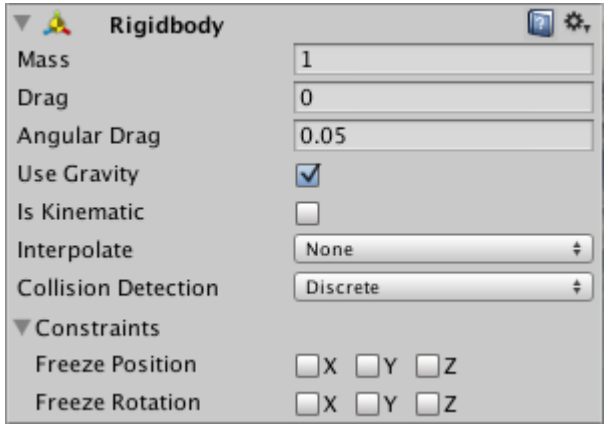
MUNDO FÍSICO

GRAVEDAD, OBJETOS RÍGIDOS Y COLISIONES

RIGID BODY

- El componente "Rigid Body" permite que cualquier Game Object actúe bajo los motores de la física y por tanto se comporte de una manera realista.
  - Se aplica desde el menú "Component>Physics>Rigid Body"
- Las propiedades que nos permite configurar son:
  - Mass: el peso (en kilogramos por defecto)
  - Drag: resistencia al viento cuando está en moimiento (si es infinito se detiene automáticamente). Todos los objetos caen a la misma velocidad, pero un Drag bajo hace que parezca más pesado (0.001 = metal sólido, 10 = pluma)
  - Angular Drag: resistencia al viento cuando gire
  - Use Gravity: si el objeto es afectado por la gravedad
  - Is Kinematic: si se activa, el objeto no es controlado por el motor de física y sólo por el componente transform (útil por ejemplo para plataformas que se desplazan y que no queremos que se vean afectadas por las físicas)
  - Interpolate: opciones para suavizar el movimiento (interploate / extrapolate), en base a los fotogramas anteriores y posteriores
  - Collision Detection: permite configurar cómo se comportan otros objetos en sus colisiones con éste
  - Constrains: permite restringir movimientos y rotaciones del objeto.



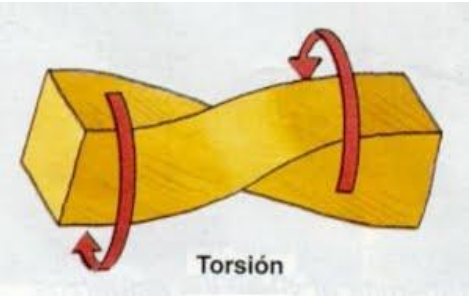
Rigid Body permite que los objetos se muevan de una manera realista, por ello, no es adecuado combinarlo con movimientos mediante el componente de Transform (por lo general, o se usa uno o se usa otro)

Lo ideal es combinarlo con algún tipo de Collider para lograr un efecto totalmente realista

FUERZAS Y TORSIONES

- En el momento que un objeto de nuestra escena se ve afectado por la física, la forma adecuada para desplazarlo (ya sea en traslación o en rotación) es aplicándoles fuerzas y torsiones
- Esto se realiza mediante código, usando los siguientes métodos que se aplican al componente RigidBody:
  - AddForce(): aplica una fuerza en la dirección indicada y del tipo indicada.
  - AddTorque(): añade una torsión al objeto lo que le provoca una rotación.
- Parámetros que deben pasarse a los métodos:
  - Dirección en X, Y, Z. Se puede sustituir por un Vector 3 (muy útil por ejemplo Transform.Forward, Transform.UP, etc)
  - Modo: mediante el método ForceMode, podemos especificar cómo debe aplicarse esa fuerza (aceleración, fuerza, impulso, cambio de velocidad)

- Tenemos que tener en cuenta que:
  - Estos métodos se deben aplicar a un componente RigidBody, para lo cual necesitamos "capturarlo" en una variable mediante el método "GetComponent"
  - Si aplicamos una vez este método (por ejemplo al pulsar un botón o en el método Start) le imprimirá el impulso y luego lo liberará, pero si lo aplicamos de forma continua (en Update) se aplicará de forma constante, por ejemplo, para desplazar un objeto
  - Alternativas a estas fuerzas son AddForceAtPosition(), y AddRelativeForce(), que permite ajustar con precisión desde dónde y cómo se aplican esas fuerzas



AddForce()

Ejemplo que aplica una fuerza hacia delante, con un valor de tipo float ("thrust")

Antes de aplicarlo, debemos obtener el componente RigidBody, que lo guardamos en la variable "rb"

```
public class ExampleClass : MonoBehaviour
{
    public float thrust;
    public Rigidbody rb;

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

    void FixedUpdate()
    {
        rb.AddForce(transform.forward * thrust);
    }
}
```

