# 1

# INTRODUCCIÓN A UNITY Y C#

## 1.1 EMPEZANDO

El objetivo principal de este libro no es aprender todos los detalles del motor de juegos Unity o fundamentos de programación o el lenguaje de programación C# en toda su profundidad. Por necesidad, cubriremos estos temas a un nivel básico al comienzo de nuestro viaje, y con más detalle en las unidades sucesivas. Sin embargo, estos temas proporcionan una forma accesible para que aprendamos el lenguaje de programación C#.

**C#** es un lenguaje de programación desarrollado por Microsoft en el año 2000, como parte de su plataforma .NET. Se creó con el propósito de ser un lenguaje moderno, simple y orientado a objetos, diseñado para competir con Java y C++. Su sintaxis está influenciada por C y C++, pero con características más avanzadas para mejorar la productividad de los desarrolladores y evitar algunos de los errores comunes en lenguajes más antiguos.

Algunas de estas características avanzadas de C# son:

En C++, la gestión manual de la memoria puede conducir a fugas de memoria o uso indebido de punteros. C# incluye un *recolector de basura* que gestiona automáticamente la memoria, liberando espacio ocupado por objetos no utilizados.

*LINQ* (Language Integrated Query): ***LINQ*** permite realizar consultas sobre colecciones de datos de forma concisa y legible, similar a SQL, pero directamente en el lenguaje de programación.

Propiedades automáticas (auto-properties): C# permite declarar propiedades con una sintaxis simplificada, eliminando la necesidad de escribir código repetitivo para ***getters*** y ***setters***. Esto mejora la claridad del código y reduce errores.

Inferencia de tipos con **var**: C# permite la inferencia de tipos, donde el compilador deduce el tipo de una variable en función del valor que se le asigna. Esto reduce el código boilerplate sin perder la seguridad de tipos, mejorando la legibilidad y manteniendo el control estricto de tipos en tiempo de compilación.

*En 2005, Unity comenzó a utilizar C#* como uno de los lenguajes principales para el scripting en su motor de desarrollo de videojuegos. La adopción de C# permitió a Unity aprovechar el ecosistema .NET, su amplia base de programadores, acceso a sus herramientas y bibliotecas. Además, el lenguaje facilitó la creación de scripts mantenibles más fácilmente, en comparación con ***UnityScript*** (basado en JavaScript) y ***Boo*** (un lenguaje menos popular que soportaba "Python-like scripting"), que también estaban disponibles en las primeras versiones de Unity.

Con el tiempo, C# se consolidó como el lenguaje principal para el desarrollo en Unity, *eliminando el soporte para UnityScript y Boo en 2017*.

*Unity* es un motor de juegos multiplataforma desarrollado por una empresa fundada en 2004 en Dinamarca. Los impulsores detrás de Unity fueron Joachim Ante, Nicholas Francis y David Helgason…

De izquierda a derecha: Joachim Ante, Nicholas Francis y David Helgason

…compartían la visión de crear herramientas accesibles para desarrolladores de juegos de todos los niveles. Originalmente, la compañía se llamó ***Over the Edge Entertainment*** (OTEE), pero *en 2007 cambió su nombre a Unity Technologies*.

Con el objetivo de democratizar el desarrollo de juegos, permitiendo que tanto desarrolladores independientes como grandes estudios pudieran crear experiencias interactivas en 2D y 3D. A lo largo de los años, Unity ha evolucionado, ampliando su soporte desde Mac OS X solamente a múltiples plataformas, incluyendo Windows, Linux, iOS, Android y consolas como PlayStation, Xbox y Nintendo Switch. Su facilidad de uso, junto con una comunidad activa y recursos abundantes, ha convertido a Unity en una de las herramientas más populares en la industria del desarrollo de videojuegos. Algunas características:

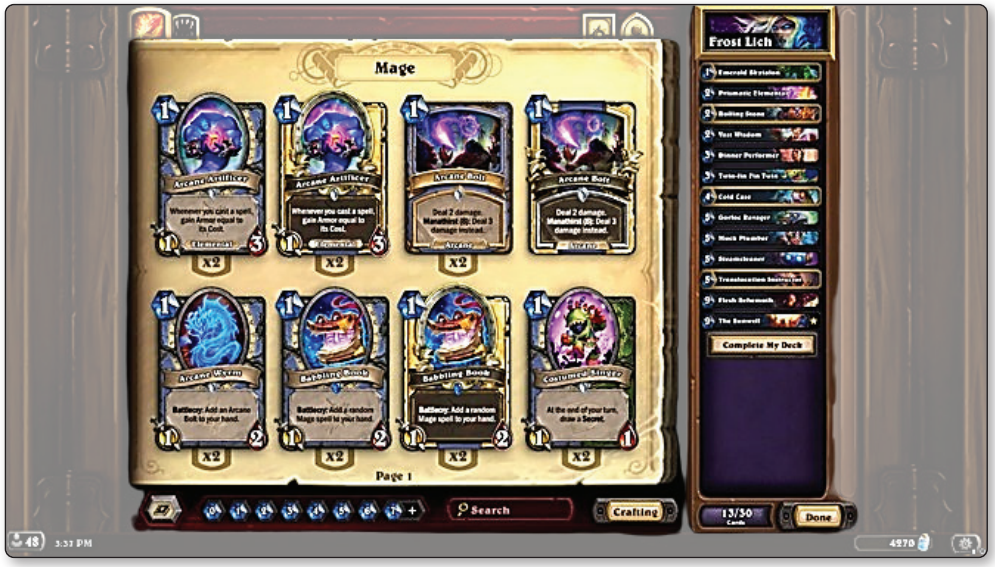
- *Comunidad y ecosistema*: se ha creado una extensa comunidad de desarrolladores y un ecosistema rico en recursos, incluyendo el Asset Store, donde se pueden encontrar assets, herramientas y extensiones creadas por otros usuarios.
- Tecnologías avanzadas: Unity ha incorporado tecnologías como realidad aumentada (AR), realidad virtual (***VR***) y herramientas para simulaciones y visualización en sectores como arquitectura, ingeniería y cine.

Unity ha atraído la atención de importantes *inversores* a lo largo de su historia:

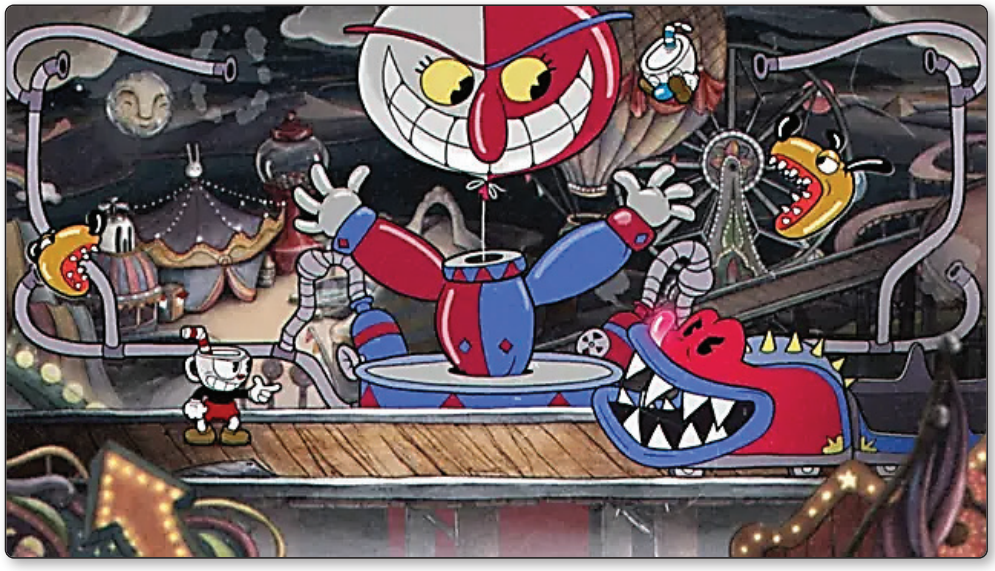
- Sequoia Capital: en 2009, esta firma de capital de riesgo invirtió en Unity, lo que ayudó a impulsar su crecimiento inicial.
- Silver Lake Partners: en 2017, Unity recibió una inversión de $400 millones, valorando la compañía en aproximadamente $2.600 millones.
- IPO en 2020: Unity Technologies salió a bolsa en septiembre de 2020, cotizando en la Bolsa de Nueva York bajo el símbolo “U”. Esta oferta pública inicial reflejó la confianza del mercado en la empresa y su posición en la industria.

La sucesión de CEOs ha sido: David Helgason (CEO hasta 2014), John Riccitiello (2014-2023), ex-CEO de Electronic Art y marcada por decisiones controvertidas como la introducción del Runtime Fee, Matt Bromberg (2023), que ha eliminado el ***Runtime Fee***.

Juegos famosos desarrollados en Unity:



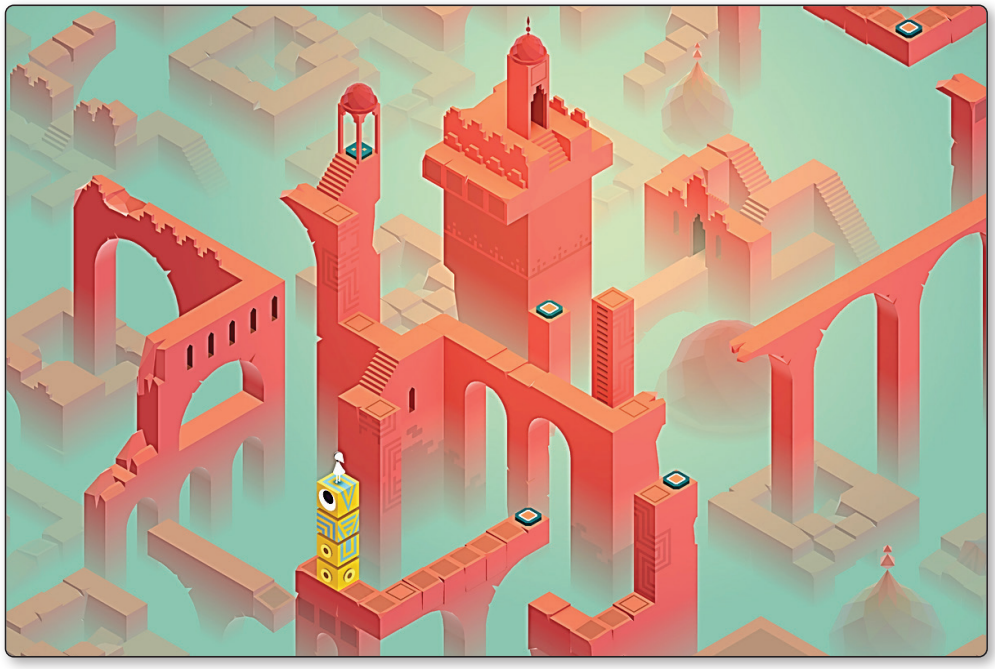

Hearthstone (Blizzard Entertainment)
Un popular juego de cartas en línea basado en el universo de Warcraft.



Cuphead (Studio MDHR)
Un juego de acción y plataformas con un estilo artístico inspirado en los dibujos animados de los años 30.


Among Us (InnerSloth)
Un juego multijugador en línea que ganó gran popularidad entre el público más joven por su divertida dinámica de deducción social.


Cities: Skylines (Colossal Order)
Un simulador de construcción de ciudades famoso por su profundidad y flexibilidad.

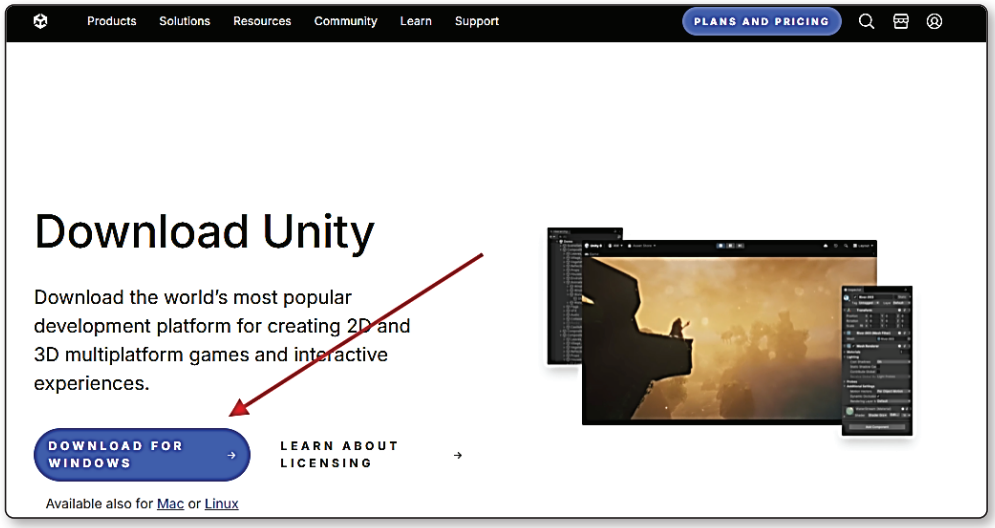


Monument Valley (Ustwo Games)
Un juego de rompecabezas conocido por su diseño artístico y arquitectura imposible.

Subnautica (Unknown Worlds Entertainment)
Un juego de exploración y supervivencia submarina en un mundo alienígena.

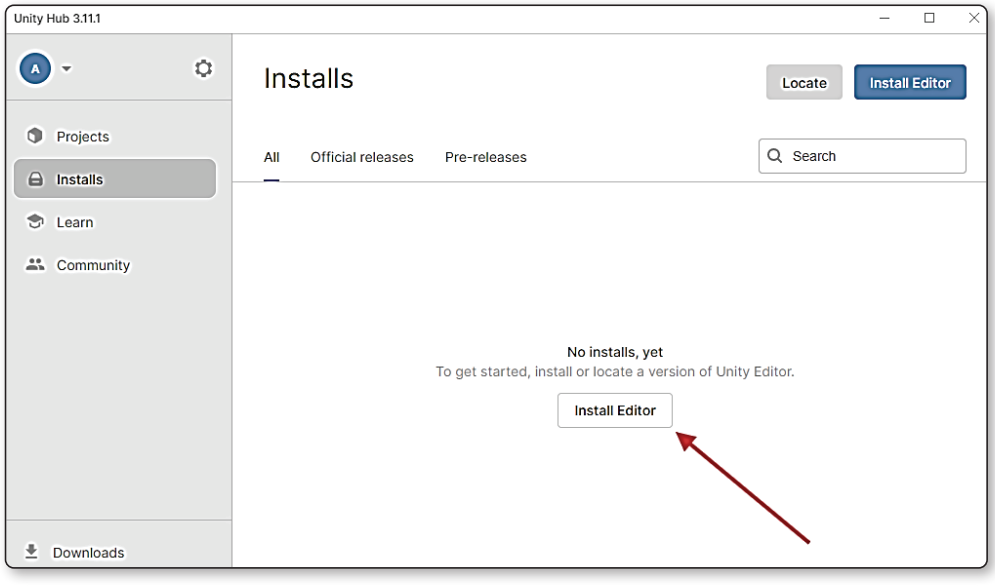
### 1.1.1 Instalación

Visita el sitio web de Unity: dirígete a *https://unity.com/download*.




Descarga Unity Hub

Instala Unity Hub: una vez descargado, abre el instalador y sigue las instrucciones para instalar Unity Hub en tu sistema.

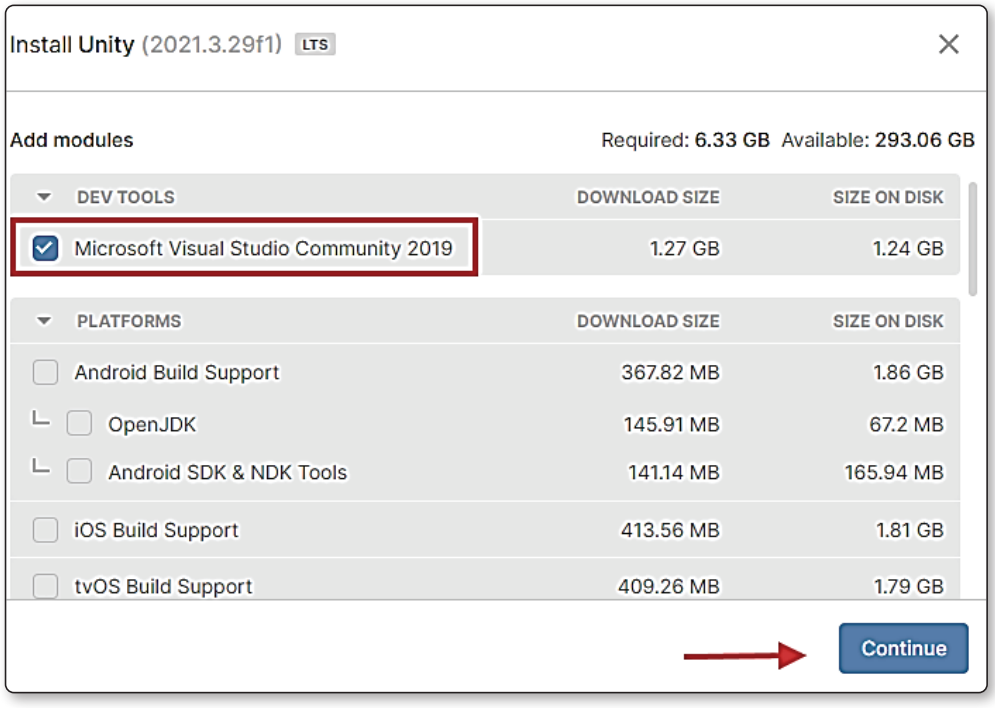



Instala Unity Editor

Abre Unity Hub y ve a la pestaña "Installs". Haz clic en ***Install Editor*** para instalar el editor de Unity.

