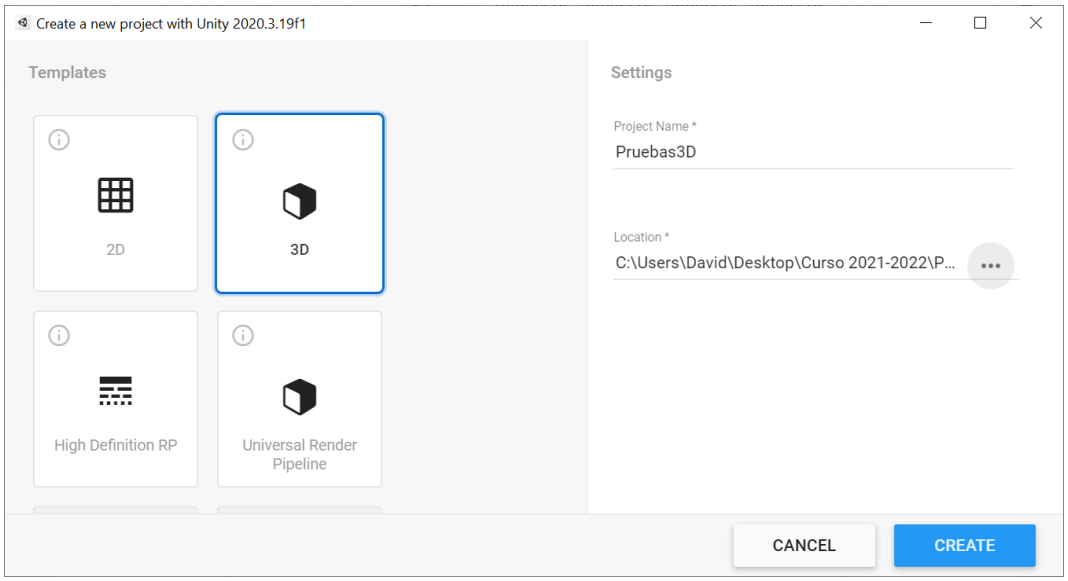
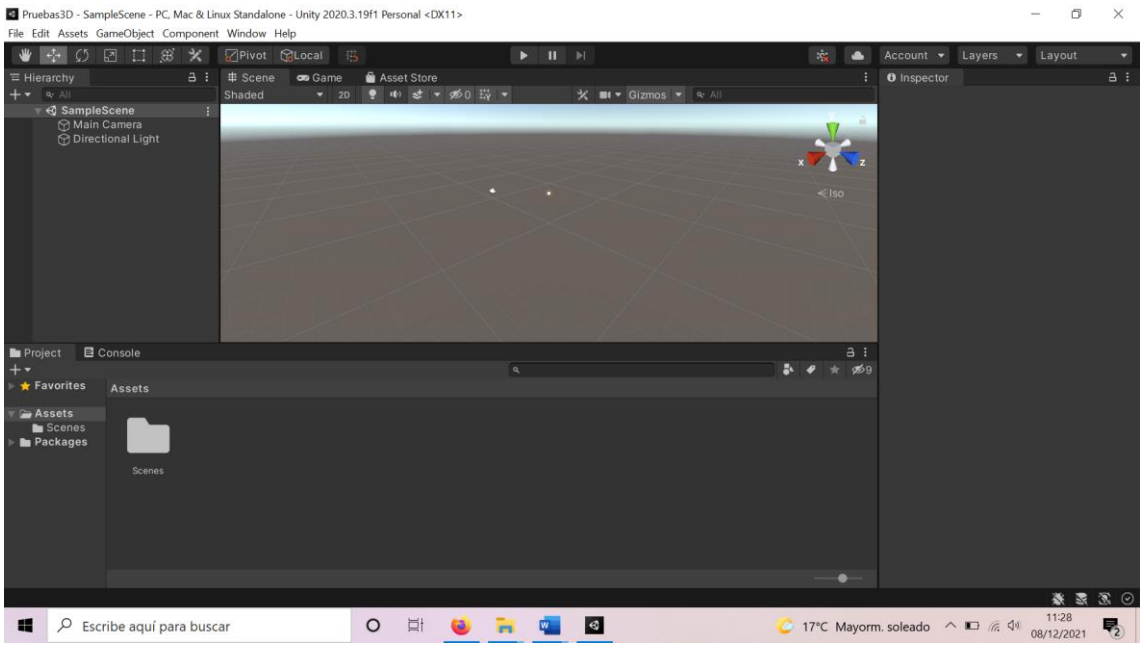
Tema 3. Primer contacto con 3D

1. El entorno 3D

Para crear un juego en 3D empezaremos un nuevo proyecto, en el cual escogeremos que sea de tipo 3D:



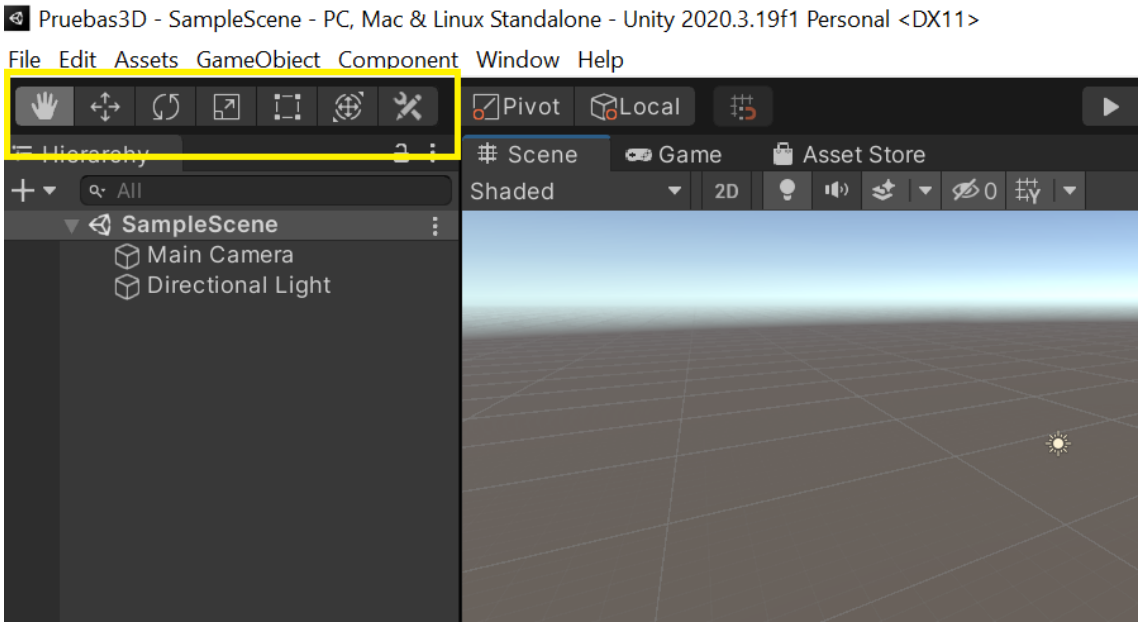
Tendremos una apariencia similar a la que teníamos en 2D, pero la parte de la escena cambia para simular un entorno 3D. Aparecerá todo en perspectiva y tendremos una cámara y un punto de luz:



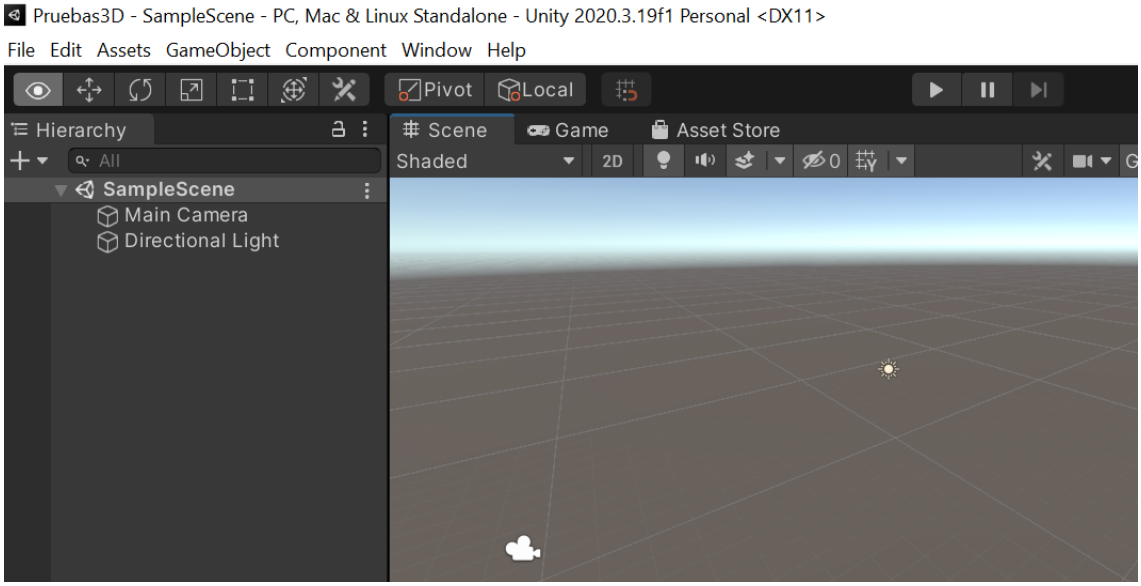
Tenemos distintas formas de “movernos” por la escena creada.

La primera es utilizar **la rueda del ratón** para acercarnos o alejarnos.

Si pinchamos la “mano” que aparece en la barra de herramientas, podremos mover la escena **utilizando el botón izquierdo** del ratón. Si al mismo tiempo pulsamos las **Mayúsculas** del teclado este movimiento será más rápido.



Si utilizamos el **botón derecho del ratón** aparecerá un "ojo" que nos permitirá **"rotar"** la escena en la dirección que nos interese:



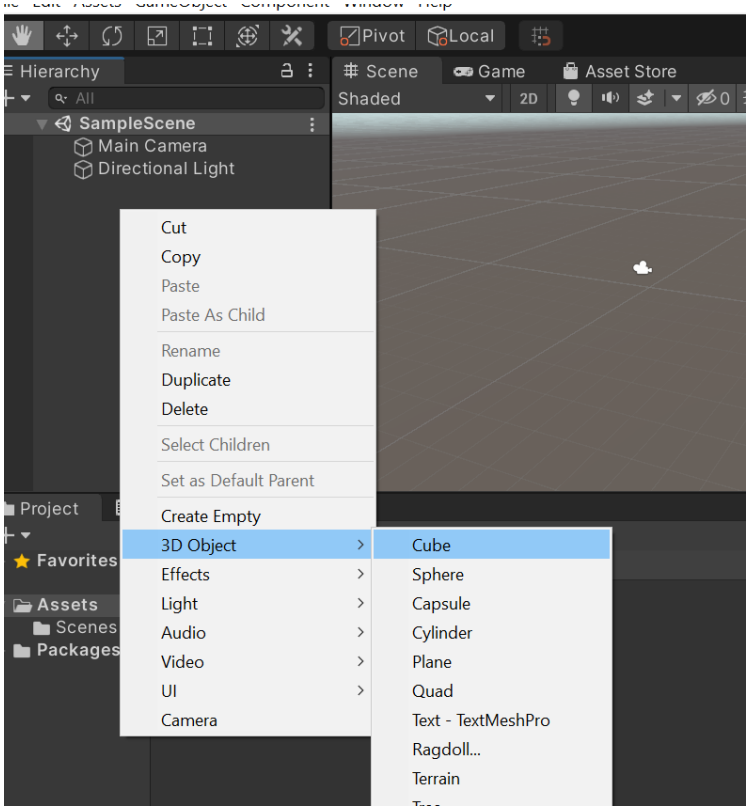
También nos podemos mover con las **flechas del teclado**.