AddTorque()

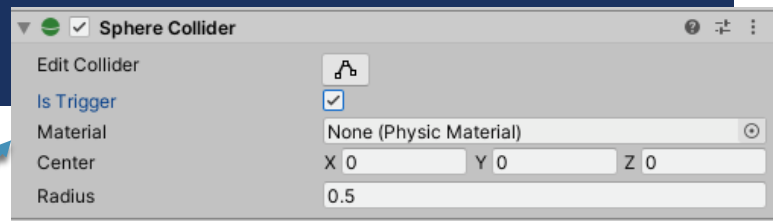
Ejemplo que nos permite aplicar torsiones, vinculadas a una fuerza de tipo float ("torque") y al control de los ejes horizontales

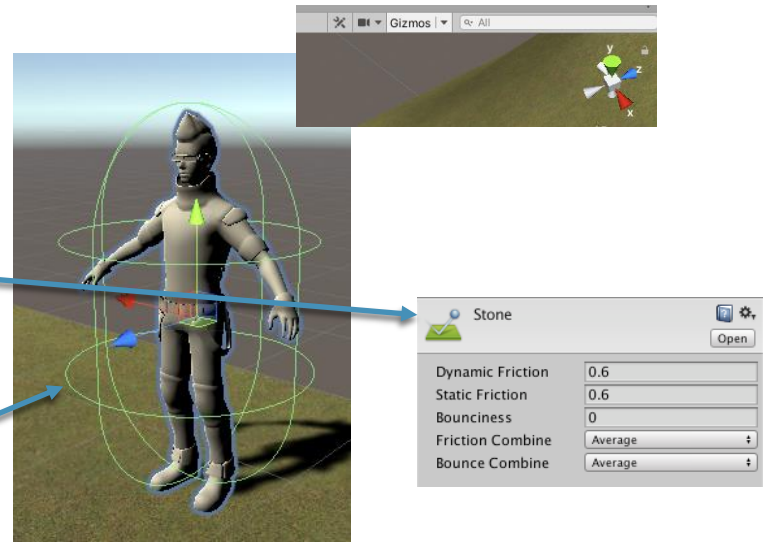
```
public class ExampleClass : MonoBehaviour
{
    public float torque;
    public Rigidbody rb;

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

    void FixedUpdate()
    {
        float turn = Input.GetAxis("Horizontal");
        rb.AddTorque(transform.up * torque * turn);
    }
}
```

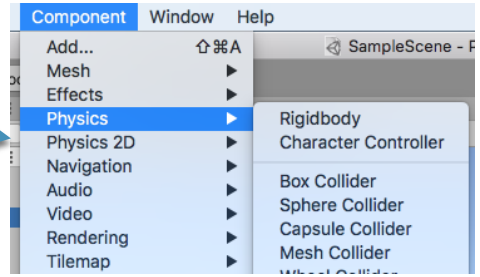
Tendrás que ajustar los valores de la fuerza ("thrust" y "torque", prueba con valores bajos y altos)
Prueba a aplicarlo en lugar de en el método de "FixedUpdate" en Start, y verás la diferencia.

COLLIDER

- Componentes que permiten detectar colisiones entre objetos que tengan aplicados un RigidBody.
- Al crear cualquier objeto 3D se crea por defecto un apartado llamado Collider, con determinados parámetros:
  - **Is Trigger**: el objeto es atravesado, pero permite lanzar disparadores cuando entra en colisión con otros objetos
  - **Material**: para saber cómo interactúa un objeto con otro, necesita saber qué tipo de material es (por ejemplo, una piedra choca, un cristal resbala, etc.) Para poder asignar uno, deberemos crear un Physics Material y asignarlo
  - **Center / Radius**: posición y extensión del Gizmo que determina qué área es sensible a colisiones.
    - El objeto tiene un área amarilla que indica la zona sensible. Al pulsar sobre el icono de "Edit Collider" podremos modificarlo a mano
    - Si no se ve, activar la visión de gizmos en la parte superior de la ventana de la escena
- Algunos objetos no tienen un collider por defecto y lo tenemos que añadir como componente (box collider / sphere collider / capsule collider / etc.)
  - Algunos son específicos, por ejemplo para terrenos (Terrain Collider), para vehículos en movimiento (Wheel Collider) o para personajes (Character Collider)







PROXIMIDAD Y COLISIONES

DETECTAR COLISIONES

- Podemos mediante código detectar que el objeto que tiene asociado el script ha entrado en contacto con cualquier otro Game Object.
- Para ello, el elemento tiene que tener activado la casilla "isTrigger" en su "Collider"
  - Podemos usar el método "OnTriggerEnter()" que se lanza al detectar una colisión
  - Le pasaremos como parámetro un variable de tipo Collider, que contiene los datos del objeto con el que ha colisionado.
    - Por ejemplo, dentro de la variable obtenida están todos los datos del Game Object del objeto que ha impactado, entre ellos sus etiquetas:

```
other.gameObject.tag
```

