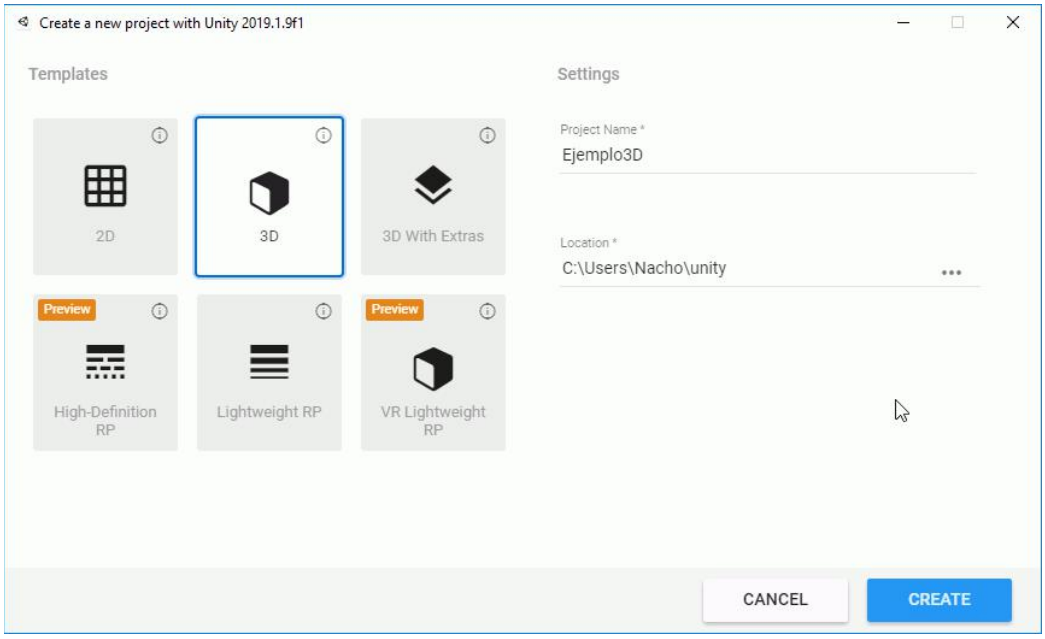
Tema 2. El entorno 3D de Unity

En este tema vamos a ver un contacto con el entorno 3D de Unity: colocaremos objetos primitivos, ajustaremos su posición y dimensiones, cambiaremos su material, hablaremos de "waypoints" en 3D, de cámaras y luces, etc.

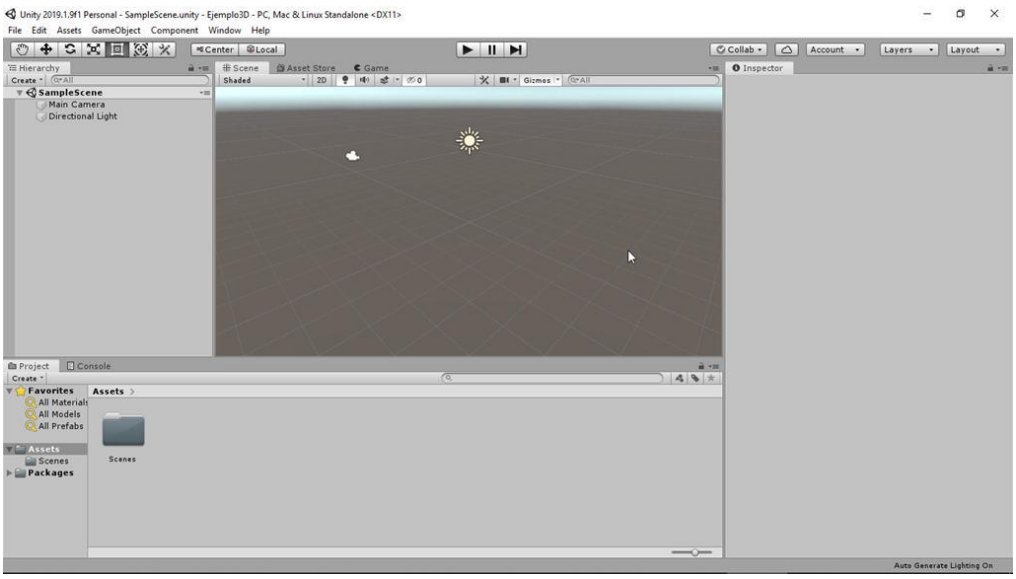
(Nota: las imágenes de este tema están tomadas de Unity 2019. Es de esperar que los cambios en Unity 2020 sean menores)

2.1. El entorno 3D

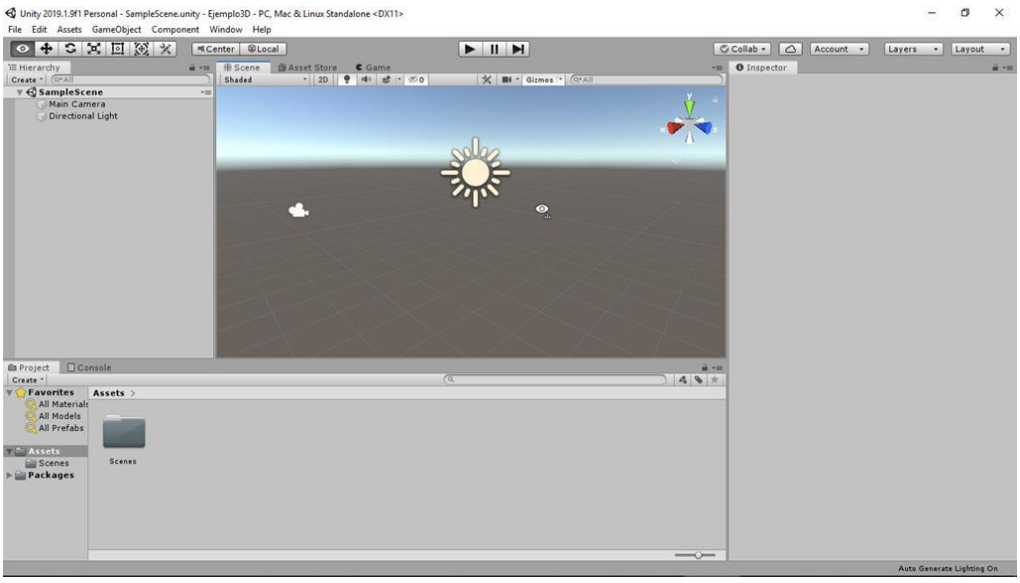
Para crear un juego 3D empezaremos, obviamente, por crear un nuevo proyecto, escogiendo como plantilla "3D":



Y la apariencia del editor del editor de Unity será similar a la de 2D, salvo que en esta ocasión el "suelo" se verá en perspectiva y se nos mostrará la ubicación de la "cámara" y de un "punto de luz":



Al igual que ocurría en 2D, podemos usar la "rueda del ratón" para acercarnos o alejarnos, y podemos usar el botón derecho del ratón para ver otras zonas del "mundo". En este caso se comportará como si moviéramos la cabeza hacia arriba, abajo y los lados (y aparecerán unos ejes cartesianos en la esquina para ayudarnos a saber la orientación actual):

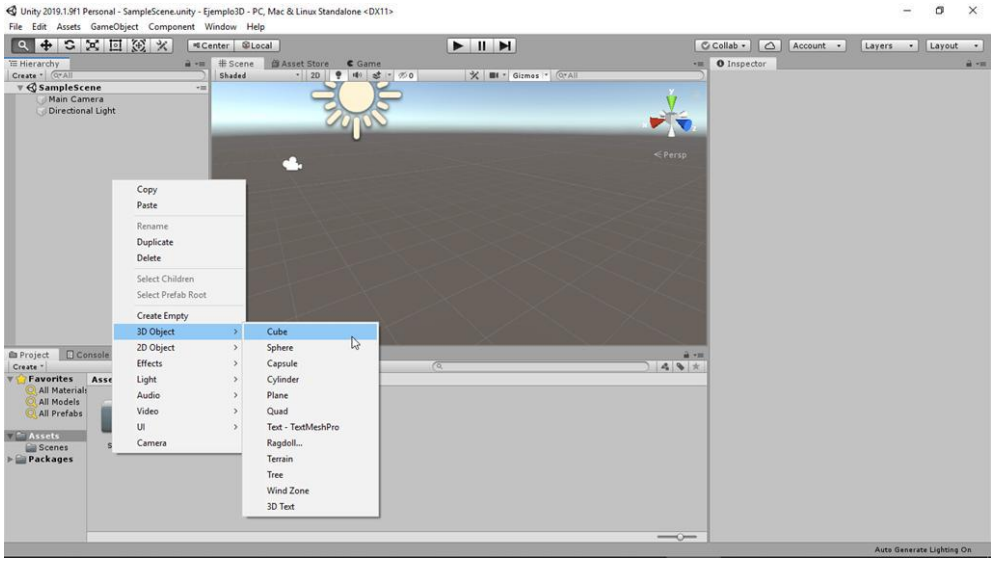


Por el contrario, arrastrar con el botón izquierdo se parecerá más a movernos hacia un lado u otro. Si pulsamos Mayúsculas a la vez que el botón, el movimiento será más rápido.

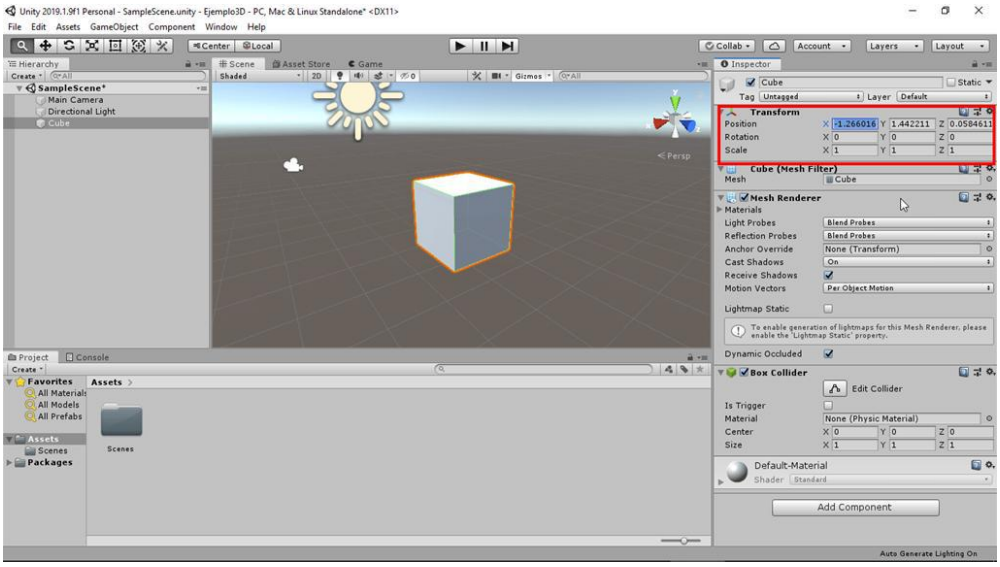
También es posible **moverse con teclas**: al pulsar el botón derecho, el icono del ratón cambia a un ojo que muestra a su lado lo que parecen unas teclas WASD o flechas del cursor, como forma de recordarnos que (manteniendo el botón apretado), nos podremos desplazar también con las teclas WASD (arriba, izquierda, abajo y derecha), además de Q y E (bajar y subir la vista):

2.2. Incluyendo un objeto primitivo y cambiando su posición y tamaño

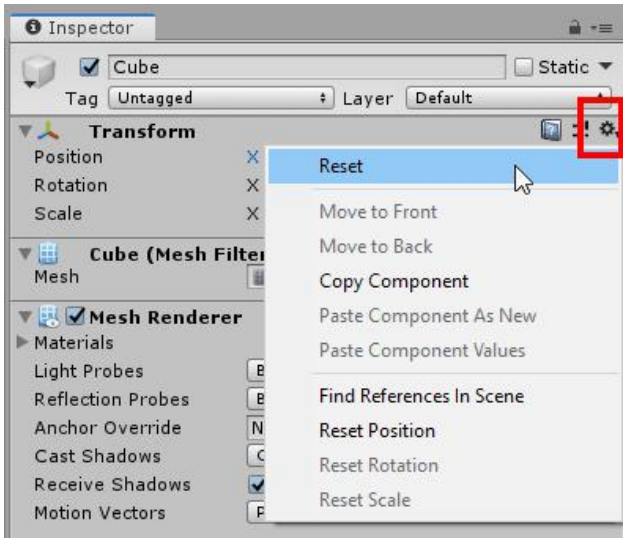
Comenzaremos por crear un cubo, que nos permita jugar con las distintas formas de cambiar posiciones, escalas y rotaciones. Para ello, bastará con pulsar el botón derecho en la jerarquía y escoger 3D Object / Cube:



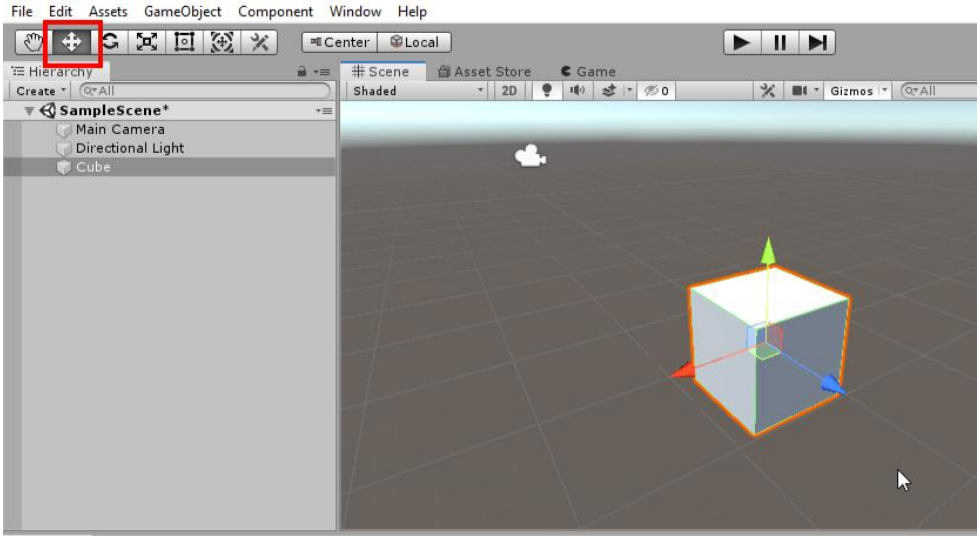
Este cubo aparecerá en posición un tanto arbitraria. Una primera forma de asegurarnos de que queda en una posición que nos interese es cambiarla desde el componente "Transform" del "Inspector":



Es habitual dejarlo en (0,0,0) y moverlo después a partir de ahí. Como ya sabemos, en ese caso realmente no necesitamos escribir un 0 en cada una de las casillas, sino que podemos usar la opción "Reset", desde el engranaje que aparece en la esquina superior derecha de cualquier componente, en el "Inspector":



A partir de ahí, una primera forma de **cambiar la posición** será precisamente modificar esas coordenadas X, Y, Z. Otra forma es pulsar la herramienta "Mover", de modo que aparecerán 3 flechas en el objeto, que nos permitirán moverlo en el sentido de cada uno de los ejes, pero también 3 planos para moverlo a la vez en dos dimensiones (en el plano XY, en el YZ o en el XZ):



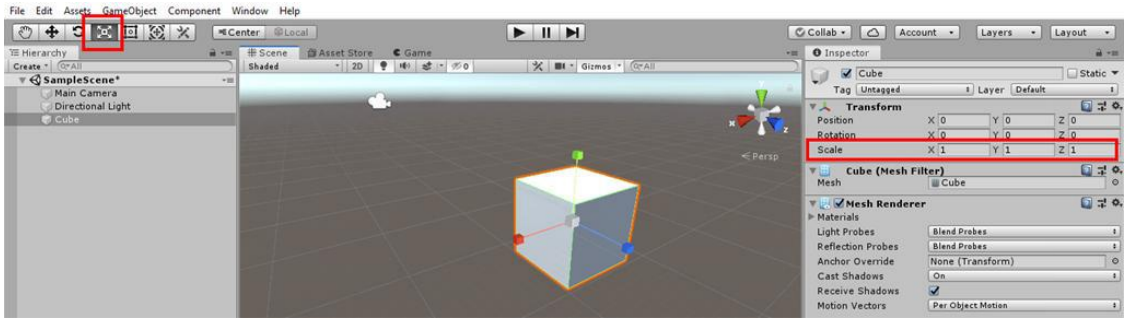
Por cierto, en 3D los significados de X, Y, Z ya "no son tan claros": en 2D usábamos la X para el movimiento horizontal, la Y para el vertical, con lo que es de esperar que la Z se refiera al movimiento "en profundidad"... pero no siempre será así (aunque intentaremos respetarlo, por simplicidad), porque podremos cambiar la posición de la cámara...

Podemos hacer que se mueva "**a saltos**" (en principio, de una unidad en una unidad), en vez de hacerlo de forma "suave". Esto puede ayudar a encajar de forma sencilla unos objetos con otros. Para conseguirlo deberemos pulsar la tecla Ctrl mientras movemos el objeto.

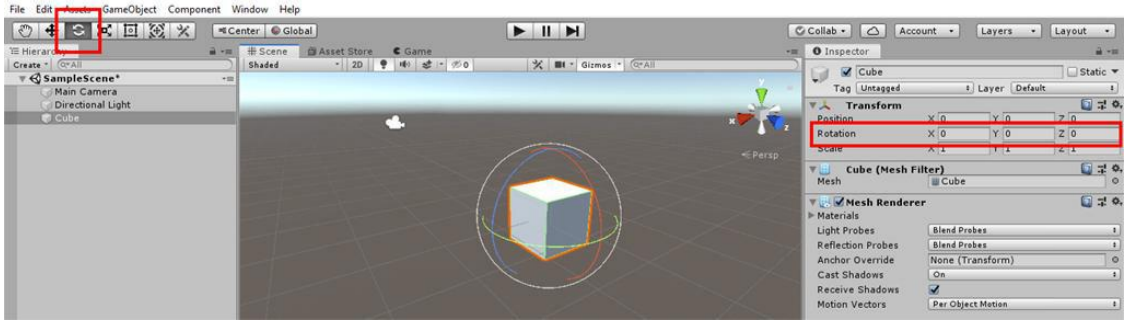
Como curiosidad, si tenemos activada la herramienta Mover, el botón izquierdo del ratón ya no servirá para "desplazar" el mundo, sino para seleccionar. En este modo, usaríamos el **botón central** (típicamente, pulsar la rueda de desplazamiento) para mover la vista.

De forma similar, podemos "**escalar**" el objeto (cambiar su tamaño): bien empleando el componente "Transform", o bien la herramienta Escalar, que mostrará 3 pequeños cuadrados

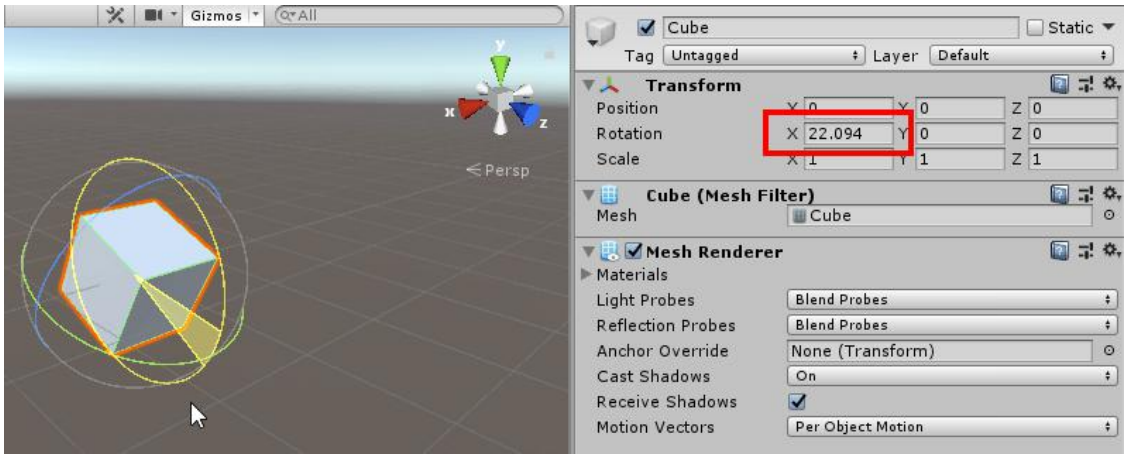
que nos permitirán "estirar" el objeto en el sentido de cualquiera de los ejes, o bien un cuadrado central para cambiar su tamaño en los 3 ejes a la vez:



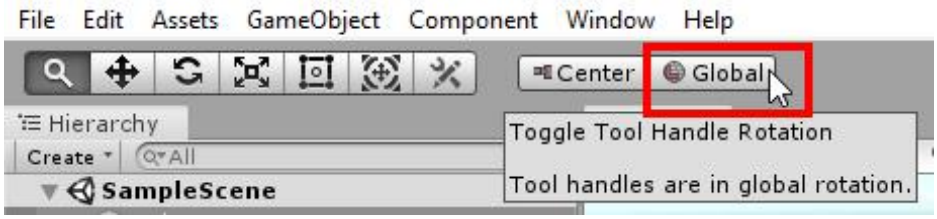
Para **rotar**, nuevamente, podemos utilizar el componente "Transform", cuando busquemos precisión, o la herramienta Rotar, que mostrará 3 círculos, en cada uno de los posibles sentidos en que podemos rotar la figura:



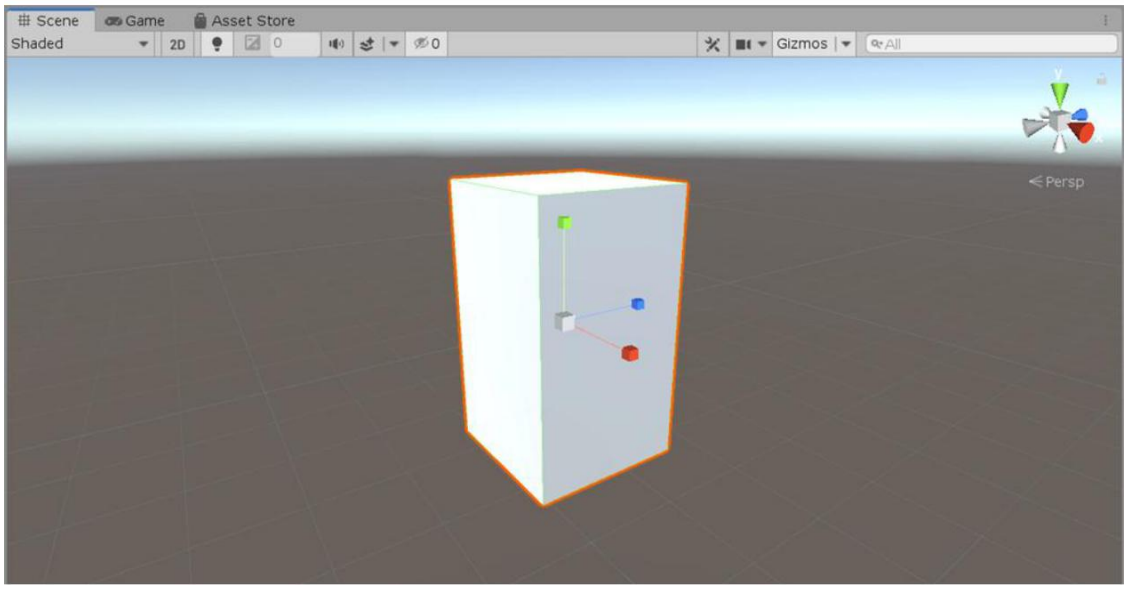
A medida que vayamos rotando, no sólo se mostrará el nuevo valor en el Inspector, sino que también se "dibujará" el ángulo rotado por la figura:



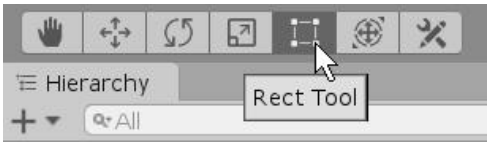
(*Detalle avanzado: Cuando un objeto está rotado, si lo movemos se moverá con relación a los ejes globales X, Y, Z; podemos hacer que se mueva con relación a sus propios ejes locales si pulsamos el botón "Global" en la barra de herramientas superior, para que pase a ser "Local"; hablaremos más de coordenadas locales y globales un poco más adelante:*)

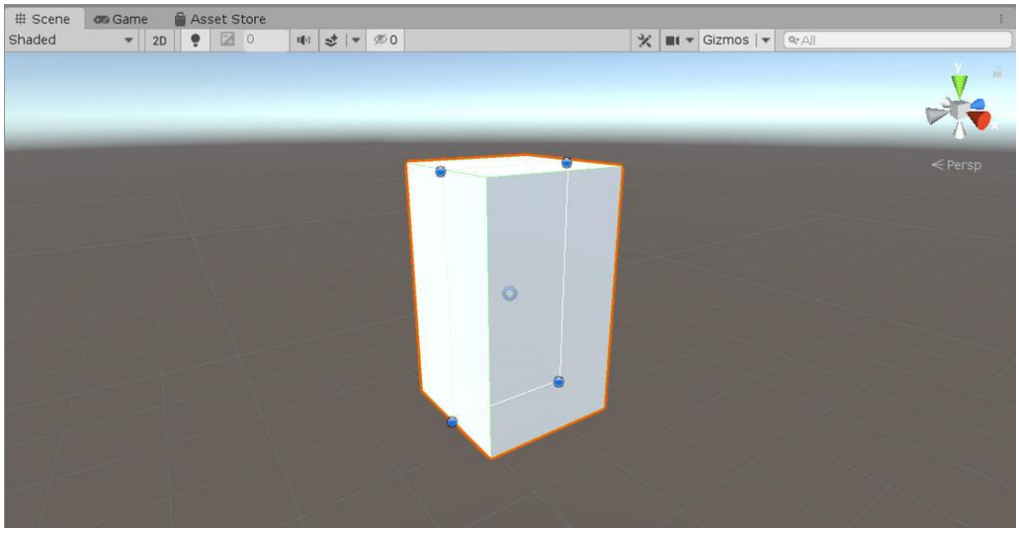


La siguiente herramienta es la de escalado, que permite cambiar el tamaño según alguno de los ejes:

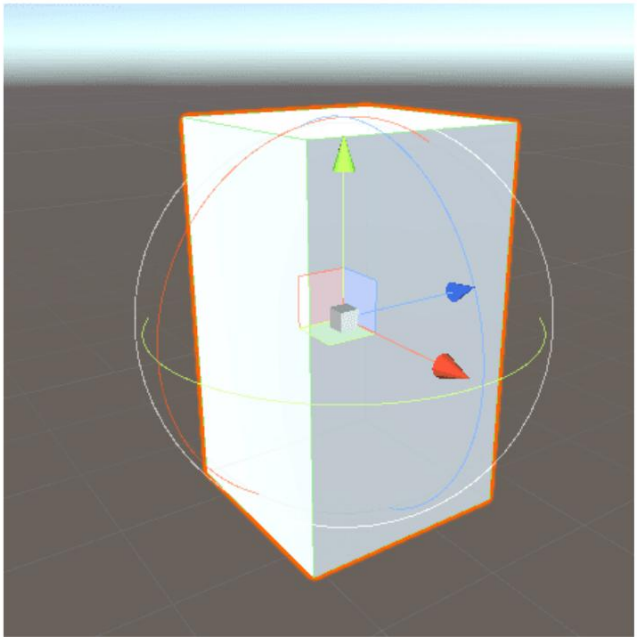


Después aparece la herramienta de rectángulo ("rect tool"), que permite estirar desde el centro de las esquinas para cambiar el tamaño en dos dimensiones a la vez:





Y la siguiente herramienta (sólo en versiones recientes de Unity) permite mover, rotar y escalar desde un único sitio:

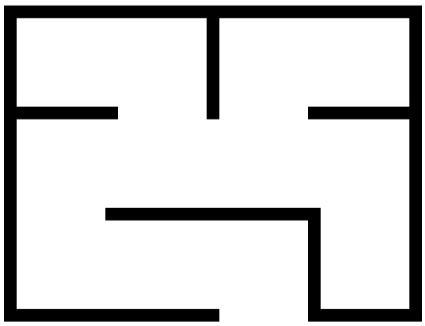


Como curiosidad adicional, las distintas herramientas de movimiento, rotación, escalado, etc., se pueden activar también desde el teclado, pulsando las teclas **QWERTY** (Q para movimiento, W para rotación y así sucesivamente).

Ejercicio propuesto 2.2.1: Crea un cubo, muévelo a las coordenadas (0,0,0) y luego redimensiónalo, para que tenga 2 unidades de alto (Y) y 20 de ancho (X), conservando 1 de profundidad (Z).

2.3. Añadiendo algún objeto más

Vamos a crear un laberinto que el jugador deba recorrer. Podemos apuntar hacia un diseño sencillo, por ejemplo de 4 casillas de ancho por 3 de alto, como el siguiente:



y en esta entrega crearemos sólo las paredes exteriores, buscando algo como

Si asignamos "5 unidades" del mundo de Unity a cada "casilla" del laberinto, necesitaríamos una pared superior de 20 unidades (que ya has creado en el ejercicio anterior), dos paredes de 15 unidades para los laterales, y otras dos de 10 + 5 para la parte inferior.

Los pasos que podríamos seguir son:

- Crear una nueva "pared", a partir de un cubo, cuyo tamaño ahora sea de 2 unidades de alto (Y), 1 de ancho (X), y 15 de profundidad (Z).
- Cambiar su nombre a algo más intuitivo, como "ParedIzquierda".
- Calcular su posición "con lápiz y papel", o bien:

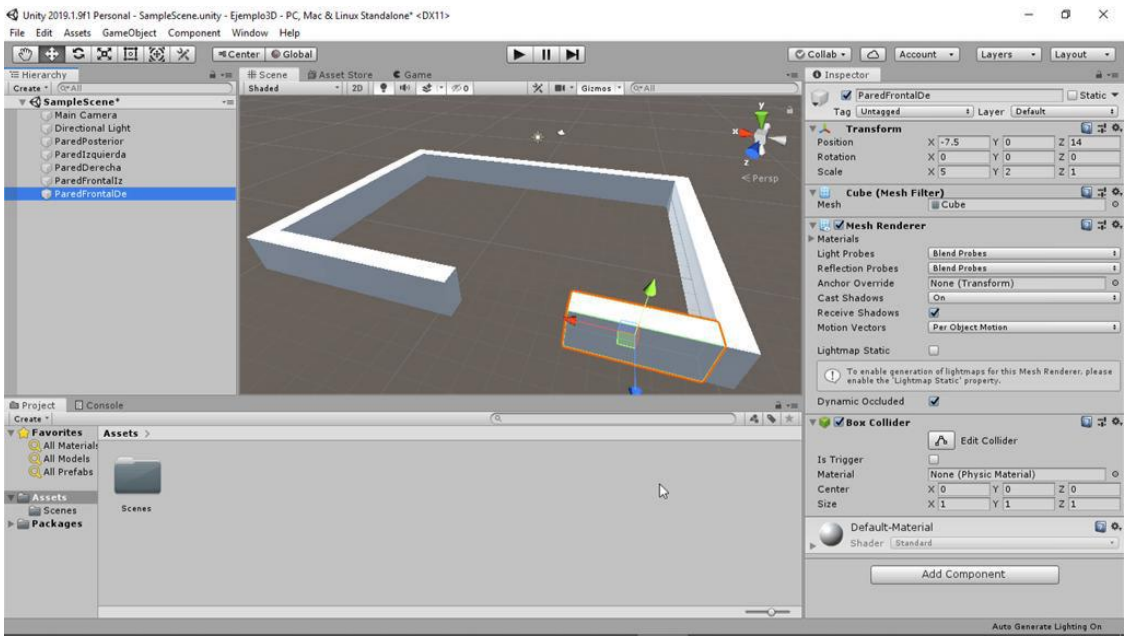
– Acercarnos a su posición, arrastrando como ya hemos visto (el resultado, por lo general, será poco preciso).
–Arrastrar pulsando Ctrl, para avanzar de casilla en casilla y poder ajustar con facilidad.
– Si pulsamos Ctrl+Mays al arrastrar, entraremos en modo de "ajuste", y el objeto se "pegará" a los laterales más cercanos de otros objetos en cada momento, algo que puede ser útil si los tamaños no son unidades exactas.

– Si los tamaños son irregulares, puede ser más práctico mantener pulsada la tecla "V" para entrar al modo de "Ajustar vértices", y entonces mover un vértice de la nueva pared hasta que coincida con otro vértice de la pared existente.

Para la siguiente pared, bastaría con duplicar la que ya hemos creado (Ctrl+D) y moverla al lado opuesto (de la forma que acabamos de hacer o bien directamente cambiando el signo de su coordenada X: si la pared izquierda está en la X -10.5, es de esperar que la pared derecha deba estar en C 10.5).

Para las dos paredes frontales, podemos duplicar la posterior, mover la copia y ajustar la escala.

El resultado debería ser algo como:

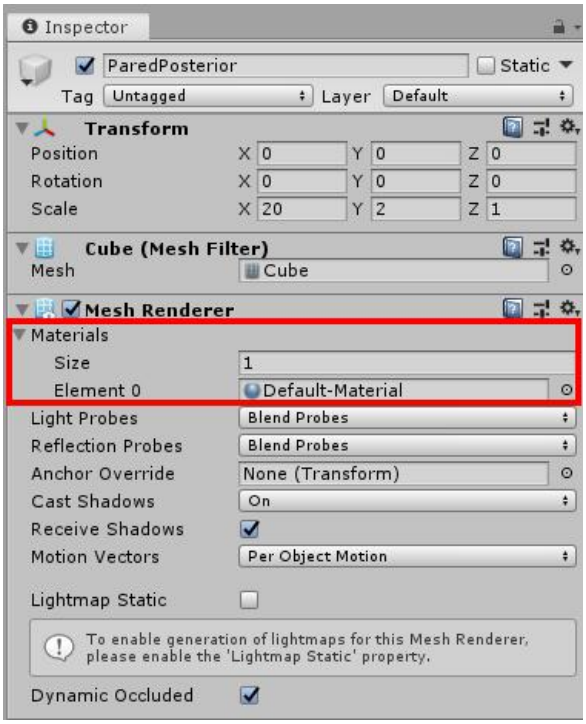


Detalle avanzado: Si un objeto está en coordenadas no exactas y queremos moverlo a la coordenada exacta más próxima, lo podemos conseguir desde el menú "Edit / Snap Settings". Desde ahí también podemos afinar el salto que se produce al mover con Ctrl+ratón, si no queremos que sea de uno en uno.

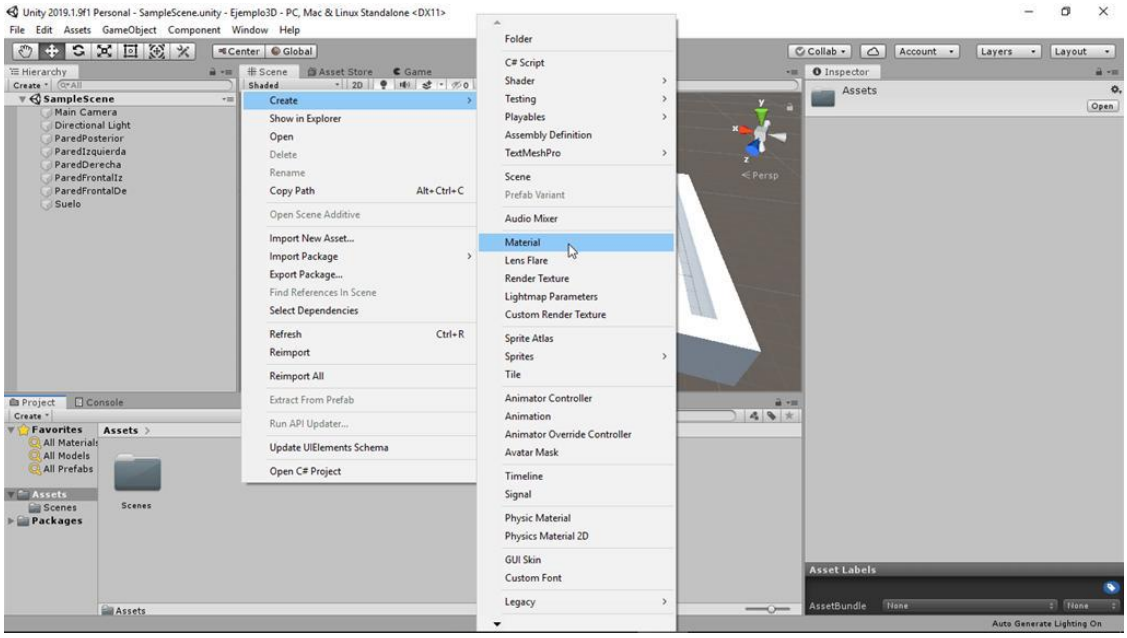
Ejercicio propuesto 2.3.1: Crea unas paredes similares a esas, así como un "suelo" por debajo de ellas, para que podamos aplicar fuerzas de gravedad a los elementos que van a participar en el juego.