Al pulsar el botón derecho (aparece el "ojo") podemos utilizar las teclas A (izquierda), D (derecha), W (acercar), S (alejar), Q (abajo), E (arriba).

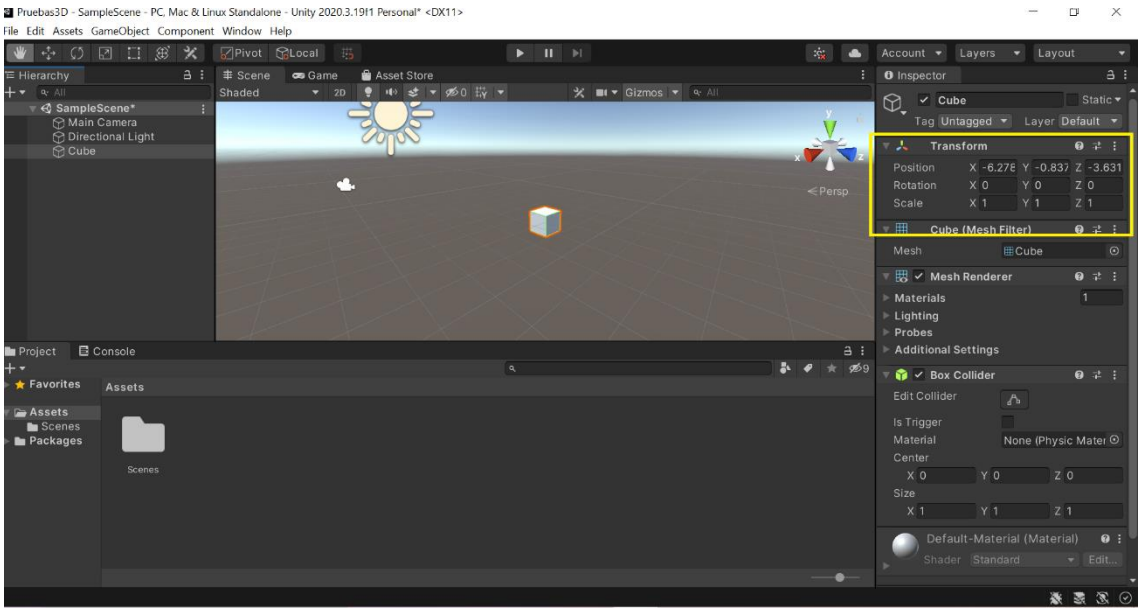
Vídeo movimiento por la escena (iniciar sesión con @alu.edu.gva.es)

2. Primer objeto 3D. Cambio de posición y tamaño.

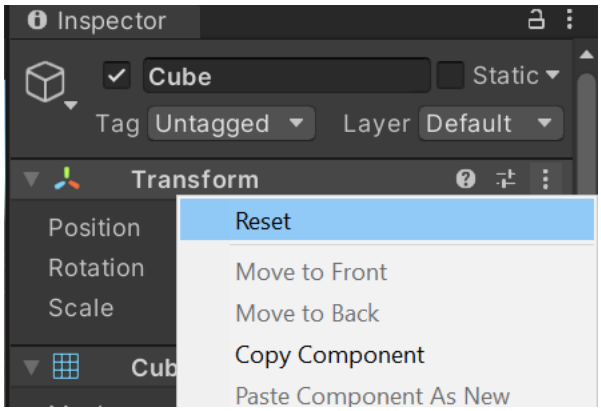
Vamos a crear nuestro primer objeto en 3D que será un cubo. Para ello, desde la jerarquía, simplemente le daremos al botón derecho, menú **3D Object->Cube**.



La posición en la que aparecerá este cubo es en el centro de lo que estamos visualizando, que no tiene que coincidir con el "centro" de la escena:

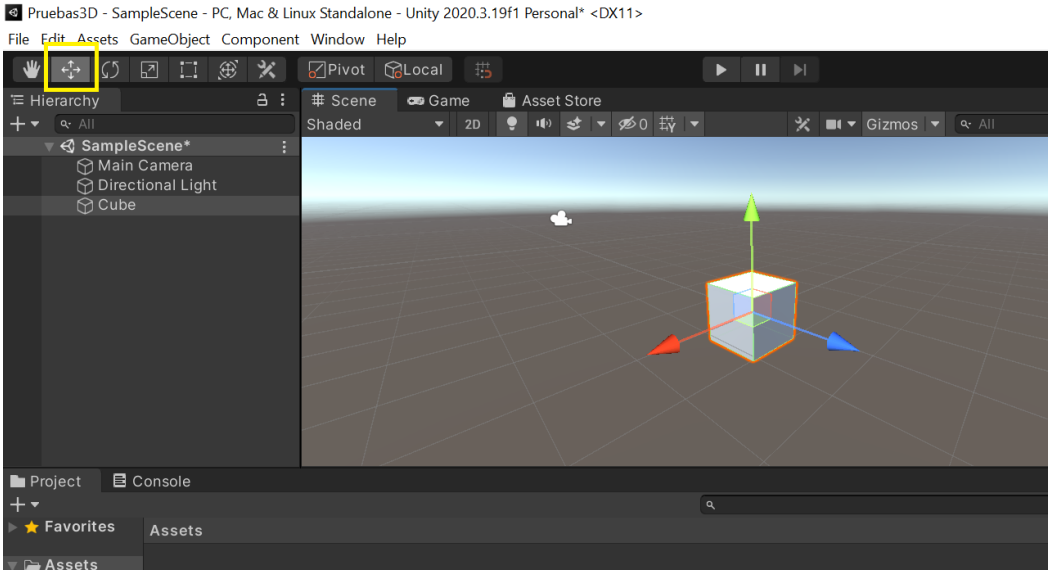


Para moverlo a una posición que nos interese, podemos utilizar su propiedad Transform. Por ejemplo, podemos moverla a la posición (0, 0, 0). Recordad que eso lo podemos hacer con la opción Reset:

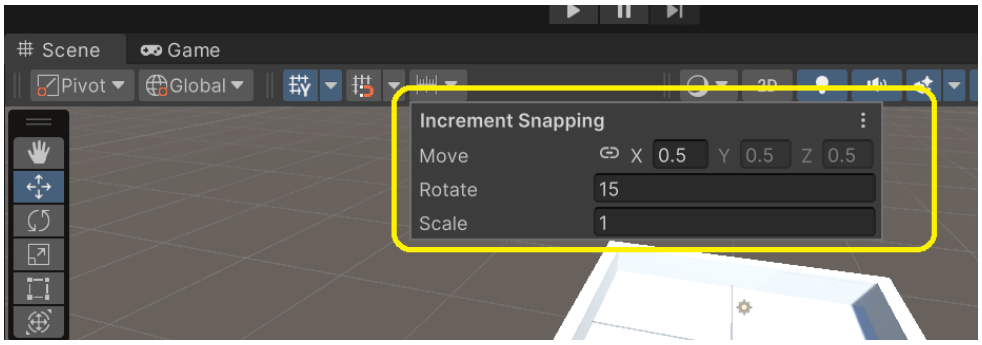


Podemos **mover** el elemento cambiando las posiciones de sus coordenadas X, Y, Z de su propiedad Transform

Otra forma de mover los elementos es pulsar, la herramienta **Mover**, de manera que aparecerán 3 flechas que nos permitirán moverlo **en cada uno de los ejes** y también 3 **planos** que nos permitirán moverlo en dos dimensiones.



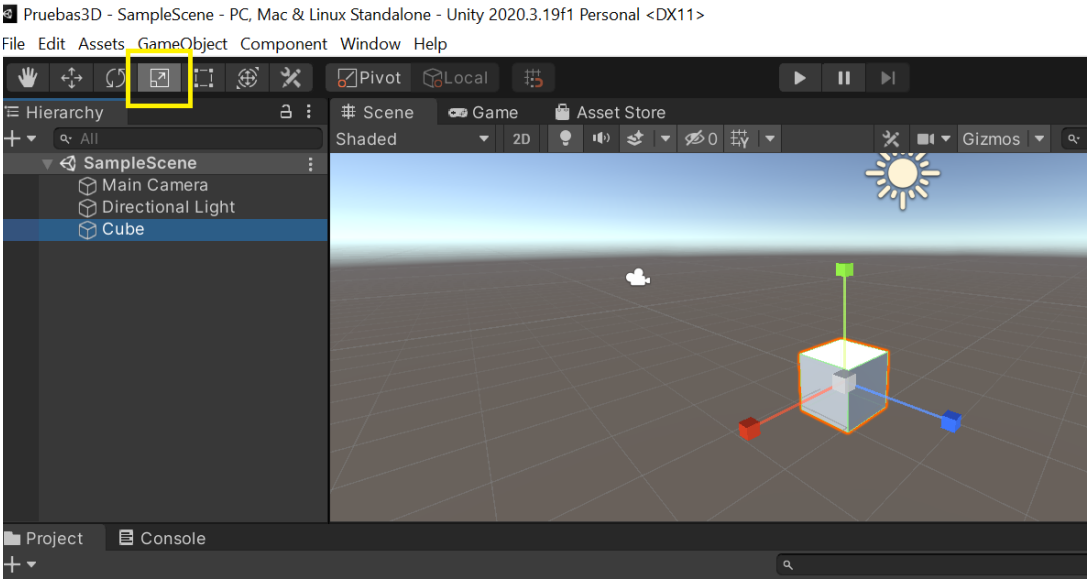
Si al mismo tiempo que movemos **pulsamos la tecla Ctrl**, lo haremos **a saltos** según la unidad que tengamos dentro de la opción Snap Increment:



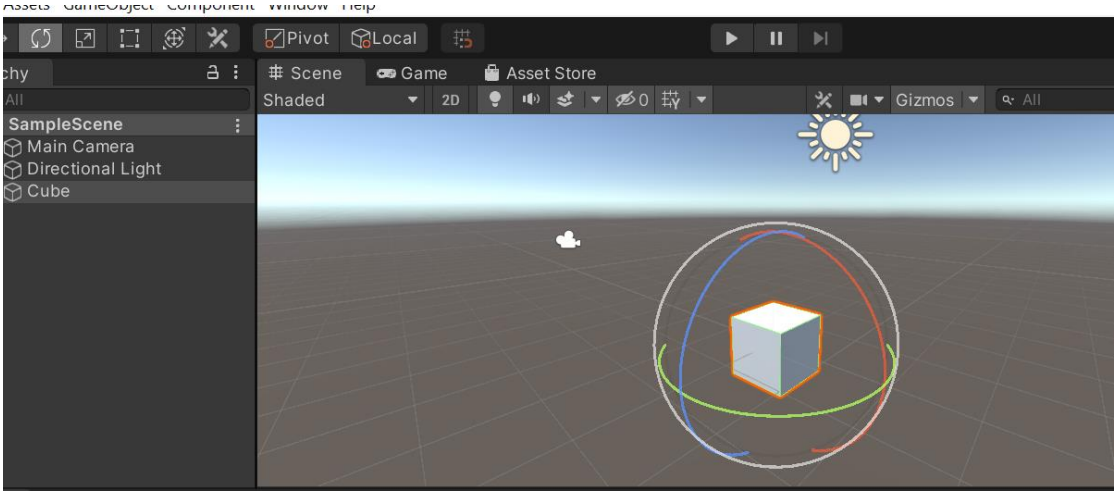
En este caso el salto será de 50 en 50 cms…

Debemos tener en cuenta que la idea de los 3 ejes es que el eje X sea el horizontal, el eje Y el vertical (al igual que pasaba en 2D) y el eje Z será la profundidad…

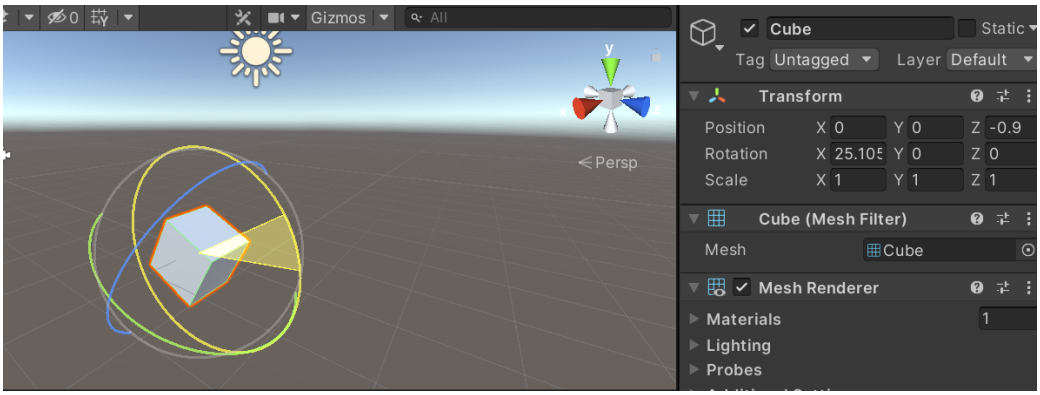
Para cambiar o escalar el tamaño del objeto lo podemos hacer de nuevo a través de su propiedad Transform (Scale), o utilizando la **herramienta Escalar** que mostrará 3 pequeños cuadrados para "estirar" el objeto en el eje que nos interese, o bien un cuadrado central que nos permitirá escalar en las 3 dimensiones:



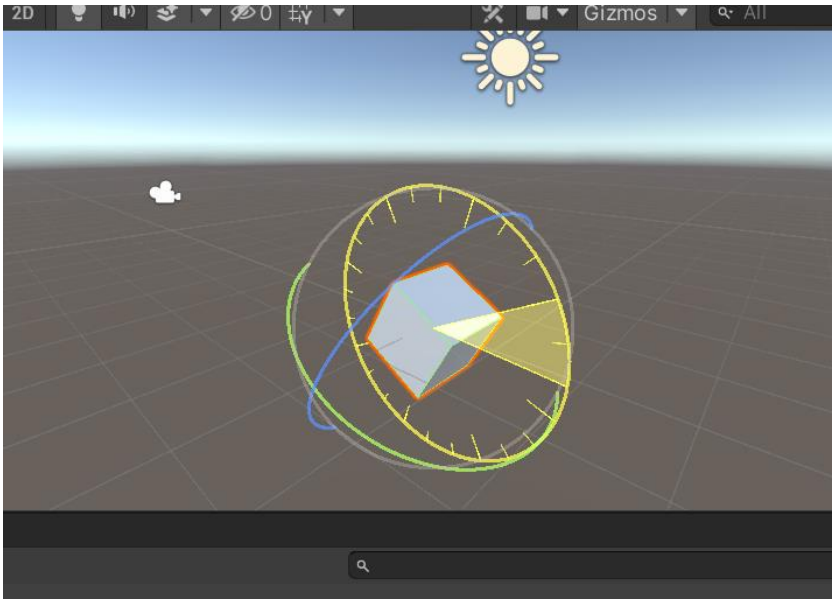
De nuevo, para rotar el elemento lo podremos hacer a través de la propiedad Transform, o bien con la herramienta Rotate, que hace aparecer tres círculos que podemos utilizar para rotar en los 3 sentidos posibles:



Conforme vayamos rotando cambiará el valor en el transform y además aparecerá un círculo indicando cuánto se ha rotado:



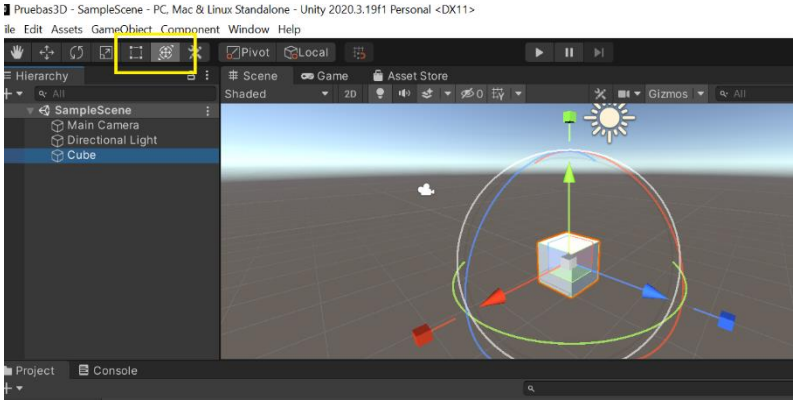
De nuevo, si conforme rotamos pulsamos la tecla Ctrl, la rotación se hace “a saltos” dependiendo del valor que tenemos en Snap Settings (ahora mismo 15 grados):



Tenemos otras dos opciones.

Rect Tool nos permite cambiar el tamaño en dos dimensiones.

Y otra opción que os permite al mismo tiempo Mover, Escalar y Rotar.



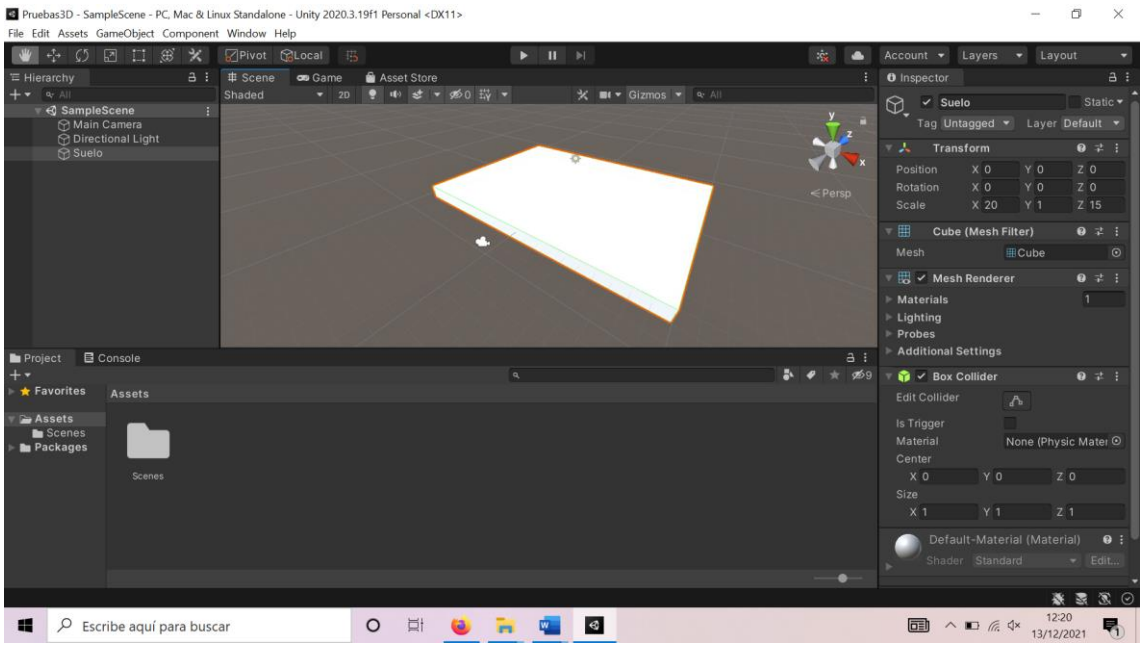
[Vídeo mover, rotar, escalar objeto.]

3. Empezando un laberinto.

Vamos a empezar a construir un laberinto para que nuestro objeto lo pueda recorrer. Para ello inicialmente vamos a crear el suelo y las paredes exteriores:

Vamos primeramente a **crear el suelo**.

Podemos hacerlo con un cubo con las siguientes dimensiones, x: 20, y: 1, z: 15 y le cambiaremos el nombre:

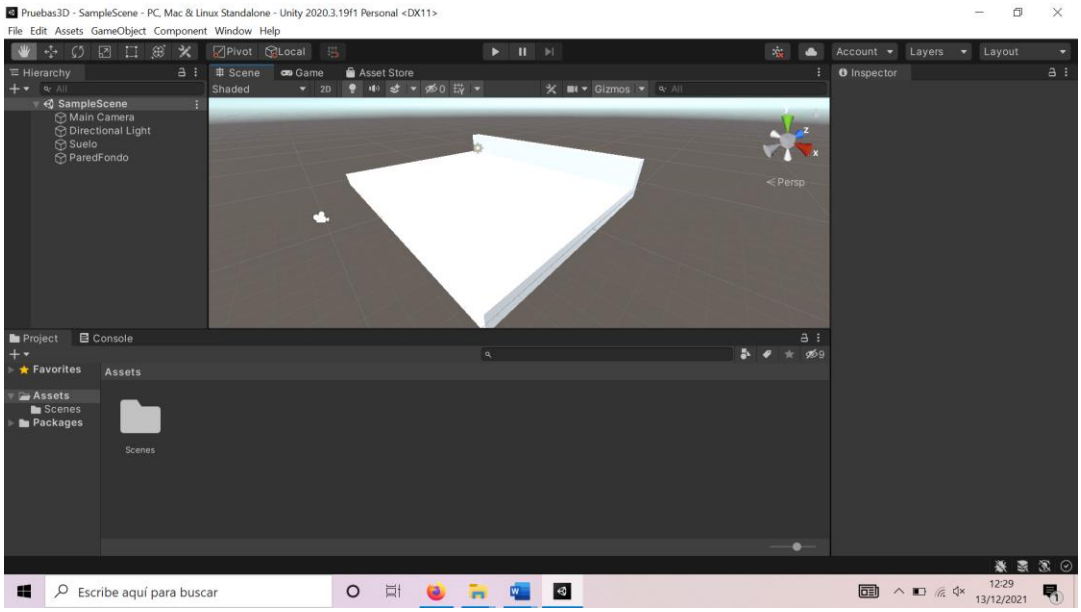


Otra posible opción es utilizar un elemento de tipo Plane (que por defecto tiene un tamaño de 10x10).

