

# **Programación de Video Juegos con Unity. Nivel inicial.**

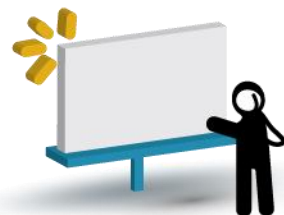
*Centro de e-Learning SCEU UTN - BA.*

*Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148*

**[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)**

## **Módulo 1: Prototipado**

### **Unidad 3: Interacción de objetos**



## ***Presentación:***

*¡Ya podemos movernos y disparar, pero queda lo más divertido, romper cosas! En esta unidad veremos cómo saber cuándo dos objetos entran en contacto, y en tal caso, como destruirlo o quitarle vida. Como siempre, los conceptos los utilizaremos para más que ello, como, por ejemplo, subir la vida, subir la velocidad, destruir a objeto concretos, etc.*

*También veremos cómo temporizar acciones para controlar el tiempo.*

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

**[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)**



## *Objetivos:*

### Que los participantes aprendan a:

- *Detectar cuándo dos objetos chocan.*
- *Modificar las propiedades de otros objetos.*
- *Temporizar acciones.*

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148  
**[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)**



### *Bloques temáticos:*

- *Colisiones.*
- *Comunicación e Interacción.*
- *Temporizadores.*

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148  
**[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)**



## *Consignas para el aprendizaje colaborativo*

*En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC\*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:*

*Los foros proactivos asociados a cada una de las unidades.*

*La Web 2.0.*

*Los contextos de desempeño de los participantes.*

*Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.*

*Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.*

*El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.*

*\* El MEC es el modelo de E-learning colaborativo de nuestro Centro.*

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148  
**[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)**



### **Tomen nota:**

*Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.*

*Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.*

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

**[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)**

## Colisiones



**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148  
[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)

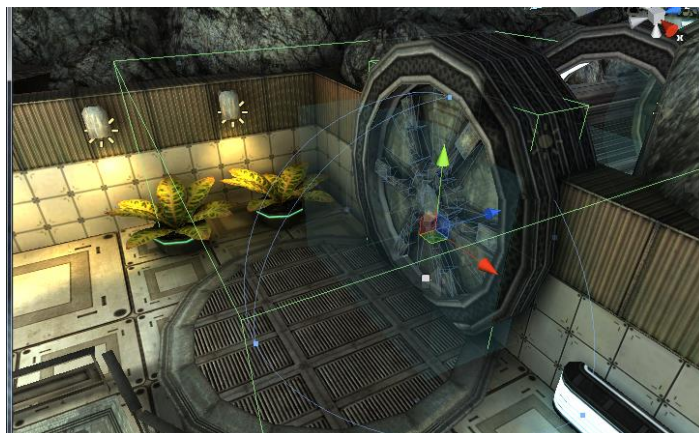


## Configurar el sistema de física

*En las unidades anteriores hablamos sobre colliders. Recordemos que los colliders sirven para representar la figura física del objeto, o sea, el volumen físico que ocupa, que el mismo es definido por una geometría. También recordemos que aparte de tener la geometría física teníamos la visual, la cual podría ser más detallada que la física ya que el proceso de dibujar un objeto es diferente al de detectar colisiones, razón principal de esta separación.*

*Cuando el juego empieza a funcionar y las cosas se empiezan a mover, van a llegar situaciones en las cuales las geometrías de 2 o más objetos entren en contacto, lo que normalmente se conoce como una “colisión”. Uno normalmente pensaría que en esta situación los objetos tendrían que bloquearse mutuamente como pasaría normalmente en la vida real, pero no siempre vamos a querer ello.*

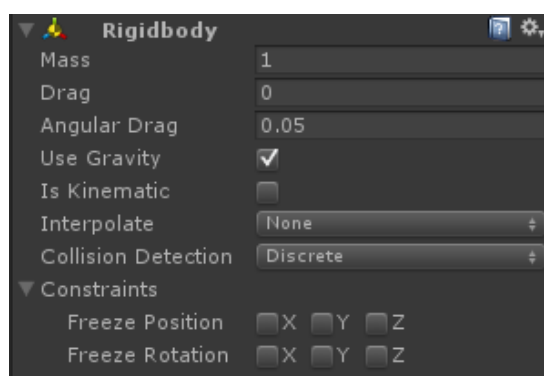
*Por ejemplo, en el caso de que el personaje choque contra la pared es deseable que el personaje no pueda pasar a través de ella, pero pensemos un caso donde el personaje tiene que pasar por una puerta automática. La puerta automática se debería abrir cuando el personaje pasa enfrente de ella, pero sin tocarla, para ello lo que se puede hacer es poner un collider enfrente de la puerta y detectar que cuando algo entro en contacto con ese collider se abra la puerta, teniendo en cuenta que el mismo debe ser invisible, quitándole el MeshRenderer. Ahora dicho collider no debería bloquear el paso del jugador porque si no, no podría pasar por la puerta. Solamente nos interesa saber si el personaje entro en esa área, no necesitamos que haya una repulsión entre los dos objetos. Este último caso normalmente se conoce como “trigger”, cuando un objeto choca con otro, pero pueden aun así pasarse entre ellos. Otros casos de trigger podría ser un área donde si el personaje está adentro lo cura, o un área que cuando la tocas aparecen enemigos, como si fuese un botón.*



Tenemos casos donde un objeto tiene que bloquear a otro, pero que puede ser arrastrado, casos donde un objeto tiene que estar fijo y nada pueda moverlo, casos donde ciertos objetos colisionan entre si y ciertos no, en resumen, hay muchas formas de interacción física entre objetos. Unity posee herramientas que nos permiten lidiar con todas estas situaciones.

### Configuraciones del Rigidbody y los Colliders

Como vimos hasta ahora, el componente Rigidbody nos permite agregarle físicas a nuestro objeto, por lo que el movimiento de nuestro objeto es gobernado por el, y es la base del sistema de colisiones. También los colliders son parte fundamental del sistema, ya que estos especifican que figura tiene el objeto a nivel físico. Este componente configurado de la manera adecuada puede lograr recrear todas las interacciones que deseemos. Veamos sus propiedades.



- **Mass:** A la hora de aplicar fuerzas físicas a objetos el peso se tiene en cuenta. Si aplicamos una fuerza minúscula a una pluma la movemos, cuando a un auto no. El tamaño del objeto afecta su peso, pero si tenemos un cubo de 1m de lado de goma espuma no es lo mismo que uno de acero. La masa sirve como multiplicador del

**Centro de e-Learning SCEU UTN - BA.**

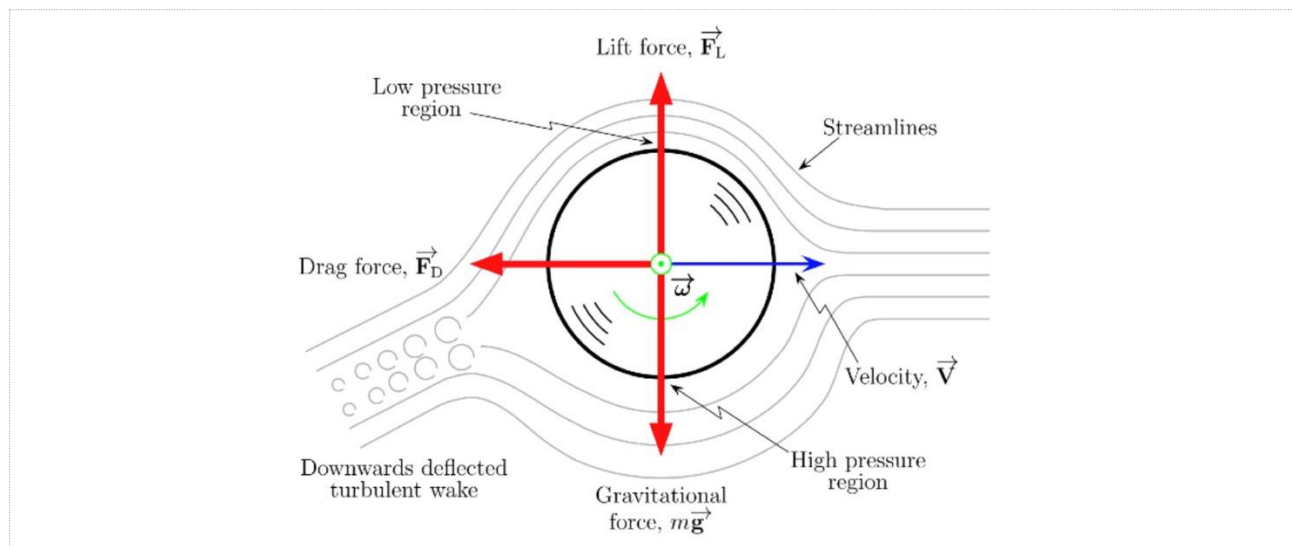
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148  
[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)



*peso, si tenemos el cubo de goma espuma que hablamos antes la masa va a ser chica (0.1 por ejemplo) y en el caso del de acero más grande (10 por ejemplo). En resumen, se usa para hacer al objeto más pesado en caso de ser necesario.*