2.4. Contacto con los materiales

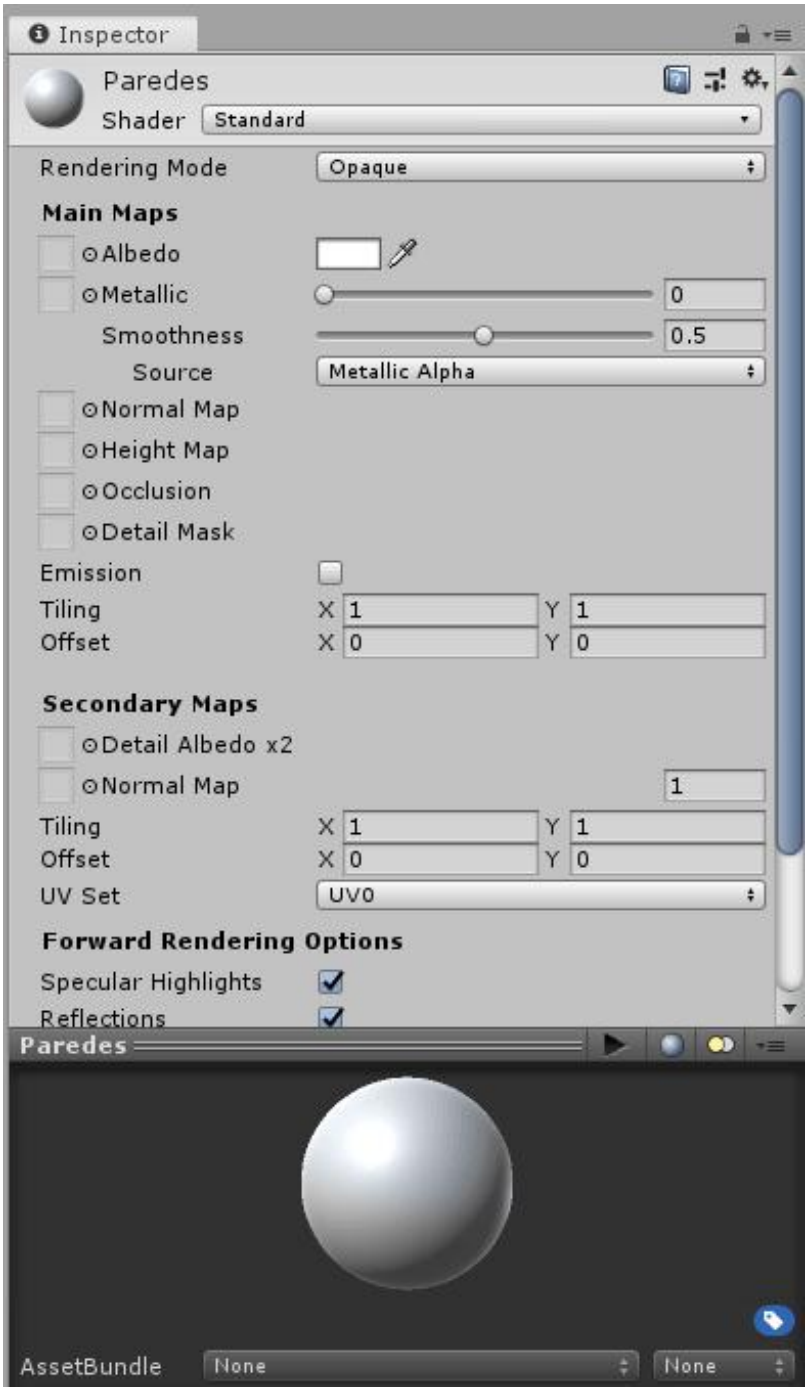
Los materiales nos servirán para indicar qué color tendrá un objeto, pero también otros detalles avanzados, como texturas o cantidad de reflejos. Los objetos que creamos aparecen con el "material por defecto", como podemos ver en el inspector:



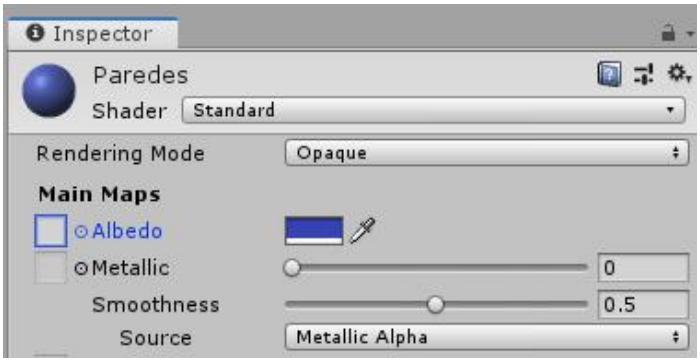
Para crear nuestro propio material, haremos clic con el botón derecho en la vista de proyecto y escogeremos la opción "Create / Material":



Y sus propiedades aparecerán en el Inspector:

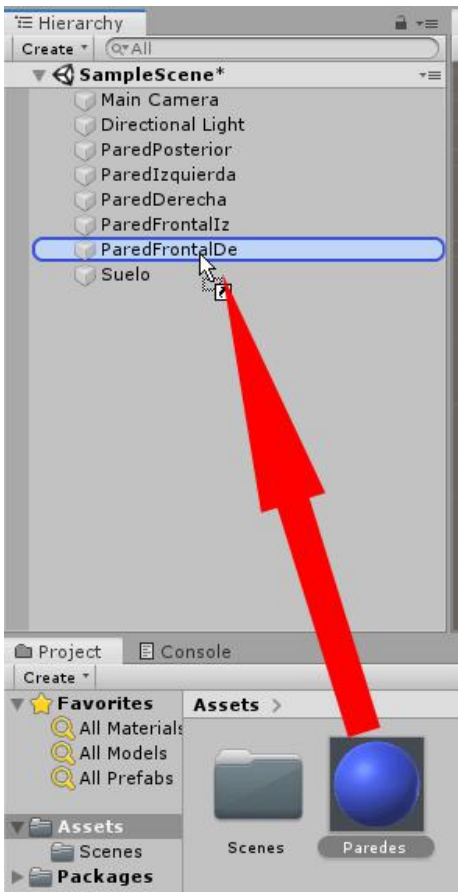


Podemos comenzar por cambiar el color base, haciendo clic en la casilla blanca que aparece junto a "Albedo", y escoger un color nuevo:

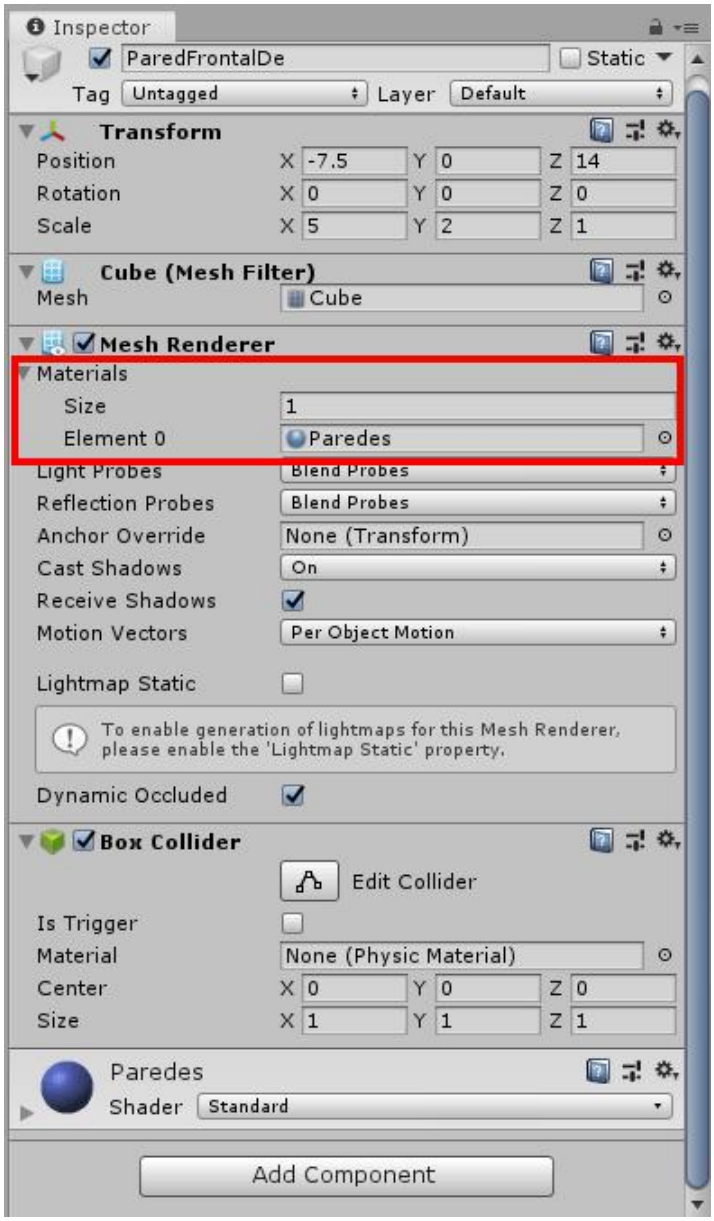


(*Nota: en términos técnicos, el "albedo" es la relación entre la cantidad de luz que recibe una superficie y la cantidad que refleja*)

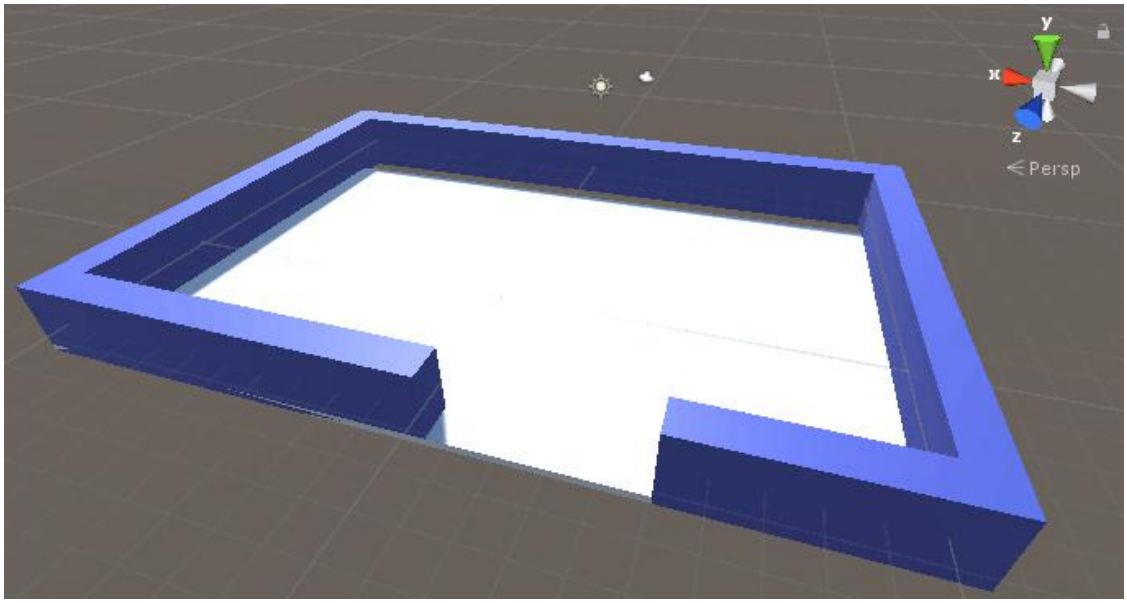
Para asignar ese material (de momento, simplemente ese color), bastará con arrastrarlo desde el panel inferior (Project) a cada uno de los elementos de la jerarquía que deseemos que empleen ese material:



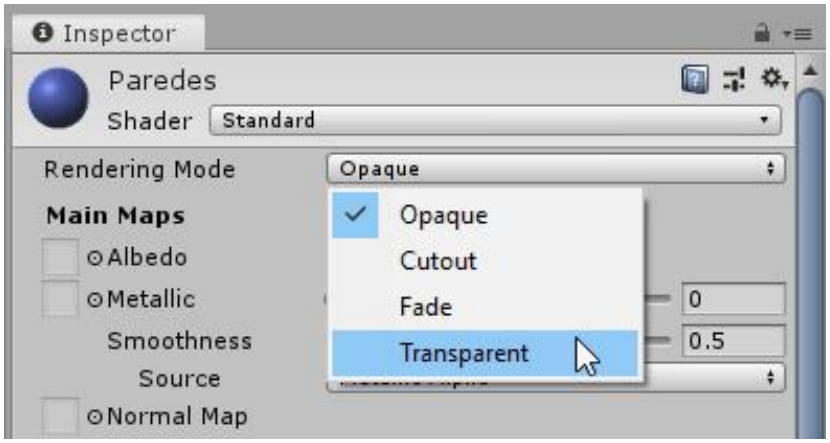
Y el material aparecerá al ver los detalles de ese objeto en el Inspector:

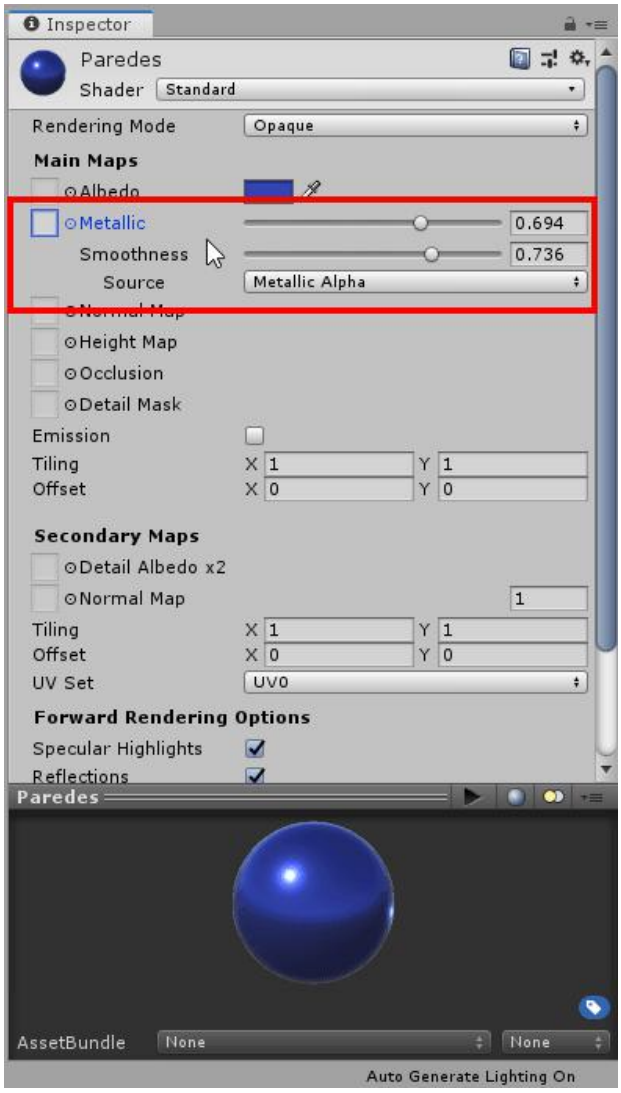


Tras aplicar esa textura a todas las paredes, debería quedar algo así:

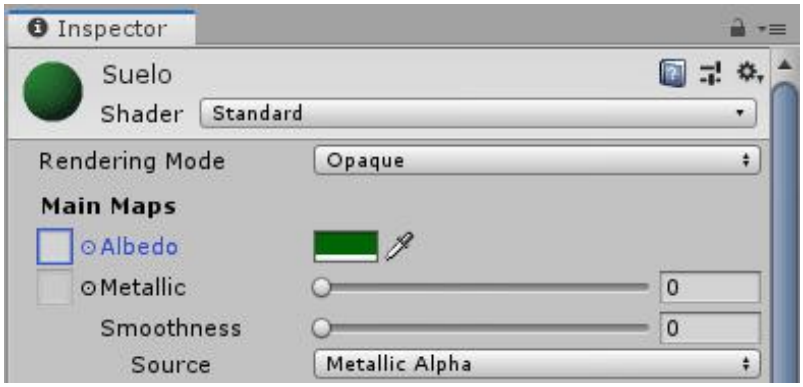


Como detalle adicional, podemos hacer que el material sea más o menos transparente, y que tenga una textura más o menos metálica (y con un brillo más difuso o más puntual, si ajustamos su "smoothness"):





Si queremos crear un textura similar a la que ya tenemos, podemos duplicar la textura actual (con Ctrl+D), en el panel de "Project", cambiarle el nombre y ajustar su color base ("Albedo").