Nuestras paredes las vamos a construir con un cubo, modificando su escala y poniendo altura 2, anchura 1 y luego jugaremos con la longitud para conseguir el tamaño que queremos.

Por ejemplo, para crear la pared del fondo vamos a seguir los siguientes pasos.

- Creamos un cubo.
- Le cambiamos las dimensiones a x: 20, y: 2, z: 1
- Le cambiamos el nombre por ParedFondo



Una vez creada podemos **ajustarla** para que esté pegada a la otra.

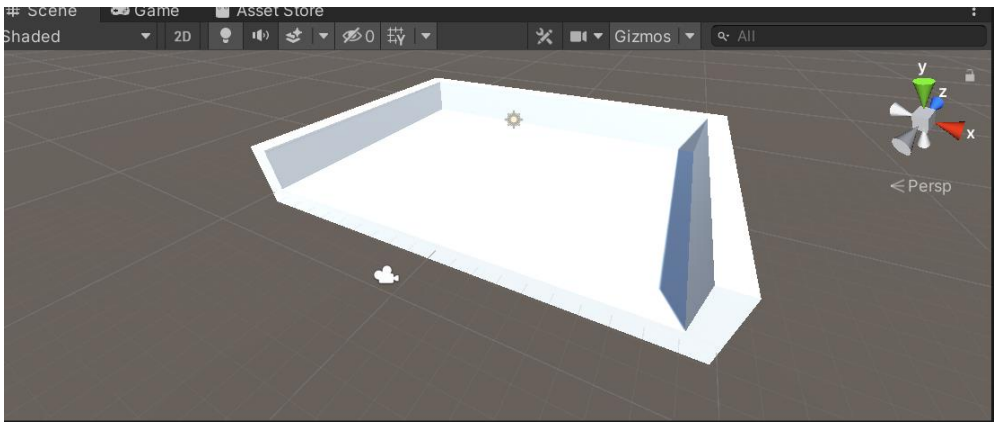
Lo podemos hacer:

- calculando su posición "con lápiz y papel"
- arrastrando con el ratón.
- Arrastrar pulsando Ctrl, avanza de casilla en casilla y se ajusta con más facilidad.
- Arrastrar pulsando Ctrl + Mays. Modo ajuste, el objeto se pegará a los laterales más cercanos de otros objetos.
- **Pulsando la tecla V. Modo "ajustar vértices", podemos mover el vértice de la pared para que coincida con otro vértice existente**.

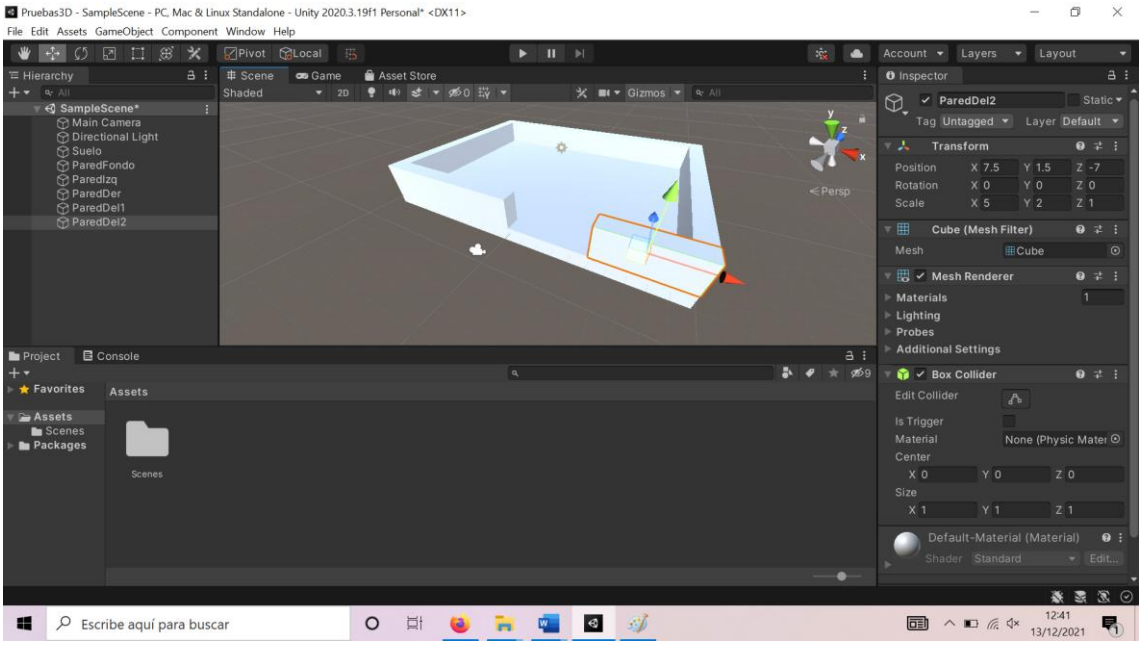
Podríamos ahora **crear la ParedIzquierda** con, por ejemplo, dimensiones x: 1, y: 2, z: 15

De nuevo la podemos ajustar (quizá lo más cómodo es con el modo "ajustar vértices") para que esté pegada a la pared del fondo y al suelo.

Para la pared de la derecha podemos duplicar esta que acabamos de crear (Ctrl + D) y moverla a la posición que nos interesa.

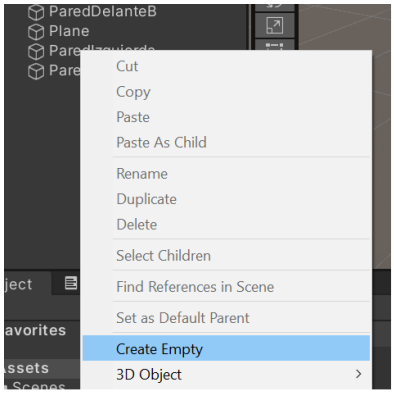


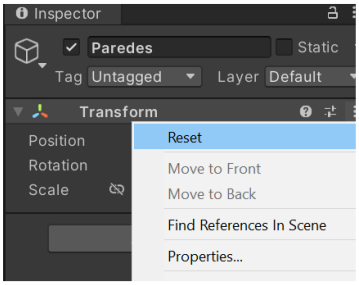
Para las paredes de delante podemos duplicar la pared del fondo y ajustar la escala y la posición:

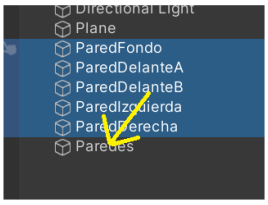


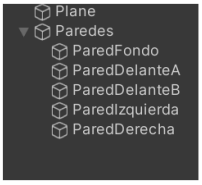
[Vídeo suelo y paredes del laberinto.]

Para que nuestra jerarquía no se llene de demasiados elementos podemos crear un elemento vacío, a continuación ese elemento hacerle reset para que esté situado en la posición 0, 0, 0, y arrastrar las paredes dentro del mismo:



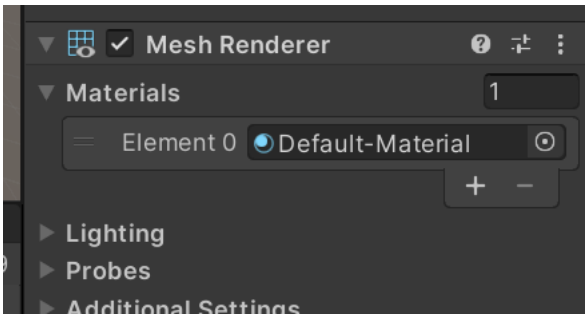






Materiales

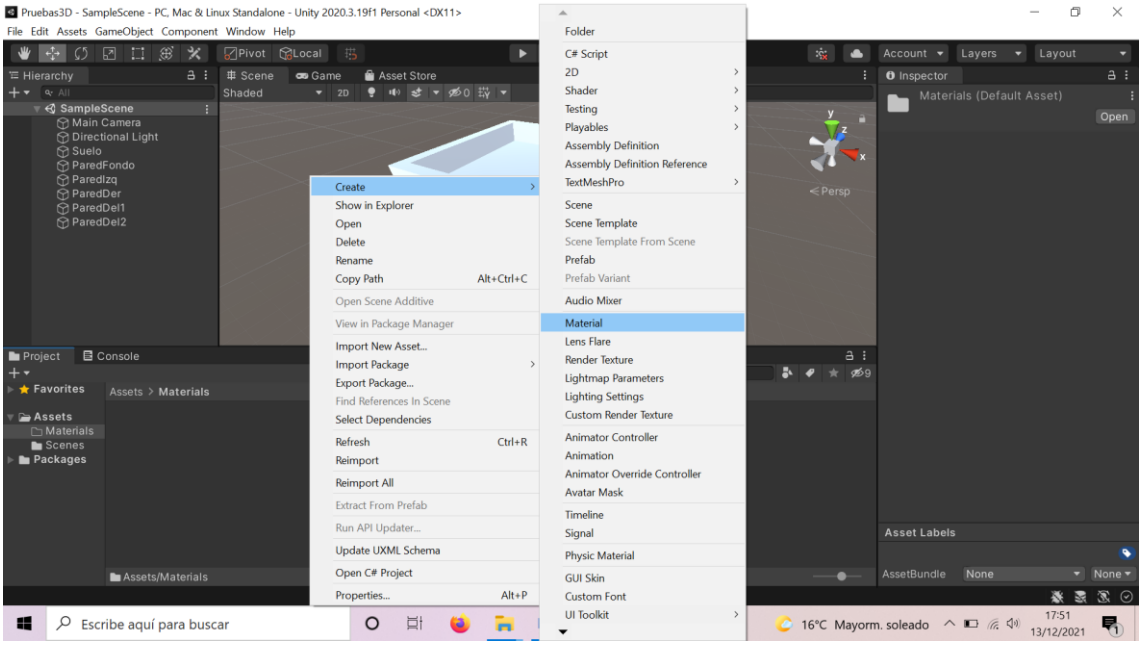
Los elementos que creamos nuevos aparecen con el material por defecto.