- *Drag: Si notan, si tiran una pelota de futbol y una hoja desde la misma altura, la pelota cae más rápido. Ahora esto rompería los conceptos básicos de la gravedad, donde independientemente del peso del objeto, la gravedad se aplica por igual independientemente del peso. Lo que sucedió acá es que no tuvimos en cuenta el rozamiento con el aire. Cuando un objeto se mueve en el vacío no hay ninguna fuerza que se le oponga, por ende, si empujamos un objeto en el vacío se va a mover infinitamente. En la tierra nosotros nos movemos en un medio gaseoso, el aire, por lo que cuando nos movemos estamos constantemente chocando contra él, a pesar de que no haya viento se opone como fuerza. Por la fuerza de movimiento de un humano esta fuerza es minúscula, pero para una hoja es titánica, o también, en el caso de un avión, al moverse tan rápido, se empieza a sentir la resistencia del aire. El valor de la propiedad drag es un multiplicador de esta resistencia, cuanto más grande el valor, más resistente es el aire, por lo que, si tiramos una pelota en un ambiente más denso, esta se va a detener más rápido. En resumen, se usa para detener el objeto más rápido o más lento, así no se queda moviendo eternamente.*
- *Angular Drag: El mismo concepto de drag aplica a angular drag. La diferencia es que angular drag como el rozamiento con el aire detiene la rotación del objeto, a diferencia de drag que controla solamente la posición. Si bien esto no es físicamente correcto del todo (dependiendo de la figura del objeto), este tipo de propiedades nos permiten modificar el sistema de físicas para que no sea preciso, pero sirva para recrear situaciones que se dan en los videojuegos, pero no en la vida real. Al igual que con la programación, el uso creativo de las propiedades puede lograr comportamientos interesantes y hasta quizás alguno esperado por el programador del componente (comportamiento emergente). En resumen, se usa para parar la rotación del objeto más rápido o más lento para que no rote eternamente.*
- *Use Gravity: Como se imaginan, simplemente sirve para especificar si se aplica la fuerza de gravedad o no. Por ejemplo, si estuviésemos en el espacio eso estaría destildado. Ahora, en otros casos quizás queramos tener una gravedad más débil, por ejemplo, si estuviésemos en el agua. Esto se podría recrear aplicando una fuerza ligera hacia arriba para contrarrestar la gravedad, apagando la gravedad y aplicar nuestra propia gravedad, o cambiar el valor de la gravedad global en el*

*Physics Manager (Administrador Físico), pero estas cosas las veremos más adelante.*



- *Is Kinematic: Esta propiedad no representa algo tan natural, pero básicamente sirve para desactivar el sistema de física, sin quitar el componente. Hay ciertas situaciones en las cuales nosotros queremos que nuestro objeto pase de moverse con código nuestro a moverse completamente por físicas (se puede hacer mezcla de ambos, pero es algo muy difícil y poco intuitivo). Por ejemplo, cuando nos desmayamos pasaríamos al conocido modo ragdoll, donde caeríamos sin control de nuestras articulaciones, hasta que recobramos el sentido y volvemos a controlar el cuerpo.*
- *Interpolate: Esta sería una propiedad avanzada que se recomienda mantener en off a excepción del personaje principal u objetos que sean muy importantes, ya que sirve para corregir errores visuales debido a que el sistema de físicas no está sincronizado con el sistema gráfico, por lo que en ciertas situaciones se pueden desfazar.*
- *Collision Detection: Otra propiedad avanzada que se recomienda dejar como está a menos que el objeto tenga problemas. Collision Detection resuelve problemas cuando objetos que se mueven muy rápido atraviesan las paredes. Esto se debe a que, si un objeto se mueve muy rápido, quizás de un frame a otro se movió de una posición, a otra a 100 metros más adelante. El sistema de físicas tiene limitaciones debido a que estamos en un juego, lo que implica que todo tiene que andar lo más*

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)

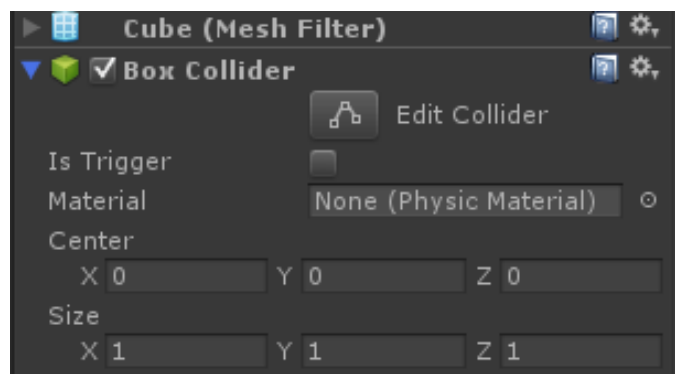
*rápido posible porque si no el juego se ve lento. Para compensar esto se baja la calidad hasta cierto punto donde se logra la fluidez requerida (esto afecta a todos los aspectos de un juego, la física, los gráficos, la IA, etc.). Si el objeto atraviesa cosas hay que poner Continuous Dynamic, y a las cosas a las que atraviesa hay que ponerle Continuous. Recalcamos el uso prudente de esto, ya que tiene un gran impacto en la performance de nuestro juego, o sea la fluidez de mismo.*

- *Constraints: Sirve para bloquear ejes en el sistema de física. Por ejemplo, si tenemos un juego de pool, donde las pelotas son 3d y se mueven con físicas 3D, pero no queremos que las pelotas puedan moverse para arriba ya que si no se irían de la tabla (sin contar que en la vida real eso se puede), podríamos bloquear el eje Y de la posición de las pelotas. Esto no implica que el objeto no puede moverse en ese eje, solamente el sistema de físicas no puede moverlo, cambiando el transform a mano o con la instrucción Translate sí. Otro ejemplo podría ser el del caso de un personaje. Generalmente a un personaje se le pone un collider de capsula para colisionar contra el nivel. Si no bloqueamos la rotación, la capsula se caería, junto con el personaje. Hay que tener en cuenta que esta propiedad hay que abrirla para verla, recordemos que, si vemos una propiedad con una flecha a la izquierda, la misma tiene más propiedades dentro.*

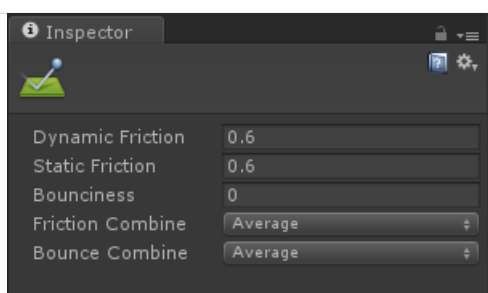
*Un aspecto importante sobre el Rigidbody es el tamaño de los objetos. Más allá de controlar el peso del objeto, el sistema de físicas toma una unidad de tamaño como 1 metro. Si nosotros escalamos todo el juego para que todo sea gigante, para un juego sin físicas va a dar lo mismo, simplemente vamos a manejar medidas muy grandes. Pero para un juego físico eso no sería válido, ya que la gravedad de los objetos se vería lenta, ya que está aplicando una gravedad minúscula a objetos gigantescos, y por más que se mueva a muchos metros por segundo, si esta todo escalado muy grande parecerían centímetros, ya que un personaje mal escalado quizás mida 1000 metros. En resumen, tener en cuenta que 1 unidad es un metro. Algo para lidiar con esto es hacer un cubo, dejarlo como viene sin modificarlo y tenerlo como referencia.*

*Ahora por ultimo nos quedarían las configuraciones de los colliders. Si bien todas las propiedades de ellos afectan al sistema de física, la mayoría son propiedades para cambiar la geometría del objeto y son particulares a cada collider. Dejando ellas de lado tenemos dos comunes a todos los tipos.*





- *Is Trigger: Esta propiedad indica si el objeto es un trigger o no. Como hablamos anteriormente, un trigger es un objeto que no bloquea a otros objetos, pero que puede reaccionar al choque si así lo programamos nosotros. Recordemos el caso de la puerta automática con un trigger delante para detectar si hay que abrir la puerta o no, o también el área de curación, en la cual cuando el personaje pase sobre ella se cura.*
- *Material: Esta propiedad permite cambiar aspectos físicos diversos de este objeto, como es la fricción que aplica a otros objetos, o que tanto rebota. Para ello se puede crear un Physics Material yendo a Assets -> Create -> Physics Material, lo cual crea un nuevo asset que lo podemos arrastrar a dicha propiedad. No vamos a entrar en detalles de cómo configurar este asset, pero pueden hacer pruebas cambiando las propiedades bounciness y friction.*



Ahora que ya vimos las propiedades, veamos configuraciones comunes para lograr diferentes tipos de interacción.

### Tipos de Interacción Física

Como vimos, las configuraciones son muchas y alguna de ellas no son muy intuitivas. De igual forma configurarlas correctamente tampoco es intuitivo. Veamos algunas de las configuraciones comunes para que nos demos una idea de cómo configurar la física.