Es importante tener en cuenta los objetos cercanos, y si estos tienen activadas mallas de colisión (ejemplo, si activamos la colisión en el suelo, cualquier objeto cuya malla de colisión se expanda hacia fuera, hará saltar la colisión automáticamente)

Este script hace que el objeto desaparezca cuando otro objeto lo toca

```
public class selfDestroy : MonoBehaviour
{
    void OnTriggerEnter(Collider other)
    {
        Destroy(this.gameObject);
    }

}
```

Este destruye el GameObject solo si el objeto colisionado tiene la etiqueta de "Enemy"

```
public class selfDestroy : MonoBehaviour
{
   void OnTriggerEnter(Collider other)
   {
        if(other.gameObject.tag == "Enemy")
        {
           Destroy(this.gameObject);
        }
    }
}
```

DETECTAR PROXIMIDAD DE UN OBJETO

- Es habitual comprobar la distancia con otro objeto del juego. Esto se hace con el método Vector3.Distance
  - Se la pasan 2 parámetros de tipo transform (Vector3) y devuelve un nº de tipo float que mide la distancia entre ambos
- En el script de la derecha, podemos comprobar en cada momento la distancia a la que se encuentra el otro objeto (capturado en una variable de tipo Transform llamada "other")
  - No es eficiente, ya que quizás no es necesario comprobarlo cada fotograma, y sería mejor ejecutarlo en una corrutina, como se muestra en la siguiente diapositiva

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour
{
     public Transform other;

     void Update()
     {
          //Nos aseguramos de que se ha declarado el otro objeto
          if (other)
          {
            float dist = Vector3.Distance(other.position,
          transform.position);
           Debug.Log("Distancia al otro: " + dist);}
     }
}
```

Detectar distancia de TODOS los enemigos

```
//Distancia a la que saltará la alarma
public float dangerDistance = 20;

void Start()
{
	//Iniciamos la corrutina
	StartCoroutine("DoCheck ");
}

function ProximityCheck()
{
	for (int i = 0; i < enemies.Length; i++)
	{
		if (Vector3.Distance(transform.position, enemies[i].transform.position) < dangerDistance)
		{
			return true;
		}
	}
	return false;
}

IEnumerator DoCheck()
{
	for(;;)
	{
		ProximityCheck;
		yield return new WaitForSeconds(.5f);
	}
}
```

Script que comprueba la distancia de todos los enemigos, pero en lugar de hacerlo cada fotograma lo hace cada medio segundo mediante una corrutina, en la cual llama en cada ciclo a la función que comprueba la distancia.

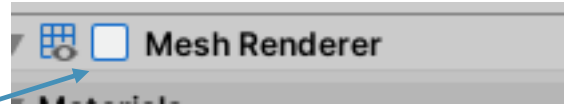
Los enemigos están contenidos en un array, creado al instanciar el prefab correspondiente

- El método OnTriggerEnter es muy apropiado para detectar presencia de un personaje en determinada zona.
- En este ejemplo vamos a crear una forma invisible que al ser atravesada por un personaje ejecuta una acción
- Para ello, crearemos un cubo, lo colocaremos en la zona por la que pasará el personaje, y para que pueda atravesarlo desactivaremos la opción de "Mesh Renderer", lo que lo convertirá en invisible
- Activaremos "Is Trigger" en el Box Collider
- A partir de ahora, podemos usar el método OnTriggerEnter(Collider other) para realizar cualquier acción si detecta que el personaje la ha cruzado.
  - También podemos destruir la caja, una vez atravesada, si no la necesitamos más

```
Destroy(this.gameObject);
```

- Podemos crear variables públicas de tipo estática que sean reconocidos por otros scripts, de esta forma este trigger puede por ejemplo activar otras acciones (ejemplo, saltar una alarma)
  - Para recuperarla en otro script solo tengo que indicar en qué clase está y su nombre. Ejemplo:

```
If(nombreDeLaClase.activarAlarma == 1) { … }
```




```
public static int activarAlarma = 0;

private void OnTriggerEnter(Collider other)
{
      activarAlarma = 1;
      Destroy(this.gameObject);
}
```

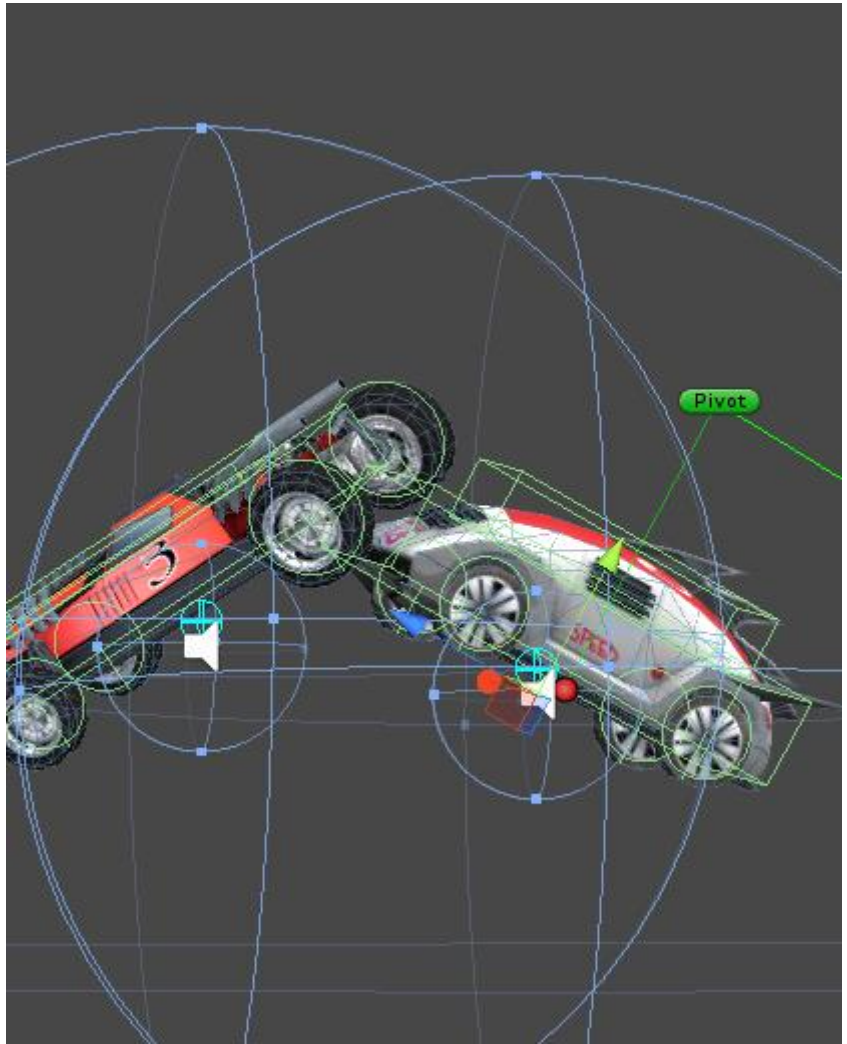
EJERCICIO: DETECTAR PRESENCIA

Activar una alarma cuando un personaje pasa por una zona de nuestra escena

DETECTAR COLISIONES

- Una alternativa al método "OnTriggerEnter" es usar el método "OnCollisionEnter()".
- Este método, a diferencia del anterior, pasa una variable de tipo "Collision", en lugar de "Collider", la cual ofrece información que puede ser de interés, como el punto de impacto, la velocidad de impacto, nº de impactos, etc.
  - Esta información es necesaria en determinadas situaciones, por ejemplo, un proyectil que impacta contra el objetivo
- Ejemplos de submétodos disponibles en la clase Collision:
  - relativeVelocity: la velocidad relativa de la colisión (si el objeto impactado está en movimiento, la fuerza del impacto depende de su velocidad y su dirección). Usaremos el submétodo "magnitude" para obtener el valor
  - contacts: nos devuelve un array con los contactos que se han producido
  - Los datos del objeto que ha colisionado los tenemos también en los submétodos "gameObject" y "transform"

- NOTA: los eventos de colisión solo se producen si uno de los objetos colisionados posee un rigid-body "non-kinematic"



Ejemplos obtenidos de la página de Unity referente a Colisiones

Script que detecta una colisión e instancia un prefab ("explosionPrefab") de un efecto de sistema de partículas en el punto exacto donde se ha producido el impacto y destruye el objeto colisionado

```
public Transform explosionPrefab;

void OnCollisionEnter(Collision collision)
{
    ContactPoint contact = collision.contacts[0];
    Quaternion rotation = Quaternion.FromToRotation(Vector3.up, contact.normal);
    Vector3 position = contact.point;
    Instantiate(explosionPrefab, position, rotation);
    Destroy(gameObject);
}
```

Script que detecta una colisión y dibuja una línea en el punto de impacto, y en caso de que la fuerza del impacto sea mayor que 2, lanza un aviso

```
void OnCollisionEnter(Collision collision)
{
    foreach (ContactPoint contact in collision.contacts)
    {
        Debug.DrawRay(contact.point, contact.normal, Color.white);
    }
    if (collision.relativeVelocity.magnitude > 2)
        print("Objeto impactado con fuerza");
}
```