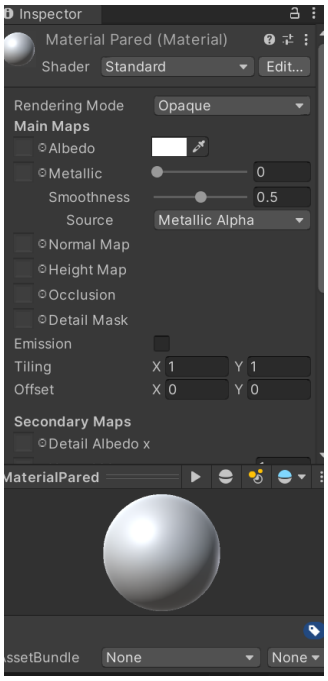
Los **materiales permiten cambiar el aspecto** de nuestros objetos, su color, texturas, reflejos…

Vamos a crear una nueva carpeta en el proyecto llamada Materials donde empezaremos a colocar nuestros nuevos materiales.

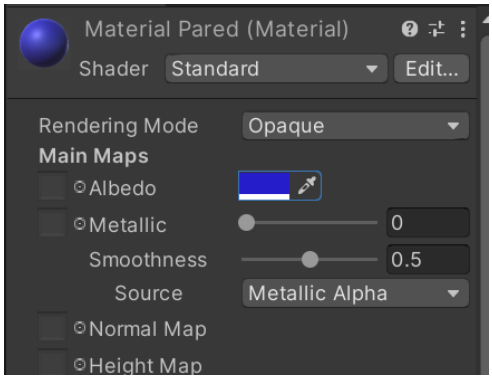
Para crear un nuevo material, sobre esa carpeta le daremos al botón derecho **Create / Material**:



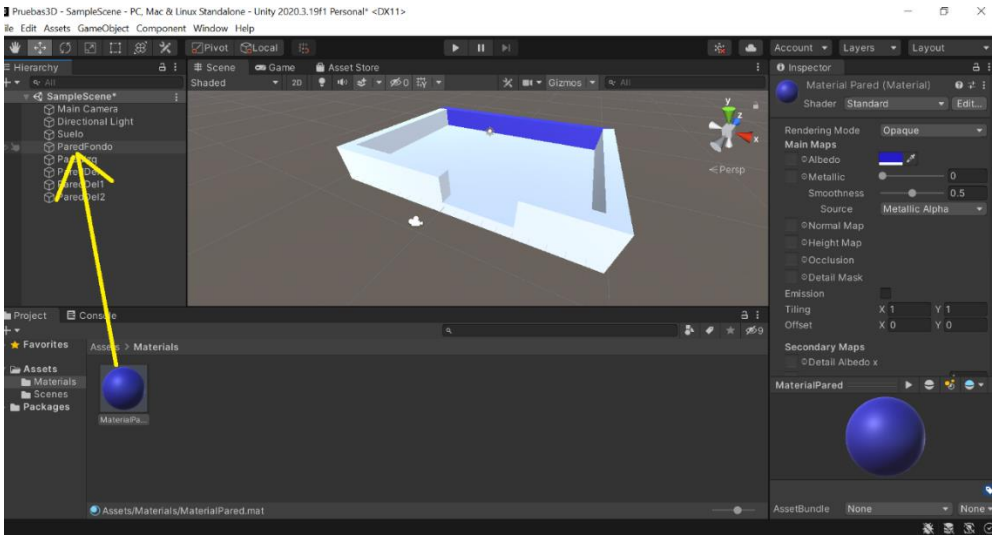
Le damos un nombre (por ejemplo, MaterialPared) y podemos ver sus propiedades en el Inspector:



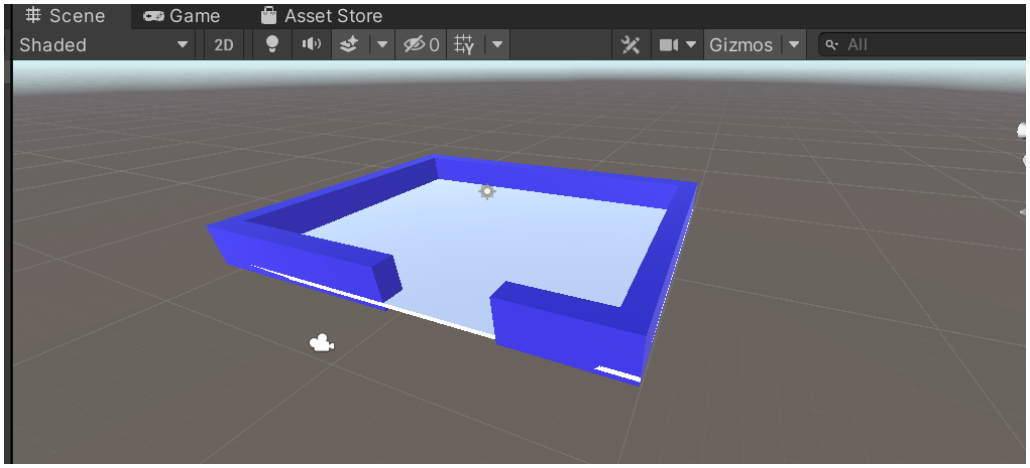
Vamos a cambiar de momento la propiedad **Albedo**, eligiendo el color que nos apetezca.



Para aplicar ahora ese material, simplemente lo tenemos que arrastrar desde la carpeta de materiales al objeto que queremos que lo tenga, bien en la jerarquía, bien en la escena:

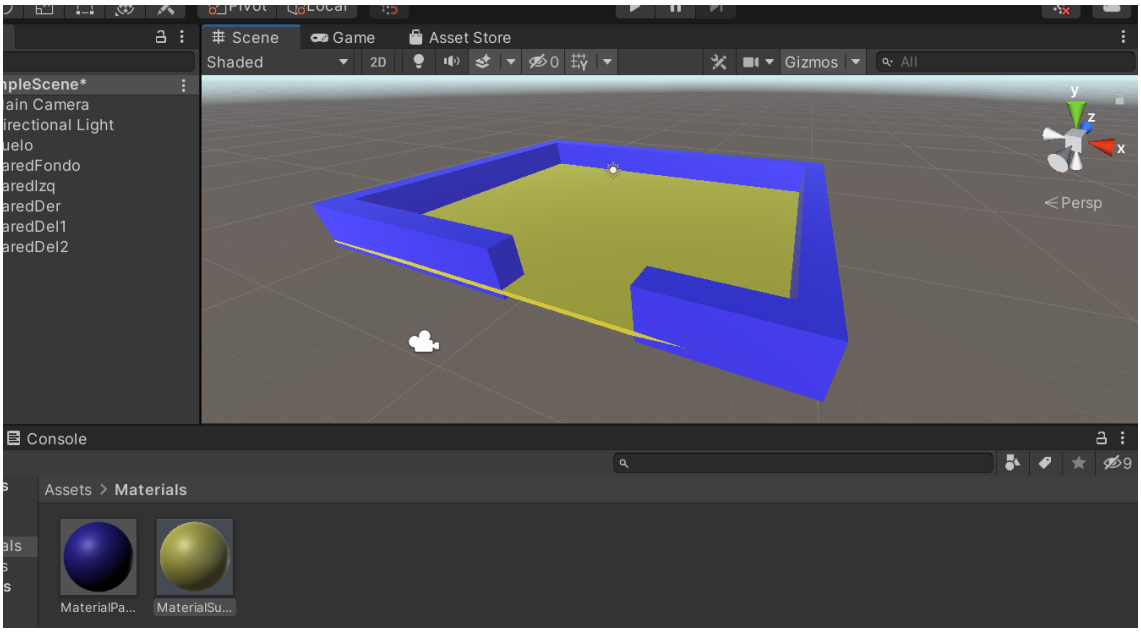


Vamos a aplicar esa textura a todas las paredes:



Podemos probar a cambiar otras propiedades del material, como el Rendering Mode, Metallic y Smothness.

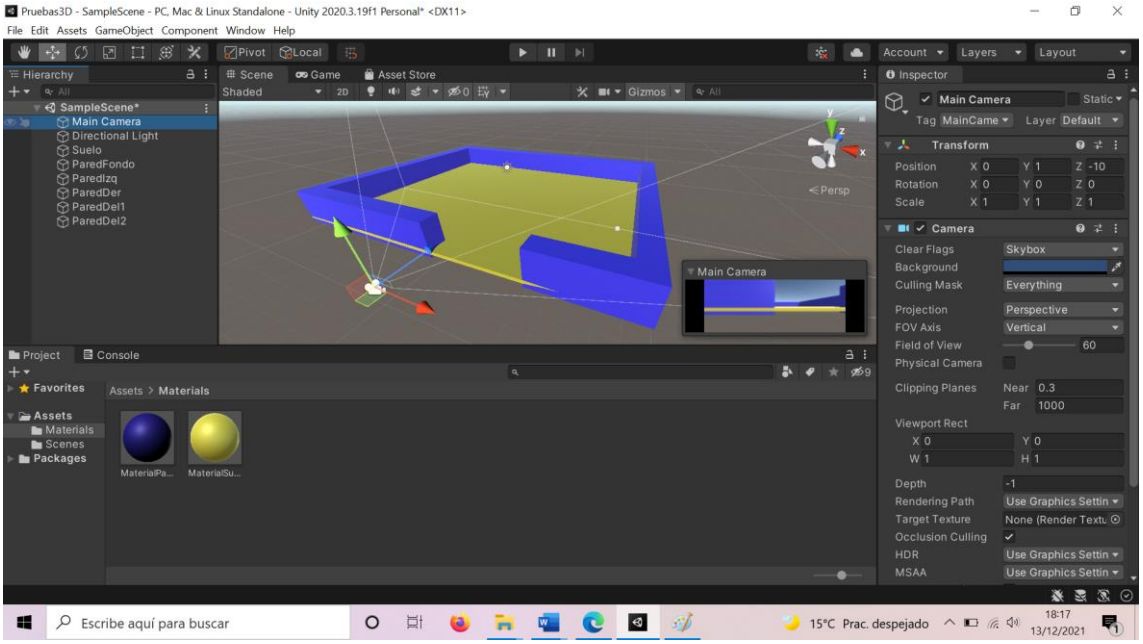
Vamos a crear otro material para el suelo:



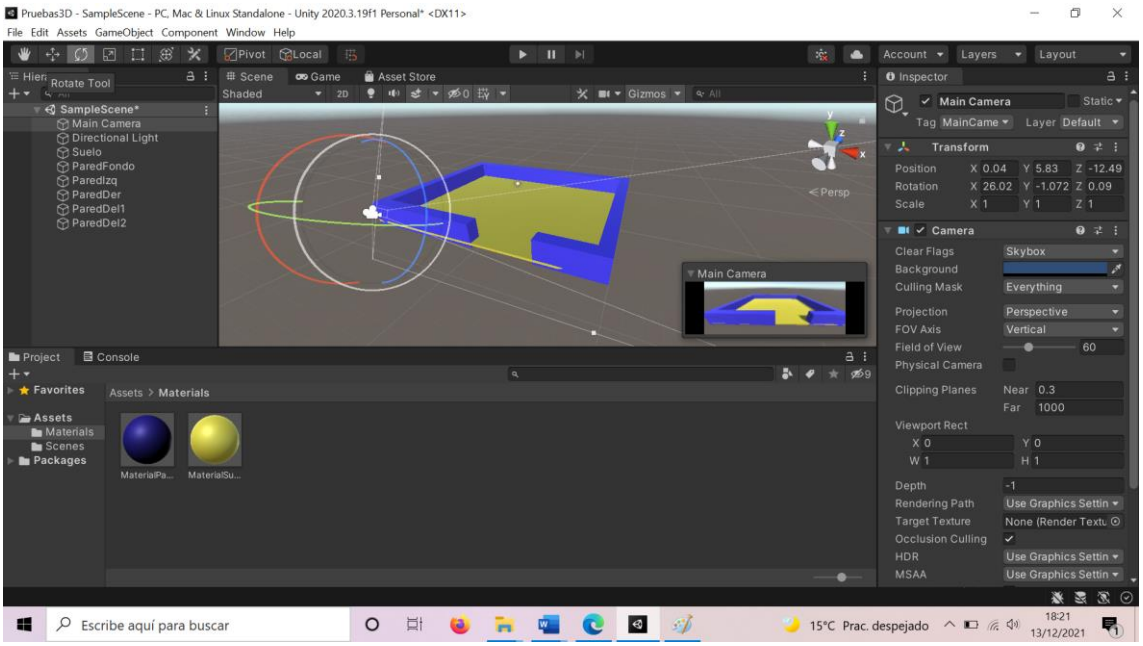
4. Lo que la cámara ve.