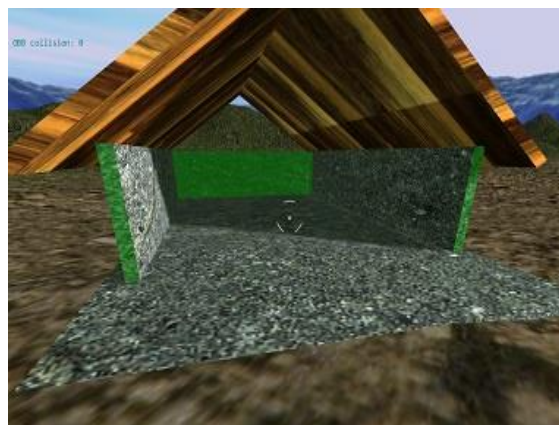
**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)



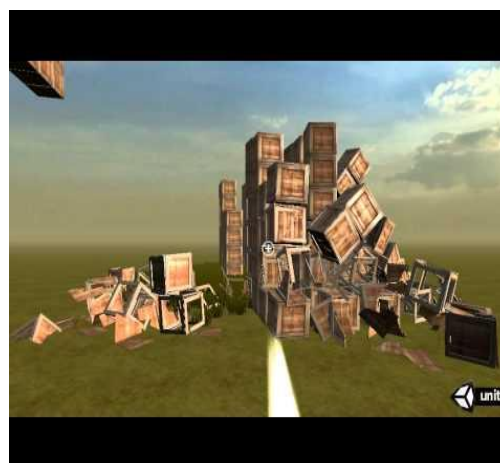
- *Estático Bloqueador: Algo que impide el paso de otros objetos, o sea, algo contra lo cual los otros objetos van a chocar, pero que a su vez no se mueve, algo que siempre va a estar en ese lugar y que bajo ninguna circunstancia cambia de posición. Cosas como el piso, las paredes, columnas, techos, piedras, etc. Para lograr este tipo de objetos no hay que poner Rigidbody, tan solo con el collider y asegurándose de que "Is Trigger" este destildado alcanza.*



- *Estático Trigger: Igual al anterior, algo que no se va a mover jamás, pero que no bloquea el paso de otros objetos. Sirve como sensor para detectar cuando algo paso por allí y reaccionar ante ello sin bloquear el paso del objeto. Al igual que el anterior, sin Rigidbody, pero con la propiedad "Is Trigger" del collider tildada.*



- *Dinámico Bloqueador Realista: Este sería un objeto que se mueve, bloquea a otras cosas, es bloqueado por otras cosas y se le aplican físicas realistas. Puede servir para pelotas, rocas que caen, puertas que se abren cuando uno las choca, autos, cualquier cosa que requiera de física realista. A este tipo de objetos generalmente se los mueve aplicando fuerzas con instrucciones de Rigidbody en vez del "transform.Translate", pero no vamos a estar usando eso por el momento. Para lograr este comportamiento basta con ponerle el Rigidbody como viene por defecto.*

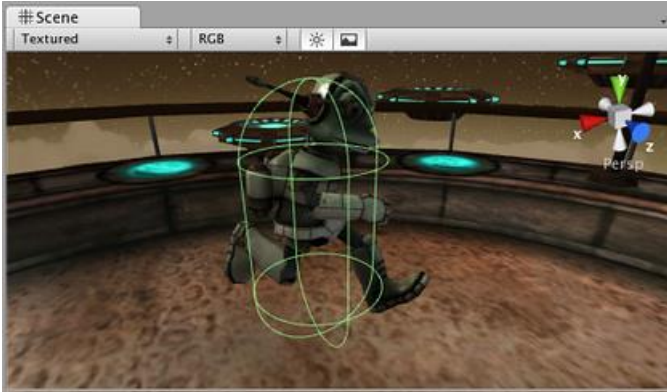



- *Dinámico Trigger Realista: Si bien es un objeto que se puede lograr, no tiene mucha utilidad práctica, ya que al ser trigger el objeto no choca contra nada, por lo*



que al aplicarle físicas realistas lo único que va a hacer es caer al infinito (si tiene tildado "Use Gravity"). Sería un objeto con Rigidbody y "Is Trigger" tildado.

- Dinámico Bloqueador:** Este tipo de objetos son los que por general vamos a querer para simular personajes u objetos móviles controlados por código. Son iguales a su versión realista, bloquean y son bloqueados por otros objetos, pero la diferencia es que en situación de impacto no van a recibir impulsos, rotaciones, ni se les va a aplicar gravedad, etc. Para lograr este tipo de objetos, tenemos que tener el "Is Trigger" del collider destildado, tener un Rigidbody, que el mismo tenga tildado los tres ejes en la propiedad Freeze Rotation de las Constraints y que en la propiedad "Drag" tenga el valor "Infinity". Tildar las Constraints de rotación anularía cualquier fuerza de rotación resultado de impactos con otros objetos, pero recordemos que aún podemos rotar el objeto con "transform.Rotate". Poner "Drag" en infinito (la palabra "Infinity" es válida como un numero) hace que se anule cualquier fuerza de movimiento resultado de impactos con otros objetos, pero aun así hace que nuestro objeto no pueda traspasar otros, o sea, que sea bloqueado.


- Dinámico Trigger:** Este tipo de objetos a diferencia de su versión realista si pueden ser útiles. Pensemos en el caso de una bola de fuego. Se mueve, necesitamos saber cuándo chocó contra algo, pero no empuja cosas, las traspasa, dañándolas en el camino. Para lograr este tipo de objetos "Is Trigger" tiene que estar tildado, el objeto tiene que tener Rigidbody y tener tildado "Is Kinematic" en el Rigidbody. Recordemos que tildar esto último anula cualquier interacción física, inclusive la parte de bloquear y ser bloqueado (aunque eso tampoco iba a estar por el "Is Trigger"), pero sigue detectando colisiones que manejaremos más tarde por código.







*Si bien estas son las configuraciones más comunes, esto no implica que sean las únicas. Y si bien vimos una forma particular de lograrlas, tampoco que implica que no haya otra forma. Esto, como todo en programación, se puede lograr de diferentes formas.*

### **Moviendo con Rigidbody**

*El único detalle a tener en cuenta ahora es que, si nosotros queremos mover un objeto con el componente Rigidbody para evitar que traspase las paredes, debemos poner el código de movimiento dentro de un evento llamado FixedUpdate. Esto se debe a que el sistema de física no corre en paralelo con los Updates comunes, pero en el nivel avanzado profundizaremos más al respecto, por ahora hay que hacer todos los Translate (cosa que también eventualmente cambiaremos) dentro del FixedUpdate o sino obtendremos un comportamiento errático.*

### **Tips del Sistema de Física**

*Si bien el sistema es poderoso, es un poco complicado de configurar. Para entender un poco mejor el porqué de estas configuraciones, veamos algunos tips a la hora de usar físicas.*

*Manejar todas las interacciones físicas, por más que bloqueemos ciertos ejes, es muy difícil, ya que por lo general va a generar efectos físicos realistas no deseados, a menos que estemos conscientes de lo que estamos haciendo y realmente los queremos. La mayoría de los juegos en muchas situaciones no usan físicas realistas, siendo una premisa general de los videojuegos que no siempre la realidad es divertida. En la vida real no podemos girar tan rápido para disparar a un enemigo, no podemos acelerar y desacelerar de una manera tan abrupta, no podemos doblar tan rápido en una curva con un auto, entre otras cosas que los juegos si necesitan hacer. Si bien el sistema de físicas es muy configurable, tenemos ciertos límites. No siempre la física realista nos va a servir para lograr la sensación que le queremos dar al jugador, en esas situaciones uno programa su física no realista a mano (cosa que no es ni muy simple ni muy complicado, pero que lo dejaremos para otro momento) pero aprovechando la detección de colisiones que ofrece el sistema.*

*Otro tema importante a destacar de las configuraciones es que, de ellas, las de los objetos que no se mueven (estáticos) ninguna necesito Rigidbody, a diferencia de las de los objetos móviles (dinámicos). Ahora, si un objeto que si tiene Rigidbody atraviesa este objeto que no, el si detecta la colisión. El Rigidbody chequea colisiones contra cualquier collider, tenga o no tildado "Is Trigger", o sea, choca tanto contra objetos estáticos como dinámicos. Esto explica porque los objetos que no se mueven no requieren de Rigidbody, mientras los que si se mueven sí.*

