Álvaro Holguera
CIFP José Luis Garci
(Madrid)

UD 03
FÍSICAS EN UNITY

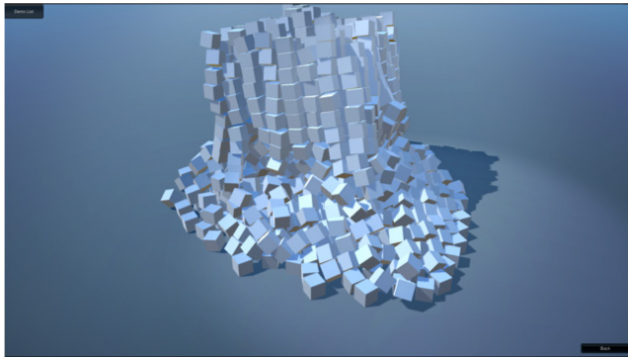
Desarrollo de Entornos Interactivos Multidispositivo

Álvaro Holguera
CIFP José Luis Garci
(Madrid)

MUNDO FÍSICO

Estamos a punto de entrar en el terreno de las físicas, donde la gravedad toma el control sobre el movimiento de nuestros objetos.

EL COMPONENTE RIGID BODY



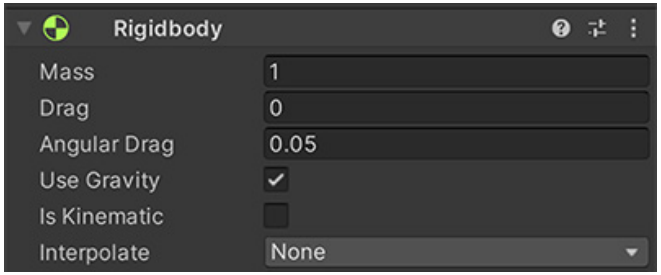
Estamos a punto de entrar en el terreno de las físicas, donde los objetos se mueven no porque cambiemos su posición, lo que se llama "movimiento Kinemático", sino porque les aplicamos fuerzas que contrarrestan otra que nos atrae en todo momento: la gravedad. Vamos, como en el mundo real.

Rigid Body

Para que un objeto se vea afectado por la física, y por las colisiones, debe tener un componente de tipo Rigid Body. Permite que cualquier Game Object actúe bajo los motores de la física y por tanto se comporte de una manera realista.

Para ver cómo funciona, crea una escena con un cubo y añádele el componente RigidBody (recuerda que tienes una herramienta para buscar por nombre al pulsar el botón "Add Component" del Inspector).

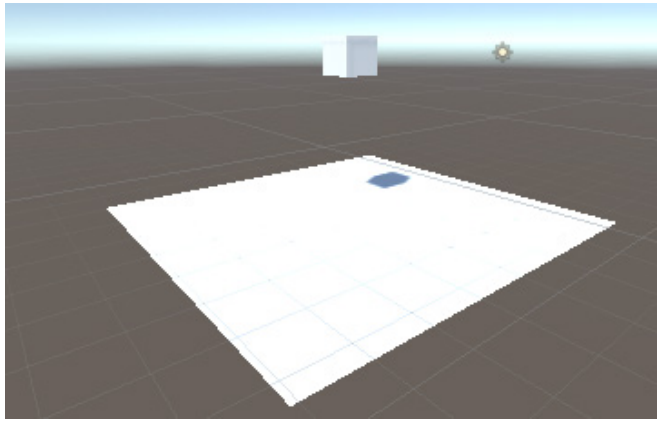
IMPORTANTE: Existe una versión RigidBody2D para objetos 2D y que veremos más tarde, no lo confundas con el RigidBody normal.



Ejemplo práctico

Pon un cubo a la escena, elévalo, y añádele el componente Rigid Body.

Prueba a lanzar el juego y verás cómo cae el cubo (le puedes poner un plano debajo que detenga la caída: GameObject > 3D Object > Plane).



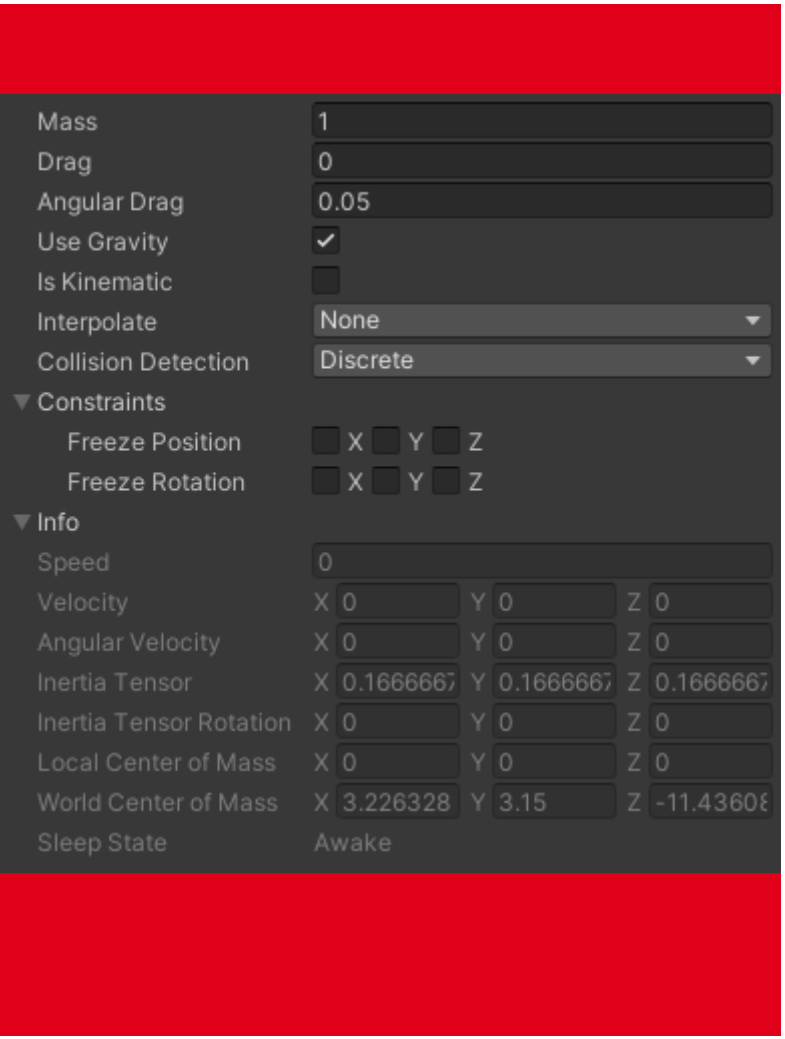
Prueba a cambiar la posición del cubo y los parámetros del componente a ver cómo afectan.