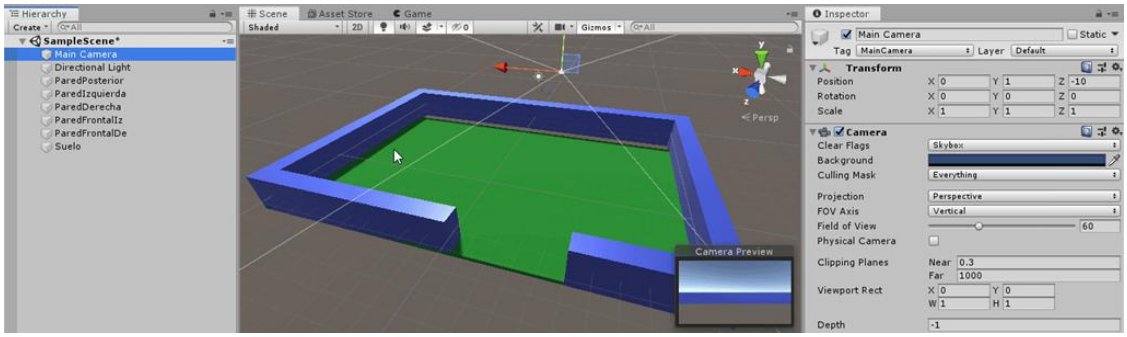


Ejercicio propuesto 2.4.1: Aplica textura a las paredes y (otra distinta) al suelo.

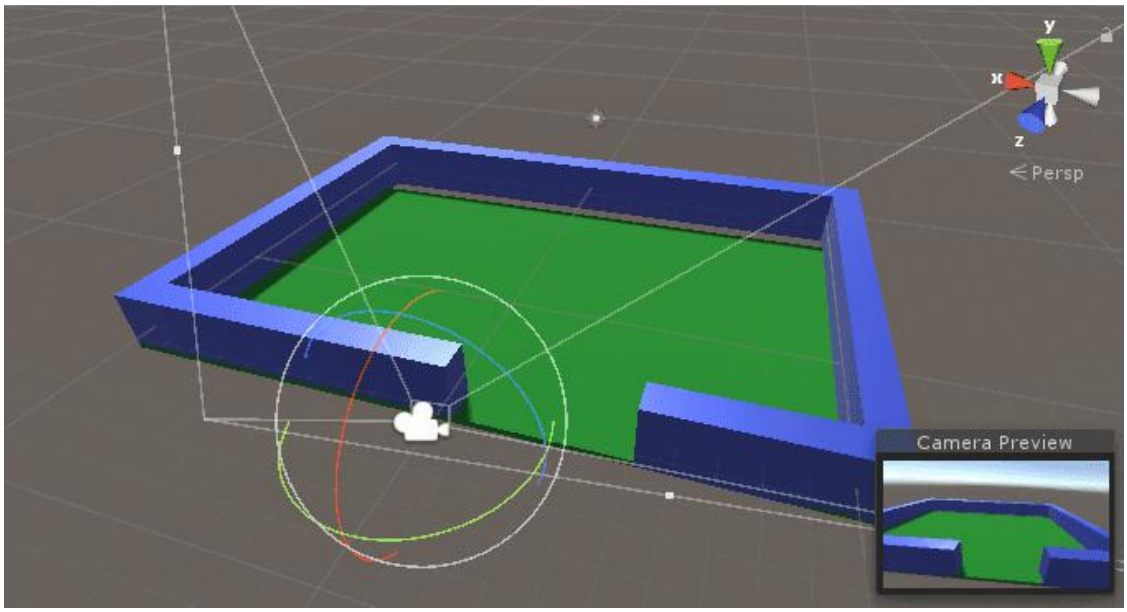
2.5. Lo que la cámara ve

Una de las formas de saber lo que se ve con la cámara es pulsar el botón "Play", para pasar al "modo de juego". Aun así, cuando se está ajustando la jugabilidad de un juego, especialmente en modo 3D, esto puede resultar muy lento.

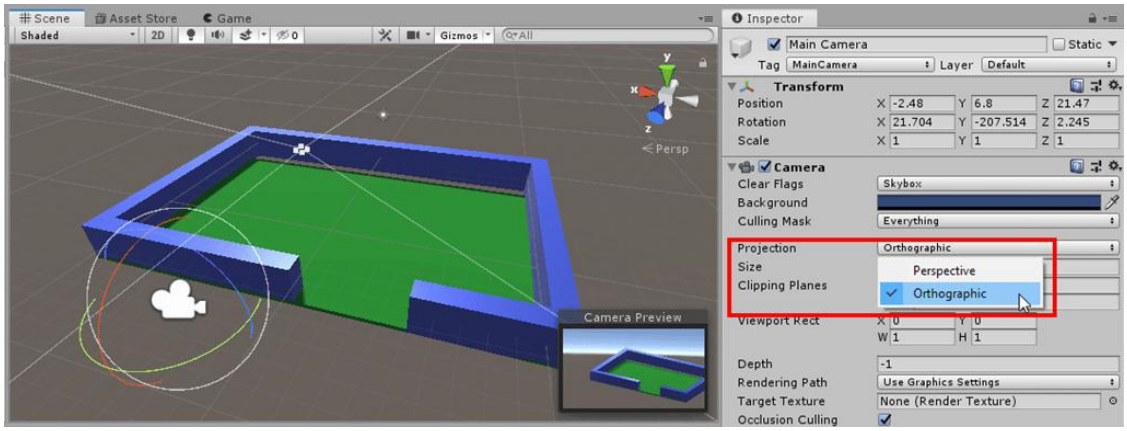
Una alternativa más sencilla es simplemente hacer un clic en la cámara, de modo que veremos en una esquina de la escena lo que la cámara mostraría en ese momento:



En este ejemplo, vemos que la cámara está en las coordenadas (0, 1, -10). Como nuestras paredes comenzaban en (0,0,0) y las hemos creado en posiciones crecientes de Z, se muestra una única pared.



Como detalle adicional, podemos escoger entre una cámara en perspectiva (lo habitual) o una cámara "ortográfica", más útil si queremos un juego en vista isométrica:



Ejercicio propuesto 2.5.1: Ajusta la posición y rotación de tu cámara, para que tu laberinto se vea lo mejor posible.

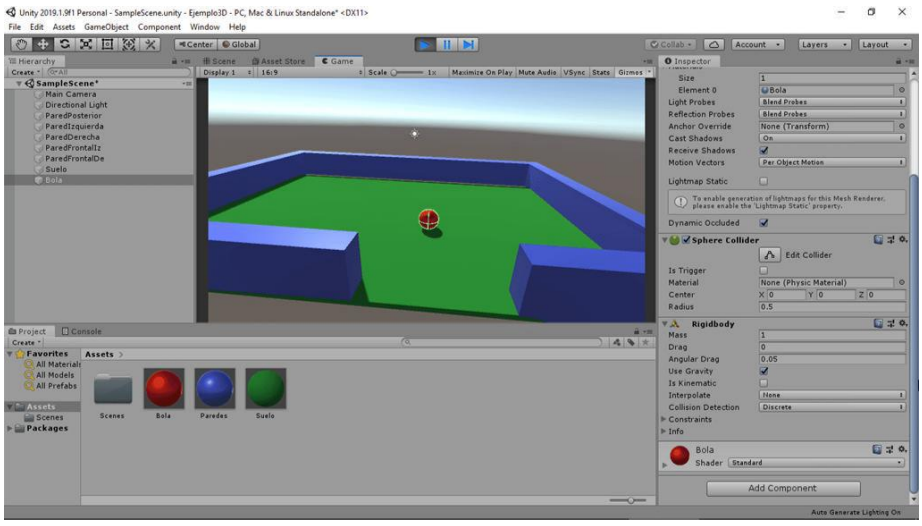
2.6. Un personaje esférico

Vamos a crear el "personaje" controlado por el usuario. Se tratará de una esfera que se moverá por el laberinto y que (más adelante) deberá esquivar obstáculos.

Crear la esfera es algo que ya sabemos hacer, porque es muy similar a las paredes:

- Creamos un objeto 3D, de tipo "Sphere".
- Ajustamos su posición y su escala.
- Creamos un material para ella y se lo asignamos.

Si la situamos un poco por encima del suelo y luego le añadimos un componente de tipo "RigidBody" (ya no "RigidBody2D", porque estamos trabajando en 3D), caerá hasta apoyarse en el suelo.



Como ya sabemos, una primera forma de hacer que se mueva nuestro "personaje" puede ser mirar el desplazamiento vertical (flecha arriba y flecha abajo) y horizontal (derecha e izquierda) de los dispositivos de entrada, y aplicar a su "RigidBody" una velocidad que tenga esas dos componentes, así:

```
using UnityEngine;

public class Bola : MonoBehaviour
{
	private Rigidbody body;
	float velocidad = 200f;

	void Start()
	{
		body = GetComponent<Rigidbody>();
	}

	void Update()
	{
		float avance = Input.GetAxis("Vertical") * velocidad
*			Time.deltaTime;
		float lado = Input.GetAxis("Horizontal") * velocidad
*			Time.deltaTime;
		body.velocity = new Vector3(avance, 0, lado);
	}
}
```

Pero quizá sea más entretenido incluir la posibilidad de **girar** si se pulsan las flechas hacia los lados (algo que se puede conseguir de varias formas, por ejemplo indicando "LookAt") y de **avanzar** si se pulsa la flecha hacia arriba (por ejemplo, sumando a su posición un vector "forward" en su sistema de coordenadas locales), así:

```
using UnityEngine;

public class Bola : MonoBehaviour
{
	private Rigidbody body;
	float velocidadAvance = 200f;
	float velocidadRotac = 10f;

	void Start()
	{
		body = GetComponent<Rigidbody>();
	}

	void Update()
	{
		float avance = Input.GetAxis("Vertical")
*			velocidadAvance * Time.deltaTime; float rotacion = Input.GetAxis("Horizontal")
*			velocidadRotac * Time.deltaTime; transform.RotateAround(Vector3.up, rotacion);
transform.position += transform.forward
*			Time.deltaTime * avance;
	}
}
```

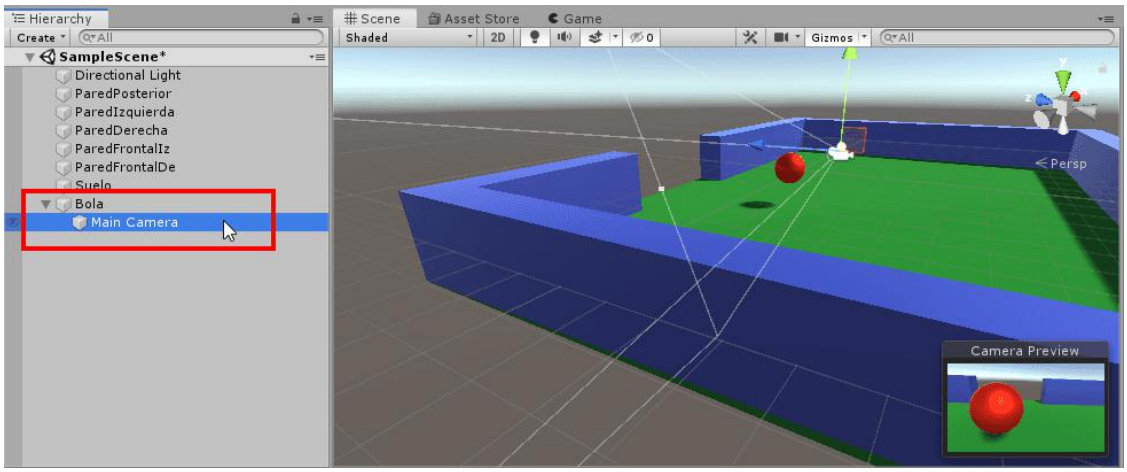
Nota: "Vector3.forward" es una forma más compacta de escribir "Vector3(0, 0, 1)", y serviría para avanzar en el sentido del eje Z. Por su parte, "transform.forward" es el equivalente en el **sistema**

de coordenadas local del objeto, por lo que hace que se mueva hacia la dirección a la que está apuntando.