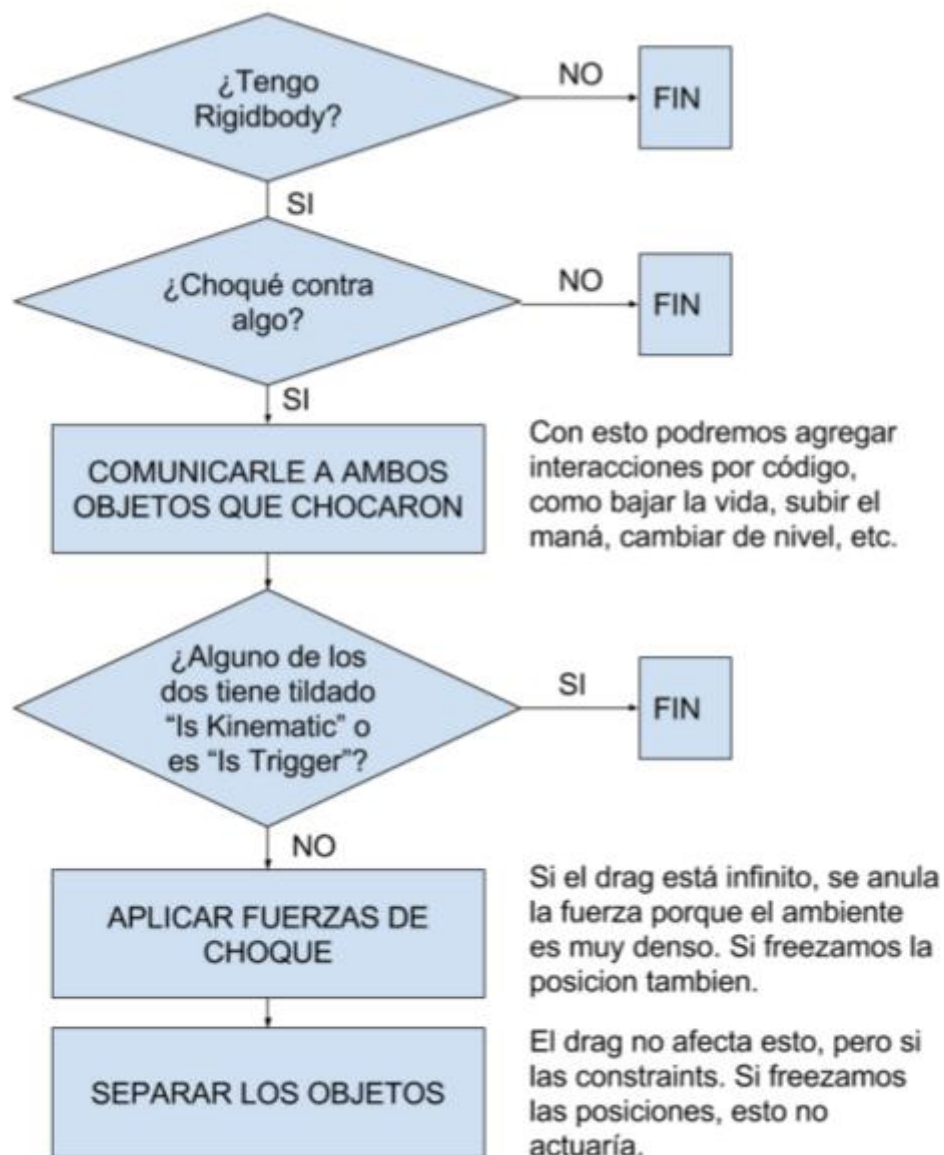
**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

**[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)**



Por ultimo veremos un pequeño diagrama conceptual que nos ayudara a resumir lo visto anteriormente. Así comprenderemos un poco más el funcionamiento interno del sistema de físicas sin entrar en demasiados detalles técnicos.





**UTN.BA**  
UNIVERSIDAD TECNOLÓGICA NACIONAL  
FACULTAD REGIONAL BUENOS AIRES

**Centro de  
e-Learning**



### **Ejercicio N°1**

- Cree un cuarto estático bloqueador (piso, paredes con cubos), pero que una de las paredes tenga una puerta, también estática bloqueadora (hace una pared hecha con 3 cubos que hagan un agujero, y otro cubo que sea la puerta). Asegurarse de que las paredes sean gruesas.
- Adentro poner una esfera Dinámica Bloqueadora Realista
- Delante de la puerta poner un cubo Estático Trigger y quitarle el MeshRender
- Crear una capsula y ponerle el script de movimiento por teclas que mueva para adelante y atrás con las teclas w y s, y que rote hacia los lados con a y d. Configurar a la capsula como Dinámico Bloqueador. No ponerle una velocidad de movimiento muy grande al script.
- Crear cubos a modo de enemigos y configurarlos también como Dinámico Bloqueador.
- Asegurarse de que la cámara pueda ver el cuarto y todo lo demás.
- Ejecutar la escena y ver como el personaje no puede atravesar las paredes, empuja a los enemigos y empuja la esfera. Por momentos se va a ver que va a tener problemas con las paredes, como que entra y sale, ello lo veremos más adelante.

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)



## Detectar colisiones por código

Ahora que ya hemos configurado el objeto de la forma que necesitábamos nos queda agregarle reglas a la interacción, cosas que sucedan cuando dos objetos colisionan, como destruir al objeto que choca un misil, dañar con una bala, curar con un powerup, entre otros, cosa que haremos por código.



### Eventos Físicos

En la unidad 1 vimos de que son los eventos. Ellos actúan como un conjunto de instrucciones a ejecutar ante determinadas situaciones. Teníamos por ejemplo el Start, evento cuyas instrucciones se ejecutan cuando el objeto se crea, o también teníamos el Update, que se ejecuta por cada frame del juego. Escribiendo código dentro de los eventos agregamos las interacciones necesarias a nuestro juego, que tiene que pasar y cuando. Hasta ahora vimos los dos eventos más básicos y fundamentales que Unity ofrece, pero cada sistema de Unity, como el de física, tiene sus propios eventos.

Cuando se produce una colisión, según como hemos configuremos los objetos que colisionan, pueden suceder varias cosas que ya vienen programadas, como bloquear el paso de un objeto y hacerlo rebotar. No siempre esas interacciones van a ser las únicas que vamos a necesitar, para ello Unity nos da eventos de colisión para agregar nuestras propias respuestas ante una colisión, como subir la vida, cambiar el nivel, etc.

Recordemos que en la colisión participan dos objetos, existen dos tipos de eventos de colisión según las configuraciones de estos objetos. Las colisiones de objetos que se bloquean mutuamente (bloquean y son bloqueados) se las conoce como "Collision". Ahora, si alguno de los dos objetos no bloquea, tanto si es porque tiene "Is Trigger" tildado y/o tiene "Is Kinematic" tildado, a este tipo de colisiones se las conoce como "Trigger", ya que, a diferencia de "Collision", en este tipo de eventos los objetos no se bloquean. Así que, en resumen, si los objetos bloquearon su paso se produjo una "Collision", sino un "Trigger". Independientemente de quien produjo la colisión, quien se movió y quien no, ambos objetos reciben el evento correspondiente, cosa de que ambos pueden tener componentes que respondan ante colisiones.



Anteriormente hablamos del concepto de ciclo de vida de un objeto. Un objeto nace (se instancia con "Instantiate"), vive (se ejecuta código en el Update) y muere (se destruye con "Destroy"). Unity tiene eventos para esos tres estados del objeto, el Start para cuando nace, el Update mientras vive y el OnDestroy para cuando se destruye. Con estos eventos podemos ejecutar acciones específicas en situaciones específicas de la vida de un objeto. Por ejemplo, en el Start le podemos dar play a un sonido de motor de auto, en el Update podemos ir cambiándolo según la velocidad para que suene diferente, y en el OnDestroy podemos pausarlo y quizás también poner en el lugar donde estaba el auto un efecto de explosión así el auto no desaparece sin más.

Tanto las "Collision" como los "Trigger" cuentan con 3 versiones que serían "Enter", "Stay" y "Exit", las mismas representando el ciclo de vida de una interacción física. El enter es el nacimiento de la interacción, el Stay el durante de la interacción y el Exit el fin de la interacción. Por ejemplo, en el enter podemos comenzar una animación de chispas haciendo alusión a que el objeto está siendo arrastrado y podemos pausarlo en el Exit, o también en el stay podemos quitar la vida gradualmente, haciendo alusión a que el objeto quita vida mientras uno lo toque.