También como experimento: añade un RigidBody al plano, pero sin que le afecte la gravedad (para que no caiga) y activa la casilla "Convex" de su MeshCollider para que el cubo la detecte. Observa qué ocurre al chocar el cubo contra el plano.

IMPORTANTE: cuando un objeto, como este plano, tiene detector de colisiones (Collider) pero no tiene Rigid Body, es lo que se denomina un objeto estático, que impide que los objetos le atraviesen pero no se ve afectado por sus fuerzas. Como una pared o el suelo.

PARÁMETROS DE RIGID BODY

En el componente Rigid Body podremos configurar una serie de parámetros que marcarán cómo se comportará ese elemento en l mundo físico. Veamos los más importantes



- **Mass**: el peso (en kilogramos por defecto)
- **Drag**: resistencia al viento cuando está en movimiento. Cuanto más alto, más rápido se detiene. Recuerda, todos los objetos caen a la misma velocidad, pero una resistencia baja hace que parezca más liviano (0.001 = metal sólido, 10 = pluma).
- **Angular Drag**: resistencia al viento cuando gira.
- **Use Gravity**: si el objeto es afectado por la gravedad. Esta fuerza por defecto es de -9,8 en el eje Y, y como otros datos referentes al mundo físico, se configura en "Edit>Project Settings>Physics"
- **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 fuerzas físicas).

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)

- **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 (depende de la velocidad del objeto)

IMPORTANTE: si uno de los objetos se mueve a gran velocidad, mejor cambiar el modo a "Continous", que consume más recursos pero es capaz de detectar esas colisiones.

- **Constrains**: permite restringir movimientos y rotaciones del objeto.
- **Info**: nos muestra en tiempo real las fuerzas que se están aplicando y cómo afectan al objeto. Muy útil por ejemplo para saber a qué velocidad se está moviendo.

MÉTODO GET COMPONENT<>

Antes de continuar debemos saber cómo actuar sobre los parámetros de cualquier componente que tenga un Game Object. Hemos visto que el componente "transform" está disponible sin tener que hacer nada, pero eso no es así con el resto.

Para acceder a un componente, ya sea propio o de otro objeto, primero debemos crear una variable del mismo tipo que ese componente (en este caso, RigidBody), y después en el método Start darle el contenido.

En el siguiente ejemplo, accedemos al componente RigidBody del mismo objeto que tiene el script:

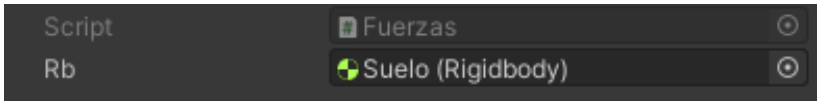
```
//Variable que contendrá el componente
Rigidbody rb;

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

Cuando queremos acceder al componente de otro objeto, solo tenemos que meter ese GameObject en una variable (por ejemplo "other") y llamar al método GetComponent dentro de él. Quedaría así.

```
rb = other.GetComponent<Rigidbody>();
```

Si bien a veces es más cómodo crear la variable "RigidBody" pública o serializada y arrastrar en Unity el GameObject que contiene ese componente:

```
[SerializeField] Rigidbody rb;
```



En cualquier caso, a partir de este momento podemos acceder a todos los atributos y métodos de este componente, como por ejemplo aplicarle fuerzas y torsiones.

IMPORTANTE: igual que accedemos a un componente como el RigidBody de otro Game Object, podemos acceder a sus scripts (al fin y al cabo, es un componente como cualquier otro, y una clase pública). El método es crear una variable de esa clase, acceder como componente:

```
[SerializeField] NombreDelScriot nombreDelScript;
```

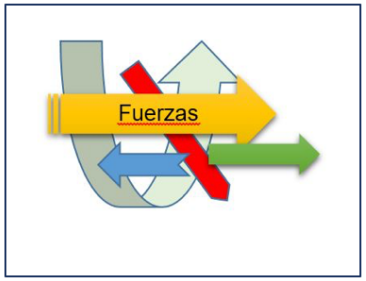
Ahora podemos arrastrar el objeto que tenga ese script y una vez "capturada" podemos ejecutar sus métodos mediante "SendMessage" y cambiar sus atributos:

Álvaro Holguera
CIFP José Luis Garci
(Madrid)

FUERZAS Y TORSIONES

Ahora que nuestro objeto responde a las leyes de la física, podemos moverlo como se hace en el mundo real: aplicando fuerzas y torsiones

FUERZAS



Comencemos con lo más básico: cómo mover un objeto aplicándole una fuerza en una dirección.

Esto se realiza mediante código, usando el método AddForce(): el cual aplica una fuerza en la dirección indicada y del tipo indicada.