Como sabemos tenemos en nuestra escena una cámara que es la que capta lo que se ve de nuestra escena.

Para poder saber **qué se está viendo** con la cámara podemos simplemente pulsar Play y empezar a jugar, o bien, podemos simplemente pinchar sobre ella en la jerarquía, y nos aparecerá en una pequeña ventana lo que "ve" la cámara:



Vamos a mover y rotar la cámara, para conseguir una buena perspectiva de nuestra escena:

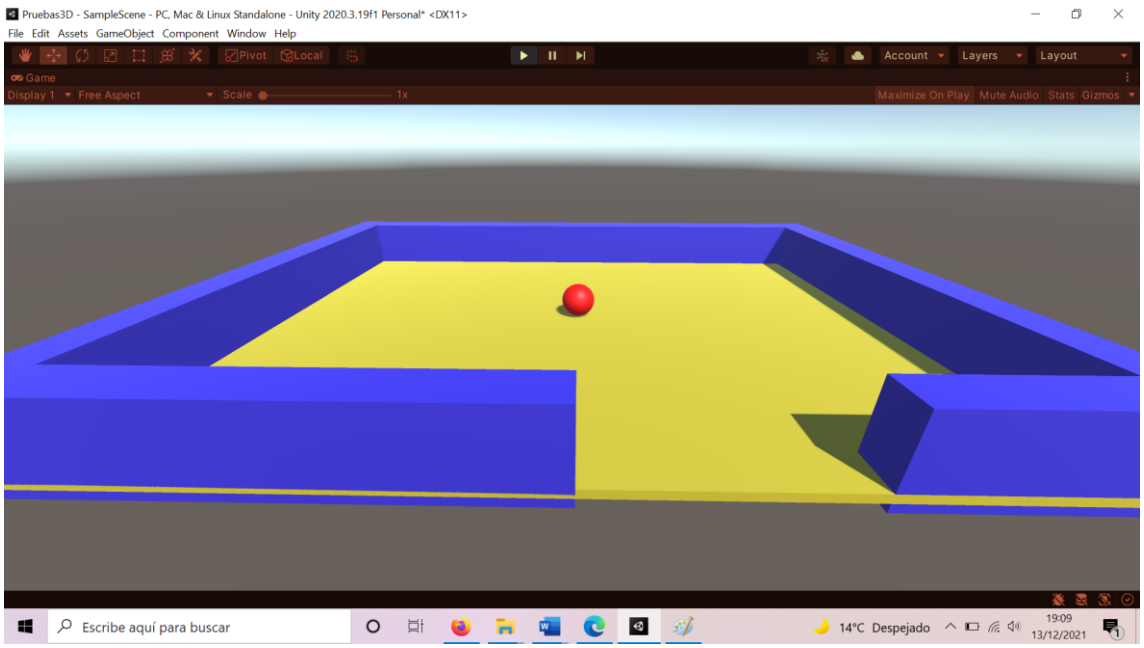


Incluso si nos gusta la perspectiva que tenemos en la Escena de muestra, podemos pulsar el botón derecho sobre la cámara y elegir **Align with View**.

5. Creando un personaje.

Vamos a crear ahora el personaje que controlará el usuario. Será una simple esfera que se moverá por el laberinto.

- Creamos un objeto **3D de tipo Sphere**.
- Ajustamos su **posición** y su escala. La posición la pondremos algo por encima del suelo, y **le añadiremos un componente de tipo RigidBody**. De esa manera, el objeto tendrá propiedades físicas y caerá hasta tocar el suelo.
- Le daremos un material.



Vamos a empezar a mover nuestro personaje.

Una primera aproximación, puede ser algo parecido a lo que hacíamos en 2D, **creando un script** para el personaje y en él, mirando el desplazamiento vertical (flecha arriba y abajo) y el desplazamiento horizontal (izquierda y derecha) y aplicar una velocidad a su RigidBody aprovechando las físicas…

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

    // Start is called before the first frame update
    void Start()
    {
        // Recogemos el Rigidbody del objeto
        body = GetComponent<Rigidbody>();
    }

    // Update is called once per frame
    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);

    }
}
```

Podemos probar qué ocurre con este funcionamiento, antes de cambiar por el código que viene a continuación.

Pero lo que vamos a hacer es que la bola **gire** si se pulsan las flechas derecha, izquierda y **avance o retroceda** si las flechas arriba, abajo.

```
public class Bola : MonoBehaviour
{
    // Este script es otra forma de mover la bola

    float velocidadAvance = 2.0f;   // 2 m/s
    float velocidadRotac = 180.0f;  // 180 grados por segundo

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        // Para coger el avance con las flechas adelante y atrás
        float avance = Input.GetAxis("Vertical")
                    * velocidadAvance * Time.deltaTime;
        // La rotación con las flechas izq y dcha.
        float rotacion = Input.GetAxis("Horizontal")
                    * velocidadRotac * Time.deltaTime;

        transform.Translate(Vector3.forward * avance);
        transform.Rotate(Vector3.up * rotacion);

    }
}
```

```
transform.Translate(Vector3.forward * avance);
```

Mueve el objeto hacia adelante (Vector3.forward equivale a Vector3(0, 0, 1)) aplicando el avance que hemos calculado con anterioridad.