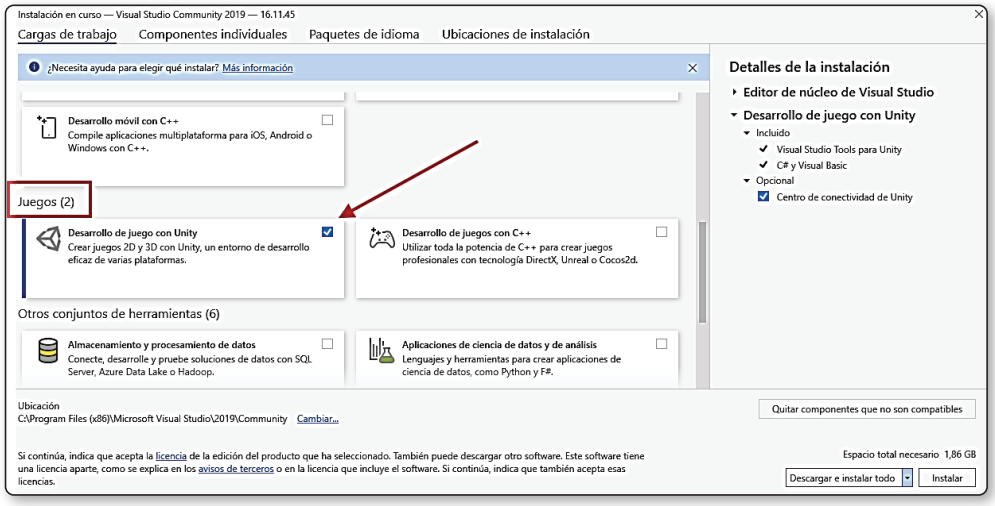
**Sugerencia**

Es completamente habitual tener varias versiones del editor de Unity instaladas en el sistema. La versión en la que haremos los ejemplos es ***2021.3.29f1***. *Es altamente recomendable no cambiar versiones una vez iniciado un proyecto*. Unity es un editor en constante evolución, lo que significa que de una versión a otra puede haber cambios que hagan que nuestro proyecto "se rompa" por lo que lo razonable es permanecer en la versión en la que se han hecho los ejemplos por seguridad.




Instalando Unity Editor & Visual Studio

Cuando te de la opción de instalar ***Visual Studio Community***, los ejemplos se han hecho en su versión 2019, misma recomendación que en el caso anterior respecto a permanecer en la misma versión. Ningún otro paquete es necesario de momento. En los últimos capítulos instalaremos los paquetes necesarios para exportar nuestros proyectos a Android, pero de momento, con marcar Visual Studio es suficiente.

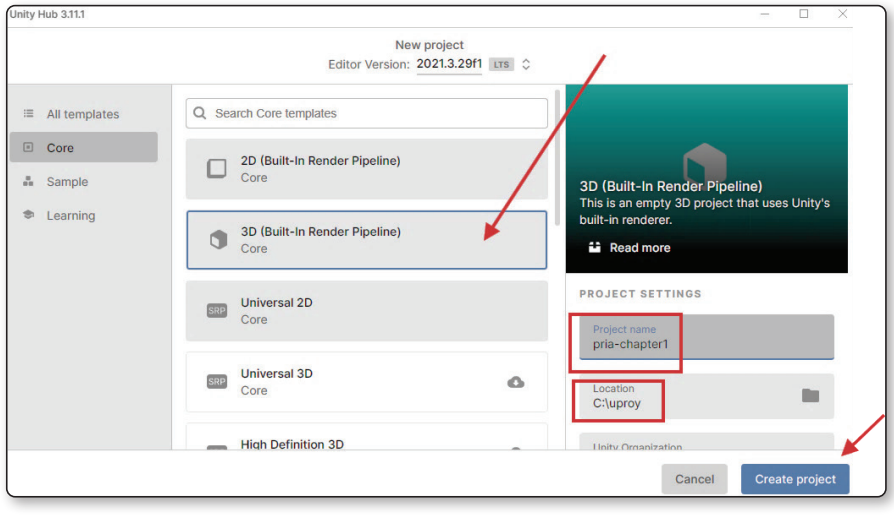



Visual Studio con Unity

Marca cuando se te pregunte la sección de instalación de Visual Studio ***Desarrollo de juego con Unity***

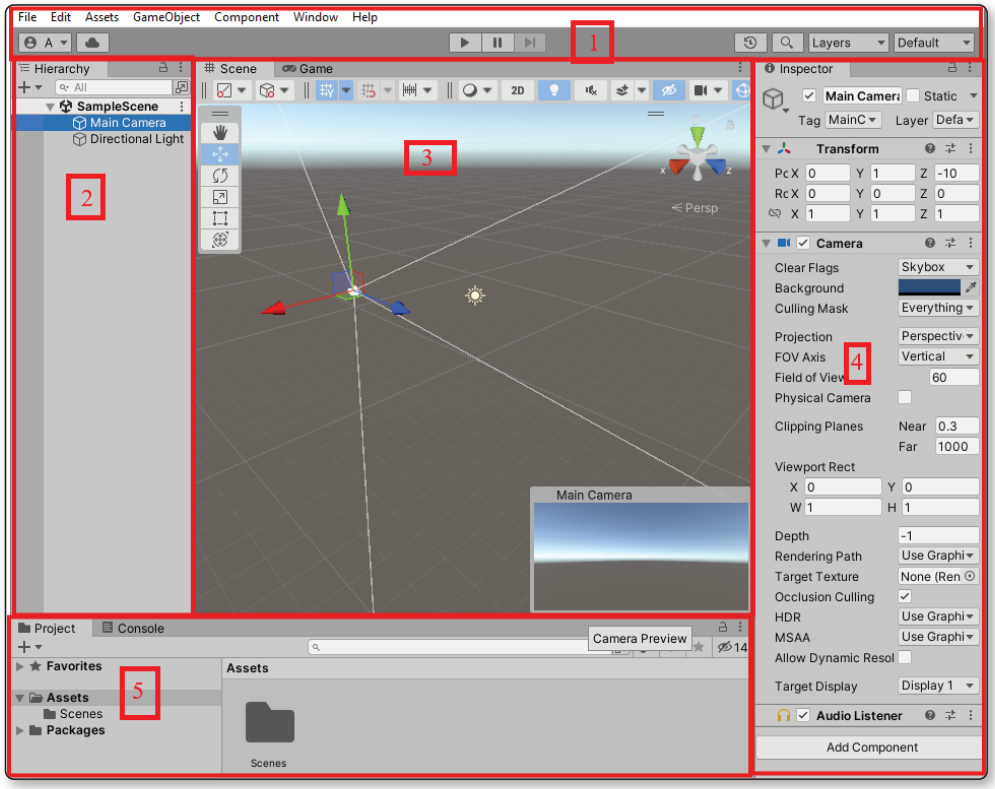
## 1.1.2 Crea un nuevo proyecto

En Unity Hub, ve a la pestaña "Projects" y haz clic en "NEW". Selecciona la plantilla "3D", asigna un nombre a tu proyecto (por ejemplo, "pria-chapter1") y elige la ubicación donde se guardará (por ejemplo, crea la carpeta "c:\uproy").




Creando nuestro primer proyecto

Es posible que la primera vez que arranquemos el editor de Unity el firewall de Windows pida confirmación para permitir el acceso a Internet, lo aceptaremos. Una vez que se crea el nuevo proyecto, se abrirá el editor de Unity. Familiarízate con las diferentes ventanas y paneles:




Navegando por el editor de Unity

1. Barra de herramientas: ubicada en la parte superior, contiene herramientas para manipular objetos y controlar la reproducción del juego.

2. Ventana de jerarquía: muestra todos los objetos presentes en la escena actual.

3. Vista de escena y vista de juego:

   - Vista de escena: donde construyes y organizas tus objetos 3D y 2D.
   - Vista de juego: muestra cómo se verá el juego cuando se ejecute.

4. Inspector: permite ver y editar las propiedades de los objetos seleccionados.

5. Ventana de proyecto: muestra todos los assets y recursos del proyecto. Y consola: donde se muestran mensajes, advertencias y errores generados por tus scripts.

**ⓘ Sugerencia**

Si deseas cambiar entre los modos claro y oscuro, ve al menú Unity

Preferencias > General y cambia el Tema del Editor:

Panel general de preferencias de Unity

## 1.2 USAR C# EN UNITY

De aquí en adelante, es importante pensar en Unity y C# como entidades simbióticas. Unity es el motor donde crearás scripts y GameObjects, pero la programación real tiene lugar en otro programa llamado Visual Studio.

### Trabajando con scripts en C#

Un script en C# es un archivo de texto en C# donde escribirás código en el lenguaje de programación C#. Estos scripts se pueden utilizar en Unity para controlar un personaje en el juego con tu teclado hasta animar objetos en tu nivel.

Hay varias maneras de crear scripts en C# desde el editor:

- Selecciona **Assets > Create > C# Script**
- Justo debajo de la pestaña **Project**, selecciona el icono + y elige **C# Script**
- Haz clic derecho en la carpeta **Assets** en la pestaña **Project** y selecciona **Create > C# Script** en el menú emergente.
- Selecciona cualquier GameObject en la ventana **Hierarchy** y haz clic en **Add Component > New Script**

De ahora en adelante, cada vez que se te indique crear un script en C#, utiliza el método que prefieras.

Además de los scripts en C#, se pueden crear otros recursos y objetos en el editor utilizando los métodos anteriores. No voy a señalar cada una de estas variaciones cada vez que creemos algo nuevo, así que solo ten en mente las opciones.

Por el bien de la organización, vamos a guardar nuestros diferentes activos y scripts dentro de sus propias carpetas con nombre.

1. Selecciona **Assets > Create > Folder** y nómbrala ***Scripts***:


Carpeta Scripts

2. Haz doble clic en la carpeta ***Scripts*** y crea un nuevo script en C#. Por defecto, el script se llamará ***NewBehaviourScript***, pero verás que el nombre del archivo está resaltado, por lo que tendrás la opción de renombrarlo de inmediato. Escribe ***Chapter1*** y presiona **Enter**:


Creando Chapter1 Script

3. Puedes usar el pequeño control deslizante en la parte inferior derecha de la pestaña **Project** para cambiar el tamaño en el que se muestran tus archivos.

Así que acabas de crear una subcarpeta llamada ***Scripts***, como se muestra en la captura de pantalla anterior. Dentro de esa carpeta principal, creaste un script en C# llamado ***Chapter1.cs*** (el tipo de archivo .cs significa **C-Sharp**(C#), por si te lo preguntabas), que ahora está guardado como parte de los activos de nuestro proyecto.

Todo lo que queda por hacer es abrirlo en Visual Studio.

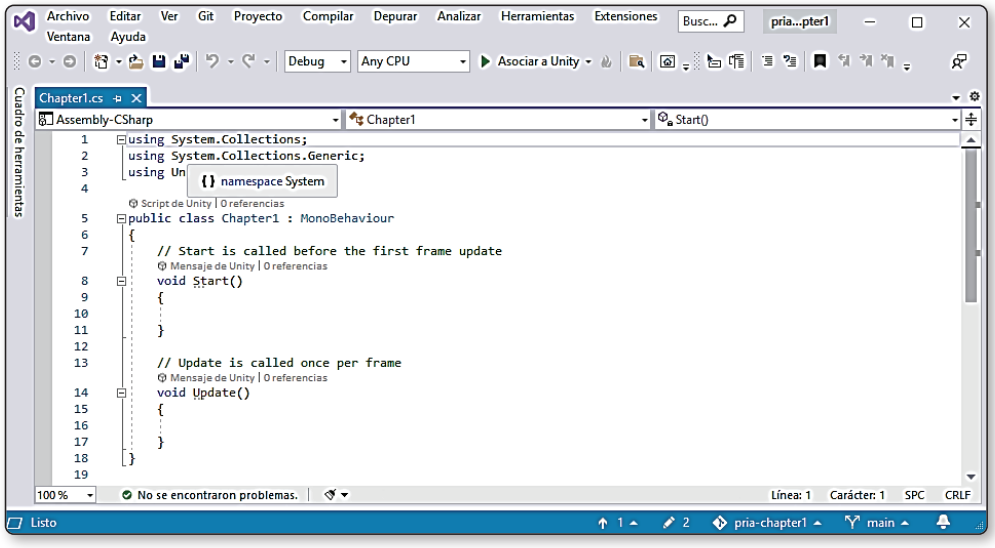
### Introducción al editor de Visual Studio

Visual Studio 2019 se debería de haber instalado junto con el Editor de Unity y se abrirá automáticamente cuando hagas doble clic en cualquier script de C# dentro del editor.

### Abrir un archivo de C#

Unity se sincronizará con Visual Studio la primera vez que abras un archivo. La manera más sencilla de hacerlo es seleccionando el script desde la pestaña **Project**. Sigue los siguientes pasos:

1. Haz doble clic en ***Chapter1.cs***, lo que abrirá el archivo de C# en Visual Studio:




Script Chapter1.cs en Visual Studio

### Cuidado con los desajustes de nombres

Un error común que cometen los nuevos programadores es el desajuste de nombres de archivo, algo que podemos ilustrar en la línea 5 de Visual Studio:

```
public class Chapter1: MonoBehaviour
```

El *nombre de la clase* ***Chapter1*** es el mismo que el nombre del archivo ***Chapter1.cs***. Este es un requisito esencial. Está bien si aún no sabes qué es una clase. Lo importante que debes recordar es que, en Unity, el nombre del archivo y el de la clase deben coincidir. Si usas C# fuera de Unity, el nombre del archivo y de la clase no tienen que coincidir.

Cuando creas un archivo de script en C# en Unity, el nombre del archivo en la pestaña **Project** ya está en modo de edición, listo para renombrarse. Es una buena costumbre renombrarlo en ese momento. Si renombramos el script más tarde, el nombre del archivo y el de la clase no coincidiría.

Si renombramos el archivo en un momento posterior, el nombre del archivo cambiaría, pero la línea 5 sería la siguiente:

```
public class NewBehaviourScript : MonoBehaviour
```

Si accidentalmente haces esto, no es el fin del mundo. Todo lo que necesitas hacer es hacer clic derecho sobre el script en la pestaña **Projects** y elegir **Rename**.

### Sincronización de archivos C#

Unity y Visual Studio se comunican entre sí para sincronizar su contenido. Esto significa que, si añades, eliminas o cambias un archivo de script en una aplicación, la otra aplicación verá los cambios automáticamente.

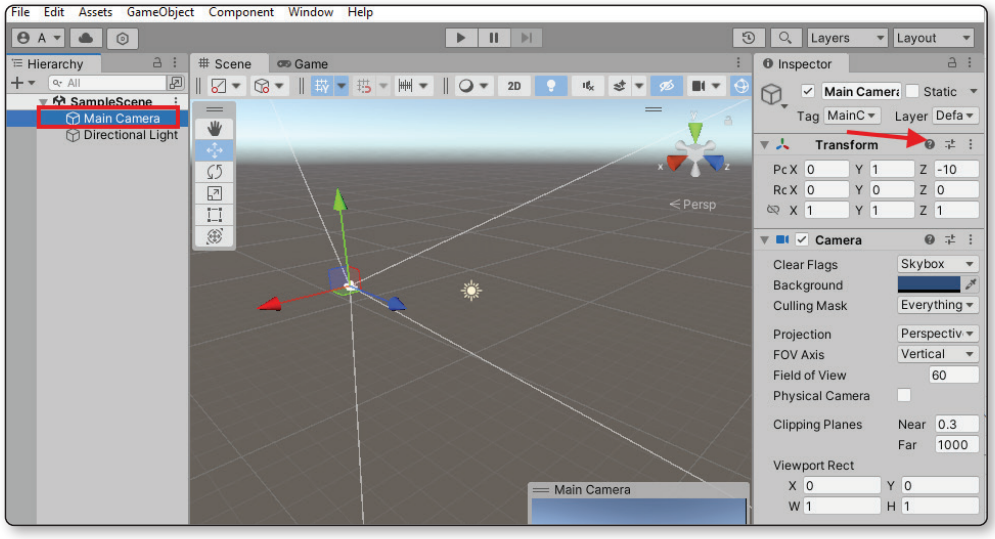
¿Qué sucede si por cualquier motivo la sincronización no parece estar funcionando correctamente? Si te encuentras en esta situación, selecciona el script problemático en Unity, haz clic derecho y selecciona **Refresh**.

