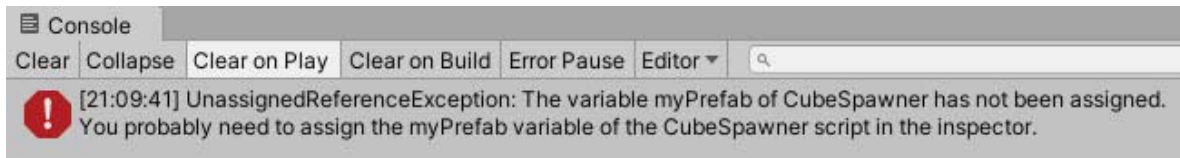
En clase hablaremos de estos métodos más detenidamente

Creación de *GameObjects* por código

- Los métodos `Instantiate` clonan el objeto que reciben
 - Clonan todo el *GameObject* con todos sus hijos, componentes y configuración
- Escribimos código para clonar el *prefab*
 4. Creamos una copia del *prefab* instanciándolo en el bloque de código adecuado y ejecutamos para probarlo
`Instantiate<GameObject> (myPrefab);`

¿Error en ejecución?

- Miramos de qué tipo de error se trata
 - Si es `UnassignedReferenceException`, estamos de suerte porque va a ser fácil arreglarlo



- Alguna referencia no ha sido asignada
 - Una variable que debería estar asignada y hacer referencia a algo para que el código funcione
 - ¿Dónde? Donde indica Unity en el texto del error
- **Arreglo:** asignar una referencia a la variable

Si queremos muchos cubos

- Seguimos
 5. Movemos la instrucción al bloque de código adecuado para crear **muchos cubos**
 - ¿Qué pasará?
 - ¿Muchos cubos en la misma posición?
 - ¿Podremos distinguirlos en la escena?
 - Ejecutamos para probarlo
 - Pausamos la ejecución y observamos el desorden de la vista Jerarquía
 - Muchos hijos sin padre ;-)

Resultado

- Vemos aparecer un sinfín de cubos cayendo
 - Esto es un poco más estimulante y vistoso que ver muchos “¡Hola Mundo!” en la consola ;-)
- ¿Por qué se comportan como una fuente?
 - Como los cubos son físicos (tienen un componente *RigidBody*), al aparecer dos cubos en la misma posición se repelen
 - Si clonásemos un cubo normal, ¿cuál sería la diferencia?
 - Pista: el cubo no tiene *RigidBody*

Motores de Videojuegos © 2020-21

Jugueteamos un poco más

- Añadamos algunos objetos más a la escena para que los cubos puedan interactuar con ellos
 - Creamos un *Suelo* y lo situamos por debajo del lugar de aparición **Suelo (Plane) P:[0, -2, 0] R:[0, 0, 0] S:[1, 1, 1]**
 - Convenio
 - **Suelo** es el nombre del *GameObject* y **Plane** es el tipo de objeto
 - **P:[0, -2, 0]** establece la posición Y a -2 y mantiene las de X y Z en 0
 - **R:[0, 0, 0]** mantiene las rotaciones X, Y, Z a 0
 - **S:[1, 1, 1]** mantiene la escala de X, Y, Z a 1
 - Añadimos más objetos a la escena y volvemos a probarlo
 - Las instancias rebotarán al toparse con los objetos estáticos de la escena. Mientras no añadamos ningún *RigidBody* a las nuevas formas, éstas serán estáticas (es decir, sólidas e inmóviles)