Sabiendo que hay 2 tipos de colisión, y ambas se dividen en 3 tipos, tenemos un total de 6 nuevos eventos para incluir a nuestro repertorio, veamos a continuación como escribirlos.

```
void OnTriggerEnter()
{
    print("Empece a colisionar con");
    print("algo que no me bloquee");
}
```

```
void OnTriggerStay()
{
    print("Sigo colisionando con");
    print("algo que no me bloquee");
}
```

```
void OnTriggerExit()
{
    print("Deje de colisionar con");
    print("algo que no me bloquee");
}
```

```
void OnCollisionEnter()
{
    print("Empece a colisionar con");
    print("algo que SI me bloquee");
}
```

```
void OnCollisionStay()
{
    print("Sigo colisionando con");
    print("algo que SI me bloquee");
}
```

```
void OnCollisionExit()
{
    print("Deje de colisionar con");
    print("algo que SI me bloquee");
}
```



### **Ejercicio N°2**

- *Crear un componente que detecte una colisión y que se autodestruya cuando eso suceda (`Destroy(gameObject)` dentro del evento `OnCollisionEnter`), o sea, que cuando cualquier cosa los toque se destruya.*
- *En la escena hecha en el ejercicio anterior agréguele el componente a las balas.*
- *Crear otro componente que tenga una propiedad de tipo `GameObject` (como las que hicimos con los Prefabs) y que cuando detecte un trigger destruya al objeto al cual está conectada la propiedad (si la propiedad se llama “objetivo”, para destruirlo usar `Destroy(objetivo)` en el evento `OnTriggerEnter`).*
- *Agregar el componente al trigger que está delante de la puerta y arrastrarle a la propiedad objetivo el objeto de la puerta.*
- *Pruebe la escena.*



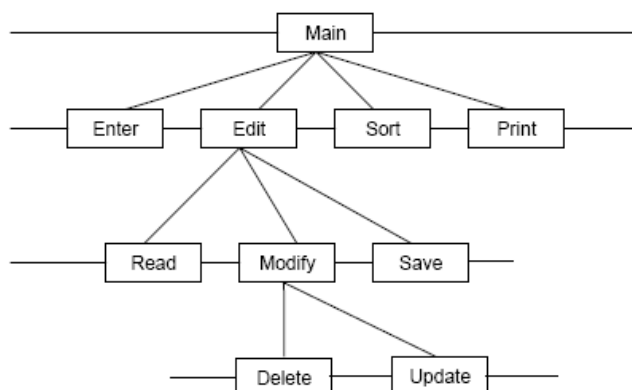
### Eventos, Instrucciones y Parámetros

*Hasta ahora hemos usado eventos e instrucciones como dos cosas diferentes, que en cierto aspecto lo son, pero en otros no. Veamos primero las cosas que sabemos de ambos y luego lo que no.*

*Sabemos que los eventos son un conjunto de instrucciones que, dependiendo de cuales, hacen que el evento haga una cosa u otra. O sea que el conjunto de instrucciones define lo que nosotros queremos que pase cuando se ejecute el evento. También sabemos que el evento al tener un nombre específico se ejecuta automáticamente en la situación asociada a dicho nombre (Update todos los frames, OnCollisionEnter en una colisión, etc.).*

*Las instrucciones son acciones que realizan una tarea específica (crear un objeto, destruirlo, mover objetos, etc.). También sabemos que, dependiendo de la instrucción, puede tener o no parámetros, o sea, datos que la instrucción necesita para poder ejecutarse. Por ejemplo, a la hora de ejecutar la instrucción de crear un objeto necesita saber que objeto va a instanciar, por lo que por parámetros se lo indicamos, o también la instrucción mover, que pide cuantas unidades va a mover. Esto hace que la instrucción pueda ejecutarse en diferentes lados con diferentes parámetros para ser usada en múltiples situaciones (como las propiedades de los componentes).*

*Ahora, lo que no sabemos de las instrucciones es que la mayoría dentro son un conjunto de otras instrucciones. Por ejemplo, la instrucción mover internamente le suma las unidades que les pasamos por parámetros a las posiciones correspondientes, o sea, la instrucción mover no es más que ejecutar tres veces la instrucción sumar a 3 propiedades diferentes (más adelante veremos cómo sumar nosotros). Otro ejemplo podría ser instanciar, que, al recibir un objeto por parámetro, internamente crea un objeto vacío, le agrega los componentes que tiene el objeto a clonar y le copia las propiedades, todo esto con otras instrucciones específicas para la situación.*



Si nos ponemos a unir las ideas, vemos que los eventos son similares, son un conjunto de instrucciones. Esto se debe a que en realidad los eventos son instrucciones que, en vez de tener que ejecutarlas a mano cuando nosotros lo deseamos, Unity las ejecuta automáticamente cuando es necesario. Los eventos al ser instrucciones que se ejecutan automáticamente comparten las mismas características que tendría una función, dentro de las cuales, la que nos interesa ahora es la capacidad de tener parámetros.

Ciertos eventos pueden recibir parámetros, o sea, datos asociados a la ejecución del evento. Vale la pena recalcar que esta vez nosotros estamos creando una instrucción, no ejecutándola, por lo que los parámetros se dice que lo estamos recibiendo, o sea, de algún lado alguien nos dio ese dato, sin importar de donde, vamos a trabajar con él. Por ejemplo, a la instrucción Translate no le interesa quien le dio los números, los toma y trabaja con ellos.

Podemos recibir parámetros en una instrucción poniéndolos entre paréntesis como la imagen debajo. Por ejemplo, si tuviésemos que crear nosotros la instrucción Translate, pondríamos dentro de los paréntesis, separados por coma, 3 parámetros. Fíjense que para crear un parámetro hay que poner primero el tipo de datos y después un nombre, al igual que las propiedades, pero sin el public. Los parámetros también son un contenedor de datos, que tienen dentro el dato que nos pasaron cuando ejecutaron la instrucción. Al ser también un dato las propiedades, podemos usarlos como tal, por ejemplo, pasarlo por parámetro a otra instrucción.

```

public void Translate(float x, float y, float z)
{
    //Codigo de translate
}
  
```

Veamos más sobre esto viendo cómo usar los parámetros de una colisión.

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)





### Parámetros de eventos de colisión

Anteriormente vimos los 6 eventos de colisión, pero hay algo que falta. Si bien sabemos que esos eventos se ejecutan en ambos objetos, en ningún momento cada objeto sabe contra quien choco, solamente sabe que choco. La mayoría de las veces con eso no nos alcanza, necesitamos saber contra que chocamos para poder, por ejemplo, sacarle o quitarle vida, subir una vida, agarrar un ítem, etc. Para ello podemos recibir por parámetros el objeto al que chocamos. Veamos primero como se hace con los triggers, ya que con las colisiones es un poco diferente.

```
public void OnTriggerEnter(Collider c)
{
    //Esto imprime el nombre del objeto
    //que nos choco o chocamos
    print(c);
}
```

En la imagen vemos que un trigger recibe por parámetro un dato de tipo Collider. Nosotros vimos anteriormente que podemos usar números, booleanos, texto y objetos como propiedades. Los parámetros al ser también datos pueden también usar esos tipos como cualquier otro.

Ahora, acá vemos que estamos recibiendo un componente por parámetro, un collider. Al igual que el tipo de datos GameObject (objetos), el cual sirve como conexión con un objeto, y cuyo valor lo configuramos desde el editor, el tipo de datos Collider sirve como conexión hacia un collider de cualquier tipo. La diferencia acá es que el valor no lo configuramos desde el editor, al ser un parámetro el valor viene de algún lado, particularmente lo da el que ejecuta esta instrucción. Al ser esto un evento el dato lo pone Unity, ya que la instrucción la ejecuta Unity automáticamente, por lo que no tenemos que preocuparnos de donde viene.

Vale la pena aclarar que los eventos no pueden recibir cualquier cosa por parámetro, solamente lo que sabemos que ese evento puede recibir, por ejemplo, Awake no puede recibir nada, por eso no tiene nada entre paréntesis. En el caso de los 3 eventos de trigger, pueden recibir un parámetro de tipo collider, como también ningún parámetro, porque así Unity lo dispone.

Por ultimo vemos que le pusimos como nombre "c" (por collider), pero como las propiedades, el nombre puede ser cualquiera que queramos, a Unity solo le interesa el

*tipo de datos y el orden de los parámetros (en Translate son 3 floats, x y z respectivamente)*

*En las colisiones cambia el tipo de datos del parámetro, en este caso recibe un Collision, que dentro, aparte de tener el collider al que chocamos, tiene también otra información de la colisión que no tendríamos en el caso de los triggers. Mas sobre esto adelante.*

```
public void OnCollisionEnter(Collision c)
{
    print(c);
}
```

## Comunicación e Interacción

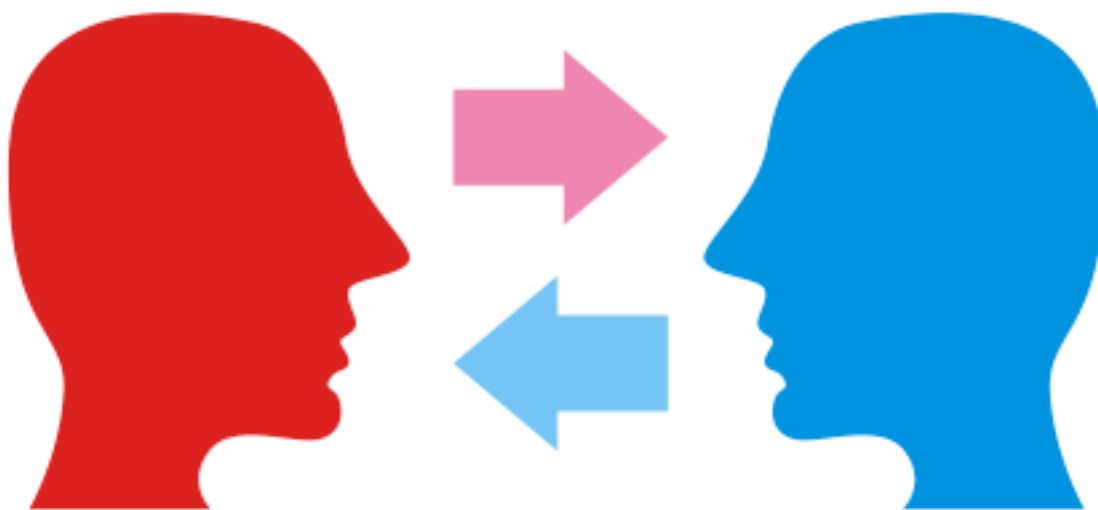
**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148  
[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)