```
transform.Rotate(Vector3.up * rotacion);
```

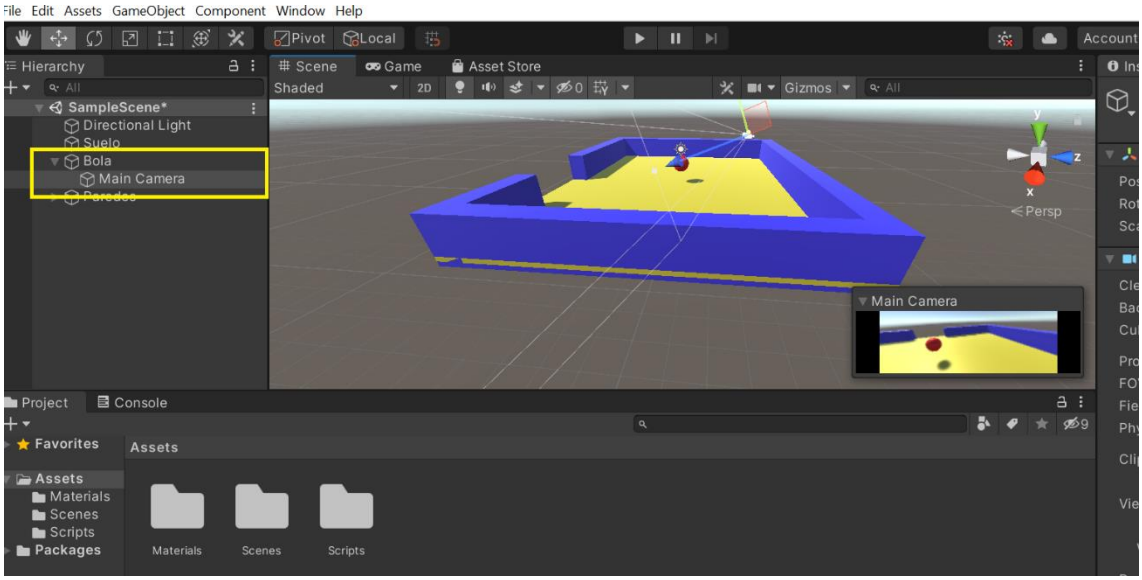
Gira el objeto 180 grados por segundo en la dirección que nos digan las flechas y en el **eje Y** ya que Vector3.up equivale a Vector3(0, 1, 0).

Si os fijáis nos encontramos con un problema en el momento que la bola choca contra una de las paredes, ya que a partir de ese momento hace un funcionamiento extraño. Es debido a que, al chocar con la pared, la bola gira físicamente y cambian sus coordenadas de rotación.

En el siguiente apartado veremos más claramente este problema y lo solucionaremos…

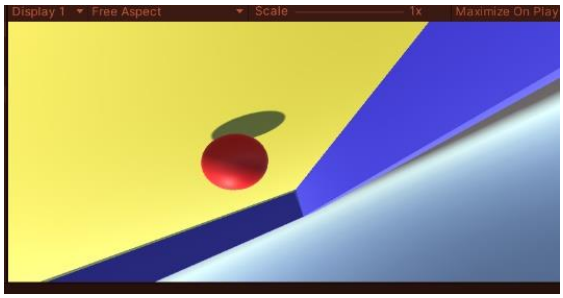
6. La cámara sigue al jugador.

Podemos hacer que la cámara siga a nuestra bola. La forma más sencilla es simplemente hacer a la cámara "hija" del objeto bola en la jerarquía, y a continuación ajustar su posición y rotación hasta conseguir el efecto visual deseado.

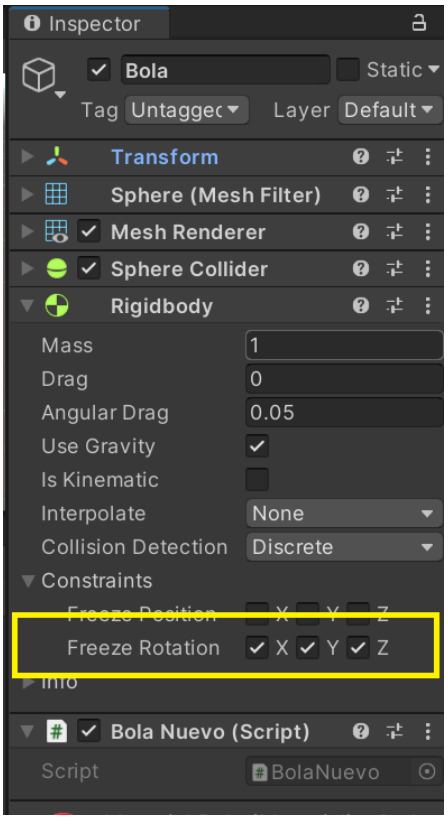


Recordad que una forma sencilla de colocar la cámara donde nos interese es visualizar esa posición en la Escena y a continuación con la cámara seleccionada en la jerarquía darle a la opción de menú: **Game Object -> Align with view**.

Como dijimos en el apartado anterior al **chocar con las paredes**, ocurren cosas extrañas que son más evidentes ahora porque la cámara al ser hija de la bola y esta girar, también gira:



Podemos evitar ese efecto bloqueando la rotación de la esfera en los distintos ejes en su RigidBody:



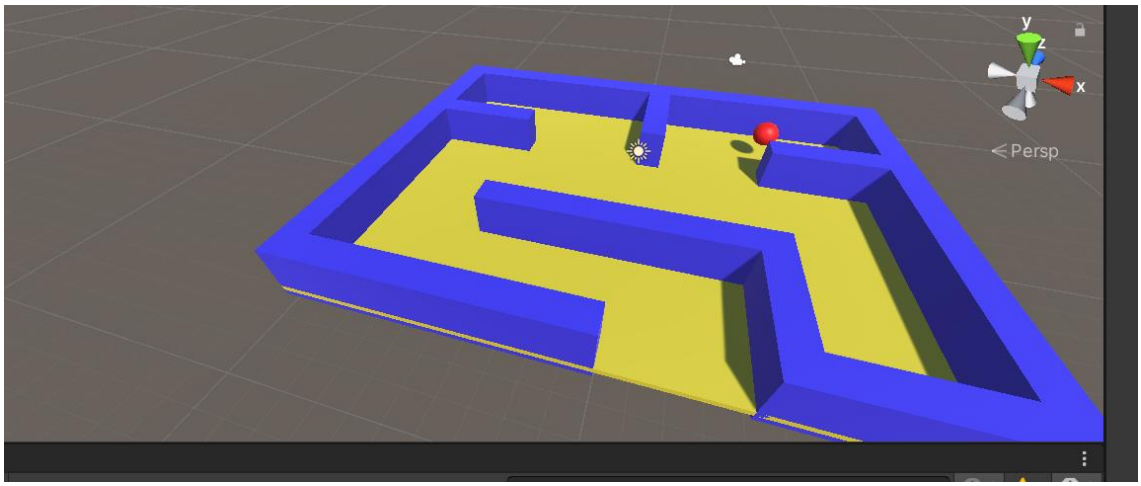
Vídeo jugador y cámara (apartados 5 y 6).

7. Diseño del laberinto.

Vamos a diseñar un “laberinto” que pueda recorrer nuestro personaje.

Ese laberinto lo podemos diseñar nosotros a mano en un papel o podemos utilizar alguno de los generadores de laberintos que encontramos online, por ejemplo https://www.mazegenerator.net/.

Podríamos implementar un laberinto a nuestro gusto:

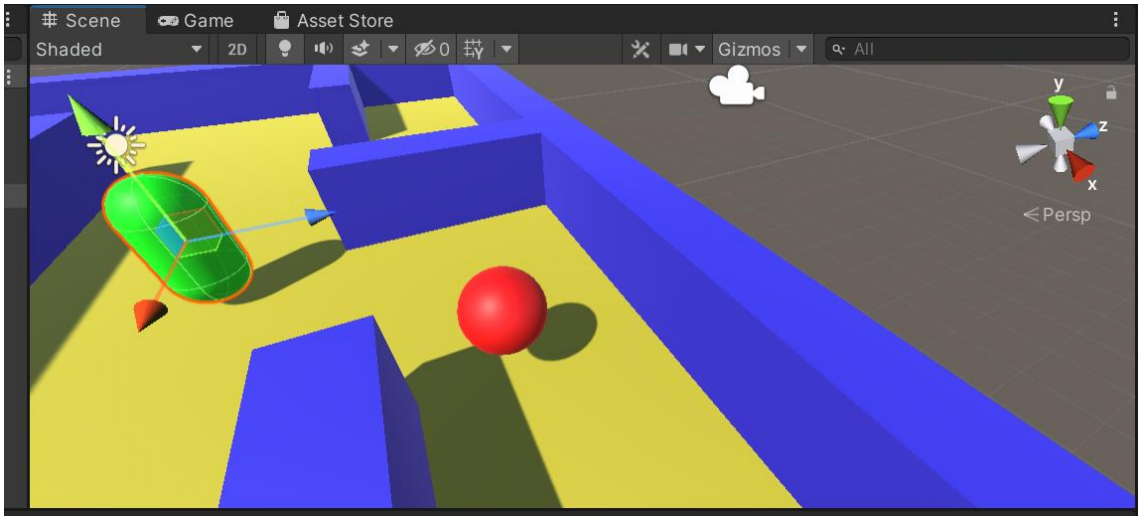


8. Creación de enemigos.

Vamos a crear ahora un “enemigo”.

Le podemos dar la forma que queramos, por ejemplo, elegir una capsula: Botón derecho->Create Object->Capsule.

Además, le ponemos un **material** con un color que nos apetezca:



Vamos a ver cómo comprobar colisiones en 3d, que es similar a como lo hacíamos en 2d.

Al añadir la esfera, Unity nos la pone con el componente Sphere Collider, y Capsule Collider en el caso de la capsula.

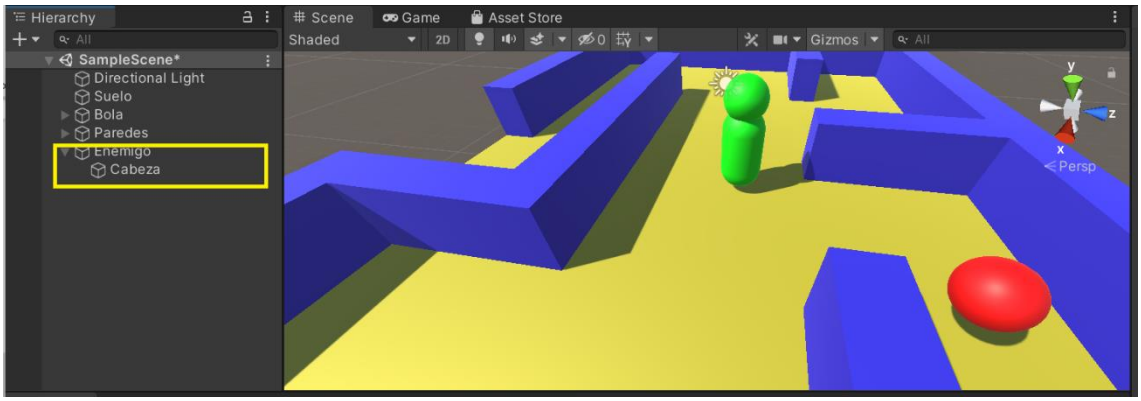
La cápsula no debe llevar RigidBody, ya que lo tiene la esfera. Sí que le activaremos IsTrigger y detectaremos la colisión con el método OnTriggerEnter (lógicamente en el script de la cápsula que llamaremos Enemigo):

```
public class Enemigo : MonoBehaviour
{
    void Start()
    {

    }
    void Update()
    {

    }
    void OnTriggerEnter(Collider other)
    {
        Debug.Log("Tocado");
        Application.Quit();
    }