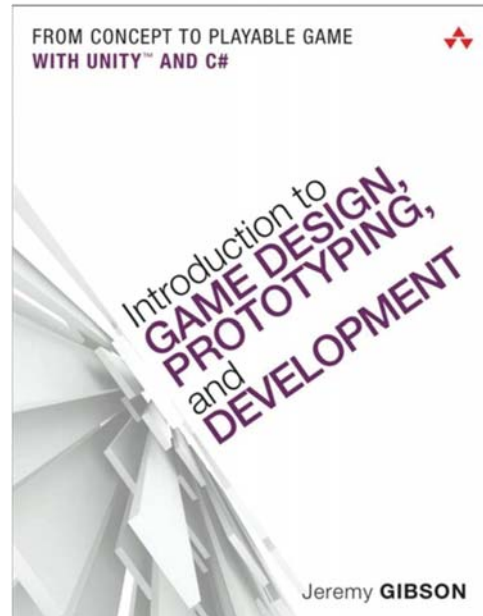
Referencias



- **Introduction to Game Design, Prototyping, and Development:** From Concept to Playable Game with Unity and C#

Jeremy **Gibson**

- *Addison-Wesley Professional, 2014*
- *[2ª edición, agosto 2017]*



- Documentación de Unity

Referencias

- *Prefabs*
 - <https://docs.unity3d.com/Manual/Prefabs.html>
- *Instantiate*
 - <https://docs.unity3d.com/ScriptReference/Object.Instantiate.html>



Lluvia de letras

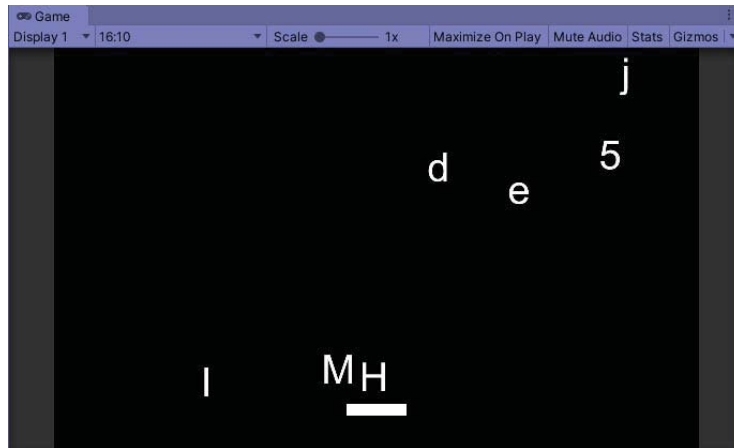
Primera práctica

Motores de Videojuegos © 2020-21 © Eva Ullán

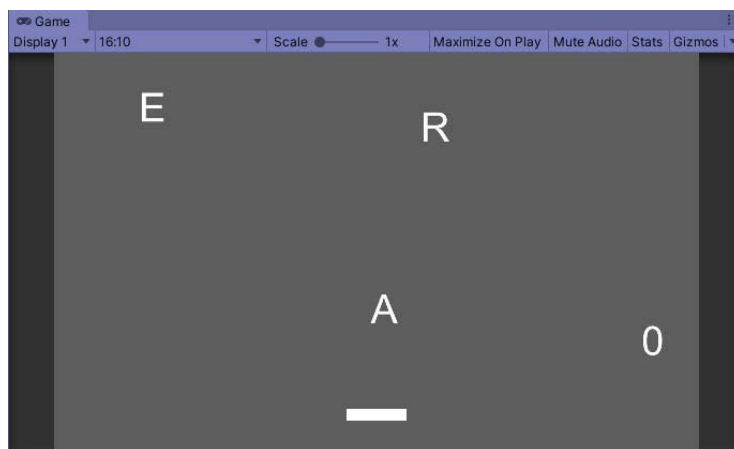
Demo en vivo del juego

- Letras que caen en vertical
- Se ganan puntos “salvando” letras
 - Escribiéndolas (jugador 1)
 - O rescatándolas con la pala inferior (jugador 2)
- Cada letra no salvada incrementa el nivel de daño
- El daño afecta al color del fondo
 - Cuando llega a 100, el fondo se vuelve blanco, haciendo el juego imposible de jugar
 - Paralelamente se va recuperando a razón de 1 por segundo