**UTN.BA**  
UNIVERSIDAD TECNOLÓGICA NACIONAL  
FACULTAD REGIONAL BUENOS AIRES

**Centro de  
e-Learning**



## **Comunicación entre componentes**

*Acabamos de ver cómo detectar colisiones y acceder al objeto que nos chocó, o por lo menos a una parte de él (el collider nada más). Ahora, el paso que sigue sería comenzar a interactuar con él, destruirlo, quitarle vida, etc. Al igual que con cualquier otro componente, se pueden usar una serie de instrucciones para poder comunicarnos con el objeto al cual nuestra propiedad o parámetro está conectado, y así acceder a sus propiedades y acciones para interactuar con él.*

### **Acceso a propiedades de componentes**

*Sabemos que al crear un componente propio podemos comenzar a agregarle propiedades para poder variar su funcionamiento y agregarle información relacionada. Por*

**Centro de e-Learning SCEU UTN - BA.**

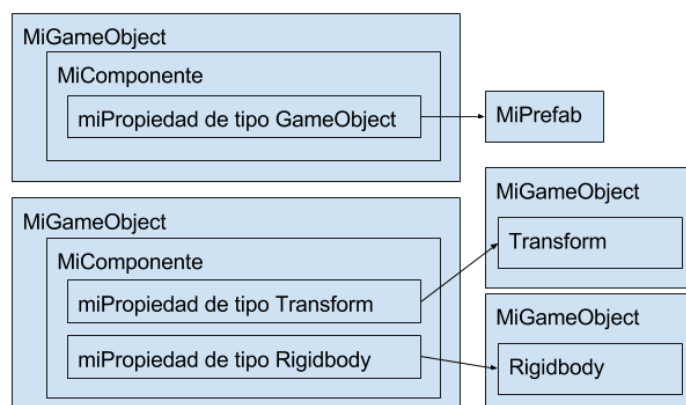
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

**[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)**

ejemplo, velocidades, objetos a copiar, etc. Otro caso sería un componente que maneje la vida, al cual le agregaríamos una propiedad numérica llamada “vida”.

También vimos que las propiedades que creamos se le suman a un repertorio de propiedades que todo componente tiene, como “gameObject”, que representa el objeto sobre el cual está puesto el componente, o también “transform.position” que representa nuestra posición.

Por último, vimos que podemos crear propiedades y parámetros de tipo componente, con esto me refiero a propiedades de tipo “Collider”, o “Rigidbody”, o también “Transform”, la que queramos. O sea, que si creamos una variable cuyo tipo se llame igual al nombre de un componente, sabemos que esa variable se puede conectar con un componente de dicho tipo, como hicimos con los Prefabs, o como acabamos de hacer con el parámetro Collider en el caso de los triggers. Poner el nombre de la propiedad o parámetro es lo mismo que referirse al objeto al cual está conectado, en el ejemplo del Trigger poner “c” (el nombre que el puse al parámetro) es lo mismo que decir el objeto al que chocamos (o el collider).

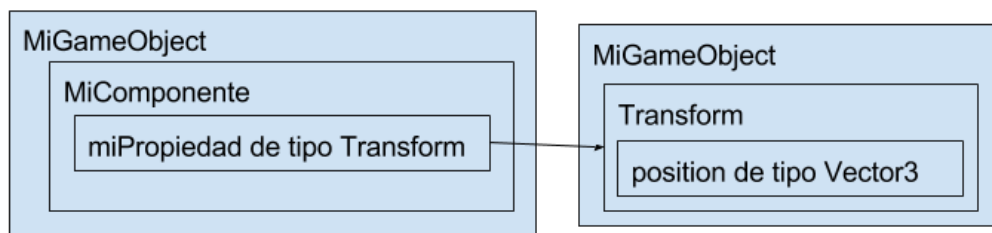


Ahora, nosotros si tenemos en cuenta que nuestra propiedad se conecta a otro objeto, que a su vez ese objeto tiene más propiedades, podemos darnos cuenta que debe haber alguna forma de acceder a las propiedades del objeto al cual está conectada la propiedad. Dicha forma se conoce como “sintaxis de punto”.

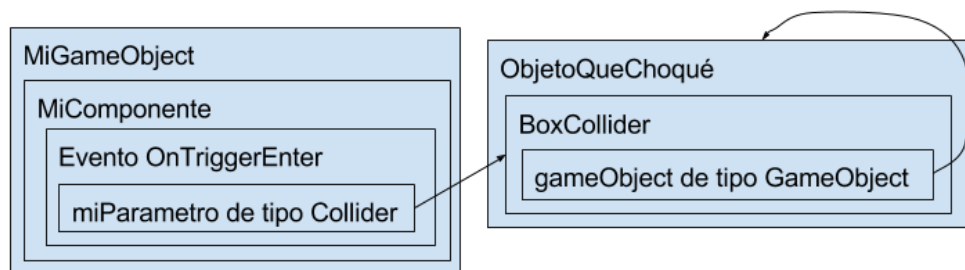
La sintaxis de punto es la forma en la cual usando un punto (“.”) podemos acceder a las propiedades de un objeto conectado a una propiedad. De hecho, ya hemos usado la sintaxis de punto sin darnos cuenta, cuando hacemos “transform.Translate”, o “transform.position” o “transform.rotation”. En el caso de “transform.position”, lo que está realmente sucediendo es que todo componente tiene la propiedad “transform”, que la misma es de tipo “Transform”, o sea, propiedad está conectada a un Transform,



particularmente al nuestro en el caso de esa propiedad. Entonces, al usar la sintaxis de punto, poniendo “.position” estamos accediendo a la posición del transform al cual está conectada la propiedad “transform”. Vale la pena destacar que “transform”, con *t* minúscula, se refiere a la propiedad transform de los componentes, cuando “Transform”, con *T* mayúscula, se refiere al tipo de datos.



Ahora, en el caso de los triggers, recibimos por parámetros una variable de tipo Collider, o sea, un componente de colisión (cualquiera de ellos). Más allá de ser específicamente un collider, es un componente, por lo que podemos acceder a las propiedades que tienen en común todos los componentes. Por ejemplo, tenemos la propiedad “gameObject”, que apunta al objeto sobre el cual está puesto el componente. Si leemos la propiedad “gameObject” del collider que recibimos por parámetros estamos accediendo al objeto al cual chocamos, en vez de solamente al collider.



Una vez que accedimos al objeto entero que chocamos en vez del componente, por ejemplo, podemos destruirlo con “Destroy”, así destruyendo al objeto entero, porque si en vez de poner “c.gameObject” poníamos solamente “c”, en ese caso estaríamos destruyendo únicamente al componente, o sea, le estaríamos quitando el componente.

```
public void OnTriggerEnter(Collider c)
{
    Destroy(c.gameObject);
}
```



**UTN.BA**  
UNIVERSIDAD TECNOLÓGICA NACIONAL  
FACULTAD REGIONAL BUENOS AIRES

**Centro de  
e-Learning**



### **Ejercicio N°3**

- *Modificar el componente de las balas del ejercicio anterior para que aparte de destruirse a sí mismo, destruya también al objeto al que choco (agregar la instrucción `Destroy(c.gameObject)` agregándole el parámetro “c” de tipo `Collider` al evento `OnCollisionEnter`, quedando dos `Destroy` a diferentes objetos).*
- *Crear componente que cuando detecte una colisión con otro objeto, acceda a la propiedad `transform` del objeto al que choco usando la sintaxis de punto y otra vez usando dicha sintaxis ejecute el `Translate` (`c.transform.Translate(0,0,10)`).*
- *Crear componente que cuando detecte una colisión con otro objeto, acceda a la propiedad `transform` del objeto al que choco usando la sintaxis de punto y le sume a la propiedad `position` la propiedad `forward` del mismo `transform`.  
(`c.gameObject.transform.position`      =      `c.gameObject.transform.position`      +*

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

**[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)**

*c.gameObject.transform.forward). Con esto estaríamos moviendo al objeto una unidad para donde está mirando.*