```
rb.AddForce(Vector3, ForceMode);
```

Le tenemos que pasar la dirección (ya sea dando un valor en los tres ejes o, más práctico, pasando un Vector3) y el tipo de fuerza. Este último se hace mediante el método ForceMode. Tiene estas variantes:

1. **Force**: el que se aplica por defecto, que aplica una fuerza continua teniendo en cuenta su masa
2. **Acceleration**: aplica una aceleración continua, ignorando su masa.
3. **Impulse**: aplica una fuerza de forma instantánea, teniendo en cuenta su masa
4. **VelocityChange**: añade una velocidad instantánea, ignorando su masa.

Cuál aplicar dependerá de la situación especial en cada momento.

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.

Veamos dos ejemplos:

```
//Variable que contendrá el componente
Rigidbody rb;
//Vector que determinará la dirección del impulso
Vector3 push = Vector3.up;
//Fuerza que le aplicaremos
[SerializeField] float thrust = 200f;

void Start()
{
    //"Capturamos" nuestro RigidBody
    rb = GetComponent<Rigidbody>();

    //Le aplicamos una fuerza al inicio
    rb.AddForce(push * thrust);
}
```

CONSEJO: puedes crear un vector normalizado y controlar su "fuerza" mediante una variable por la que multiplicar ese vector.

Prueba a cambiar los parámetros del RigidBody, como la masa o la resistencia al aire, o incluso la fuerza de la gravedad, para ver cómo afecta.

Como verás, no hemos pasado el parámetro del ForceMode, por lo que ha aplicado el que usa por defecto, Force, que está más pensado para empujar un objeto de forma continua. Si quieres aplicar un impulso inicial, prueba a aplicarlo de esta forma a ver qué pasa:

```
rb.AddForce(push * thrust, ForceMode.
Impulse);
```

Ahora vamos a hacer que se mueva:

```
//Variable que contendrá el componente
Rigidbody rb;
//Vector que determinará la dirección del impulso
Vector3 push = Vector3.forward;
//Fuerza que le aplicaremos
[SerializeField] float thrust = 20f;

void Start()
{
    //"Capturamos" nuestro RigidBody
    rb = GetComponent<Rigidbody>();

}

//RECUERDA: aquí usaremos FixedUpdate en lugar de Update
void FixedUpdate()
{
    //Le aplicamos una fuerza pero de forma constante
    rb.AddForce(push * thrust);
}
```

Cuando movemos objetos mediante físicas, es recomendable usar el método FixedUpdate, en lugar del Update clásico, para evitar movimientos poco realistas.

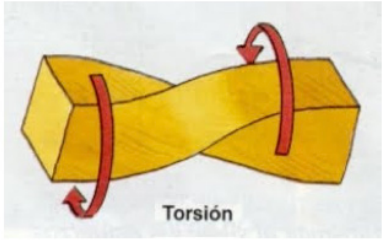
IMPORTANTE: si movemos un objeto dentro del método FixedUpdate y lo estamos siguiendo con una cámara, es importante que esa cámara se mueva también con ese método, o habrá un ligero parpadeo en el renderizado del objeto.

TORSIONES

Se aplican para hacer torcer un objeto en movimiento, ya que aplica fuerzas laterales.

El método utilizado es AddTorque(), y al igual que en el anterior hay que pasarle un vector de dirección, pero no admite más parámetros:

```
rb.AddTorque(transform.up * torque * turn);
```



NOTA: fíjate que aquí en lugar de crear un Vecto3.up, lo hemos hecho a través del mismo atributo "up" pero de la clase "transform", ya que también lo admite, pero siempre con relación al objeto.

Hay que tener en cuenta la resistencia que ofrece el objeto al giro, expresada en el parámetro Angular Drag de su RigidBody.

Podemos vincular los movimientos del joystick a las fuerzas de torsión aplicadas a nuestro vehículo (como un avión o una nave) y así simular un vuelo más realista.

Recuerda que los vectores de giro se corresponden con los ejes sobre los que bascula el vehículo:

- Yaw: eje Y
- Roll: eje Z
- Pitch: eje X

RECUERDA: todo lo que hemos visto para el método RigidBody es válido para elementos 2D, pero utilizando el componente RigidBody2D, que es igual salvo porque los objetos solo se pueden mover en los ejes X e Y, y que solo pueden girar en el eje perpendicular a ese plano.

RigidBody Velocity

Una alternativa a mover un objeto mediante fuerzas es aplicando una "velocidad", utilizando el método "RigidBody.velocity". Es como si un objeto se desplazara, pero sin recibir una fuerza (cosa de fantasmas), por lo que es menos realista.

Aplica un Vector3 para cambiar la dirección y la velocidad con la que cambia la posición del RigidBody (de una forma similar al método Translate).

Este sistema es menos realista, pero permite aplicar movimientos bruscos y cambios de velocidad rápidos (en algunos casos nos puede interesar, por ejemplo para saltos de personajes)

La sintaxis es sencilla:

```
rb.velocity = new Vector3(x,y,x);
```

Donde rb es el componente RigidBody del GameObject y el Vector3 indica la dirección y la velocidad a la que moveremos el objeto, medido en metros por segundo (convencionalmente)

PARA NOTA: Alternativas a estas fuerzas son AddForceAtPosition(), y AddRelativeForce(), que permite ajustar con precisión desde dónde y cómo se aplican esas fuerzas

EJERCICIO

Con todo lo que hemos aprendido, y unas nociones básicas de aerodinámica, vamos a hacer volar nuestro avión, pero como lo hacen los de verdad

Instrucciones

- Añade un RigidBody al "avión", y configura la masa y las resistencias de forma realista.
- Automáticamente la nave se precipitará contra el suelo.
- Añade un control de velocidad que empuje la nave hacia adelante. Al hacerlo, aumentará su velcidad en el eje Z.
- Usa la velocidad para aplicar un empuje hacia arriba (como los aviones de verdad), que hará que si te paras, el avión cae, pero si avanzas rápido subirá. Recuerda, el parámetro "Velocity" no solo puedes cambiarlo, también leerlo. Ejemplo:

```
float pushUp = rb.velocity.z;
```

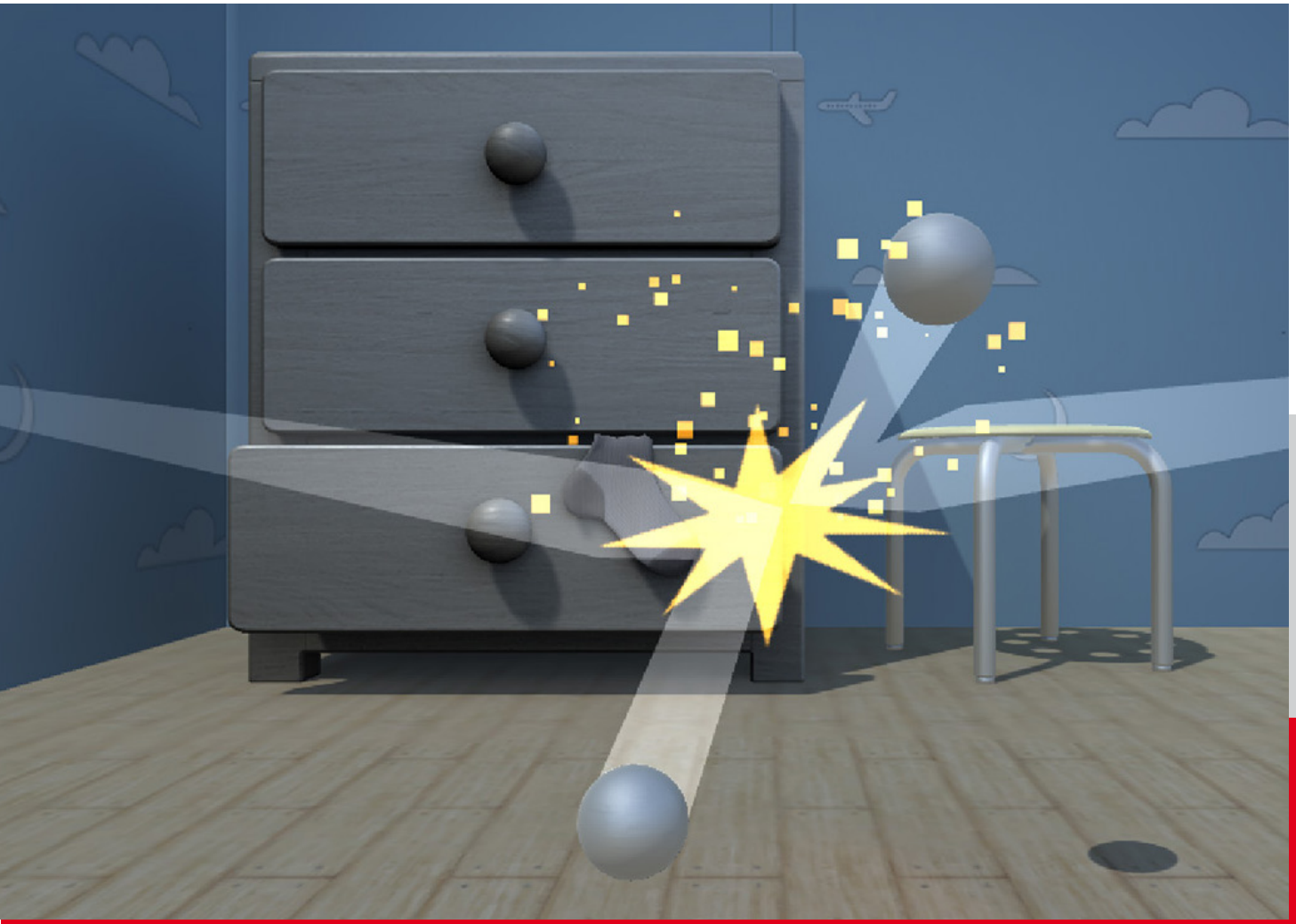
TRUCO: para empujar hacia arriba puedes usar el ForceMode.Acceleration que no tiene en cuenta la masa, lo que resulta cómodo en esta ocasión.

- Ahora añade torsiones para poder controlar también su giro.

RECUERDA: es bueno tener variables que gestionen la velocidad de desplazamiento y de giro, si pueden ser públicas o serializadas para ajustarlas en la ejecución del juego.

PARA NOTA: los aviones tienen 2 giros: uno con los flaps de las alas, y otro con el alerón de cola, el primero lo hace girar en torno al eje Z, y el segundo en torno al eje Y. ¿Podrías crear unos controles para poder manejar la nave en ambos ejes?

Álvaro Holguera
CIFP José Luis Garci
(Madrid)



Parte 2
COLISIONES

Hasta ahora hemos visto cómo hacer que los objetos respondan a las físicas, pero eso no implica que colisionen unos con otros.

COMPONENTE COLLIDER (Tipos)

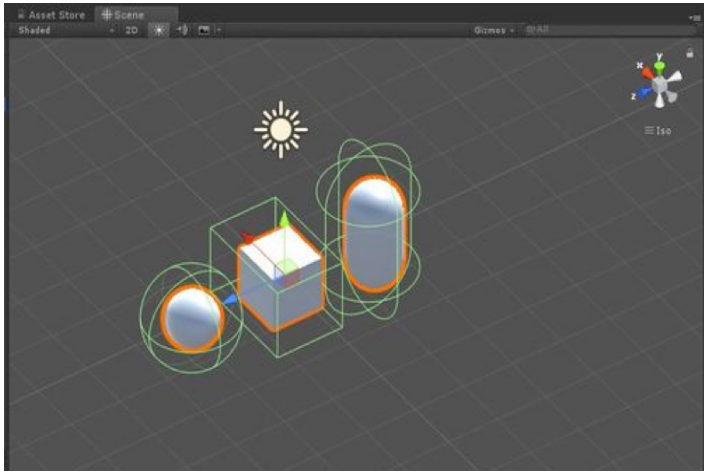
Ser afectado por las físicas no implica colisionar con otros objetos. Para ello, existen los colisionadores -ó "Colliders"-, que actúan como mallas invisibles que rodean al objeto y detectan cuándo entran en contacto con otro colisionador.

Como en la vida real, el comportamiento de los objetos al colisionar depende de su velocidad relativa y de su masa.

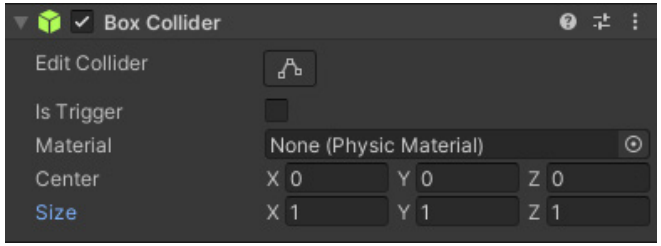
Al añadir un componente de tipo "collider" veremos que hay muchos tipos (incluso algunos ya vienen por defecto en las primitivas de Unity). Veamos los más habituales:

Box/Sphere/Capsule collider.

Colisionadores con formas básicas, a menudo integrados en las primitivas aunque podemos añadirlos nosotros a cualquier GameObject, incluso a un empty.



Veamos algunos de sus parámetros, comunes a casi todos los colliders:



- **Edit Collider**: si pulsamos este botón, se activarán los "asideros" del colisionador para poder modificar sus dimensiones manualmente.
- **Is Trigger**: si activamos esta casilla, los demás objetos lo atravesarán como si no tuviese materia, pero permite lanzar disparadores cuando entra en colisión con otros objetos (mediante OnTriggerEnter, OnTriggerStay y OnTriggerExit) como veremos más adelante
- **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. Lo veremos a contunación.
- **Center / Radius / Size:** posición y extensión del Gizmo que determina qué área es sensible a colisiones. Si no se ve la zona amarilla del colisionador, activar la visión de gizmos en la parte superior de la ventana de la escena. Ten cuidado de no dar valores negativos en el tamaño del colisionador.

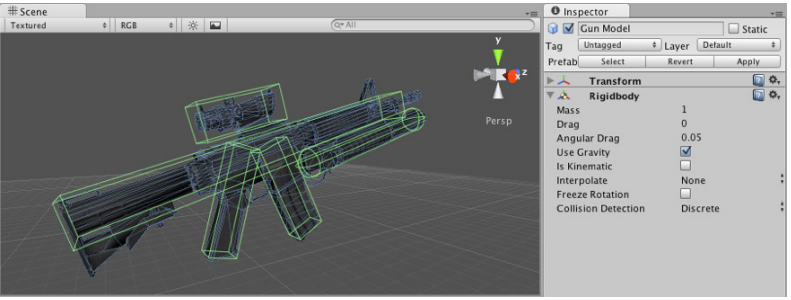
IMPORTANTE: para que dos objetos colisionen, los dos tienen que tener algún tipo de colisionador, y al menos uno de ellos un componente RigidBody. Y si los dos objetos tienen RigidBody, tendremos que activar en uno de ellos la detección de colisión "continua", en lugar de discreta.