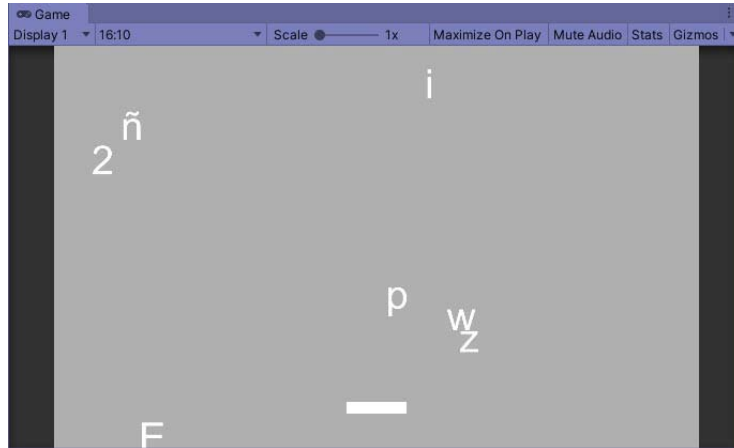
Sin daño



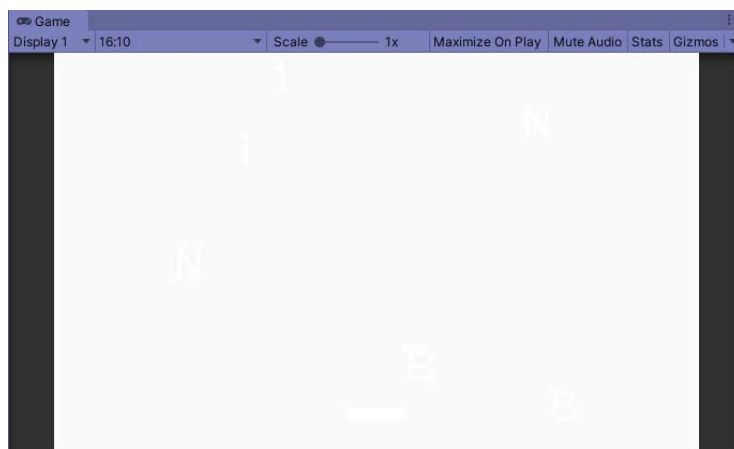
Con algo de daño



El daño complica las cosas



Hasta que ya no es jugable





Por el principio

¿Por dónde empezar?

Motores de Videojuegos © 2020-21 © Eva Ullán

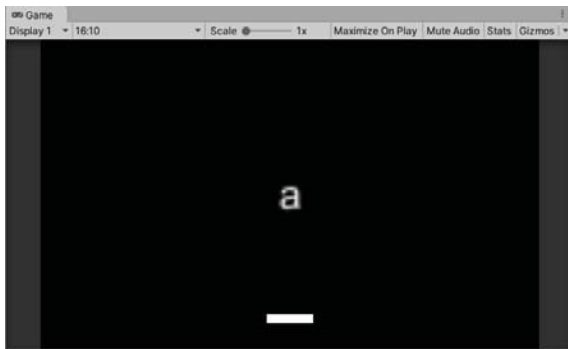
Las cosas evidentes

- Tipo de proyecto
 - Proyecto 2D con luz direccional
 - Para que se vea la pala del segundo jugador
- Fondo de la cámara negro
- Crear la pala del segundo jugador (cubo alargado)

Crear una letra

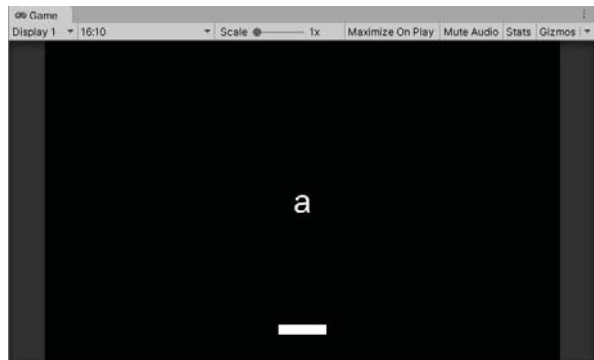
Sin nitidez

- Añadir objeto 3D Text que tiene un componente *Text Mesh*



Con nitidez

- Para mejorar la calidad del texto
 - Tamaño de la fuente a 100
 - Tamaño del carácter a 0.1



¿Dimensiones del mundo?