## 1.3 EXPLORANDO LA DOCUMENTACIÓN

Una vez que comiences a escribir scripts en serio, usarás la documentación de Unity con frecuencia, por lo que es útil saber cómo acceder a ella desde el principio. El **Reference Manual** te dará una *visión general* de un componente, mientras que los *ejemplos específicos* de programación se encuentran en la **Scripting Reference**.

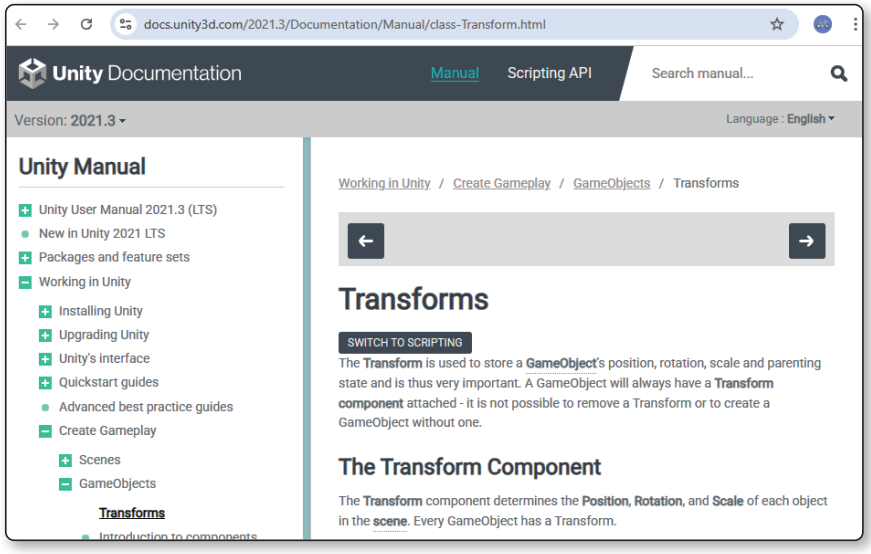
Cada ***GameObject*** (un elemento en la ventana ***Hierarchy***) en una escena tiene un componente ***Transform*** que controla su posición, rotación y escala. Vamos a buscar el componente ***Transform*** de la cámara en el **Reference Manual**:

1. En la pestaña ***Hierarchy***, selecciona el ***Main Camera GameObject***.
2. Ve a la pestaña ***Inspector*** y haz clic en el icono de información (signo de interrogación, ?) en la parte superior derecha del componente ***Transform***:




Main Camera GameObject seleccionado en el Inspector

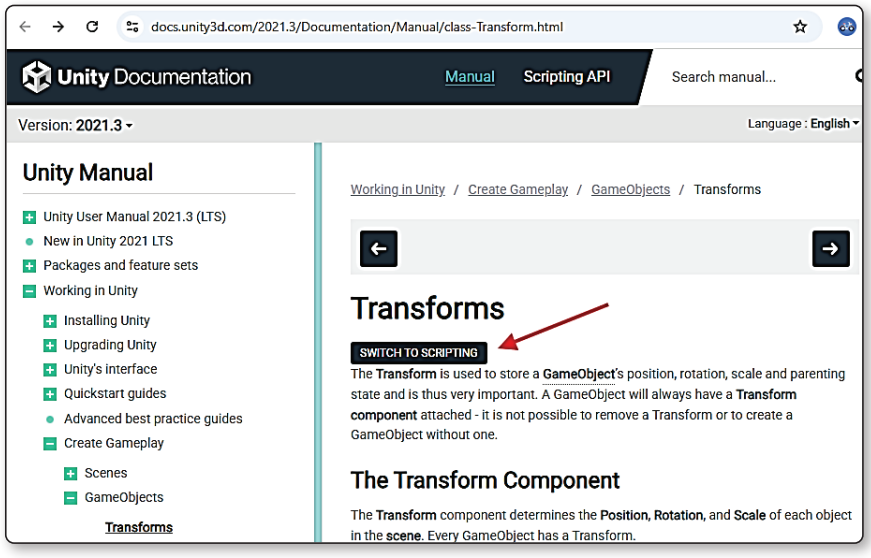
3. Verás que se abre un navegador web en la página del ***Transform*** en el **Reference Manual** (Unity Technologies, 2025):




Manual de Referencia de Unity

Así que tenemos el **Reference Manual** abierto, pero ¿qué pasa si queremos ejemplos de codificación concretos relacionados con el componente ***Transform***? Necesitamos consultar la **Scripting. Reference**:

1. Haz clic en el enlace **SWITCH TO SCRIPTING** debajo del nombre del componente o clase (en este caso, ***Transform***):




Manual - SWITCH TO SCRIPTING

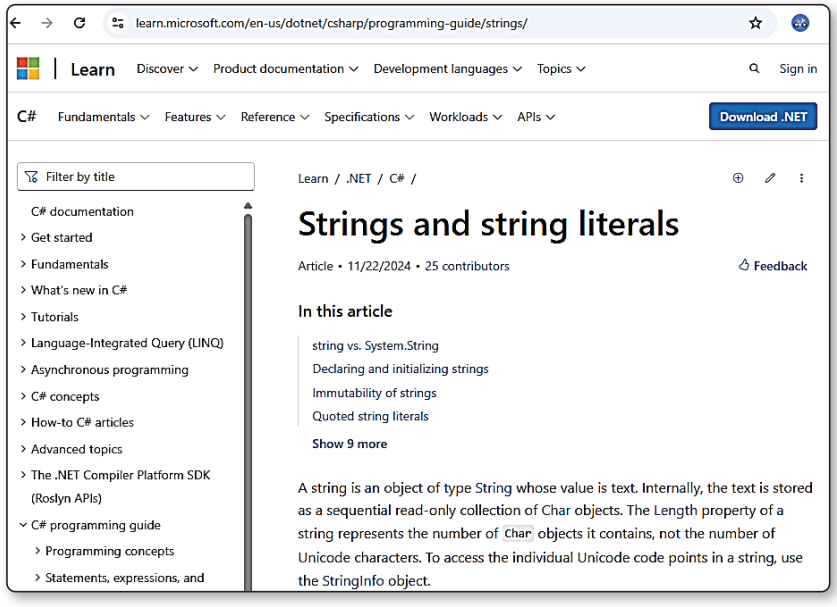
2. Al hacerlo, el **Reference Manual** cambia automáticamente a la **Scripting Reference**. También hay una opción (**SWITCH TO MANUAL**) para volver al **Reference Manual**.

### Localizando recursos de C#

Ahora que hemos cubierto los recursos de Unity, echemos un vistazo a algunos recursos de C# de Microsoft. Para empezar, la documentación de Microsoft Learn en *https://docs.microsoft.com/en-us/dotnet/csharp* (Microsoft, 2025) tiene una gran cantidad de tutoriales, guías rápidas y artículos instructivos. También puedes encontrar excelentes resúmenes de temas individuales de C# en: *https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/index* (Microsoft, 2025).

Por ejemplo, sigamos el enlace de la guía de programación y busquemos la clase ***String*** en C#. Haz una de las siguientes acciones:

- Ingresa ***Strings*** en la barra de búsqueda en la esquina superior izquierda de la página web.
- Desplázate hasta ***Language Sections*** y haz clic directamente en el enlace ***Strings***:




Navegando por la guía de referencia de C# de Microsoft

A diferencia de la documentación de Unity, la información de referencia y scripting de C# está toda unificada.

## 1.4 FUNDAMENTOS DE C#

Nos centraremos en los siguientes temas a lo largo de este capítulo:

- Definir Variables y Métodos.
- Estructuras de control while-do, if-then-else.
- Introducción a las Clases.

### 1.4.1 Variables y métodos

Una variable es una pequeña sección de la memoria que contiene un valor asignado. Cada variable rastrea dónde se almacena su información (esto se llama dirección de memoria), su valor y su tipo, por ejemplo: números, cadenas de texto o listas.

*https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/variables* (Microsoft, 2021)

#### 1.4.1.1 CREANDO TU PRIMERA VARIABLE

Vamos a crear una variable en el script ***Chapter1*** que habíamos creado.

1. Haz doble clic en ***Chapter1.cs*** desde la ventana ***Project*** en Unity para abrirlo en Visual Studio.

2. Agrega un espacio entre las líneas 6 y 7 e inserta la siguiente línea de código para declarar una nueva variable:

```
public int PrimeraVariable = 44;
```

3. Dentro del método ***Start***, agrega dos ***Debug Logs*** para imprimir los siguientes cálculos:

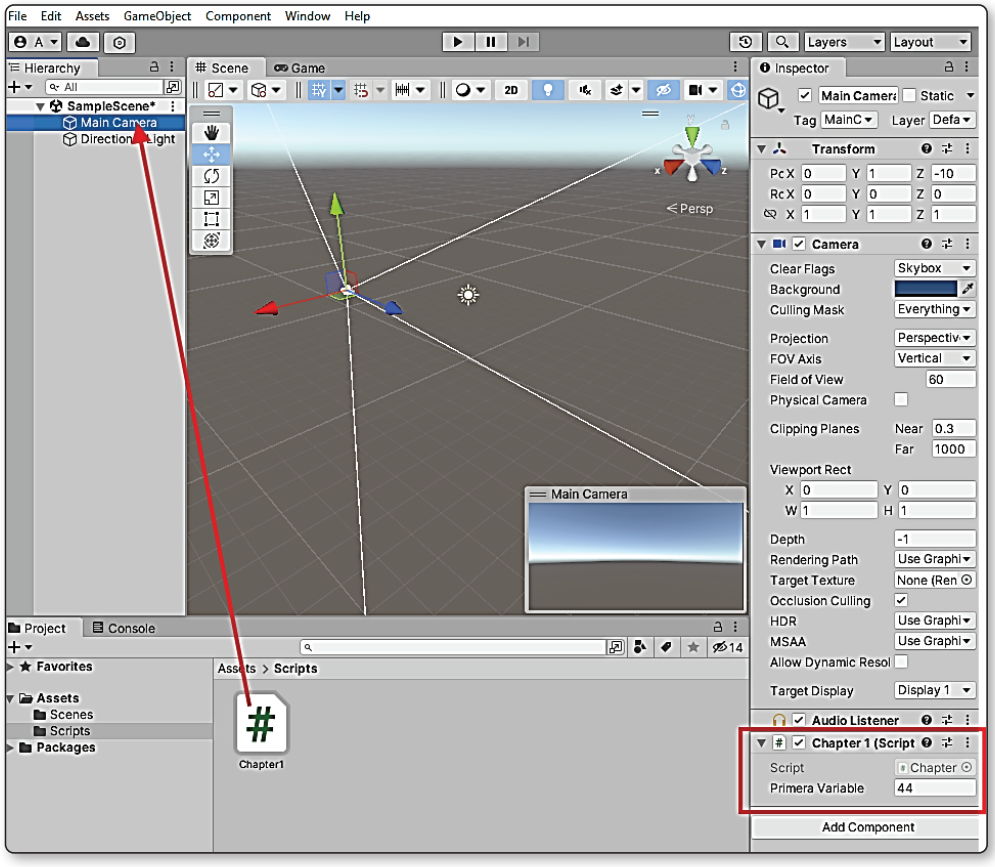
```
Debug.Log(PrimeraVariable + 1);
```

Es importante hacer notar que las variables **public** aparecen en el ***Inspector*** de Unity, mientras que las ***private*** no. Para finalizar, guarda el archivo en VS utilizando **Editor > File > Save**. Unity solo reconoce los cambios guardados en el editor.

Para que los scripts se ejecuten en Unity, deben estar adjuntos a ***GameObjects*** en la escena. Unity considera que todo en tu juego es un ***GameObject***: luces, personajes, objetos, edificios, todo.

Por defecto, la escena tiene una cámara para renderizar la escena y una luz direccional para iluminar, así que vamos a adjuntar ***Chapter1*** a la cámara por simplicidad.

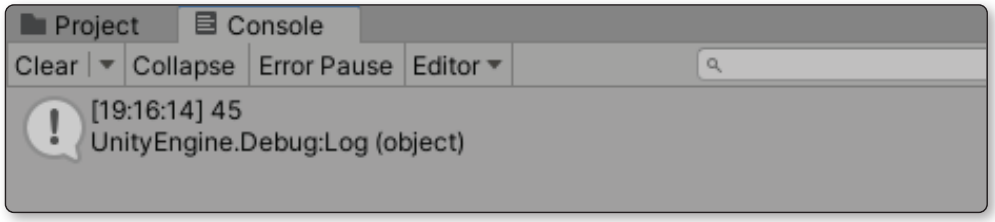
1. Arrastra y suelta ***Chapter1.cs*** sobre la ***Main Camera***.




Arrastrando Chapter1.cs al GameObject Main Camera

2. Selecciona la ***Main Camera*** para que aparezca en el panel ***Inspector***, y verifica que el script ***Chapter1.cs*** esté correctamente adjunto.

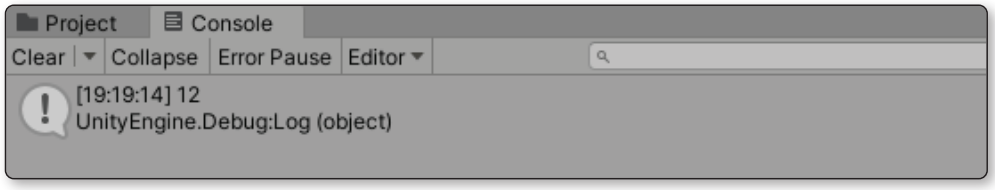
3. Haz clic en **Play** y observa el resultado en el panel ***Console***:




Salida por Consola

Ahora:

1. Detén el juego haciendo clic en el botón **Play** si la escena sigue corriendo.
2. Cambia ***PrimeraVariable*** a 14 en el panel ***Inspector*** y vuelve a ejecutar la escena, observando la nueva salida en el panel Console:




Segunda Salida por Consola

La segunda salida ahora es 15 porque cambiamos el valor en el ***Inspector***.

### 1.4.1.2 ENTENDIENDO LOS MÉTODOS

Debemos aclarar un pequeño punto de terminología. En el mundo de la programación, comúnmente verás que los términos *método* y *función* se usan indistintamente, especialmente en Unity.

Un método es un *bloque de código* que se ejecuta cuando se invoca al método por su nombre. Los métodos pueden recibir *argumentos* (también llamados *parámetros*), que se pueden usar dentro del *ámbito* del método.

*https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods*

1. Abre Chapter1 en Visual Studio.
2. Agrega una nueva *variable.*

```
public int SegundaVariable = 1;
```

3. Agrega un nuevo *método* que sume ***PrimeraVariable*** y ***SegundaVariable*** y muestre el resultado:

```
void Suma() {
  Debug.Log(PrimeraVariable  + SegundaVariable);
}
```

4. *Invoca al nuevo método* dentro de ***Start*** con la siguiente línea:

```
void Start() {
  Suma();
}
```

5. Guarda el archivo y luego regresa y haz clic en **Play** en Unity para ver la nueva salida en la ***Console***. Prueba diferentes valores de variable en el panel ***Inspector*** para ver esto en acción.

### 1.4.2 Clases y comentarios

Una clase es un contenedor tanto de variables como de métodos.

- **Conceptualmente**, una clase almacena *información relacionada*, acciones y comportamientos dentro de un solo contenedor.
- **Técnicamente**, las clases son estructuras de datos que pueden contener variables, métodos y todo esto puede referenciarse cuando *se instancia un objeto de la clase*.

*https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/classes*

Ya hemos estado trabajando con clases. Por defecto, cada script creado en Unity es una clase, lo que puedes ver desde la palabra clave **class**:

```
public class Chapter1: MonoBehaviour {
  ...
}
```

***MonoBehaviour*** significa simplemente que esta clase puede adjuntarse a un ***GameObject*** en la escena de Unity, y las dos llaves marcan los límites de la clase: *cualquier código dentro de esas llaves pertenece a esa clase*.

Los términos ***script*** y ***clase*** se usan de manera indistinta en Unity, pero por aclararnos, llamaremos los archivos de C# como ***scripts*** si están adjuntos a ***GameObjects***, y como ***clases*** si son independientes.