COMPONENTE COLLIDER (Tipos)

Collider compuesto.

Podemos añadir varios colisionadores a un mismo objeto, para lograr una forma más compleja. Lo ideal es añadir hijos con los colisionadores, ya que todos serán detectados por el padre, como si fuera una forma compleja.

Partir el objeto en varias mallas y unirlas mediante jerarquía en la escena, de forma que cada una tiene un colisionador de malla más simple. Muy útil usar el componente Articulation Body



IMPORTANTE: no es recomendable añadir dos colisionadores del mismo tipo a un mismo Game Object. Y áun siendo de diferente tipo (por ejemplo, un box collider y un capsule collider), deben usarse para el mismo propósito. Y si son varios hijos, y el padre tiene RigidBody, es el padre el que detecta la colisión Trigger.

Existen paquetes en la Asset Store que permiten crear colisionadores compuestos de forma fácil.

Mesh Colliders

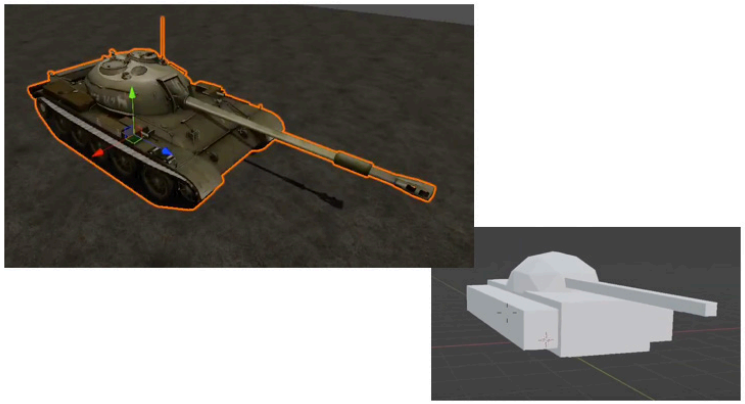
Los colisionadores de malla (o Mesh Colliders) toman la malla del objeto para generar un colisionador complejo y mucho más preciso

El problema es que si la malla tiene muchos polígonos afectará al rendimiento. Para evitarlo, el inspector de estos colisionadores tienen la opción de añadir una malla que actúe como colisionador (normalmente, una versión de baja poligonación del mismo objeto)

Tienen características propias:

- Poseen un checkbox "Convex" que permite que un colisionador de malla colisione con otro mesh collider (solo se puede aplicar a objetos con menos de 256 polígonos)
- Posee una opción llamada "Cooking Options", que determina cómo de preciso será la detección de colisiones en esa malla (mayor precisión exige más procesamiento, claro). Consultar el manual para más detalles.
- Mesh: a menudo, un objeto tiene una malla compleja que no tiene que trasladarse a su colisionador. En esos casos tenemos varias opciones:

Crear una malla más simple que se adapte a la compleja y que sea la que use para definir la zona que colisiona, ya que Unity permite añadir una malla distinta para el colisionador. Esta opción es la más adecuada siempre que sea posible



Fuente: https://www.youtube.com/watch?v=zLEhpVjgJb8

Otros

Terrain Collider: específico para suelos de terreno

Wheel Collider: pensado para las ruedas de los vehículos que tocan el suelo

Controller Collider: cuando usemos el Character Controllerpara manejar personajes, descubriremos que detectar sus colisiones no es tan sencillo.

DETECTAR COLISIONES

Además de la reacción propia de un objeto chocando contra otro (que depende de sus masas, su velocidad, su trayectoria, si uno de los dos no tiene RigidBody y se comporta como una pared, etc.), es importante que nosotros podamos detectar cuándo se ha producido una colisión, y actuar en consecuencia.

OnCollisionEnter vs. OnTriggerEnter

Unity tiene dos métodos heredados principales para ello:

1. OnCollisionEnter(): detecta cuándo hemos chocado con otro objeto.
2. OnTriggerEnter(): si activamos la casilla "IsTrigger" del collider, el objeto no interactuará físicamente (será atravesado), y este método lo detecta. Usado por ejemplo para trampas, activadores o cuando queremos sencillamente que algo desaparezca cuano lo toquen (pero que no rebote).

Cada uno de ellos tienen su correspondiente método que detecta cuándo ha dejado de producirse la colisión: OnCollisionExit() y OnTriggerExit(), e incluso un método que nos avisa mientras se está produciendo esa colisión, OnCollisionStay().

Ambos al ejecutarse pasan una variable (de tipo Collision o de tipo Collider, dependiento del método usado) que contiene todos los datos de la colisión,

Entre los datos que devuelve, por ejemplo, está el GameObject con el que hemos chocado, con todos sus atributos. En el siguiente ejemplo lo puedes ver:

```
//Detectaos una colisión y mostramos el nombre del objeto colisionado
private void OnCollisionEnter(Collision collision)
{
    string nombre = collision.gameObject.name;
    print("He chocado con: " + nombre);
}

private void OnTriggerEnter(Collider other)
{
    //Detectamos un "trigger" y mostramos la etiqueta del objeto
    string nombre = other.gameObject.tag;
    print("Me ha atravesado un objeto con la etiqueta: " + tag);
}
```

IMPORTANTE: si no se activa la casilla "IsTrigger" en el colisionador del componente, el método OnTriggerEnter no saltará.

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). Por eso a menudo debemos saber contra qué hemos colisionado y, mediante un if, actuar en consecuencia.

Ejemplo: mi nave va a colisionar con muchos objetos, pero dependiendo de si es un obstáculo, un Power Up o el mismo suelo, pasará una cosa u otra. Aquí son muy útiles las etiquetas en los objetos.

NOTA: los eventos de colisión solo se producen si uno de los objetos colisionados posee un rigid-body "non-kinematic", es decir, guiado por las físicas.

OnCollisionEnter()

Datos precisos de la colisión

Usar el método "OnCollisionEnter()" nos permite obtener datos de la colisión que a veces son neesarios (por eso es un método que consume más recursos), 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"

PRACTICA

Ahora que sabemos cómo detectar colisiones, vamos a añadir algo más de jugabilidad a nuestro juego de Zaxxon:

- Añade colisionadores a la nave (puedes usar los que tienen sus hijos, o añadirle uno al prefab con una forma aproximada). Eso sí, tienen que ser de tipo trigger.
- Añade un detector de colisiones a la nave, de forma que si choca con un obstáculo se autodestruya. Para ello, esos obstáculos tienen que tener también colliders, y la etiqueta adecuada. Este sería un ejemplo de código:

```
if(other.gameObject.tag == "obstaculo")
    Destroy(this.gameObject);
```

Lo ideal es que al chocar contra un obstáculo la nave se detuviera, pero no es ella la que se mueve,

PARA NOTA: Revisa en la documentación de Unity toda la información que nos devuelve esa variable de colisión, porque puede serte muy útil en el futuro.: https://www.youtube.com/watch?v=WY-mk-ZGAq8

Álvaro Holguera
CIFP José Luis Garci
(Madrid)

DETECTAR PRESENCIA

Aunqueesalgoquenecesitaremos más adelante, es bueno aprender algunas técnicas para calcular distancias y detectar presencias

CALCULAR DISTANCIAS Y RAYCAST

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 siguiente 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"):

```
[SerializeField] Transform other;
float dist;

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

NOTA: no es eficiente, ya que quizás no es necesario comprobarlo cada fotograma, y sería mejor ejecutarlo en una corrutina.

Raycast

Otra alternativa para detectar objetos próximos es utilizar el "Raycast" (tanto en 3D como en 2D). Esta herramienta lanza un "rayo invisible", dese una posición, en una dirección y a una distanci. Se puede indicar incluso la capa en la que buscará o la profundidad en Z. Veamos un ejemplo:

```
//Indicamos que el rayo solo choque con objetos en la capa 8
//Opcinalmente, podráimos indicar que fuese en todas menos la 8 layerMask = ~layerMask;
int layerMask = 8;

//Creamos la variable que nos dará los datos de la colisión
RaycastHit hit;

void FixedUpdate()
{

    //Si el Raycast devuelve true, es que hemos chocado con algo. Metemos los datos en la variable hit
    //Los parámetros son: origen, dirección, (OPCIONAL: variable Raycast con los datos de la colisión
    if (Physics.Raycast(transform.position, -Vector3.up, out hit, 100.0f))
    {
        print("He chocado con: " + hit.collider.gameObject.name);
        print("Está a una distancia de: " + hit.distance);
    }
}
```

Nos devolverá una variable, que si su atributo "collider" es distinto de null, ha colisionado con algo y toda la información de ese objeto

IMPORTANTE: si el rayo colisiona con nuestro propio colisionador, nos dará nuestros datos, por lo que es buena idea separarlo por capas (layers) y desactivar colisiones entre ellas.

En el ejemplo anterior, el Raycast que se lanza hacia abajo nos devuelve una booleana, que será true si choca con algo a 100 de distancia.

Los datos de la colisión se meten en la variable "hit" (añadiendo "out" antes) que es de tipo RaycastHit, que nos permite saber entre otras cosas la distancia a la que ha chocado.

Dentro del atributo "collider" tenemos los datos del objeto colisionado, por ejemplo el GameObject colisionado (**`hit.collider.gameObject`**)

Raycast2D

Este método tiene su versión es pecífica para 2D: RaycastHit2D. Los parámetros que podemos pasar son la posición inicial, la dirección y la distancia. Por ejemplo nos permite ver si tenemos algún objeto debajo nuestro a menos de 2 metros:

```
RaycastHit2D hit = Physics2D.
Raycast(transform.position, -Vector2.
up, 2f);
```

Si la variable hit no devuelve "null" puedo obtener por ejemplo la distancia de ese objeto debajo nuestro:

```
float distance = Mathf.Abs(hit.point.y -
transform.position.y);
```

CHECK Y OVERLAP

CheckSphere

Este método incluido en Unity dentro de la clase Physics, nos permite crear una esfera virtual de un radio determinado a partir de un punto determinado y –opcional- dentro de una capa determinada y nos devuelve si hay objetos dentro (y si es necesario, qué objetos)

Devuelve una booleana, pero nos permite también saber si hay objetos dentro de esa esfera, y qué objetos son

```
if (Physics.CheckSphere(transform.
position, sphereRadius, checkLayer));
```

Muy útil por ejemplo para detectar si un objeto está tocando el suelo. Para llevarlo a cabo debemos:

1. Crear un Empty a los pies de nuestro objeto, al que accedermos a su componente Transform.
2. Una variable float que determina el radio de esta esfera de comprobación
3. Una capa en la que meteremos los objetos que queremos comprobar (normalmente el suelo, para evitar que detecte colisiones con otros). Debemos crear esa Layer y poner en ella el suelo. En el código, al crear una variable de tipo LayerMask serializada nos peritirá elegirla en Unity
4. Una booleana que nos dirá si estamos tocando el suelo

Una vez tenemos esto, podemos comprobar en todo momento si estamos tocando el suelo y pasarlo a nuestra booleana. Debajo tienes un ejemplo de código.