}
```

Relaciones padre/hijo.

Podemos crear un **objeto** que sea **hijo de otro**. Lo arrastramos sobre el otro objeto y a partir de ese momento sus coordenadas serán locales o relativas a las del padre. Si movemos o rotamos al padre, también lo hará el hijo.

Vamos a añadir una esfera sobre el enemigo, que además será hijo de este, de manera que si movemos al enemigo los dos objetos se moverán juntos:

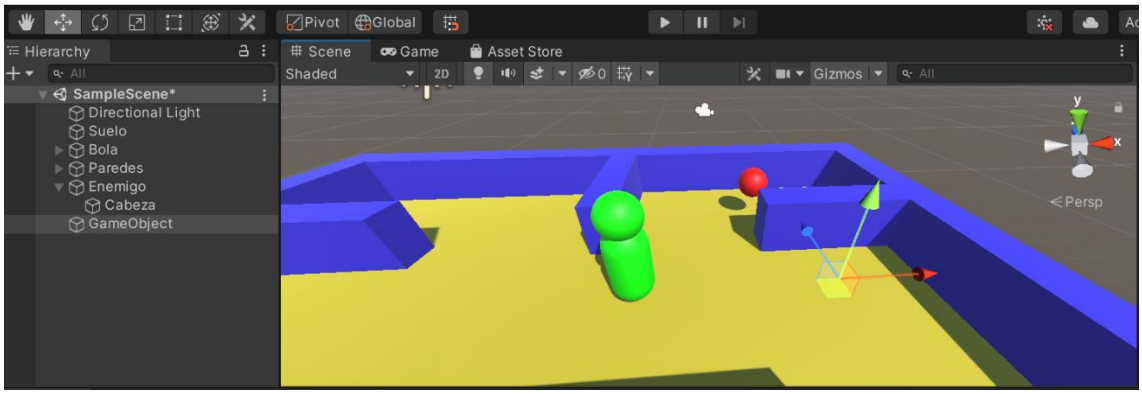


Movimiento del enemigo. Waypoints.

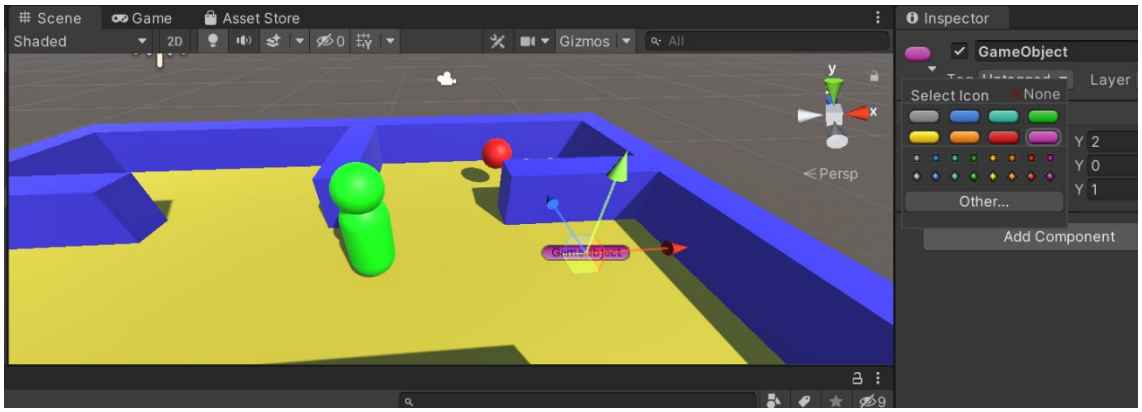
Vamos a conseguir en este apartado mover al enemigo. A diferencia a como hicimos en el tema anterior, en el que los enemigos se movían de lado a lado, vamos a utilizar unos **puntos de referencia** o **waypoints** que el enemigo irá recorriendo.

Vamos a crear una lista de waypoints.

Para empezar a crear un waypoint, crearemos un objeto vacío (botón derecho -> Create Empty). Le pondremos la misma altura Y que al enemigo.

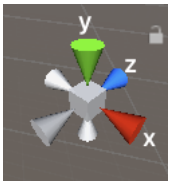


Nos encontramos que, de esa forma, el punto solo es visible cuando está seleccionado. Vamos a darle un color haciendo click en el cubo que aparece al lado de su nombre en el Inspector:



Podemos además cambiarle el nombre por algo como Punto1 o P1.

Para situar el punto y los siguiente podríamos pasar a vista superior pulsando el eje Y en los ejes que aparecen arriba a la derecha en la vista de la escena:



De esa manera veremos así nuestra escena:

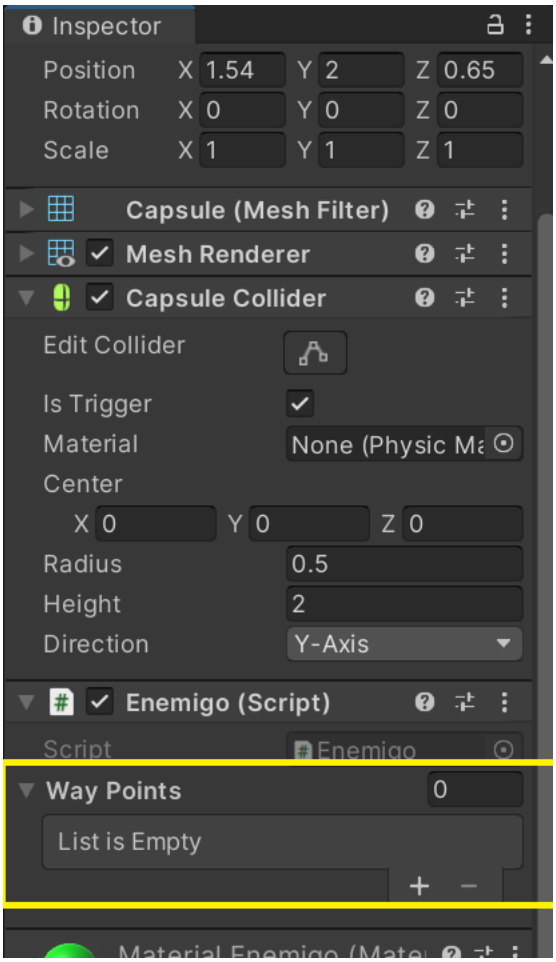
Podemos duplicar nuestro punto para hacer una lista de puntos:

Una vez creados los waypoints, los vamos a poner como un array de Transforms en el script del enemigo, utilizando [SerializeField] para que sea accesible desde el editor:

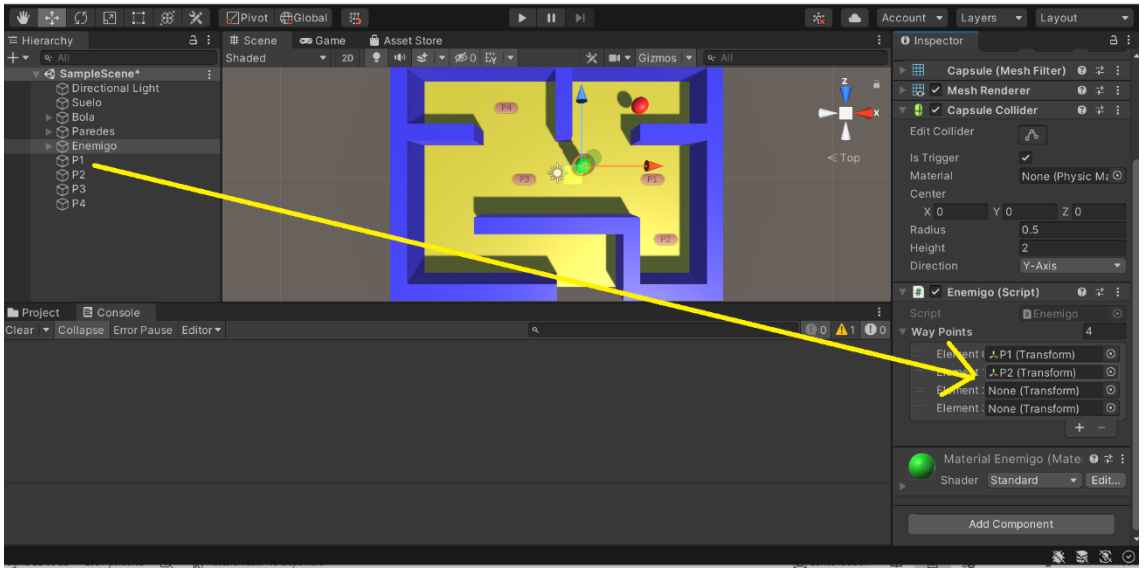
```
[SerializeField] Transform[] wayPoints;
```

En este momento desde el editor haremos dos pasos.

Indicaremos el tamaño Size del array:



Y una vez que tengamos el tamaño colocamos los puntos en el orden en el que queramos que se recorran después, arrastrándolos desde la jerarquía:



Ahora vamos a crear la lógica de seguimiento. Tendremos una variable en la que guardaremos la siguiente posición a la que vamos a ir, poniendo en Start la primera:

```
    Vector3 siguientePosicion;

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

En Update utilizaremos Vector3.MoveTowards para calcular un vector que va desde la posición actual a la siguiente, utilizando cierta velocidad:

```
    float velocidad = 2;

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

A continuación, vamos a hacer que cuando nos encontremos muy cerca del punto cambiemos al siguiente punto que corresponda.

Para ello definimos dos variables, una para controlar la distancia al punto, y otra para la posición en el vector de la siguiente posición.

```
    float distanciaCambio = 0.2f;
    int numeroSiguientePosicion = 0;
```

Con el siguiente código, cambiaríamos la siguiente posición cuando la distancia sea muy cercana al waypoint de la posición a la que vamos:

```
    void Update()
    {
        // Nos movemos hacia la siguiente posición
        transform.position = Vector3.MoveTowards(transform.position,
                                siguientePosicion,
                                velocidad * Time.deltaTime);

        // Si la distancia al punto es corta cambiamos al siguiente
        if (Vector3.Distance(transform.position,
                             siguientePosicion) < distanciaCambio)
        {
            numeroSiguientePosicion++;
            if (numeroSiguientePosicion >= wayPoints.Length)
                numeroSiguientePosicion = 0;

            siguientePosicion = wayPoints[numeroSiguientePosicion].position;
        }

    }
```

De esta manera conseguimos que el enemigo se mueva consecutivamente entre los puntos de referencia que hemos elegido.

El código completo de la clase Enemigo quedaría como sigue:

```
public class Enemigo : MonoBehaviour
{
    [SerializeField] Transform[] wayPoints;

    float velocidad = 2;

    float distanciaCambio = 0.2f;
    int numeroSiguientePosicion = 0;
    Vector3 siguientePosicion;


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

    void Update()
    {
        // Nos movemos hacia la siguiente posición
        transform.position = Vector3.MoveTowards(transform.position,
                                siguientePosicion,
                                velocidad * Time.deltaTime);

        // Si la distancia al punto es corta cambiamos al siguiente
        if (Vector3.Distance(transform.position,
                             siguientePosicion) < distanciaCambio)
        {
            numeroSiguientePosicion++;
            if (numeroSiguientePosicion >= wayPoints.Length)
                numeroSiguientePosicion = 0;

            siguientePosicion = wayPoints[numeroSiguientePosicion].position;
        }
    }

}
```

Vídeo creación enemigo

9. Jugador con varias vidas.

Vamos a conseguir a continuación que nuestro jugador tenga **varias vidas**.

Hasta el momento lo que hacemos es que cuando el jugador choca con el enemigo termina la partida. Vamos a cambiarlo para que cuando el jugador choque con el enemigo pierda una vida y además vuelva a la posición inicial.

Como no tenemos todavía ninguna clase que represente a todo el juego, esas vidas las guardaremos en el propio jugador.

Además, guardaremos en él al principio la posición inicial.

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

    // Start is called before the first frame update
    void Start()
    {
        xInicial = transform.position.x;
        zInicial = transform.position.z;
    }
```

Vamos a crear un método llamado **PerderVida**, en el que volveremos a la posición inicial y quitaremos una vida al jugador.

```
    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();
        }
    }
```

Por tanto, el Script del jugador quedaría así de momento:

```
public class Bola : MonoBehaviour
{
    // Este script es otra forma de mover la bola siguiendo el vídeo del curso
cefire

    float velocidadAvance = 5f;    // Se supone que son 5 unidades/s
    float velocidadRotac = 180.0f;  // 180 grados por segundo

    float xInicial, zInicial;
    int vidas = 3;

    // Start is called before the first frame update
    void Start()
    {
        xInicial = transform.position.x;
        zInicial = transform.position.z;
    }
```

```
    // Update is called once per frame
    void Update()
    {
        // Para coger el avance con las flechas adelante y atrás
        float avance = Input.GetAxis("Vertical")
                    * velocidadAvance * Time.deltaTime;
        // La rotación con las flechas izq y dcha.
        float rotacion = Input.GetAxis("Horizontal")
                    * velocidadRotac * Time.deltaTime;

        transform.Rotate(Vector3.up * rotacion);
        transform.Translate(Vector3.forward * 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();
        }
    }
}
```

Lo que tenemos que hacer ahora es llamar al método PerderVida desde el OnTriggerEnter del Enemigo.

Podemos pensar que se podría hacer:

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

Pero eso **nos da un error de compilación** en Unity.

Si lo queremos hacer a través del Collider other, tendremos que utilizar SendMessage, indicando el método al que queremos llamar…

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

10. Posibles mejoras del juego

Podríamos pensar distintas mejoras del juego desarrollado hasta el momento:

- Añadir una pantalla de bienvenida.
- Crear un laberinto Nivel 2 más grande, con más enemigos y que se muevan a mayor velocidad.
- Pasaremos del nivel 1 al nivel 2 al llegar a la “puerta” de salida del laberinto.
- El jugador tendrá 3 vidas, que no se reiniciarán al pasar de nivel.
- Al perder las tres vidas aparecerá un mensaje de Game Over.
- Al alcanzar la puerta del segundo nivel, terminará el juego apareciendo el texto Fin y volviendo tras 5 segundos a la pantalla de Bienvenida.
- Utilización de prefabs para los enemigos y para el jugador.