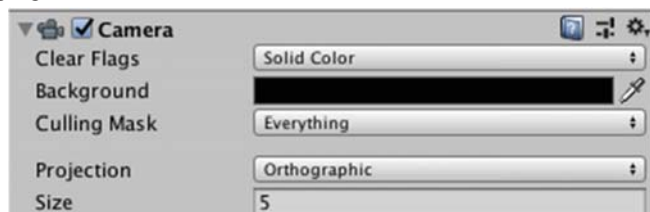
- Concretaremos

- Vista de escena en 2D



- Configuración de la cámara

- Color de fondo
- Negro (0, 0, 0, 0)
- Proyección ortográfica
- **Tamaño 5**



- Nº de unidades desde el centro hasta el tope superior (1/2 altura)

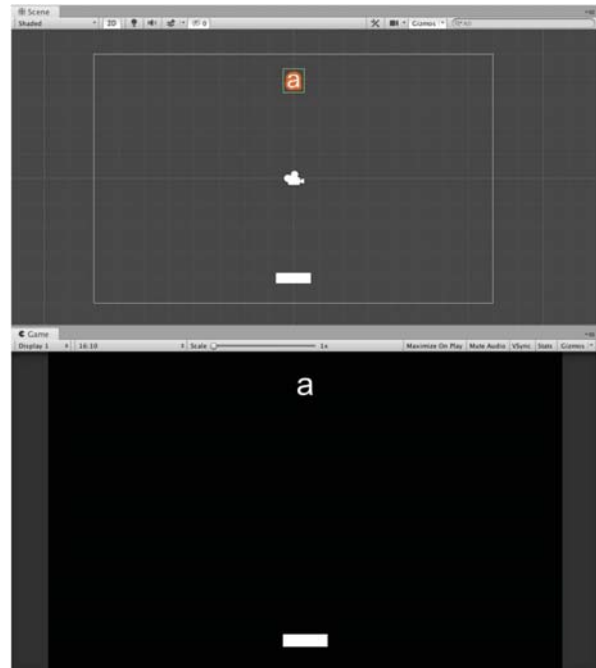
- Vista de juego

- Concretaremos un *Aspect Ratio: 16:10*
 - ¿Cuál es la anchura del mundo?
 - » ¿Y si el tamaño fuera 10?



Dimensiones del mundo

- Mundo
 - Parte de la escena visible a través de la cámara
 - Centro: (0, 0, 0)
 - Alto: 10
 - Ancho: 16
- Pala
 - Eje X: 0
 - El ancho se lo pone el diseñador
 - ¿Cómo lo consultamos?

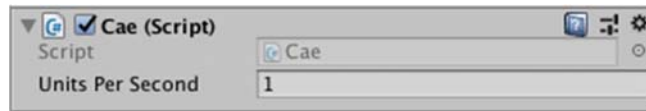


Parte de ellos

Scripts
2CLIB12

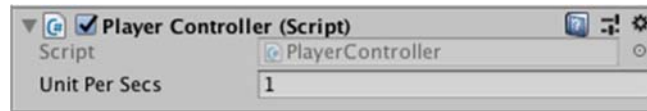
Cae

- Dota al *GameObject* de la capacidad de caer en el eje Y a una velocidad configurable desde el Inspector
 - Si la velocidad es negativa, subirá en lugar de bajar
 - `public float unitPerSecs;`