### 1.4.2.1 UN EJEMPLO DE CLASE SIN HERENCIA DE *MONOBEHAVIOUR*

Pensemos en una oficina de correos local. Es un entorno que tiene propiedades, como una dirección física (una variable), y la capacidad de ejecutar acciones, como enviar tu correo (métodos).

```
public class Restaurant {
    // Variables
    public string name = "Delicious Eats";
    public string location = "123 Tasty Ave.";

    // Métodos
    public void PrepareFood(){}
    public void ServeFood(){}
}
```

La clave aquí es que, cuando la información y los comportamientos siguen un plano predefinido, se vuelven posibles acciones complejas y la comunicación entre clases. Por ejemplo, si tuviéramos otra clase que quisiera preparar comida a través de nuestra clase ***Restaurant***, simplemente podría llamar a la función ***PrepareFood*** de la siguiente manera:

```
Restaurant re = new Restaurant();
re.PrepareFood();
```

O podrías usarla para buscar la dirección de la oficina de correos, para saber dónde enviar tus cartas:

```
var direccion = re.location;
```

Esta notación se llama *notación de puntos.*

*Comunicación entre clases:* cualquier variable, método u otro tipo de dato dentro de una clase puede accederse con la notación de puntos. Cada vez que una clase necesita información sobre otra clase o quiere ejecutar uno de sus métodos, se utiliza la notación de puntos. La notación de puntos a veces se denomina el "operador de acceso".

### 1.4.2.2 TRABAJANDO CON COMENTARIOS

```
// Este es un comentario de una sola línea
```

Las líneas que comienzan con dos barras diagonales (sin espacios) no son compiladas, por lo que puedes usarlas tanto como necesites para explicar tu código a otros o a tu futuro yo.

Los comentarios de *una sola línea* sólo se aplican a una línea de código. *Si quieres comentarios de varias líneas*, deberás usar una barra inclinada y un asterisco, (/* y */ como caracteres de apertura y cierre, respectivamente) alrededor del texto del comentario:

```
/* este es un
comentario de varias líneas */
```

#### Añadiendo comentarios

Abre ***Chapter1*** y añade tres barras diagonales encima del método ***Suma()***:

```
/// <summary>
 ///
 /// </summary>
 void Suma() {
     Debug.Log(PrimeraVariable + SegundaVariable);
 }
```

Comentario de triple línea generado automáticamente

Deberías ver un comentario de tres líneas con un espacio para una descripción del método generado por Visual Studio, entre dos etiquetas **<summary>**. Puedes, por supuesto, cambiar el texto o añadir nuevas líneas presionando **Enter**, como lo harías en un documento de texto; hay muchas más etiquetas disponibles para xmldoc, es recomendable echar un vistazo además al menos a **<param>** y **<return>**.

*https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/xmldoc/recommended-tags*

Lo útil de estos comentarios detallados es evidente cuando quieres saber algo sobre un método que has escrito. Si has usado un comentario de triple barra diagonal, todo lo que necesitas hacer es colocar el cursor sobre el nombre del método en cualquier lugar donde se llame dentro de una clase o script, y Visual Studio mostrará tu resumen.

## 1.4.3 Estructuras de control

Las sentencias condicionales **if-else** y **switch** te permiten especificar qué bloque de código debe de ejecutarse basándose en una o más condiciones.

### 1.4.3.1 IF-ELSE

Es la forma más común de tomar decisiones en el código. La idea básica es: si mi condición se cumple, ejecuta este bloque de código; opcionalmente, se puede agregar una declaración **else** para ejecutar otro bloque de código en caso de que la condición NO se cumpla.

Ejemplo: sustituye tu script en ***Chapter1.cs*** por este:

```
public class Chapter1: MonoBehaviour {
    public bool hasEggs= true;

    void Start() {
        if(hasEggs) {
            Debug.Log("Vamos a hacer una tortilla");
        } else {
            Debug.Log("No puedo hacer una tortilla");
        }
    }
}
```

Otro ejemplo:

1. Abre el archivo ***Chapter1*** y añade una nueva variable **public bool** llamada ***Monedas***. Asigna su valor entre 1 y 100. Crea un método público sin valor de retorno, llamado ***Asalto.*** Dentro de la nueva función, añade una declaración **if** para verificar si ***Monedas*** es mayor que 50, y escribe un mensaje en la consola si esto es verdadero. Añade una declaración

**else-if** para verificar si ***Monedas*** es menor que 15, con un mensaje de depuración diferente. Añade una declaración **else** sin condición y un mensaje final por defecto. Guarda el archivo y haz clic en **Play.**

```
public int Monedas = 23;

 public void Asalto() {
   if(Monedas > 50) {
     Debug.Log("¡Cuántas monedas!");
   } else if (Monedas < 15) {
     Debug.Log("No hay mucho para el saco");
   } else {
     Debug.Log("Ni mucho ni poco");
   }
 }
```

Dado que 23 no es menor que 15 ni mayor que 50, ninguna de las condiciones anteriores se cumple, por lo que la declaración **else** se ejecuta y se muestra el tercer mensaje de depuración.

### Anidación de declaraciones

Las declaraciones **if-else** pueden anidarse unas dentro de otras.

```
public class Chapter1 : MonoBehaviour {
    public bool tengo_huevos = true;
    public string complemento = "patatas";

    void Start() {
        if(tengo_huevos ) {
            if(complemento == "patatas") {
                Debug.Log("Hacer tortilla de patatas");
            }
        } else {
            Debug.Log("Hacer tortilla francesa");
        }
    }
}
```

Si la primera declaración **if** se evalúa como falsa, el código saltaría a la declaración **else** y su mensaje de depuración. Si la segunda declaración **if** se evalúa como falsa, no se imprime nada porque no hay una declaración **else**.

#### 1.4.3.2 SWITCH

Cuando tienes más de tres o cuatro acciones anidadas el código se convierte en difícil de seguir y actualizar.

Las declaraciones **switch** aceptan expresiones y nos permiten escribir acciones para cada resultado posible. Requieren los siguientes elementos:

- La palabra clave **switch** seguida de un par de paréntesis que contengan su condición.
- Un par de llaves.
- Una **case** para cada valor que termina con dos puntos, líneas individuales de código o métodos, seguidas de la palabra clave **break** y un punto y coma.
- Un **default** que termina con dos puntos, líneas individuales de código o métodos, seguidas de la palabra clave **break** y un punto y coma.

Por ejemplo:

```
public string Complemento = "fideos";

 void Start() {
   Cocina();
 }

 public void Cocina() {
   switch(Complemento) {
     case "patatas": Debug.Log("Cocinar patatas fritas"); break;
     case "fideos":  Debug.Log("Hacer sopa");     break;
     default:        Debug.Log("Hoy pasaremos hambre");  break;
   }
}
```

Dado que ***Complemento*** tiene como valor ***"fideos"***, la declaración **switch** ejecuta el segundo caso e imprime su registro de depuración: "Hacer sopa".

## 1.4.4 Collections

Hasta ahora, solo hemos necesitado variables para almacenar un valor, pero hay muchas situaciones en las que se requerirá un grupo de valores. Las ***Collections*** en C# incluyen **arrays**, **list** y **dictionaries**; cada uno tiene sus puntos fuertes.

### 1.4.4.1 ARRAYS

Piensa en ellos como contenedores para un grupo de valores, llamados elementos, cada uno de los cuales puede accederse o modificado individualmente. También son referidos normalmente como arreglos o vectores.

- Los ***arrays*** pueden almacenar cualquier tipo de valor; *todos los elementos deben ser del mismo tipo*.
- La *longitud*, o el número de elementos que un arreglo puede tener, se establece cuando se crea y *no se puede modificar después*.

Veamos un ejemplo donde necesitamos almacenar los tres puntajes más altos de nuestro juego:

```
int[] TopPuntos = new int[3];
```

El array ***TopPuntos*** almacenará tres elementos de tipo entero. Como no añadimos valores iniciales, cada uno de los tres valores en ***TopPuntos*** tienen inicialmente el valor 0.

Puedes asignar valores directamente a un arreglo cuando se crea.

```
int[] TopPuntos = new int[] {323, 57, 4};
```

Siempre puedes verificar la longitud de un arreglo, es decir, cuántos elementos contiene, con la propiedad ***Length***:

```
TopPuntos.Length;
```

### Indexación

Cada elemento del arreglo se almacena en el orden en que se asigna, lo que se conoce como su índice. Los arreglos tienen índice cero, lo que significa que el orden de los elementos comienza en 0 en lugar de 1.

En ***TopPuntos***, el primer entero, 323, está ubicado en el índice 0, 57 en el índice 1 y 4 en el índice 2:

```
// El valor de score se establece en 323
int score = TopPuntos[0];
```

También se puede usar para modificar directamente un valor:

```
TopPuntos[1] = 55;
```

Los ejemplos de arreglos anteriores solo contienen un elemento por índice, por lo que son unidimensionales. Si quisiéramos que un arreglo contuviera, por ejemplo, una matriz de 3x2, sería:

```
int[,] MiMatriz3x2 = new int[3,2] {
 {5,-64},
 {11,7},
 {300,12}
};
```

## 1.4.4.2 LISTS

Las listas pueden cambiar de tamaño dinámicamente, lo que significa que puedes agregar, eliminar y modificar elementos. Este tipo de flexibilidad hace que las listas sean una de las ***Collections*** más populares.

- Las listas pueden almacenar cualquier tipo de valor y, al igual que los arrays, *todos los elementos deben ser del mismo tipo*.
- Las listas *SÍ pueden cambiar de tamaño dinámicamente* (en tiempo de ejecución).
- Las listas tienen una serie de métodos integrados que facilitan la manipulación de los elementos, como **Add**, **Remove** y **Sort**.

Por ejemplo, para crear una lista que almacene puntuaciones de jugadores, el código sería:

```
List<int> TopPuntos = new List<int>();
```

Esta declaración crea una lista vacía que puede almacenar enteros, debido al operador diamante <> (llamado así porque si lo miras de lado parece un diamante) ***<int>***. Ahora, veamos algunas operaciones básicas:

### Agregar elementos a una lista

Para agregar un elemento a una lista, utiliza el método ***Add***. Siguiendo con nuestro ejemplo de puntajes de jugadores, podríamos agregar un puntaje con el siguiente código:

```
TopPuntos.Add(323);
TopPuntos.Add(57);
TopPuntos.Add(4);
```

### Eliminar elementos de una lista

Si necesitas eliminar un elemento de la lista, puedes usar el método ***Remove***. Por ejemplo, para eliminar el puntaje de 323:

```
TopPuntos.Remove(323);
```

### Acceder a elementos de una lista

Puedes acceder a los elementos de una lista utilizando su índice. Las listas también son indexadas desde cero:

```
int puntos = TopPuntos[0]; // Obtiene el primer puntaje
```

### Otras operaciones

Las listas vienen con una amplia variedad de métodos útiles, como ***Sort***, que organiza los elementos en orden ascendente:

```
TopPuntos.Sort();
```

Otra operación común es verificar si un elemento está en la lista con ***Contains***:

```
bool tieneLaPuntuacion = TopPuntos.Contains(57);
```

### 1.4.4.3 DICTIONARIES

Los diccionarios son otro tipo de colección en C# que almacenan pares clave-valor. Piensa en un diccionario como una tabla donde cada valor se asocia con una *clave única*. Esto permite un acceso a los valores mediante sus claves, en lugar de depender de un índice numérico como en los arrays o las listas.

Por ejemplo, para crear un diccionario que almacene nombres de jugadores y sus puntajes, el código sería:

```
Dictionary<string, int> TopPuntos = new Dictionary<string, int>();
```

#### Agregar elementos a un diccionario

Para agregar un par clave-valor a un diccionario, utiliza el método `Add`. Siguiendo con el ejemplo de puntajes de jugadores:

```
TopPuntos.Add("Juan", 323);
TopPuntos.Add("Maria", 57);
TopPuntos.Add("Fer", 4);
```

#### Acceder a valores en un diccionario

Para acceder a un valor en un diccionario, puedes usar la clave:

```
int puntos = TopPuntos["Fer"];
```

#### Eliminar elementos de un diccionario

Para eliminar un elemento de un diccionario, utiliza el método `Remove` con la clave asociada al elemento:

```
TopPuntos.Remove("Juan");
```

### Otras operaciones con diccionarios

Puedes verificar si una clave existe en el diccionario usando el método ***ContainsKey***:

```
bool estaM = TopPuntos.ContainsKey("Maria");
```

### Resumiendo las Collections

Los **Arrays** son útiles cuando sabes que los datos no cambiarán, pero son menos flexibles. Las **List** ofrecen más flexibilidad al permitir agregar y eliminar elementos dinámicamente. Finalmente, los **Dictionaries** son ideales para cuando necesitas asociar valores con claves únicas y realizar búsquedas rápidas. Aunque hay más tipos de colecciones, como BitArray, HashSet, Queue<T>, Stack<T> y otros, para nuestros propósitos nos valdrá con los mostrados.

## 1.4.5 Bucles

Solemos necesitar recorrer una ***Collection*** elemento a elemento. Esto se llama iteración, y C# proporciona varios tipos de sentencias que nos permiten recorrer una colección.

### 1.4.5.1 FOR

El bucle **for** se usa comúnmente cuando un bloque de código necesita ejecutarse un número determinado de veces antes de que el programa continúe.

Veamos un ejemplo práctico con la lista ***Equipo***

```
List<string> Equipo= new List<string>() { "John",  "Ned", "Bart" };
int n = Equipo.Count;

for (int i = 0; i < n; i++) {
  Debug.LogFormat("Índice: {0} - {1}", i, Equipo[i]);
}
```

Repasemos cómo funciona este bucle:

- Se establece como una variable local de tipo **int** llamada ***i*** con un valor inicial de 0.
- Se asegura de que solo se ejecute otra vez si ***i*** es menor que el número de elementos en de ***Equipo***, (arrays ***Length***. listas ***Count***).
- ***i*** se incrementa en 1 cada vez que se ejecuta el bucle con el operador ++.
- Dentro del bucle **for**, imprimimos el índice y el elemento de la lista de ese índice.

### 1.4.5.2 FOREACH

Los bucles **foreach** toman cada elemento de una colección y lo almacenan en una variable local, haciéndola accesible dentro de la sentencia.

Veamos el ejemplo de la lista ***QuestPartyMembers*** y hagamos una lista de asistencia para cada uno de sus elementos:

```
List<string> Equipo= new List<string>() { "John",  "Ned", "Bart" };
foreach(string miembro in Equipo) {
    Debug.LogFormat("{0} listo", miembro);
}
```

Se crea una variable local, llamada ***miembro***, para contener cada elemento a medida que el bucle se repite.

### 1.4.5.3 WHILE

Un ejemplo de una simple cuenta atrás:

```
public int RocketLaunch = 10;

void Start()
{
    CountDown();
}

public void CountDown()
{
    while(RocketLaunch > 0)
    {
        Debug.LogFormat("Cuenta atrás {0}", RocketLaunch );
```

```
            RocketLaunch--;
        }
        Debug.Log("¡Lanzamiento!");
    }
```

Con ***RocketLaunch*** comenzando en 10, el bucle **while** se ejecutará 10 veces. Durante cada iteración, se mostrará el mensaje de depuración "Cuenta atrás" seguido del número de la variable ***RocketLaunch*** y se restará uno a ***RocketLaunch***.

## 1.5 MANOS A LA OBRA EN UNITY

El juego en el que trabajaremos durante el resto de este capítulo está diseñado para que adquieras unos conocimientos básicos del editor de Unity. Es bastante simple y no requerirá algo tan detallado como un ***GDD*** (Game Design Document) o TDD (Technical Design Document). En su lugar, crearemos un ***One-Pager*** para realizar un seguimiento de nuestros objetivos y establecer algunos antecedentes del proyecto.

**Concepto**

Prototipo enfocado en evitar enemigos y recolectar ítems de salud, con un poco de FPS(First Person Shooter) como complemento.

**Jugabilidad**

Utilizar la línea de visión para mantenerse un paso adelante de los enemigos que patrullan y recolectar los objetos necesarios. El combate consistirá en disparar proyectiles a los enemigos. Los enemigos nos descubrirán por proximidad o por ataques, lo que desencadenará una respuesta por parte de ellos.

**Interfaz**

El control para el movimiento utilizará las teclas WASD o las flechas, y la cámara se moverá con el jugador. La mecánica de disparo usará la barra espaciadora, y la recolección de objetos funcionará a través de colisiones con los objetos. El HUD (Head Up Display) simple mostrará la barra de salud.

**Estilo**

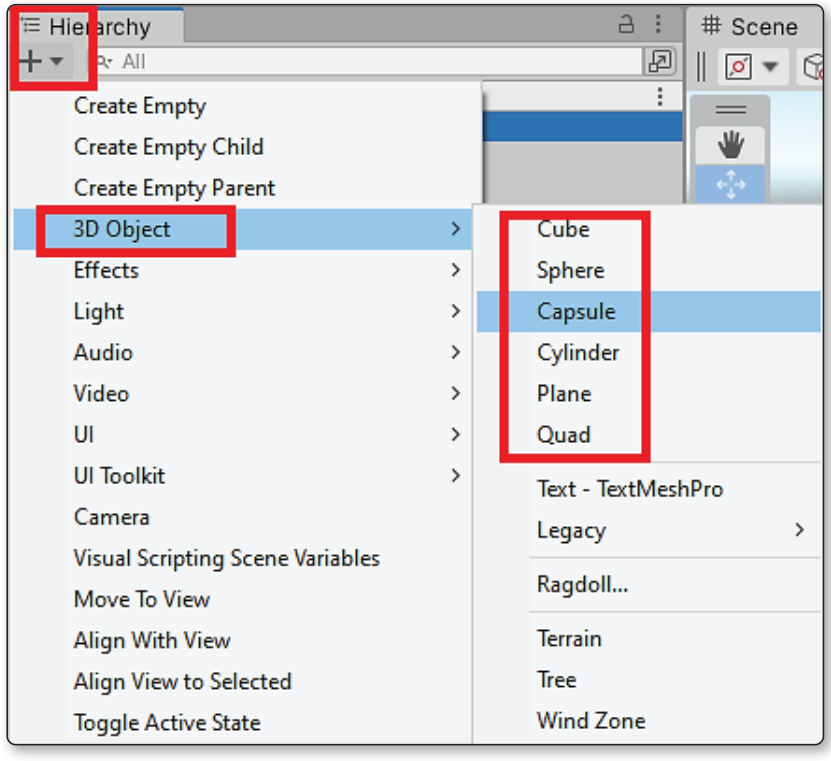
La arena y los personajes consistirá en GameObjects primitivos.