Tiene una versión similar pero con forma de cápsula, aunque en este caso hay que pasarle dos centros en lugar de uno.

```
[SerializeField] Transform checkGround; //el empty object
bool isGrounded; //Booleana que me dirá si estoy tocando el suelo
float checkRadius = 0.3f;
[SerializeField] LayerMask groundLayer; //Mostrará un desplegable con las capas

void Update()
{
    //Comprobamos en todo momento si estamos tocando suelo
    isGrounded = Physics.CheckSphere(checkGround.position, checkRadius, groundLayer);

}
```

Overlap Sphere / Box / Capsule

Es un método muy útil en Unity, dependiente de la clase Physics, que nos permite detectar los objetos que están cerca de nosotros, a partir de una esfera que nos rodea o una caja o una cápsula

Dibuja ese elemento a partir de una posición dada y un radio, y nos devuelve una lista de los objetos que están dentro

Debemos recogerlos en un array de tipo Collider:

Una vez tenemos los elementos los podemos listar como cualquier otro array. En el siguiente ejemplo de Unity, se crea una función a la que hay que pasar el centro de la esfera y su radio, y con ello crea una variable de tipo Collider (“hitColliders”) en las que mete todos los objetos que hay en esa esfera.

```
void ExplosionDamage(Vector3 center, float radius)
{
    Collider[] hitColliders = Physics.OverlapSphere(center, radius);
    foreach (var hitCollider in hitColliders)
    {
        hitCollider.SendMessage("AddDamage");
    }
}
```

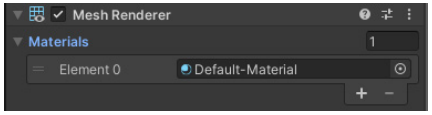
MATERIALES FÍSICOS

Aunque los materiales asociados a texturas los veremos más adelante, vamos a adelantar algunos conceptos para poder analizar así este material que está vinculado directamente con las físicas.

Aplicando materiales

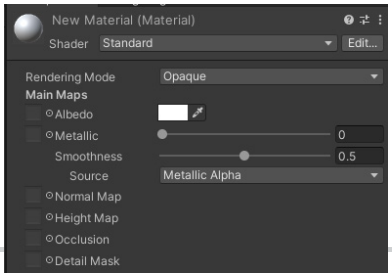
Como ya deberíamos saber, la apariencia física de un objeto 3D lo da su material, basado en fórmulas matemáticas que indican cómo responde a la luz (los shaders) y a la vez en las imágenes 2D que lo "recubren" (las texturas).

Cualquier objeto 3D que queramos que se vea en cámara, debe tener un componente llamado "Mesh Renderer" encargado de que se renderice (si lo desactivas verás que desaparece, aunque siga existiendo, algo que puede ser útil en el futuro).



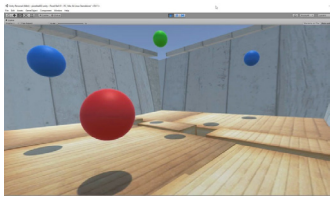
Ese componente es el que tiene el material asociado, que en el caso de las primitivas es un material preexistente llamado Default Material.

Tú puedes crear materiales propios, con sus propios shaders y texturas, para aplicarlo a este Mesh Renderer: botón derecho del ratón en el proyecto > Create > Material:



Pero este es algo que veremos más adelante.

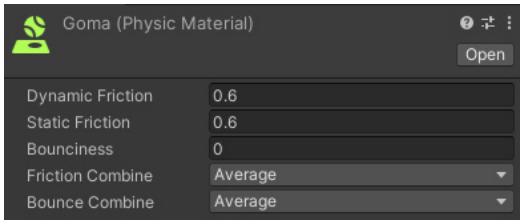
Materiales físicos (Physics Materials)



Este tipo de materiales están diseñados para añadir efectos elásticos a nuestros objetos, ya que no reacciona igual una esfera de metal que una de goma.

Van vinculados a los colliders del Game Object, ya que se aplican a la forma que tienen de interactuar con otros objetos.

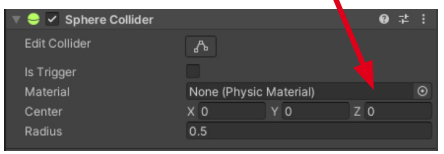
Los podemos crear en el menú de Assets>Create o con el botón derecho en el proyecto > Create > Phisic Material. En este caso, hemos creado uno llamado "Goma"



Los parámetros que permite configurar son:

- **Dynamic Friction**: el rozamiento y la resistencia que presenta el material cuando un objeto se desplaza por él (a mayor valor, antes lo frenará al aumentar la resistencia)
- **Static Friction**: similar al anterior, pero se aplica cuando el objeto está quieto en contacto con el material. Por ejemplo, dificulta que un objeto en contacto comience su movimiento
- **Bounciness**: efecto de goma, ya que hace que los objetos reboten al entrar en contacto con él
- **Friction / Bounce Combine**: cuando un objeto con un physic material entra en contacto con otro, las propiedades de ambos se combinan, ya sea sumándose, haciendo una media, tomando el valor que sea máximo o el que sea mínimo

Ahora podemos arrastrarlo al Material que encontramos en nuestro collider:



Prueba a crear una pelota de goma y hazla rebotar y rodar por una superficie cambiando los parámetros, a ver qué ocurre.

ENLACES A VÍDEOS

Aquí podrás acceder a los vídeos que explican y/o profundizan sobre los conceptos vistos, por si te son de utilidad.

Mundo físico

Introducción:

https://www.youtube.com/watch?v=YkVzFK7fjI8

Colisiones:

https://www.youtube.com/watch?v=hBX2xP4ERfY

Arrays

Aunque no está relacionado directamente, veamos cómo se pueden usar los arrays en Unity:

https://www.youtube.com/watch?v=L8QpQ_mg2Zk