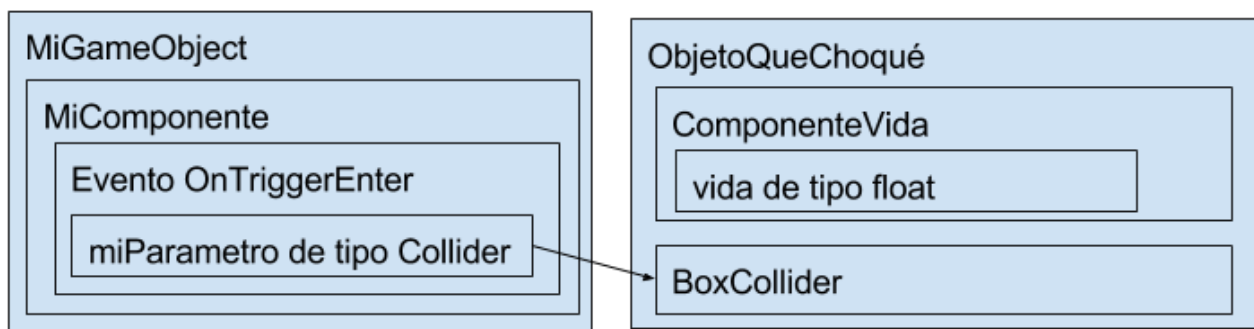
- *Probar los componentes poniéndolos en el jugador y chocando cosas.*
- *Crear componente que tenga un parámetro de tipo GameObject llamado "objetivo". Hacer que cuando detecte colisión ejecute el Translate del objetivo (objetivo.transform.Translate(0,0,10). Ponérselo al trigger que está delante de la puerta y arrastrarle la puerta.*

### Acceso a otros componentes

*Siguiendo el ejemplo del disparo, acabamos de ver efectivamente como destruir al objeto que chocamos, simplemente ejecutando la instrucción Destroy y pasándole por parámetros la propiedad gameObject del parámetro "c", siendo "c" el collider al que chocamos, y la propiedad "gameObject" el objeto del collider. Ahora, supongamos que tenemos un tipo de bala que no destruye objetos de un solo disparo, sino que tenemos que disparar varias veces para poder destruir al enemigo. El código escrito hasta el momento entonces no nos sirve, ya que el mismo destruiría inmediatamente.*

*Para ello necesitaríamos que, en vez de destruir al objeto chocado, quitarle vida progresivamente. Ningún objeto viene con la propiedad vida, para agregársela necesitaríamos agregarle un componente que tenga una propiedad numérica que represente la vida del objeto.*





*Aparte de esto tenemos la bala que creamos previamente, la cual tiene un componente que detecta la colisión. Anteriormente este componente destruía al objeto, ahora necesitamos que acceda al componente que maneja la vida del objeto chocado. Pero el problema está en que nosotros solamente tenemos acceso al collider del objeto al que chocamos, y cuanto mucho al objeto entero. Para acceder a un componente específico de un objeto tenemos la instrucción “GetComponent”. GetComponent es una instrucción que permite acceder a un componente que está en un objeto, si es que este lo tiene.*

```

public void OnTriggerEnter(Collider c)
{
    OtroComponente otro = c.gameObject.GetComponent<OtroComponente>();
}
  
```

*Analicemos toda la línea, ya que hace muchas cosas.*

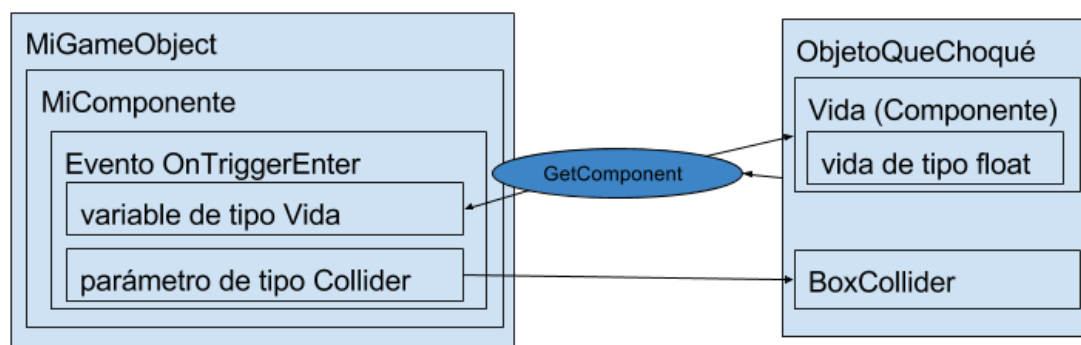
*En principio lo primero que nos llamaría la atención es la parte que está a la izquierda del símbolo “=”. Hay dos partes, primero una que dice “OtroComponente” y la otra que dice “otro”. En la primera parte sabemos que es un tipo de datos, ya el propio editor de código lo resalta. El tipo de datos “OtroComponente” es un tipo de datos que representa a un componente que hice llamado así. A la derecha de ese tipo aparece la palabra “otro”, está siendo un nombre. Vemos que esa sintaxis ya la hemos visto anteriormente, en el caso de las propiedades y parámetros, donde iba un tipo de datos y un nombre uno segundo del otro. ¿Qué estamos creando? ¿Una propiedad, un parámetro? Ninguna, a esto se le llama variable local.*

*Las variables locales son similares a las propiedades y parámetros. Sabemos que las propiedades son datos pertenecientes al componente, y que los parámetros son datos que recibimos en la función. Ahora, las variables locales también son datos, y*



generalmente se las usa como un contenedor de datos temporal, un dato que solo va a servir dentro del evento o función, por ejemplo, el resultado parcial de una operación matemática o también, en este caso, un componente que sacamos de otro objeto.

La idea acá es utilizar la variable local para guardar en su interior el resultado de la función `GetComponent`, o sea, en esa variable guardamos el componente que encontramos, para luego operar sobre él. Para ello primero fue crear una variable local poniendo su tipo y nombre, y luego le ponemos dentro (con el operador "=") el resultado de la función `GetComponent` al estar esta función a la derecha de una variable. Esto nos indica que hay ciertas instrucciones que dan un resultado, como si se tratasen de una fórmula o función matemática. Recuerden la fórmula  $y = f(x)$  (se lee  $f$  de  $x$ ), donde " $x$ " es un parámetro de la función " $f$ ", e " $y$ " es una variable donde guardamos el resultado de " $f$ ".



En resumen, lo que hicimos fue acceder a un componente de otro objeto para poder modificarlo. Para ello usamos la instrucción `GetComponent` sobre un objeto para pedirle por un componente y lo guardamos en una variable temporal para su posterior uso.

### Operadores para modificación de propiedades

Ahora, ya que guardamos el componente que le pedimos al objeto que chocamos, podemos usar la sintaxis de punto para poder acceder a sus propiedades en caso de querer modificarlas. Por ejemplo, en el caso de que chocamos a un objeto y somos una bala, vamos a querer quitarle vida. En ese caso, una vez que accedimos al componente "Vida" del objeto que chocamos, usamos la sintaxis de punto para restarle vida. Lo que nos falta saber es cuál es la sintaxis para modificar el valor de una variable.

```
public void OnTriggerEnter(Collider c)
{
    Vida vidaDelObjeto = c.gameObject.GetComponent<Vida>();
    vidaDelObjeto.cantidad = vidaDelObjeto.cantidad - 10;
}
```

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148  
[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)

Como vemos en la imagen, la segunda línea hace lo que queremos, analicémosla. Primero tenemos “vidaDelObjeto.cantidad”, siendo “vidaDelObjeto” una variable local que guarda en su interior el componente vida del objeto que chocamos, y “cantidad” una propiedad numérica que tiene el componente vida, la cual indica la cantidad de vida que tiene el objeto.

Luego el operador “=” el cual sirve para cambiar el valor, fíjense como el mismo operador sirve tanto para variables locales como para propiedades. Luego del “=” tenemos nuevamente “vidaDelObjeto.cantidad”, o sea, otra vez nos referimos a la cantidad de vida que tiene el objeto que chocamos, pero esta vez está el operador de resta seguido del número 10. El operador “-” claramente indica resta. Por ende, el poner “vidaDelObject.cantidad – 10”, el resultado de esa operación se guardaría en la misma variable cantidad, así cambiando su valor, por su mismo valor menos 10.

En resumen, lo que hicimos fue tomar el valor de la cantidad de vida del objeto, restarle 10, y el resultado de eso guardarlo en la misma propiedad cantidad, reemplazando su valor por su mismo valor menos 10.

Recordemos que hicimos algo similar con Time.deltaTime en la unidad 1, pero el resultado de la operación lo usamos como parámetro de Translate. De esa forma nosotros podemos modificar el valor de una propiedad, simplemente igualando su valor por su propio valor modificado por una instrucción.