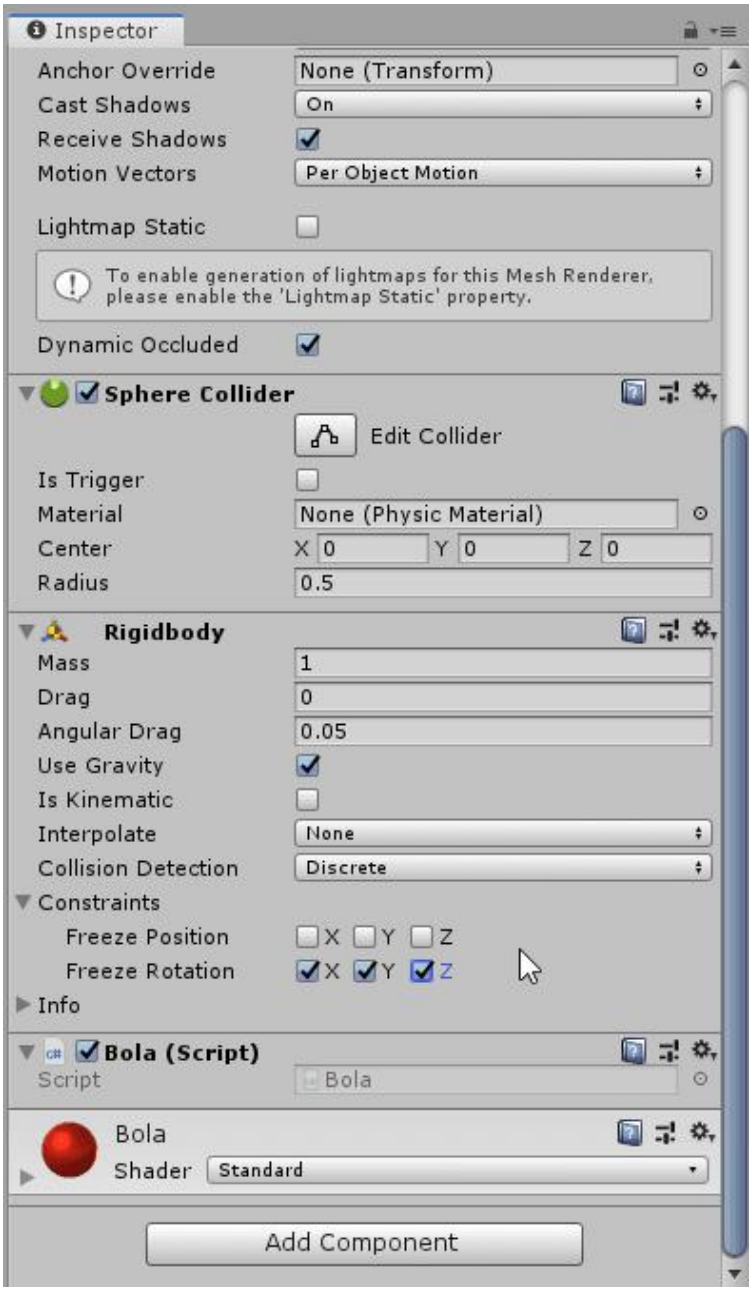
Ejercicio propuesto 2.6.1: Añade un personaje esférico que deba recorrer tu laberinto.

2.7. Una cámara que sigue al protagonista

Conseguir que la cámara siga al "personaje" puede ser tan sencillo como hacer que la cámara sea "hija" del personaje, en la jerarquía, y luego ajustar su posición y rotación hasta conseguir el efecto visual que deseemos.



Ahora debemos movernos para comprobar que el efecto visual es bueno en todo momento. El instante más crítico es el de las colisiones con los laterales, porque la bola puede girar en esos momentos, creando un efecto no deseado en la cámara. Lo podemos solucionar bloqueando la rotación de la esfera alrededor de los ejes X, Y, Z o todos ellos:



Ejercicio propuesto 2.6.1: Haz que la cámara siga a tu personaje.

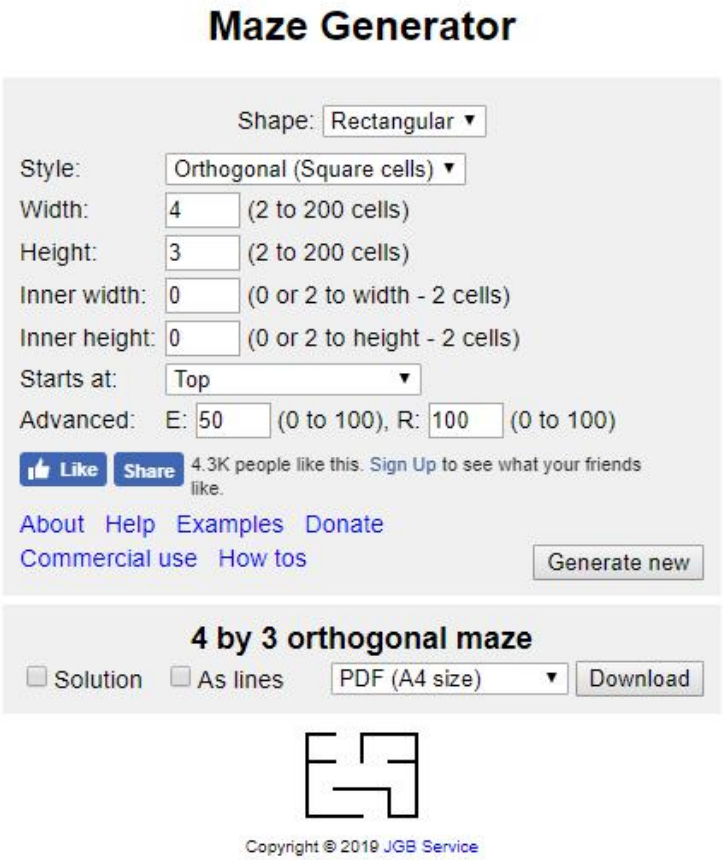
2.8. Diseñando un laberinto

Vamos a crear un laberinto que el usuario deba recorrer. Para eso, podemos partir de diseñar nosotros un laberinto "en papel", preferiblemente con la ayuda de una hoja cuadriculada, pero también podemos optar por usar generadores de laberinto online, como los que puedes encontrar en:

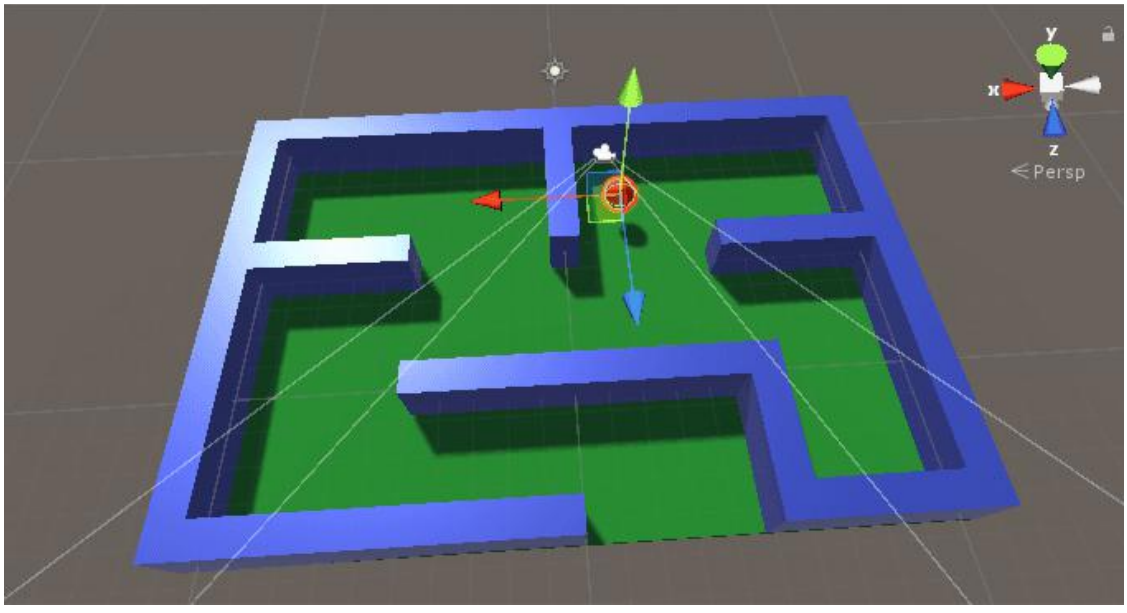
http://www.mazegenerator.net/

http://puzzlemaker.discoveryeducation.com/code/BuildMaze.asp

La ventaja de estos generadores es que se pueden usar para crear de forma automática laberintos de distintos tamaños, y luego sólo tendríamos que retocar "a mano" detalles menores, como por ejemplo en el caso de que no queramos que tengan una entrada y una salida, sino que se parta de un punto central y sólo exista una salida.



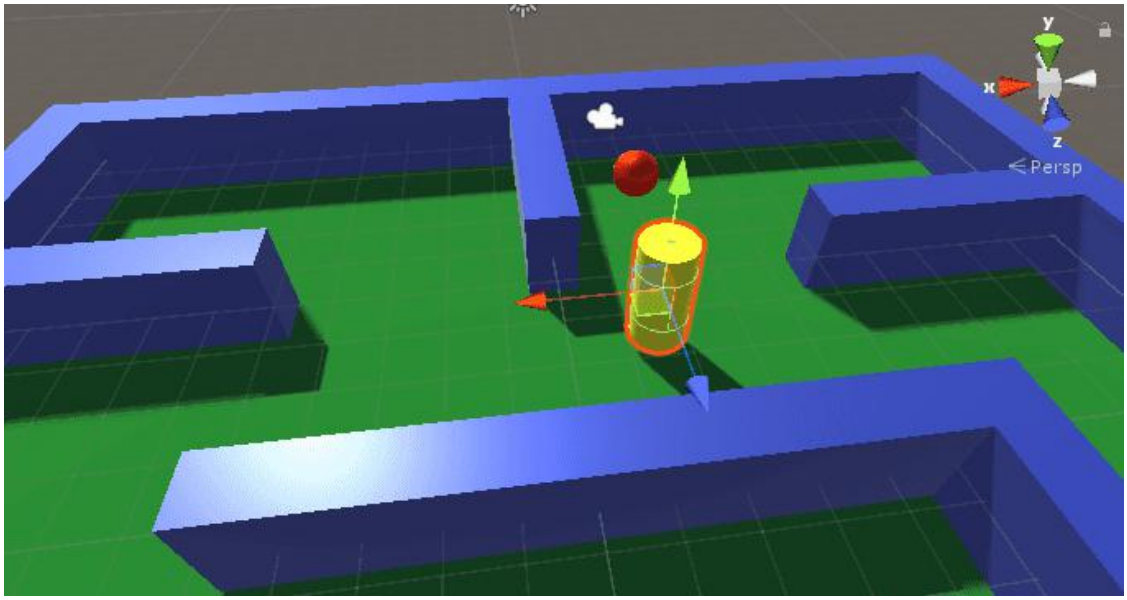
A partir de un diseño como ése, podríamos implementar algo como:



Ejercicio propuesto 2.8.1: Diseña tu propio laberinto.