- Probadlo con la letra

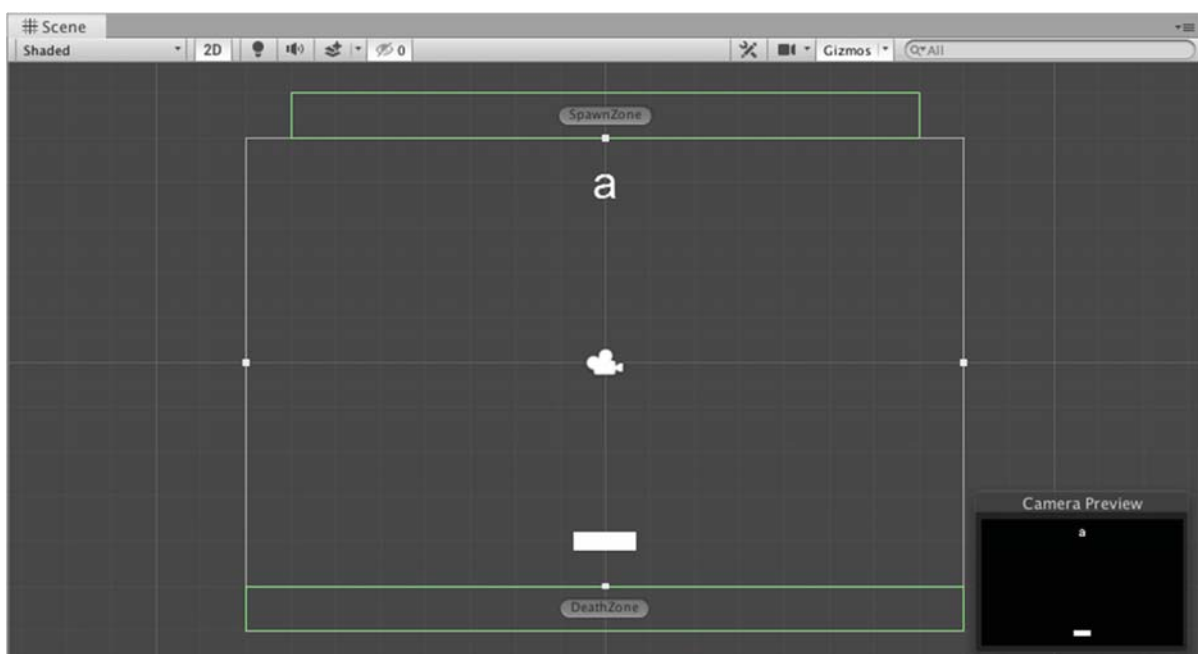
PlayerController

- El jugador 2 se mueve horizontalmente con las teclas de dirección a una velocidad configurable desde el Inspector
 - `public float unitPerSecs;`
- 
- No debe obedecer a las teclas 'a' ni 'd'
 - Viable ajustar propiedades del eje quitando teclas alternativas
 - No debe salirse de los límites horizontales del mundo
 - `const float anchoMundo = ...;`
- Opcional
 - El jugador 2 puede moverse también con el ratón
 - Además de con las teclas de dirección

Elementos invisibles de la escena

- *SpawnZone*
 - Cubo invisible en la parte superior de la escena
 - Lugar de nacimiento de las letras
- *DeadZone*
 - Cubo invisible en la parte inferior de la escena
 - Lugar de destrucción de las letras “no salvadas”

Vista escena



Spawner

- Genera instancias de un *prefab*
 - Lo hace en una **posición aleatoria** en el eje X dentro de los límites del objeto al cual esté asociado
 - Es muy fácil determinar esos límites (si podemos asumir que el *GameObject* al que va a estar asociado tiene su centro en $X = 0$)
 - Otra forma, que no presupone nada y es algo más complicada, requiere **investigar la clase `Bounds`**
 - El **ritmo de creación** de instancias es **aleatorio** dentro de un intervalo configurable desde el editor
 - `public float minRitmoCreacion;`
 - `public float maxRitmoCreacion;`
 - **Investigar la clase `Random`**
 - Para conseguir un valor **aleatorio** dentro de un intervalo

Continuará...