One-Pager

### 1.5.1 Diseño del juego, construcción de un nivel

Con Unity, puedes usar **formas 3D básicas** para crear entornos simples, la herramienta más avanzada **ProBuilder**, o una combinación de ambas. Incluso puedes importar modelos 3D de otros programas, como ***Blender***, para utilizarlos como objetos en tus escenas. También puedes utilizar **Assets** de la ***Unity Asset Store***. Sin embargo, como nuestro objetivo es familiarizarnos primero con el entorno básico, nos quedaremos con un escenario simple, fácil de recorrer, pero con algunos rincones para esconderse. Utilizarás primitivas, que son formas de objetos base proporcionadas por Unity, aprendiendo a crearlas, escalarlas y posicionarlas en una escena.

Si abres Unity, puedes ir al panel **Hierarchy** y hacer clic en **+ > 3D Object**, y verás todas las opciones disponibles. Solo la mitad de estas son primitivas o formas comunes, como se indica en la siguiente captura de pantalla:

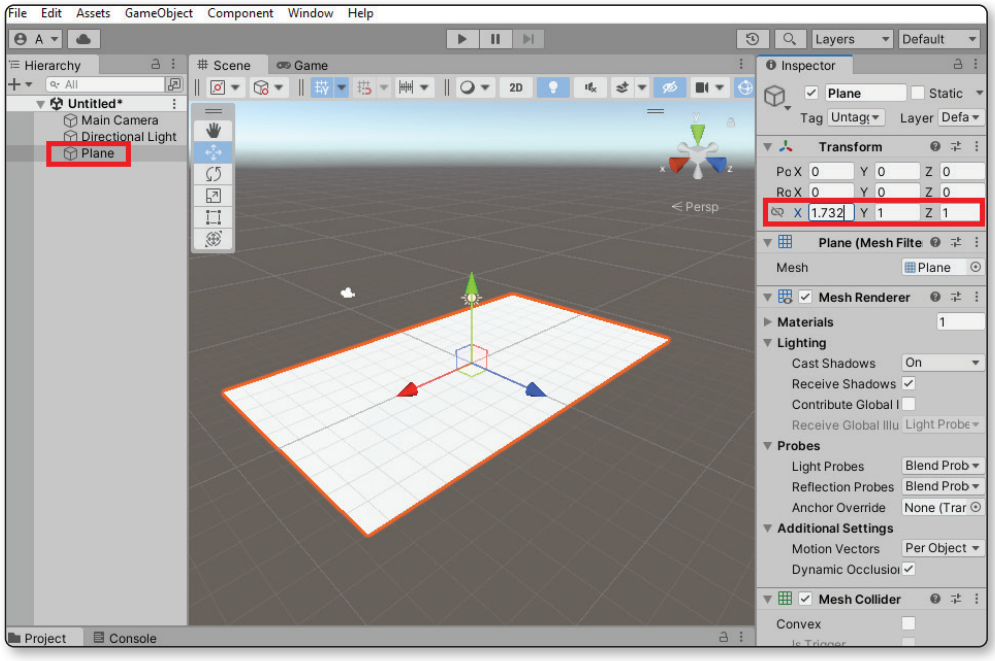



Hierarchy con la opción de 3D Object seleccionada

### Creación de un plano de suelo

Lo primero: para moverse debes tener un suelo debajo de ti. Así que comencemos creando un plano de suelo para nuestra arena siguiendo estos pasos:

- En el panel **Hierarchy**, haz clic en **+ > 3D Object > Plane**.
- En el menú desplegable ***Transform***, cambia la escala en X a 1.732, deja Y y Z a 1



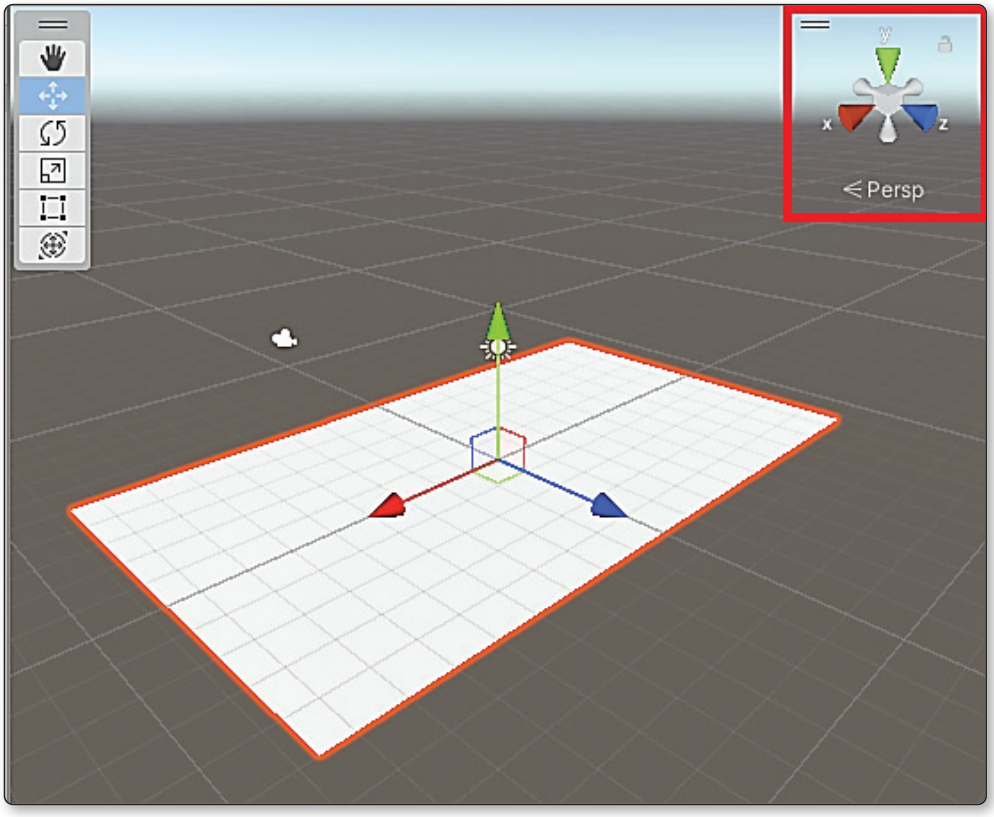

Editor de Unity con un plane

- Si la iluminación en tu escena parece más tenue o diferente, selecciona ***Directional Light*** en el panel **Hierarchy** y ajusta el valor de ***Intensity*** a 1.

### Pensando en 3D

Ahora que tenemos nuestro primer objeto en la escena, podemos hablar sobre el espacio 3D, específicamente cómo se comportan la posición, la rotación y la escala de un objeto en tres dimensiones. En la esquina superior derecha del panel **Scene**, verás un icono geométrico con los ejes X, Y y Z marcados en rojo, verde y

azul, respectivamente. Todos los ***GameObjects*** en la escena mostrarán sus flechas de eje cuando estén seleccionados en la ventana **Hierarchy**.



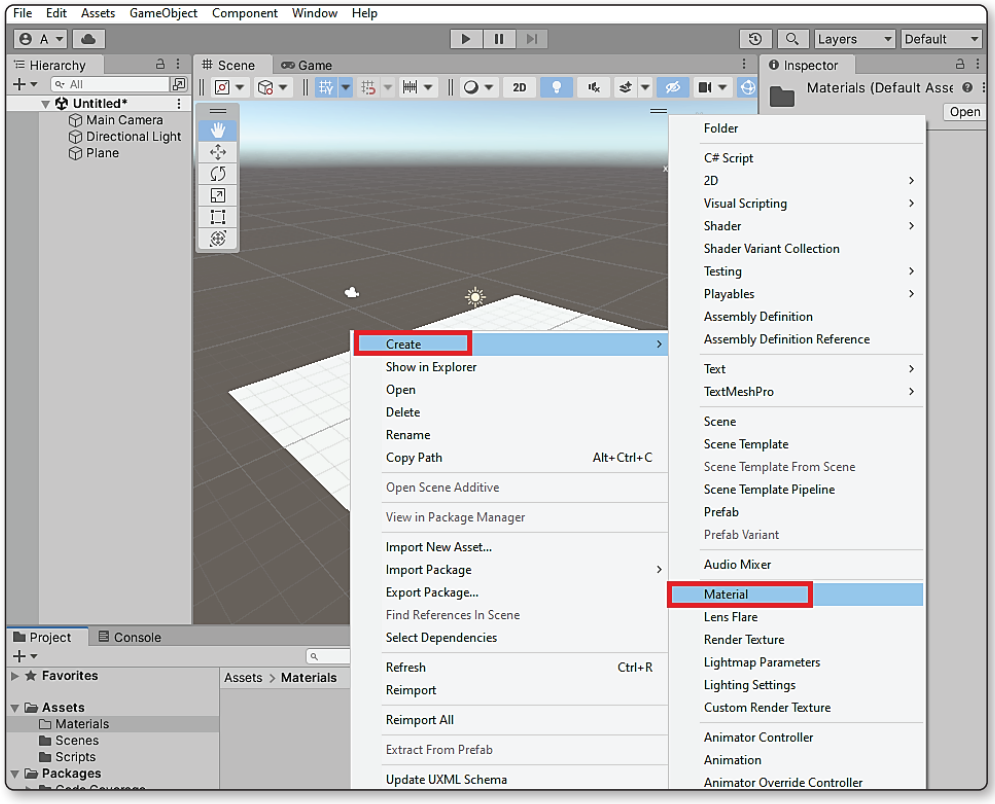

Vista de la escena con el gizmo de orientación

## Materiales

Podemos usar **Materials** para darle un poco de vida al nivel. Los materiales se encargan de configurar propiedades del ***GameObject***, como el color y la textura. El material se pasa al ***Shader***, que usa este último para renderizar las propiedades del material en la pantalla. Piensa en los ***Shaders*** como responsables de combinar los datos de iluminación y textura en una representación de cómo se verá el material. Para cambiar el color de un objeto, necesitamos crear un material y arrastrarlo al objeto que queremos modificar.

- Crea una nueva carpeta en el panel **Project** haciendo clic derecho > **Create > Folder**, y nómbrala ***Materials***.

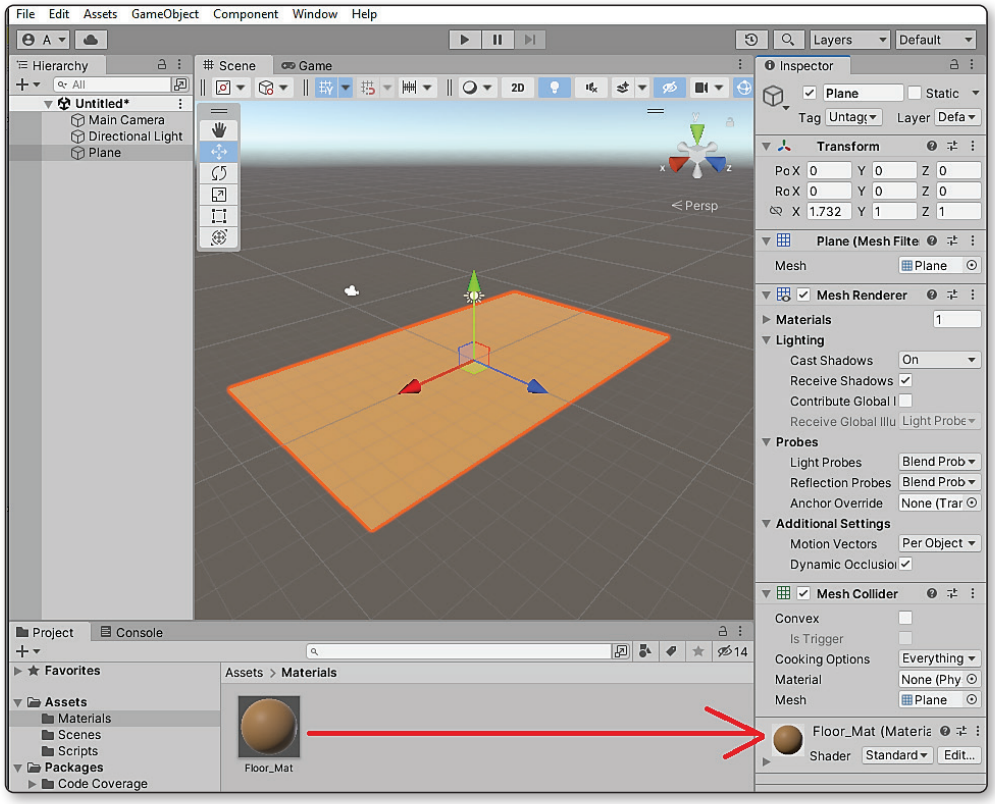
  Dentro de la carpeta ***Materials***, haz clic derecho > ***Create > Material***, y nómbralo ***Floor_Mat***.




Creando un Material

- Selecciona el nuevo material en el panel ***Project*** y mira sus propiedades en el **Inspector**. Haz clic en la caja de color junto a la propiedad ***Albedo***, selecciona tu color en la ventana emergente de selección de color (por ejemplo Marrón #D69E5A) y luego ciérrala.

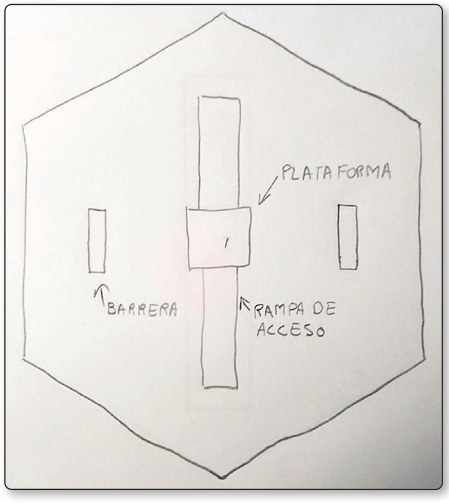
- Arrastra el objeto ***Floor_Mat*** desde el panel **Project** y suéltalo sobre el ***GameObject Plane*** en el panel ***Hierarchy***.




Plane con el color actualizado

## White-boxing

El término *white-boxing* se refiere a la práctica de diseñar niveles usando objetos en cierta posición para tener una idea de cómo quieres que se vea. Esta es una excelente manera de comenzar, especialmente durante las etapas de creación de prototipos. Veamos un boceto:

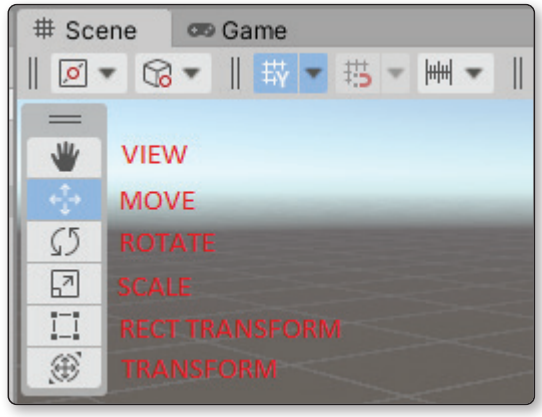



Boceto de la arena

Lo importante es plasmar tus ideas en papel para asentarlas en tu mente antes de ponerte a trabajar en Unity. Hay que familiarizarte con algunas bases del editor de Unity para que el proceso de **white-boxing** sea más fácil.

### Herramientas del editor

Puedes encontrarlas en la esquina superior izquierda del editor de Unity:




Editor Toolbar

Herramientas disponibles en la barra de herramientas:

- **View**: te permite desplazarte y cambiar tu posición en la escena haciendo clic y arrastrando el ratón.
- **Move**: te permite mover objetos a lo largo de los ejes X, Y y Z arrastrando sus respectivas flechas.
- **Rotate**: te permite ajustar la rotación de un objeto girando o arrastrando sus marcadores respectivos.
- **Scale**: te permite modificar la escala de un objeto arrastrándolo hacia los ejes específicos.
- **Rect Transform**: combina la funcionalidad de mover, rotar y escalar en una sola herramienta.
- **Transform**: te da acceso a la posición, rotación y escala de un objeto simultáneamente.