2.9. Enemigos en el laberinto. Colisiones 3D

Para añadir enemigos en el laberinto, debemos escoger una forma y un color. Por ejemplo, puede ser un cilindro de color amarillo:



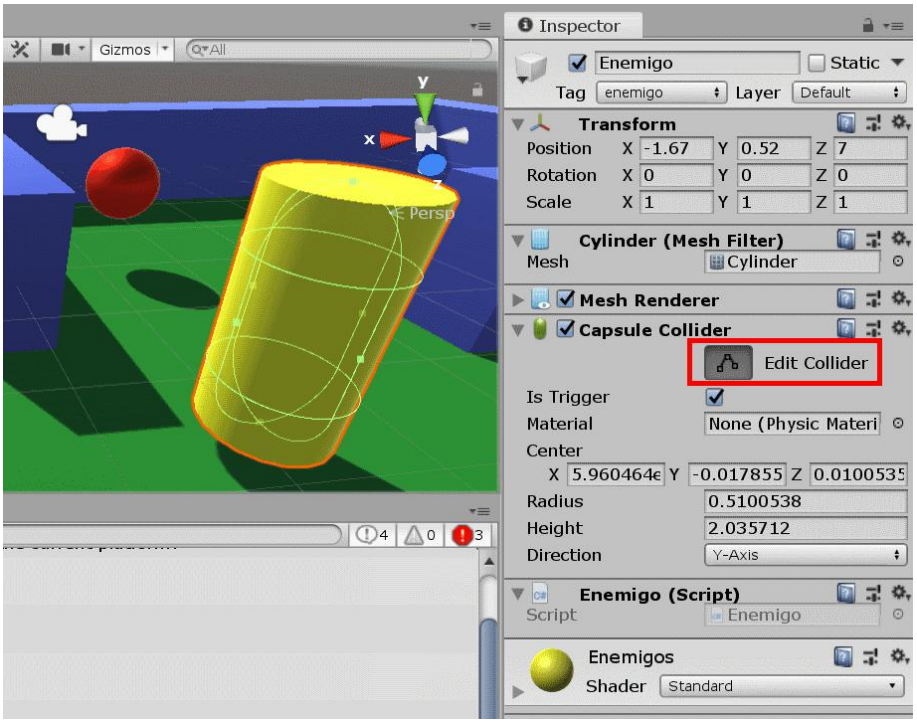
Y comprobar colisiones en 3D es casi igual que en 2D, pero parte del trabajo ya está hecho para nosotros: al añadir crear la esfera, Unity le habrá incluido un "SphereCollider" y con el cilindro habrá preparado un "CapsulleCollider". No es necesario añadir un RigibBody, porque la esfera ya tiene uno, y, de lo contrario, el cilindro "se caería" al ser golpeado por la bola. En vez de eso, activaremos "IsTrigger" en el cilindro y emplearemos el método "OnTriggerEnter".

```
using UnityEngine;

public class Enemigo : MonoBehaviour
{
	void OnTriggerEnter(Collider other)
	{
		Debug.Log("Tocado");
		Application.Quit();
	}

}
```

Y podemos afinar el tamaño del "collider", si lo deseamos, pulsando el botón "Edit Collider" en el inspector:



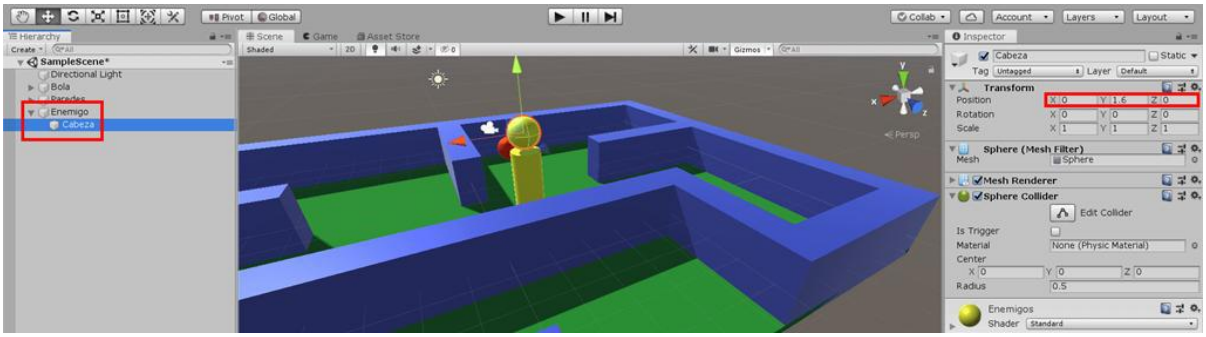
Nota: Application.Quit(); interrumpe la ejecución cuando se ha lanzado el ejecutable, pero se ignora en el editor.

Ejercicio propuesto 2.9.1: Añade un enemigo con la forma que desees. Ajusta el tamaño de su collider, según lo estrictas que quieras que sean las colisiones.

2.10. Relaciones padre/hijo. Coordenadas globales o locales

Al igual que ocurre en 2D, podemos crear un objeto que sea "hijo" de otro: si lo arrastramos "sobre" otro objeto en la jerarquía, pasará a depender de aquél, y, desde ese instante, sus coordenadas serán "locales", relativas a las del "padre", y si movemos o rotamos el padre, también se desplazará o rotará el "hijo".

Por ejemplo, podemos añadir una esfera "flotando" por encima del enemigo, como si éste estuviera formado por dos partes. A partir de ese momento, si movemos el enemigo, la esfera se desplazará junto con el cilindro:



Ejercicio propuesto 2.10.1: Crea un objeto vacío "Paredes", y añade como hijos suyos todas las paredes que has creado hasta ahora (y el suelo).

2.11. Varias vidas. Paso de mensajes.

En la mayoría de juegos, una colisión (con el enemigo, con un disparo, etc) no supone el fin del juego, sino perder una vida o una cierta cantidad de energía.

Vamos a hacer una primera aproximación, en la que todavía no usaremos ninguna clase que represente a todo el juego, sino que el propio "personaje" llevará cuenta de cuántas vidas le quedan y se encargará de volver a su posición inicial tras una colisión. De esta colisión le avisará el enemigo, que es el "Game Object" que las está comprobando.

Hacer que vuelva a su posición inicial es sencillo: creamos atributos para almacenar la X y la Z iniciales (la Y es la altura, que no va a cambiar en este juego), así como la cantidad de vidas:

```
float xInicial, zInicial;
int vidas = 3;
```

Y guardamos los valores iniciales de X y de Z en "Start":

```
xInicial = transform.position.x;
zInicial = transform.position.z;
```

Y podemos crear un método "PerderVida", que devuelva la bola a la posición inicial, reste una vida y compruebe si se han gastado todas:

```
public void PerderVida()
{
	Debug.Log("Una vida menos");
	transform.position = new Vector3(xInicial,
		transform.position.y, zInicial);
	vidas--;
	if (vidas <= 0)
	{
		Debug.Log("Partida terminada");
		Application.Quit();
	}
}
```

De modo que el script completo para la "bola" que maneja el jugador podría ser:

```
using UnityEngine;

public class Bola : MonoBehaviour
{
	float velocidadAvance = 200f;
	float velocidadRotac = 100f;
	float xInicial, zInicial;
	int vidas = 3;

	void Start()
	{
		xInicial = transform.position.x;
		zInicial = transform.position.z;
	}

	void Update()
	{
```

```
            float avance = Input.GetAxis("Vertical") * velocidadAvance
*                   Time.deltaTime;
            float rotacion = Input.GetAxis("Horizontal") * velocidadRotac
*                   Time.deltaTime;
            transform.Rotate(Vector3.up, rotacion);
            transform.position += transform.forward
                  * Time.deltaTime * avance;
      }

      public void PerderVida()
      {
            Debug.Log("Una vida menos");
            transform.position = new Vector3(xInicial,
                  transform.position.y, zInicial);
            vidas--;
            if (vidas <= 0)
            {
                  Debug.Log("Partida terminada");
                  Application.Quit();
            }
      }
}
```

Falta que el enemigo avise de que debe perder una vida. Puede que nuestro primer intento fuera hacer un "forzado de tipos", para indicar que el otro objeto es de tipo "Bola" y que, por tanto, tiene un método "PerderVida()" al que queremos llamar, pero ese planteamiento no funcionará en Unity, nos dirá que no puede convertir un "GameObject" en un objeto de tipo "Bola":

```
using UnityEngine;

public class Enemigo : MonoBehaviour
{
      void OnTriggerEnter(Collider other)
      {
/                 Aproximación que NO funciona en Unity ((Bola) other.gameObject).PerderVida();
      }
}
```

La "manera Unity" es usar el método "SendMessage" para indicar (como string) el nombre del método al que queremos llamar:

```
using UnityEngine;

public class Enemigo : MonoBehaviour
{
      void OnTriggerEnter(Collider other)
      {
            other.SendMessage("PerderVida");
      }
}
```

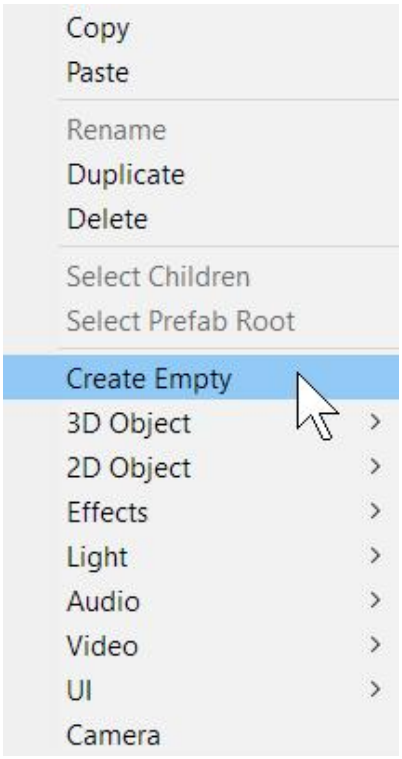
Ejercicio propuesto 2.11.1: Haz que tu personaje tenga varias vidas, que pierda una colisión, y crea una pantalla de bienvenida, a la que se volverá cuando ya no queden vidas.

2.12. Waypoints 3D

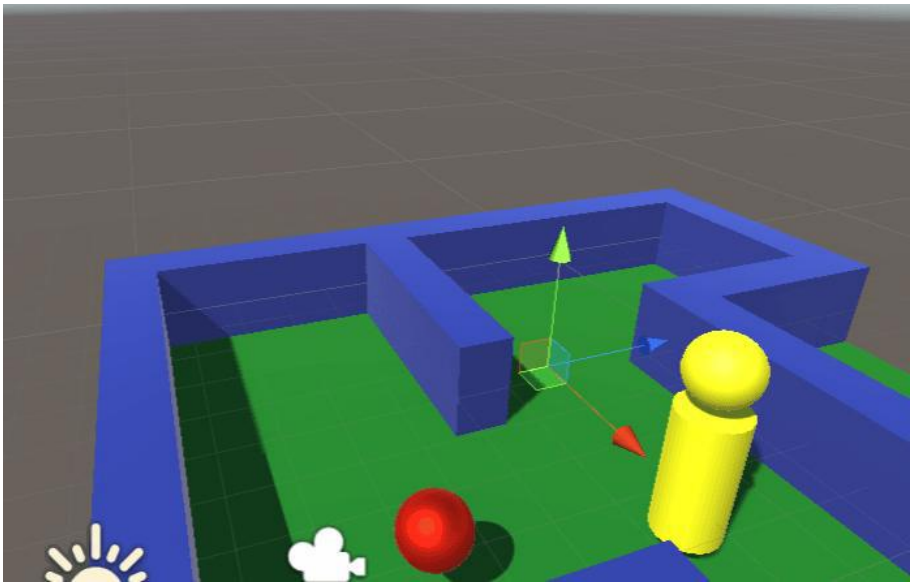
Vamos a hacer que el enemigo se mueva. Pero esta vez, en vez de moverse simplemente de un lado a otro, como hicimos con el juego en 2D, crearemos varios "puntos de referencia" ("waypoints") que irá recorriendo en orden.

Vamos a comenzar por la alternativa "artesanal" es crear una lista (o array) de GameObjects. Más adelante veremos la forma "moderna", usando "rejillas de navegación" ("NavMesh").

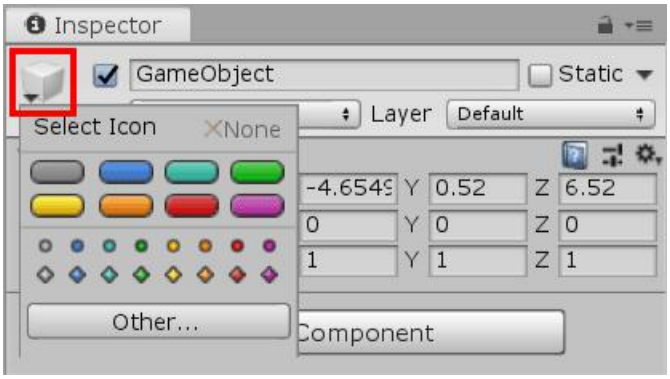
Podemos empezar por añadir un primer "GameObject" vacío al laberinto (botón derecho en la jerarquía, "Create Empty");



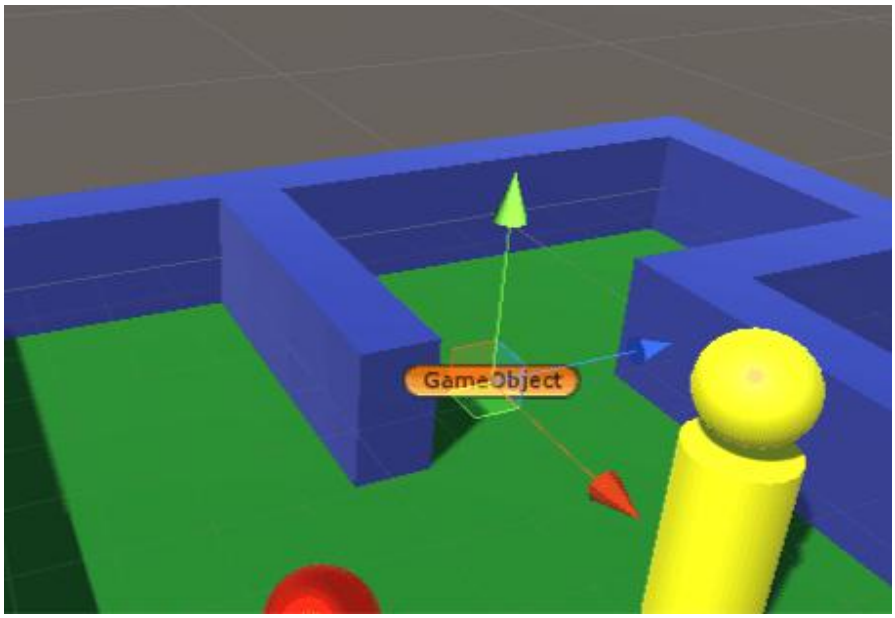
Y darle la misma coordenada Y (altura) que a nuestro enemigo. Pero los objetos vacíos sólo son visibles cuando están seleccionados:



Así que podemos asignarle un color, haciendo clic en el cubo que aparezca junto a su nombre en el inspector:



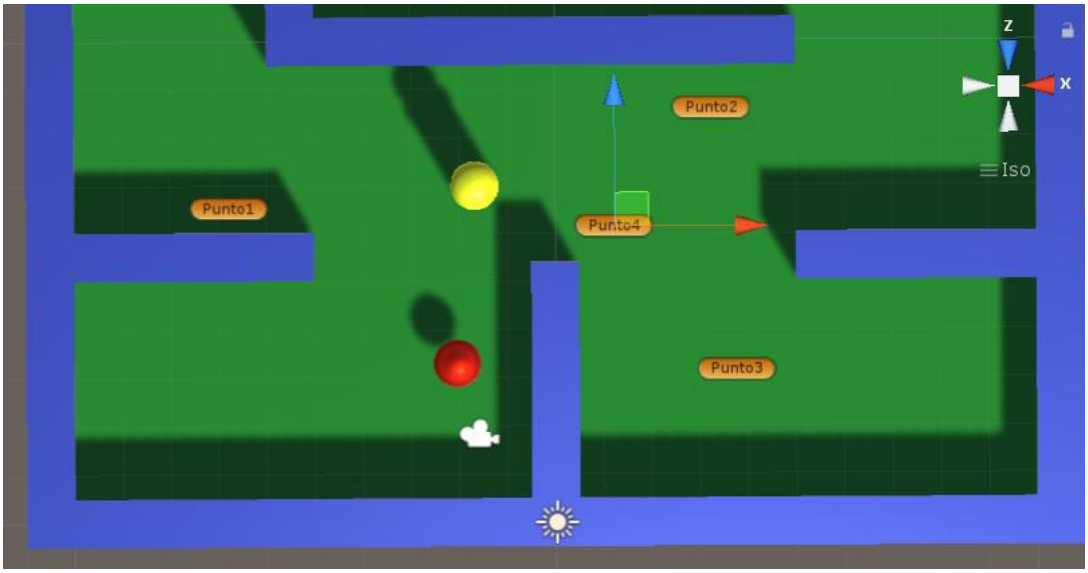
Y entonces pasará a ser visible en tiempo de diseño:



Convendría cambiarle el nombre por algo como "Punto1". Para ayudarnos a situar el objeto en una posición adecuada, podemos pasar a la vista superior. La forma de conseguirlo es acercarnos a los ejes que aparecen en la esquina superior derecha de la escena:

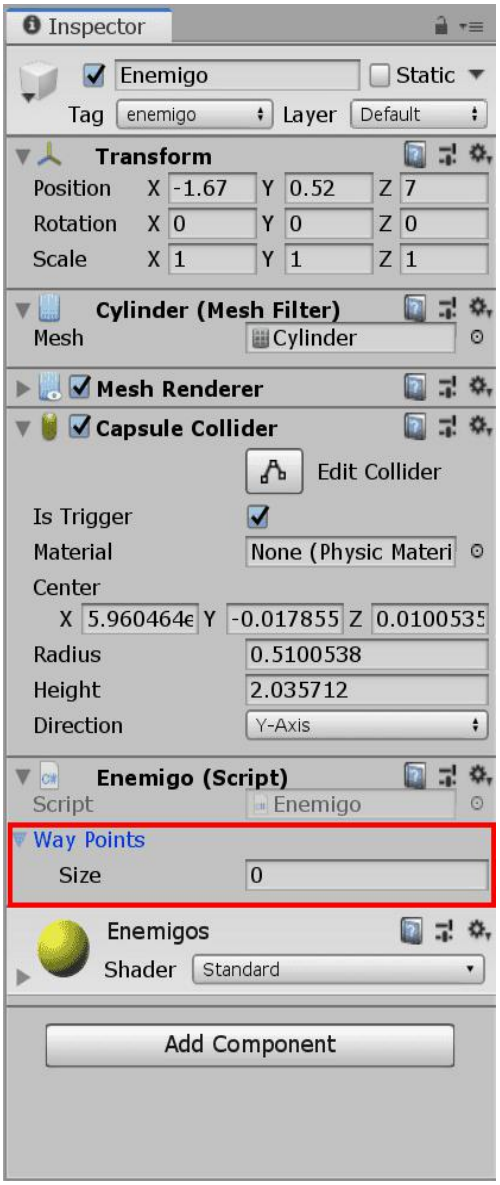
y hacer clic en el eje Y:

Con la ayuda de esta vista superior, podemos duplicar ese punto de referencia para crear varios más y repartirlos por la escena, en la secuencia en que queramos que recorra el enemigo:

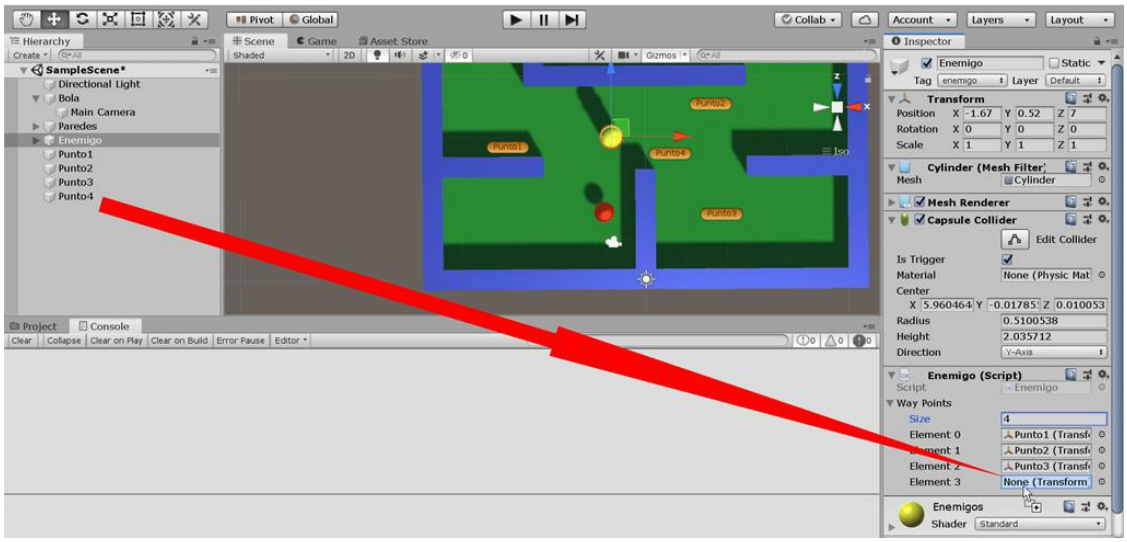


Una vez creados los waypoints en el interfaz visual, podemos definirlos como array de "Transforms" en el script el Enemigo, usando "SerializeField" para que sea accesible desde el editor:

```
[SerializeField] Transform[] wayPoints;
```



Y entonces podremos indicar que su tamaño (size) es 4, y arrastrar los puntos individuales desde la jerarquía al editor para enlazarlos:



Y ahora tenemos que crear la lógica de seguimiento. Almacenaremos siempre la siguiente posición a la que vamos a ir, para lo que crearemos un vector como atributo:

```
Vector3 siguientePosicion;
```

En "Start" le daremos como valor la posición inicial:

```
private void Start()
{
	siguientePosicion = wayPoints[0].position;
}
```

Y en "Update" usaremos "Vector3.MoveTowards" para calcular una nueva posición que esté entre la actual ("transform.position") y la siguiente ("siguientePosicion), usando una cierta velocidad de desplazamiento:

```
private void Update()
{
	transform.position = Vector3.MoveTowards(
		transform.position,
		siguientePosicion,
		velocidad * Time.deltaTime);
}
```

Y esa velocidad podría tener un valor inicial como

```
float velocidad = 2;
```

o incluso estar etiquetada con "SerializeField" para poderse modificar desde el editor.

Eso hará que se acerque hasta la posición 0 de nuestro array de "waypoints". Sólo falta que cuando lleguemos a ese punto (que podemos saber con "Vector3. Distance"), cambiemos de dirección hacia el siguiente punto, algo que podríamos conseguir con algo como

```
if (Vector3.Distance(transform.position,
	siguientePosicion) < distanciaCambio)
{
	numeroSiguientePosicion++;
	siguientePosicion = wayPoints[numeroSiguientePosicion].position;
```

```
}
```

Donde ese "número de la siguiente posición" debería empezar en 0, porque comenzamos dirigiéndonos a la primera posición del array de "waypoints":

```
byte numeroSiguientePosicion = 0;
```

Y esa distancia de cambio de destino sería un nuevo atributo, que también podríamos dejar accesible desde el editor si nos interesa:

```
float distanciaCambio = 0.2f;
```

Sólo falta tener en cuenta que cuando lleguemos a la posición 3, que es la última del array, deberemos volver a apuntar hacia la primera, que es la 0, por lo que el método "Update" realmente debería ser así:

```
private void Update()
{
	transform.position = Vector3.MoveTowards(
		transform.position,
		siguientePosicion,
		velocidad * Time.deltaTime);

	if (Vector3.Distance(transform.position,
		siguientePosicion) < distanciaCambio)
	{
		numeroSiguientePosicion++;
		if (numeroSiguientePosicion >= wayPoints.Length)
			numeroSiguientePosicion = 0;
		siguientePosicion = wayPoints[numeroSiguientePosicion].position;
	}
}
```

Con eso ya conseguimos que el Enemigo se vaya moviendo en secuencia entre 4 posiciones predefinidas (que realmente podrían ser tantas como quisiéramos).

El fuente completo de la clase Enemigo podría ser así:

```
using UnityEngine;

public class Enemigo : MonoBehaviour
{
	[SerializeField] Transform[] wayPoints;
	Vector3 siguientePosicion;
	byte numeroSiguientePosicion;
	float distanciaCambio = 0.2f;
	float velocidad = 2;

	private void Start()
	{
		siguientePosicion = wayPoints[0].position;
		numeroSiguientePosicion = 0;
	}

	private void Update()
	{
		transform.position = Vector3.MoveTowards(
			transform.position,
```

```
                siguientePosicion,
                velocidad * Time.deltaTime);

        if (Vector3.Distance(transform.position,
                siguientePosicion) < distanciaCambio)
        {
                numeroSiguientePosicion++;
                if (numeroSiguientePosicion >= wayPoints.Length)
                        numeroSiguientePosicion = 0;
                siguientePosicion = wayPoints[numeroSiguientePosicion].
                        position;
        }
    }

    void OnTriggerEnter(Collider other)
    {
        other.SendMessage("PerderVida");
    }
}
```

Ejercicio propuesto 2.12.1: Añade waypoints a tu enemigo.

2.13. Niveles de dificultad: FindObjectOfType, Time.timeScale

Si queremos que el juego sea cada vez más difícil, tenemos la posibilidad de más "escenas", cada una de ellas con más enemigos (y más rápidos) que los niveles anteriores.

Como ya hemos visto, para cargar otra escena usaríamos algo como "SceneManager.LoadScene("Nivel2");". Lo haremos en el próximo tema.

Otra alternativa más sencilla es reiniciar el nivel, cuando el personaje alcance la puerta de salida, pero cada vez con mayor velocidad. Esta velocidad se puede incrementar de dos formas:

- Podemos cambiar sólo la velocidad del Enemigo. Para eso, su atributo "velocidad" podría ser público, o, mejor, podríamos crear un método público encargado de incrementar su valor:

```
public void IncrementarVelocidad()
{
    velocidad += 0.5f;
}
```

Y, cuando sea el momento, el personaje (o la puerta, o el juego, según decidamos), podría "buscar al enemigo" y enviarle el mensaje de que debe incrementar su velocidad:

```
FindObjectOfType<Enemigo>().SendMessage("IncrementarVelocidad");
```

Si tuviéramos más de un enemigo, podríamos obtener todos ellos como array con "FindObjectsOfType<Enemigo>()", y recorrer ese array mandándoles el mensaje de acelerar.

- Una segunda alternativa más sencilla (aunque menos versátil) es acelerar todo el juego. Para eso podemos cambiar el valor de "Time.timeScale". que inicialmente es 1. Si lo convertimos a 1.1, estaremos acelerando todo el juego en un 10%, o un 20% si lo convertimos a 1.2:

```
Time.timeScale += 0.2f;
```

El inconveniente de esta última forma de trabajar es que también acelerará el personaje (hasta el punto de poder llegar a atravesar paredes), las pausas que hubiéramos previsto serían más cortas, etc.

Ejercicio propuesto 2.13.1: Añade una comprobación de colisiones al llegar a la salida, y haz que se reinicie el personaje con mayor velocidad.