Navegar por la escena:

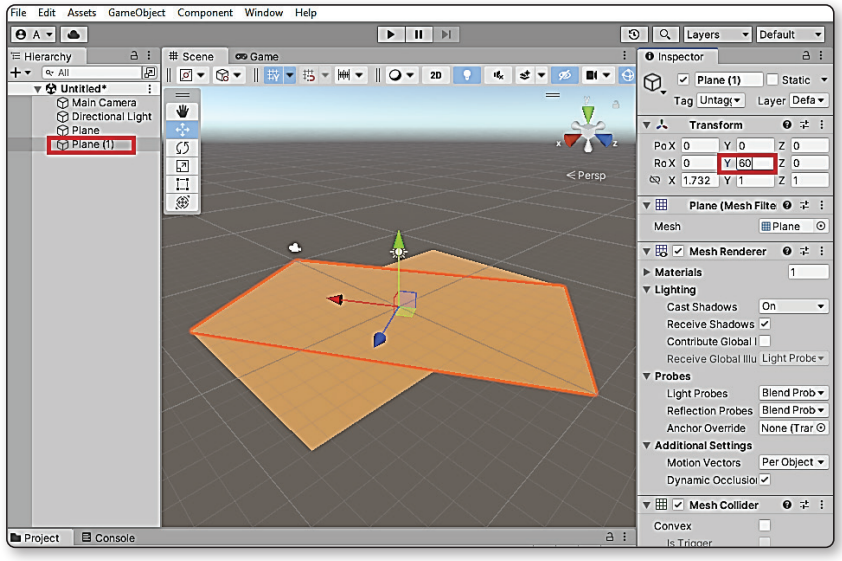
- Para mirar alrededor, mantén presionado el botón derecho del ratón y arrástralo para desplazar la cámara.
- Para moverte mientras usas la cámara, mantén presionado el botón derecho del ratón y utiliza las teclas **W, A, S y D** para moverte hacia adelante, atrás, izquierda y derecha, respectivamente.
- Presiona la tecla **F** para hacer zoom y enfocar un ***GameObject*** seleccionado en el panel **Hierarchy**.

Más información en: *https://docs.unity3d.com/Manual/SceneViewNavigation.html* (Unity Technologies, 2025).

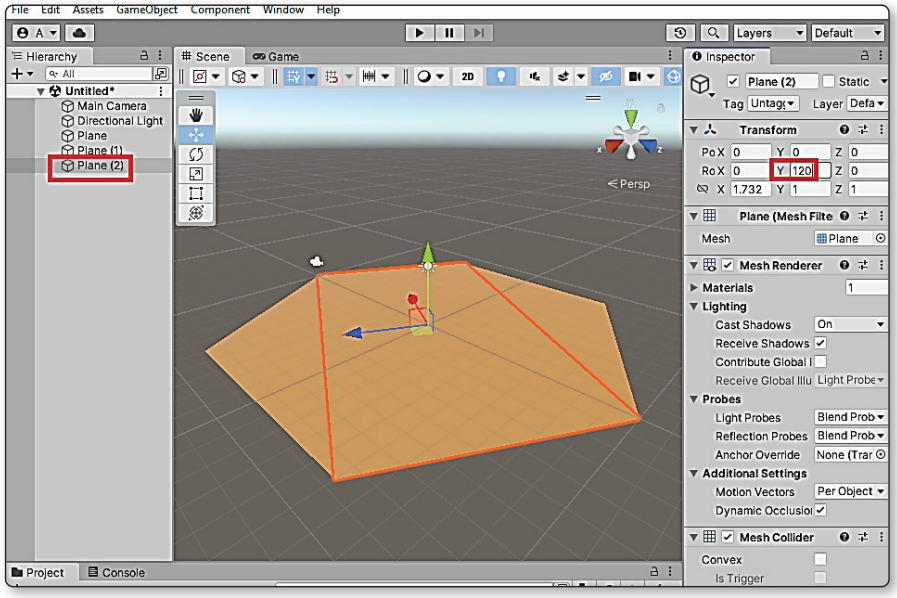
**ⓘ Nota para los interesados en geometría**

¿Por qué el valor 1.732 (√3)? Un hexágono regular contiene 6 triángulos equiláteros. La altura h de ese triángulo es h = ***s*·*sin***(60°) = ***s***·√3⁄2. Dos alturas (una arriba y otra abajo del centro) dan la distancia entre lados opuestos: 2·(***s***·√3⁄2) = ***s***·√3. Por eso, si el lado del hexágono ***s*** vale 1u, el ancho del plano tendrá el valor √3 ≈ 1.732. Tras duplicar el plano y rotarlo ±60°, los tres planos encajan formando un hexágono.

Dado que queremos una arena hexagonal, vamos a triplicar el rectángulo para formar un hexágono, rotando 60 y 120º en el eje Y. Puedes duplicarlo seleccionando el ***Plane*** y pulsando **Ctrl+D**.

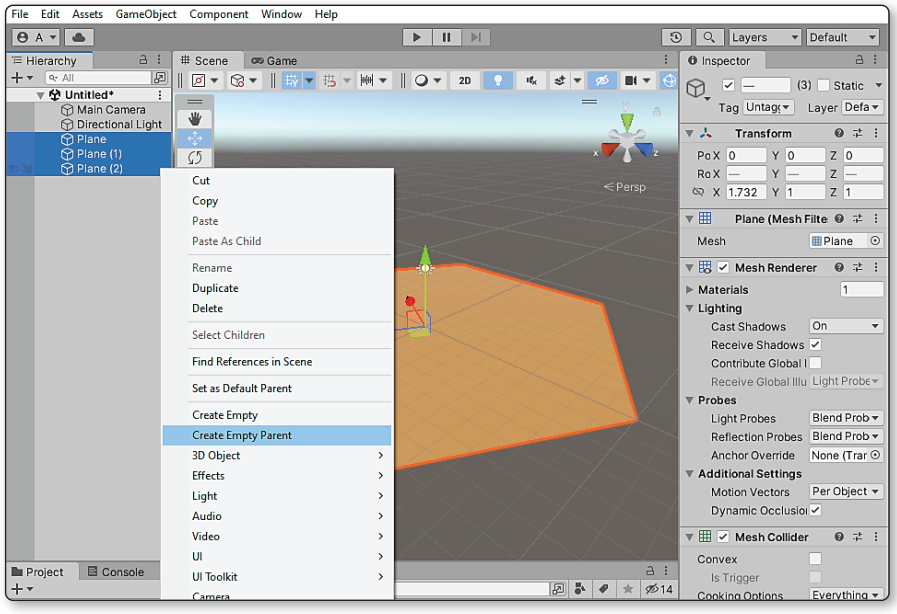



Duplicando un Plane




Formando un hexágono

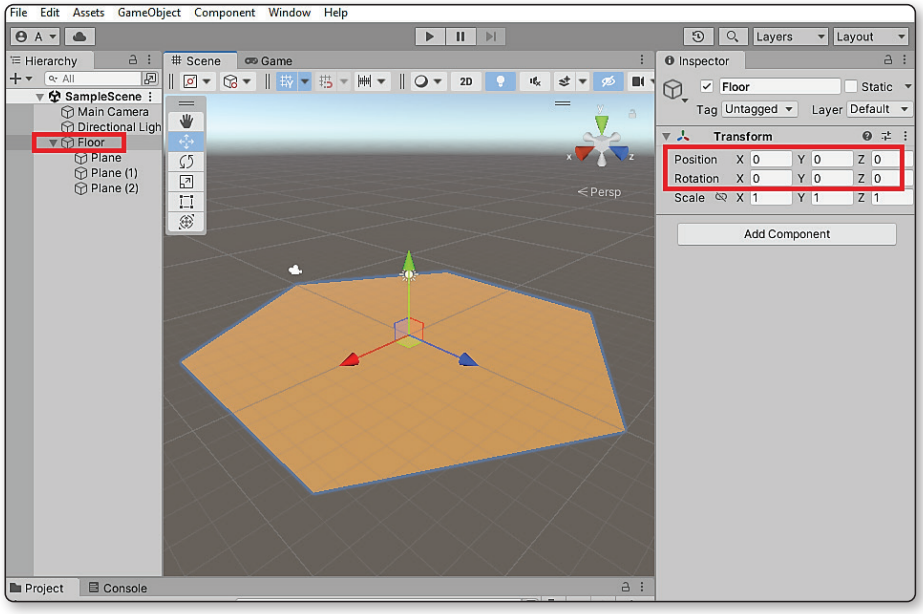
*Vamos a adquirir el hábito de mantener tu jerarquía de objetos organizada*. Podemos agrupar los elementos del suelo Seleccionando los 3 ***Plane*** con **Hierachy -> Create Empty Parent** y llamarlo ***Floor***.




Agrupando planes en el elemento Floor

Es importante configurar las posiciones X, Y y Z del objeto ***Floor*** en 0, ya que las posiciones de los objetos hijos ahora son relativas a la posición del padre. Esto nos lleva a una pregunta interesante: ¿cuáles son los puntos de origen de estas posiciones, rotaciones y escalas que estamos configurando? La respuesta es que dependen del espacio relativo que estemos utilizando, que en ***Unity*** puede ser ***World*** o ***Local***:

- **World space** utiliza un punto de origen establecido en la escena como referencia constante para todos los ***GameObjects***. En Unity, este punto de origen es (0, 0, 0), es decir, 0 en los ejes X, Y y Z.
- **Local space** utiliza el componente ***Transform*** del objeto padre como su origen, cambiando esencialmente la perspectiva de la escena. Unity también establece este origen local en (0, 0, 0). Piensa en esto como si el ***Transform*** del padre fuera el centro del universo, con todo lo demás orbitando en relación con él.




World Space vs Local Space

Es hora de construir paredes alrededor de la arena para tener un área de movimiento confinada.

## Construyendo las paredes

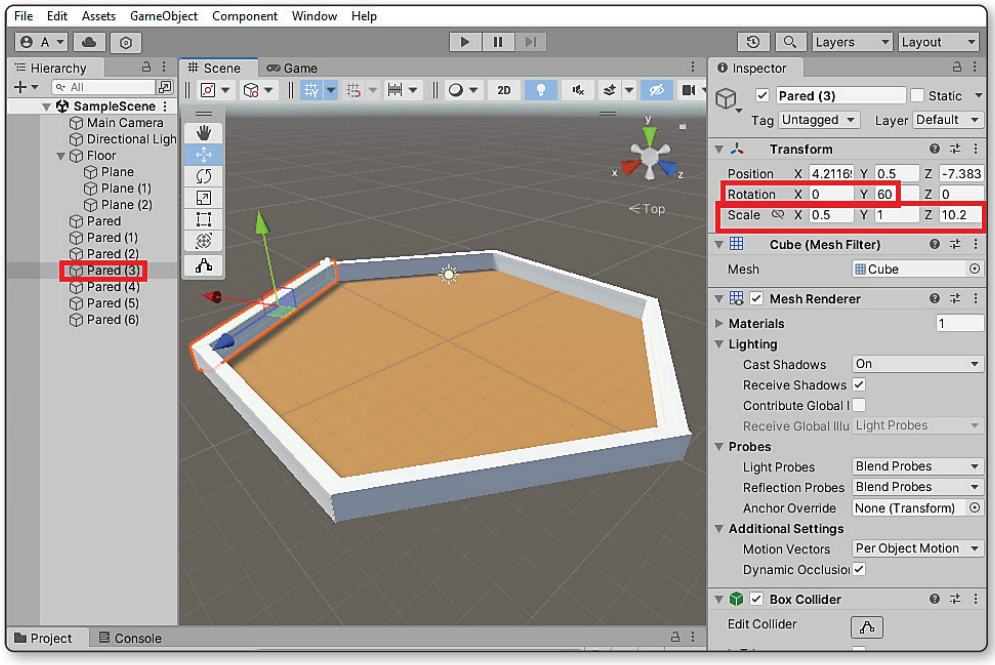
Usando cubos primitivos y la barra de herramientas, posiciona seis paredes alrededor del nivel utilizando las herramientas de Mover, Rotar y Escalar para delimitar la arena principal:

- En el panel **Hierarchy**, selecciona + > **3D Object** > **Cube** para crear la primera pared y nómbrala **Pared**.
- Configura su escala a 0,5 para el eje X, 1 para el eje Y y 10.2 para el eje Z.

**Nota**

Los ***planes*** en Unity (objetos planos que se usan comúnmente como suelos o superficies) tienen una escala diferente a otros objetos 3D. En Unity, un ***plane*** tiene un tamaño base que es 10 veces más grande que otros objetos 3D, como cubos o esferas. Por lo tanto, si escalas un ***plane*** a un tamaño de 1x1.732, su tamaño será equivalente al de otro objeto 3D que tenga un tamaño de 10x17.32.

- Con el objeto ***Pared*** seleccionado en el panel **Hierarchy**, cambia a la herramienta de ***Position*** en la esquina superior izquierda y usa las flechas rojas, verdes y azules para posicionar la pared en el borde del plano del suelo. Si pulsas **Ctrl** mientras arrastras, se irá intentando ajustar a la cuadrícula. Ten en cuenta que las paredes estarán rotadas 60º y 120º en el eje Y en el hexágono.
- Repite los pasos hasta que tengas seis paredes rodeando tu área:




Arena hexagonal con paredes

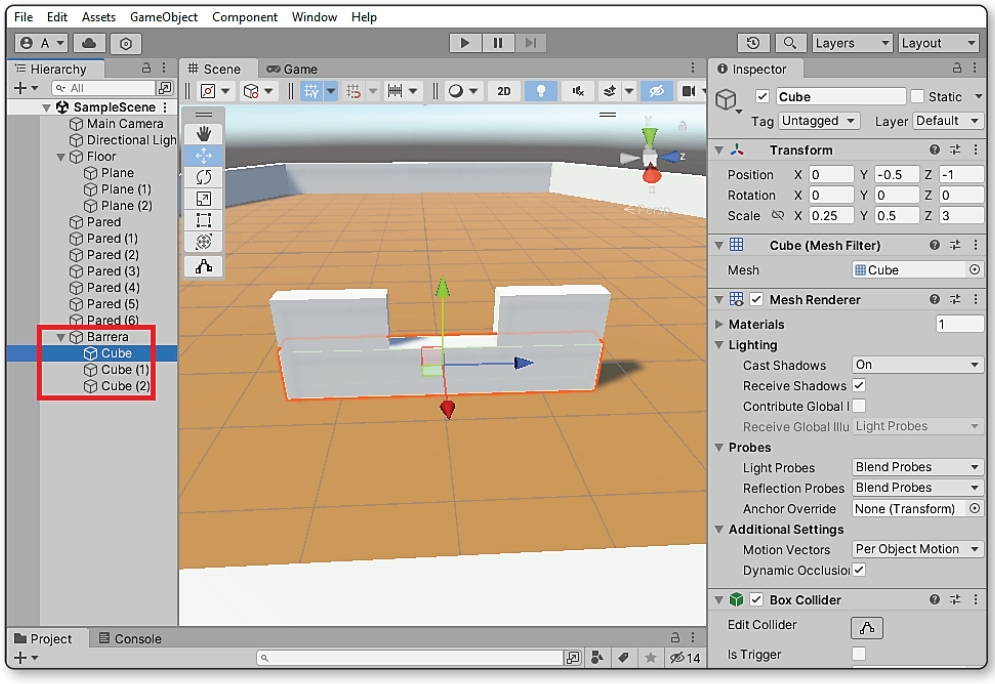
Eso fue un poco de construcción, pero la arena está comenzando a tomar forma.

### 1.5.1.1 TRABAJANDO CON PREFABS

Los ***Prefabs*** son ***GameObjects*** que se pueden guardar y reutilizar con todos sus objetos hijos, componentes, scripts C# y configuraciones de propiedades. Una vez creado, un ***Prefab*** es como una plantilla. Como resultado, cualquier cambio en el ***Prefab*** base también afectará a todas las instancias activas no modificadas en la escena.

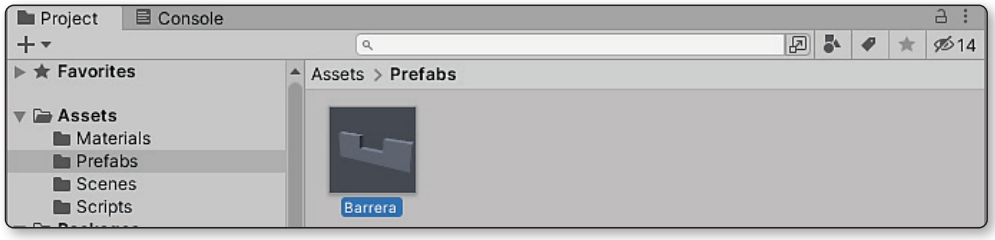
Queremos dos barreras idénticas en dos lados de la arena, son un caso ideal para un ***Prefab***. Creemos la barrera con los siguientes pasos:

- Crea un objeto padre vacío + > **Create Empty** y nómbralo ***Barrera***.
- Crea tres cubos seleccionando + > **3D Object** > **Cube**, luego posiciónalos y escálalos como una base en forma de "U". La base puede ser de 0.25x0.5x3. Coloca los otros dos encima de los extremos de la base de la barrera.




Barrera

- En la carpeta principal ***Assets***, crea una nueva carpeta llamada ***Prefabs*** y arrastra el ***GameObject Barrera*** desde el panel **Hierarchy** a la carpeta ***Prefabs*** en la vista del proyecto.



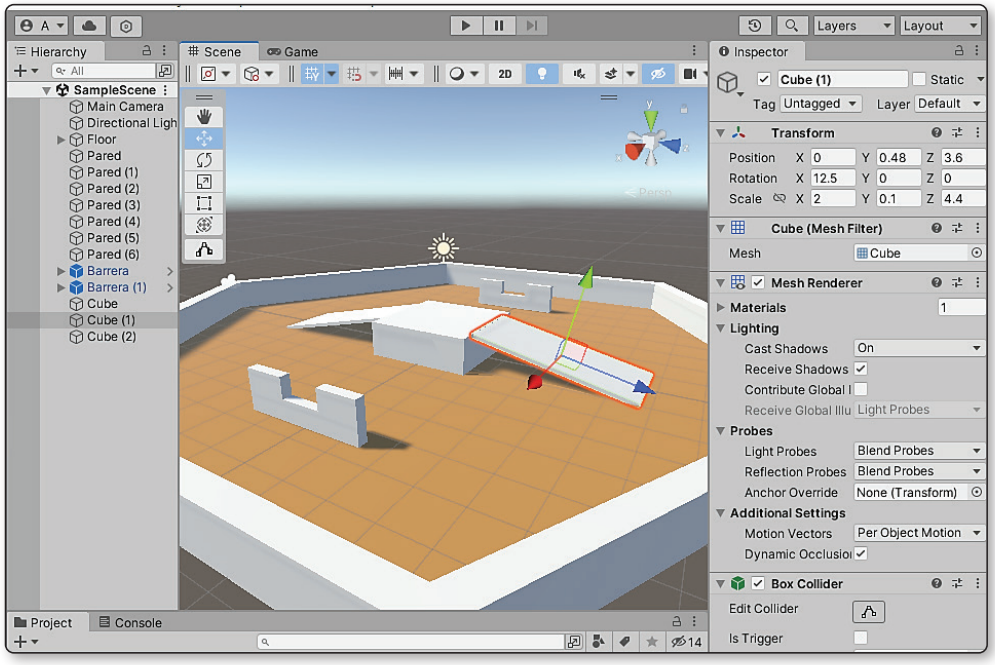

Barrera en la carpeta Prefabs

La ***Barrera*** y todos sus objetos hijos ahora son un ***Prefab***, lo que significa que podemos reutilizarla arrastrando copias desde la carpeta ***Prefabs*** o duplicando la que está en la escena. El ***Barrier*** se debería de haber vuelto de color azul en la pestaña **Hierarchy** para señalar su cambio de estado y también añadió una fila de botones de función de ***Prefab*** en la pestaña **Inspector** debajo de su nombre.

### Completar el nivel

Ahora que tenemos un ***Prefab*** reutilizable, vamos a construir el resto del nivel para que coincida con el boceto inicial:

- Duplica el ***Prefab*** y coloca cada otro en un lado diferente de la arena. Puedes hacerlo arrastrando múltiples objetos ***Barrera*** desde la carpeta ***Prefabs*** a la escena o haciendo clic derecho en ***Barrera*** en **Hierarchy** y seleccionando **Duplicate**.
- Crea un ***Cube*** y escálalo (3x1x3) para formar una plataforma.
- Crea un ***Cube*** y escálalo (2x0.2x4.4), será una rampa, rota alrededor del eje X para crear una inclinación, y luego posiciónala para que conecte la plataforma con el suelo.
- Duplica el objeto de la rampa usando **Ctrl + D** en Windows. Luego, repite los pasos de rotación y posicionamiento.

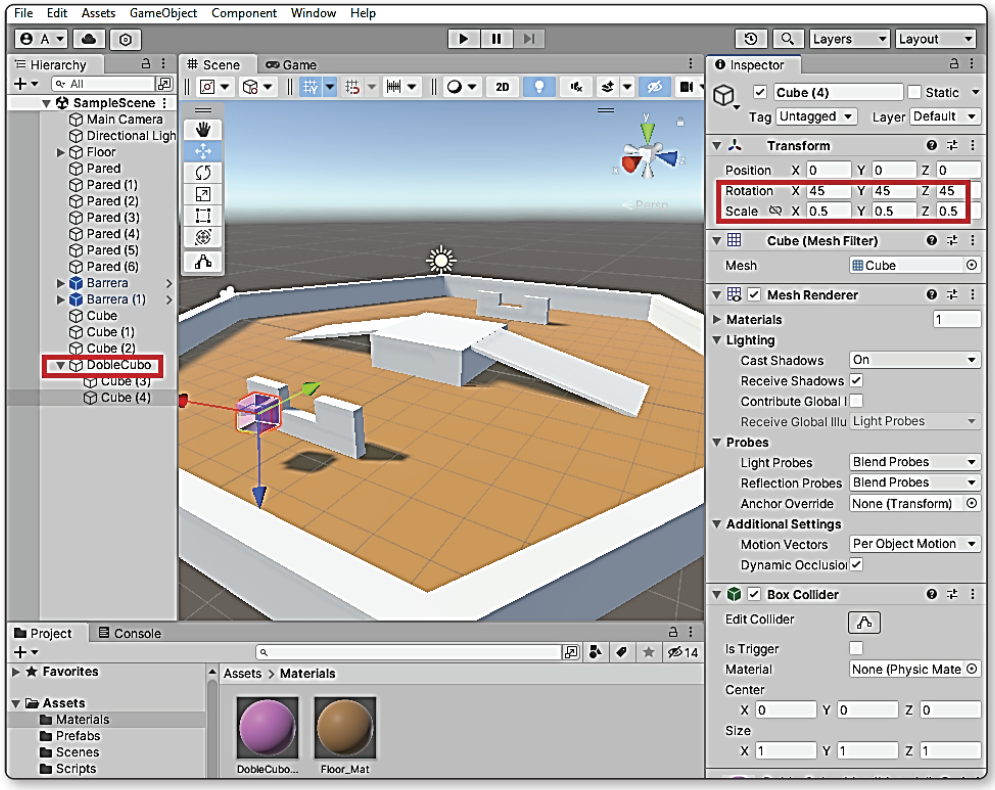



GameObject plataforma elevada

## Creación de un objeto de salud como Prefab

Los juegos suelen tener objetos que los jugadores pueden recoger o con los que pueden interactuar. Crearemos un cubo doble, uno rotado respecto al otro, de forma que tenga 36 caras y lo convertiremos en un ***Prefab***.

- Crea un ***GameObject Cube*** seleccionando **+ > 3D Object > Cube**. Duplícalo y rótalo 45º en X, y y Z. Crea un ***Empty Parent*** con los dos ***Cubes*** y llámalo ***DobleCubo.***
- Ajusta la escala a 0.5 para los ejes X, Y y Z, y luego cambia a la herramienta de mover para posicionarlo cerca de una de tus barreras.
- Crea un material y asígnale un color llamativo, rosa por ejemplo (#FF9AFF). Asígnalo a ***DobleCubo***.
- Arrastra el objeto ***DobleCubo*** desde el panel **Hierarchy** a la carpeta **Prefab**.




Objeto para coger DobleCubo

Eso concluye nuestro trabajo de diseño y disposición del nivel por ahora. En la siguiente sección introduciremos la iluminación en Unity y aprenderemos a animar nuestro objeto.

## 1.5.2 Iluminación básica

La iluminación en Unity es un tema amplio, pero puede resumirse en dos categorías: **en tiempo real** y **precomputada**. Ambos tipos de luces tienen en cuenta propiedades como el color, la intensidad y la dirección en la que se proyecta la luz, todas configurables en la ventana del **Inspector**. La diferencia radica en cómo el motor de Unity calcula el comportamiento de las luces:

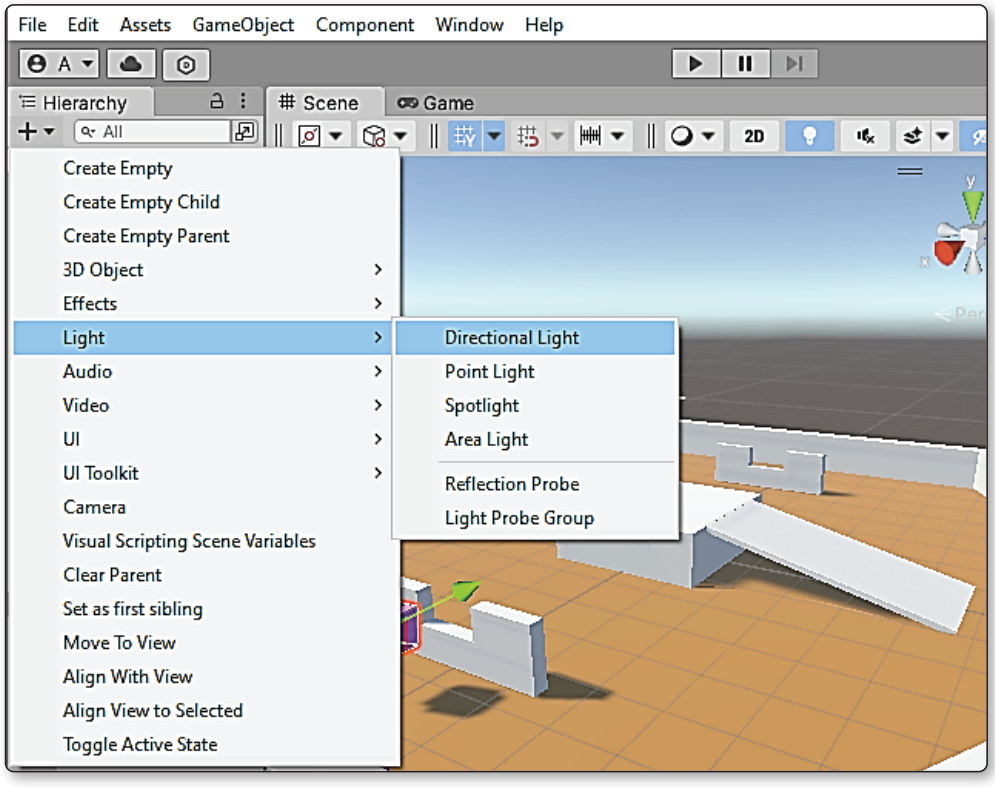
- **Iluminación en tiempo real**: se calcula en cada fotograma, lo que permite que cualquier objeto que cruce su camino proyecte sombras realistas y se comporte como una fuente de luz del mundo real. Sin embargo, esto puede ralentizar considerablemente el juego, especialmente si hay muchas luces en la escena.

- **Iluminación precomputada**: almacena la iluminación de la escena en una textura llamada ***lightmap***, que luego se aplica (o se "hornea"-**baked**-) en la escena. Esto ahorra potencia de cálculo, pero la iluminación horneada es estática, lo que significa que no cambia ni reacciona a los movimientos de los objetos en la escena.
- Una tercera opción, HDRP (High Definition Render Pipeline) ofrece características como **Iluminación Global (GI)**, con sombras más detalladas y reflejos realistas.

En nuestro proyecto utilizaremos iluminación en tiempo real.

## Creación de luces

Cada escena en Unity viene por defecto con un componente de luz direccional como fuente principal de iluminación, pero también puedes crear luces adicionales en la jerarquía como cualquier otro ***GameObject***. Las luces se pueden posicionar, escalar y rotar según sea necesario.




Menú de creación de luces

### Tipos de luces en Unity

1. **Directional lights**: simulan la luz natural, como la luz del sol. No tienen una posición fija en la escena; la luz incide en todo como si siempre estuviera orientada en la misma dirección.
2. **Point lights**: emiten luz desde un punto central en todas direcciones, como una bombilla flotante.
3. **Spotlights**: dirigen la luz en una dirección específica, limitada por un ángulo, enfocándose en un área determinada, como los focos en la vida real.
4. **Area lights**: son rectangulares y emiten luz desde un lado de su superficie.

Puedes crear una luz puntual seleccionando **+ | Light | Point Light** y experimentar con sus ajustes. Después de probar, puedes eliminar la luz haciendo clic derecho en ella en el panel **Hierarchy** y seleccionando **Delete**.

## 1.5.3 Animación

Animar objetos en Unity puede ir desde un simple efecto de rotación hasta movimientos y acciones complejas de los personajes. Puedes crear animaciones mediante código o con las ventanas de ***Animation*** y ***Animator***:

- La ventana de ***Animation*** es donde se crean y gestionan segmentos de animación, llamados ***clips***, utilizando una línea de tiempo. Las propiedades del objeto se registran a lo largo de esta línea de tiempo y luego se reproducen para crear un efecto animado.
- La ventana de ***Animator*** gestiona estos clips y sus transiciones usando objetos llamados controladores de animación.

En este breve recorrido por las animaciones en Unity, crearemos el mismo efecto de rotación *tanto en código como usando el Animator*.

### Crear animaciones en código

Para empezar, vamos a crear una animación en código para rotar nuestro ***DobleCubo***. Dado que todos los ***GameObjects*** tienen un componente ***Transform***, podemos tomar el componente ***Transform*** de nuestro objeto y hacerlo girar indefinidamente.

Para crear una animación en código, debes realizar los siguientes pasos:

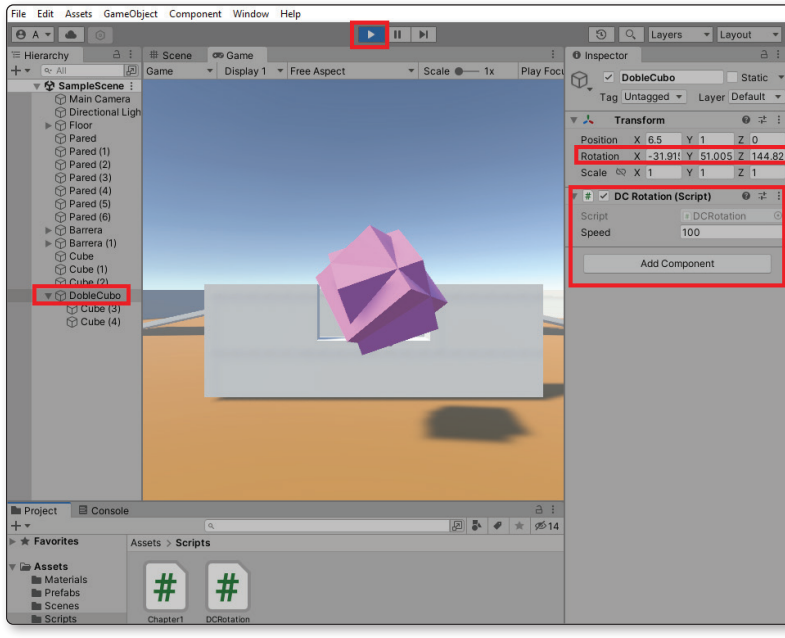
- Crea un nuevo script dentro de la carpeta Scripts, llámalo ***DCRotation*** y ábrelo en Visual Studio.
- En Unity, selecciona el objeto ***DobleCubo*** en la carpeta de ***Prefabs*** y desplázate hasta la parte inferior de la ventana del **Inspector**. Haz clic en ***Add Component***, busca el script ***DCRotation*** y presiona **Enter**. De esta forma, podremos acceder a componentes del ***GameObject DobleCubo*** como ***Transform*** desde el propio Script, con `this.transform`.
- En la parte superior del script y dentro de la clase, añade una variable pública de tipo **int** con valor 44 llamada ***Speed***:

```
public int Speed = 44;
```

- Dentro del método Update(), llama a `this.transform.Rotate`. Este método de la clase Transform toma tres ejes, uno para las rotaciones en x, y y z.

```
this.transform.Rotate(-Speed*Time.deltaTime,
 Speed*1.5f* Time.deltaTime, -Speed*5*Time.deltaTime);
```

- Mueve la **Main Camera** para que puedas ver el objeto ***Health_Pickup*** y haz clic en **Play**.




Cámara enfocada en el DobleCubo

Ahora, el ***DobleCubo*** gira continuamente sobre sus ejes.

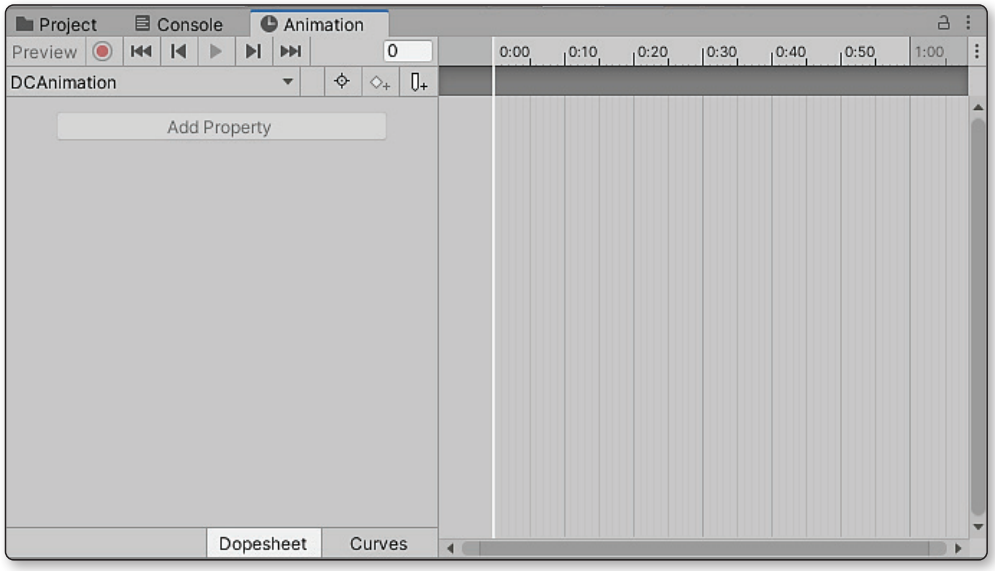
### Crear animaciones en la ventana Animation de Unity

Es importante que *elijas entre el método por código o el sistema de animación de Unity* para una única animación, ya que *estos dos sistemas pueden entrar en conflicto*.

Cualquier GameObject al que quieras aplicar un clip de animación necesita tener un componente **Animator** con un **Animation Controller** asignado. Si no hay un controlador en el proyecto cuando se crea un nuevo clip, Unity creará uno y lo guardará en el panel de **Proyect**.

A continuación, se describen los pasos para crear un nuevo clip de animación para el ***DobleCubo***:

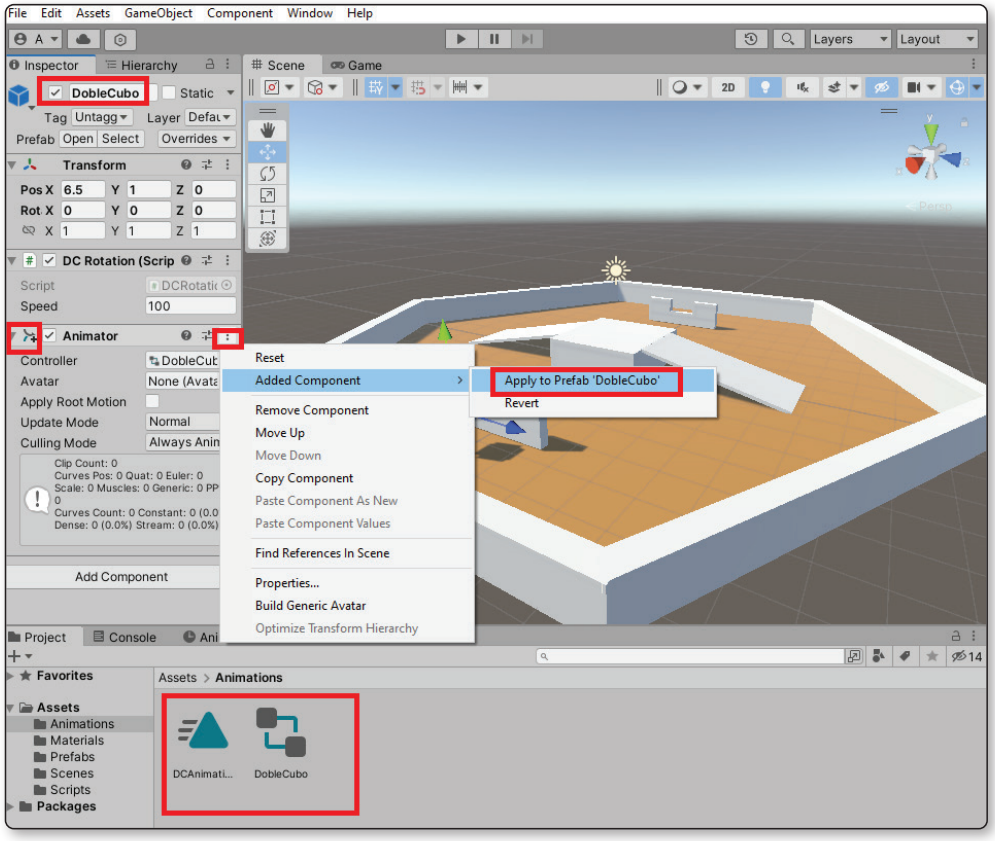
- Navega a **Window > Animation > Animation** para abrir el panel de **Animation**.
- Asegúrate de que el objeto ***DobleCubo*** esté seleccionado en el **Hierarchy** y haz clic en **Create** en el panel de **Animation**.




Ventana de Unity Animation

- Crea una nueva carpeta debajo de **Assets** llamada ***Animations*** y nombra el nuevo clip ***DCAnimation***.

- Dado que no teníamos ningún controlador de **Animator**, Unity creó uno para nosotros en la carpeta de Animación llamado ***DobleCubo.controler***. Con ***Health_Pickup*** seleccionado, observa en el panel del Inspector que, al crear el clip, también se agregó un componente **Animator** al ***Prefab***, pero aún no se ha guardado oficialmente en el ***Prefab***. Observa que el ícono de "+" aparece en la parte superior izquierda del componente **Animator**, lo que significa que aún no forma parte del Prefab ***Health_Pickup***:




Componente Animator en el panel Inspector

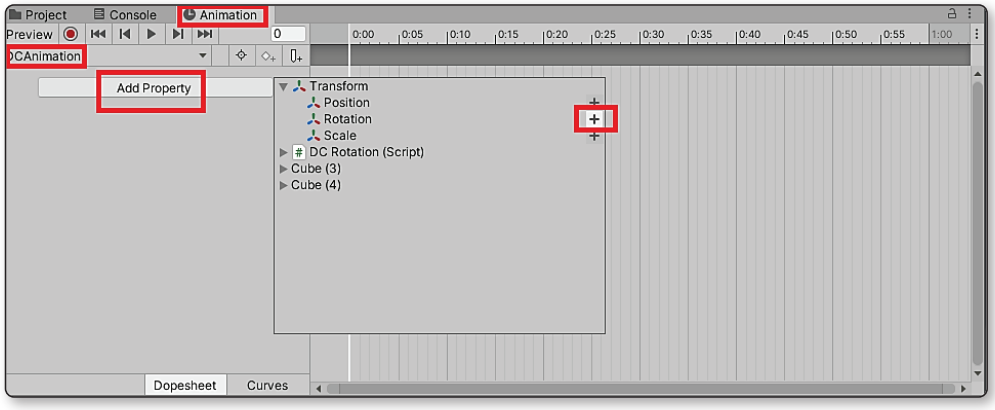
- Selecciona el ícono de los tres-puntos-verticales en la parte superior derecha y elige **Added Component > Apply to Prefab 'DobleCubo'**:

Ahora que has creado y agregado un componente Animator al Prefab ***Health_Pickup***, es hora de comenzar a grabar algunos fotogramas de animación. Cuando piensas en clips de movimiento, como en las películas, puedes pensar en *fotogramas*. A medida que el clip avanza a través de sus fotogramas, la animación progresa, dando el efecto de movimiento. Así que necesitamos grabar nuestro objeto en diferentes posiciones clave (keyframes) para que Unity pueda reproducir el clip.

### Grabar fotogramas clave (keyframes)

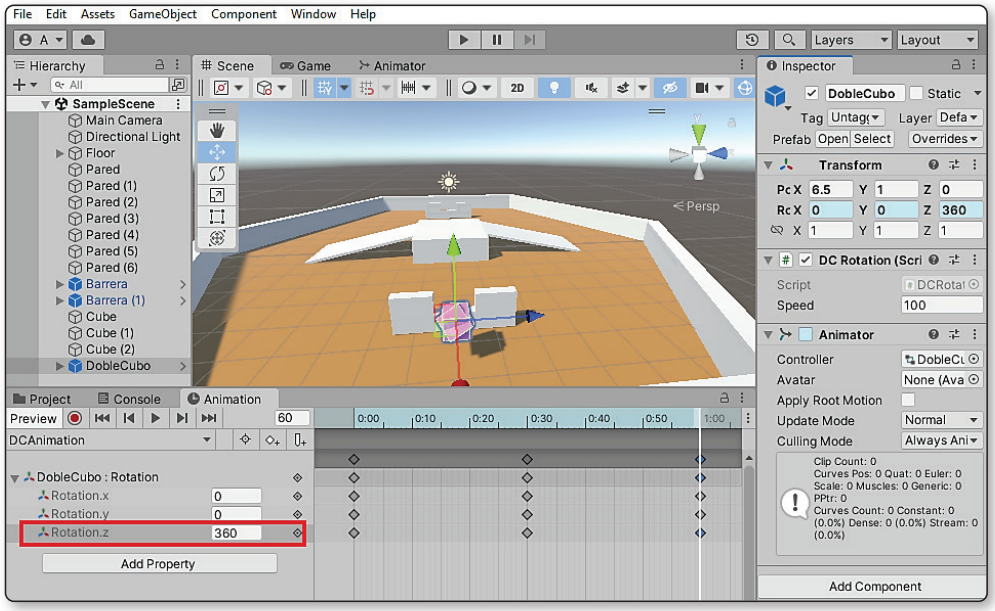
Ahora que tenemos un clip, vamos a registrar la rotación del objeto en diferentes fotogramas para que Unity pueda reproducir el clip y dar el efecto de movimiento. Queremos que el objeto realice una rotación completa sobre un eje cada segundo:

1. Selecciona el objeto ***DobleCubo*** en la ventana **Hierarchy**, elige **Add Property > Transform** y luego haz clic en el signo "+" junto a ***Rotation***:



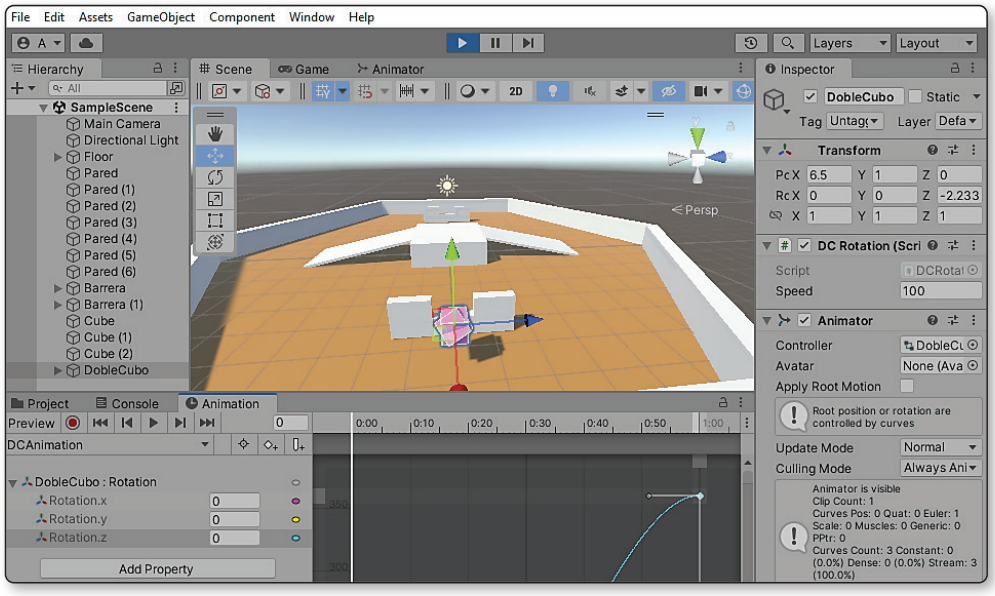

Agregando una propiedad de Transform para la animación

2. Haz clic en el botón **Record** para comenzar la animación:
   - Coloca el cursor en 0:00 en la línea de tiempo, pero deja la rotación en z del Prefab ***Health_Pickup*** a 0 en el **Inspector**. Los campos modificables aparecerán en rojo para que no animes una propiedad por error:




Introduciendo keyframes de rotación en Z

- Coloca el cursor en 0:30 en la línea de tiempo y ajusta la rotación en z a 180.
- Coloca el cursor en 1:00 en la línea de tiempo y ajusta la rotación en z a 360:




Probando la animación1 del DobleCubo

3. Haz clic en el botón **Record** para finalizar la animación.

4. Haz clic en el botón **Play**, que está a la derecha del botón **Record**, para ver el bucle de la animación.

Notarás que nuestra animación en el Animator sobrescribe la que escribimos en código anteriormente. No te preocupes, esto es un comportamiento esperado. Puedes hacer clic en la casilla de verificación junto a cualquier componente en el panel del Inspector para activarlo o desactivarlo. Si desactivas el componente Animator, el ***Health_Pickup*** rotará nuevamente sobre el eje x utilizando nuestro código.

El objeto ***Health_Pickup*** ahora rota en el eje z entre 0, 180 y 360 grados cada segundo, creando una animación de giro en bucle. La animación se ejecutará indefinidamente hasta que el juego se detenga.

## 1.6 MOVIMIENTO, CONTROL DE CÁMARA Y COLISIONES

Una de las primeras cosas que hace un jugador al empezar un nuevo juego es probar el movimiento del personaje y los controles de la cámara. El ***Player*** será un objeto cápsula que puede ser movido y rotado usando las teclas W, A, S, D o las flechas de ↑ ← ↓ → respectivamente.

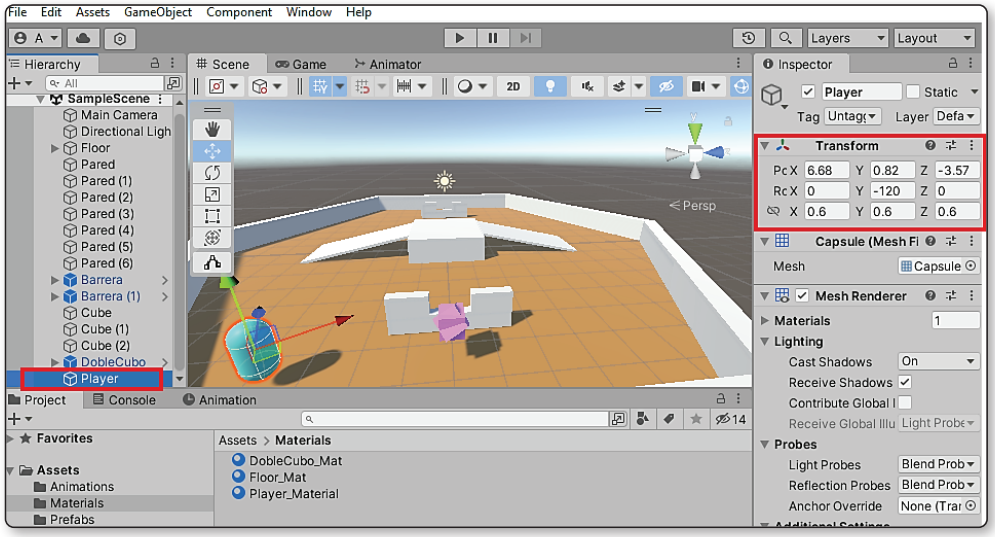
Cuando movamos al jugador, la cámara lo seguirá desde una posición ligeramente por detrás y por encima del jugador, lo que facilitará la puntería cuando implementemos la mecánica de disparo. Por último, vamos a explorar cómo *las colisiones y las interacciones físicas* son manejadas por el motor de físicas ***PhysX*** de Unity, trabajando con nuestro prefab en la recogida de objetos.

También nos va a dar nuestra primera muestra de C# utilizado para programar las características del juego mediante:

- Movimiento y rotación de ***GameObjects***.
- Gestión del teclado (***inputs)*** para mover al jugador.
- Movimientos de cámara.
- Físicas (***rigidbodies)*** y fuerzas aplicadas.
- Colisionadores (***colliders)*** básicos y detección de colisiones.

### 1.6.1 Creando el jugador

Crea un material (del color ***-albedo-*** que quieras), un objeto cápsula con **+ > 3D Object > Capsule** y posiciónalo en la arena. Juega un poco con los tamaños y rotación del transform de la cápsula hasta que estés satisfecho, asígnale el material y nómbralo ***Player***:




Creando el GameObject Player

Finalmente arrastra el Player hasta la carpeta Prefabs.

### 1.6.2 Añadir una cámara que sigue al jugador

Crea un nuevo script en C# en la carpeta ***Scripts***, nómbralo ***Cam*** y arrástralo al objeto ***Main Camera*** en el panel de Jerarquía.

```
using UnityEngine;
public class Cam : MonoBehaviour {
  public Vector3 CamOffset = new Vector3(0f, 1.2f, -2.6f);  // 1
  private Transform p; // 2

  void Start() {
    p = GameObject.Find("Player").transform; // 3
  }

  void LateUpdate() { // 4
    this.transform.position = p.TransformPoint(CamOffset); // 5
    this.transform.LookAt(p); // 6
  }
}
```