Por ejemplo, si hubiésemos querido sumarle vida en vez de restarle, hubiésemos hecho esto:

```
public void OnTriggerEnter(Collider c)
{
    Vida vidaDelObjeto = c.gameObject.GetComponent<Vida>();
    vidaDelObjeto.cantidad = vidaDelObjeto.cantidad + 10;
}
```

O si hubiésemos querido dividir la vida por 10:



```
public void OnTriggerEnter(Collider c)
{
    Vida vidaDelObjeto = c.gameObject.GetComponent<Vida>();
    vidaDelObjeto.cantidad = vidaDelObjeto.cantidad / 10;
}
```

*Si hubiésemos querido multiplicar su vida por 10:*

```
public void OnTriggerEnter(Collider c)
{
    Vida vidaDelObjeto = c.gameObject.GetComponent<Vida>();
    vidaDelObjeto.cantidad = vidaDelObjeto.cantidad * 10;
}
```

*Si hubiésemos querido quitarle toda su vida:*

```
public void OnTriggerEnter(Collider c)
{
    Vida vidaDelObjeto = c.gameObject.GetComponent<Vida>();
    vidaDelObjeto.cantidad = 0;
}
```

*Por último, si hubiésemos querido multiplicarle la vida por 2 y sumarle 10*

```
public void OnTriggerEnter(Collider c)
{
    Vida vidaDelObjeto = c.gameObject.GetComponent<Vida>();
    vidaDelObjeto.cantidad = vidaDelObjeto.cantidad * 2 + 10;
}
```

*Como vemos, tenemos todos los operadores matemáticos básicos como suma (+), resta (-), multiplicación (\*) y división (/), y podemos combinarlos para hacer formulas complejas que no solo usen números fijos, sino también variables locales, propiedades y parámetros, ya que ellos son datos.*

*Como siempre vale la pena recordar que, si bien todas las instrucciones aprendidas hasta el momento las hemos utilizado con el único propósito de quitarle vida a un objeto, al igual que todo lo que vamos a aprender a lo largo del curso, sirve para múltiples propósitos. Por ejemplo, podríamos hacer esto para quitarle vida al objeto que chocamos, o también sumarle vida, o subirle la velocidad si es un Powerup de velocidad, o contar puntaje porque agarramos una moneda, etc.*

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148  
[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)

*Recordemos que programar involucra el uso creativo de las herramientas aprendidas más allá de como las aprendimos, y eso lo aprendemos cada uno empezando a encontrarnos con problemas en nuestro camino a terminar el juego.*



#### **Ejercicio N°4**

- *Crear un componente que cuando detecte una collision con otro objeto, acceda al Rigidbody (`Rigidbody rb = c.gameObject.GetComponent<Rigidbody>()`) y que ejecute la función `AddRelativeForce` del Rigidbody pasándole 100 en el eje Z*

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148  
[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)



*(rb.AddRelativeForce(0,0,100)). Agregarlo al personaje y probar la escena haciendo que el personaje choque contra la pelota con física realista.*

- *Crear componente que cuando detecte un trigger contra un objeto, acceda al componente encargado del movimiento del personaje (si el componente se llama "Movimiento", `Movimiento m = c.gameObject.GetComponent<Movimiento>()`), y sumarle 1 al movimiento (si tiene una propiedad velocidad, `m.velocidad = m.velocidad + 1`) y que finalmente destruya al objeto que chocó (`Destroy(c.gameObject)`). Agregarle el componente a una caja "Trigger Estática".*
- *Crear componente que le quite vida al objeto que choco (como el código arriba) y agregarlo a las balas.*
- *Crear componente que tenga una propiedad de tipo `GameObject` llamada objetivo que cuando detecte un trigger acceda al componente `Rigidbody` del objeto al que está conectado (`Rigidbody rb = objetivo.GetComponent<Rigidbody>()`) y le agregue fuerza a ese objeto (`rb.AddRelativeForce(0,0,100)`). Ponerlo a una caja "Estática Trigger" y arrastrarle la pelota como objetivo.*

### Comparaciones y Condiciones

*Hasta ahora hemos podido detectar cuando dos objetos colisionan y también hemos hecho que un objeto le quite vida al otro, por lo que estamos encaminados para el último paso, que el objeto se destruya. Para ello vamos a necesitar saber si en algún momento la propiedad numérica que indica la cantidad de vida restante llegó a cero. Para ello necesitamos aprender a comparar.*

*Para comparar vamos a necesitar los operadores de comparación. Vimos que existen operadores matemáticos (suma, resta, etc.) que sirven para crear fórmulas matemáticas*

**Centro de e-Learning SCEU UTN - BA.**

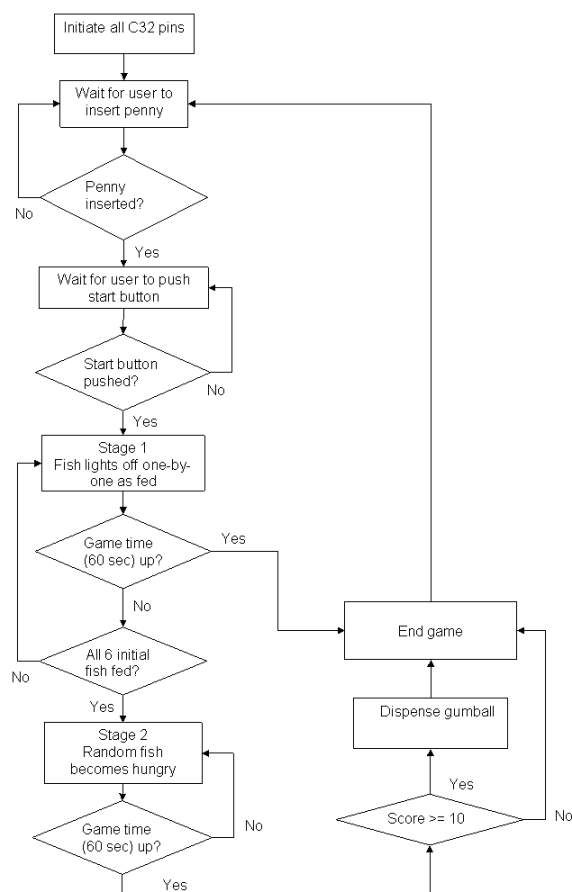
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)



tanto simples (ej. multiplicar algo por dos), como complejas (ej. multiplicar por la suma de una variable más la mitad de otra). Hay otros operadores como ellos pero que, en vez de servir para hacer fórmulas matemáticas, sirven para hacer lo que personalmente me gusta llamar formulas booleanas, comúnmente conocidas como condiciones.

Las condiciones, como su nombre lo indica, son situaciones que contemplamos. Si un conjunto de situaciones se da, la condición es cierta, si no, es falsa. Por ejemplo, si la vida del personaje llegó a cero, entonces es cierto que murió, sino no, o también, si la cantidad de balas es mayor que cero, es cierto que podemos disparar, o si el tiempo que nos queda para terminar el nivel llegó a 0, es cierto que perdimos. Si bien hemos visto todos ejemplos con números, pueden llegar a necesitarse otras condiciones, como, por ejemplo, si el jugador se llama de una manera determinada, prohibir su entrada al juego, a modo de bloqueo. Más allá de la situación determinada, como vemos, todas son situaciones comunes que pueden suceder en un juego, y vamos a necesitar contemplarlas de alguna forma.



Nosotros anteriormente hemos visto cómo detectar si se presionó una tecla usando la instrucción `GetButton`. Si recuerdan, `GetButton` era una instrucción que, al igual que una suma o `GetComponent`, devolvía un resultado de un tipo determinado. `GetButton` se transformaba en un valor de tipo `bool` o booleano, el cual, justamente, sirve para determinar verdades. Recordemos que los booleanos son tipos de datos que solamente pueden guardar en su interior los valores "true" o "false".

Al usar `GetButton` dentro de un `if`, estábamos chequeando si se cumplía una condición, está siendo que, si el usuario presiona una tecla, pase algo. Particularmente lo que vaya a pasar por ahora no es lo importante, lo importante es que fuimos capaces de detectar que se haya cumplido la condición. Nosotros disponemos de otras instrucciones similares, que



reciben parámetros y devuelven booleanos, y las mismas se llaman operadores booleanos.

### Operadores Booleanos

Recordemos que los booleanos sirven para definir verdades. Un operador booleano es algo que sirve para determinar una verdad, o sea, una condición, a partir de la comparación de valores. Por ejemplo, si queremos saber si el jugador perdió, deberíamos preguntar si la vida llegó a ser 0, o sea, si es menor o igual a 0, o también, si queremos saber si podemos tirar un poder, tendríamos que preguntar si tenemos energía para ello, o sea, preguntar si la energía del personaje es mayor o igual a la cantidad de energía necesaria para tirar la skill o, por último, si quisiésemos disparar, antes tenemos que saber si nos quedan balas, o sea, si la cantidad de balas es mayor a 0. Veamos algunos ejemplos de operadores booleanos.

```
public class Vida : MonoBehaviour
{
    public float cantidad;

    public void Update()
    {
        if(cantidad < 0)
        {
            Destroy(gameObject);
        }
    }
}
```

En el ejemplo anterior vimos una posible forma de como programar el componente que administre la vida. Nosotros anteriormente vimos que podíamos quitarle vida al objeto obteniendo este componente y restándole vida. Ahora, lo único que hicimos fue hacer que un número decremente su valor, nunca hicimos que el objeto realmente se destruya.

En este ejemplo hemos usado un if dentro de un Update, lo cual implica que todos los frames nuestro componente pregunta algo. Dentro de los paréntesis del if, donde va la condición, pusimos el nombre de la propiedad cantidad seguida del operador "<", el cual significa "menos que", y luego un 0. Entonces, lo que estamos haciendo es preguntando todo el tiempo si es que la vida llegó a ser menos que 0, si es que eso sucede, se ejecuta el Destroy interno.

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148  
[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)



Lo que hace el operador "<" es comparar los valores que tiene a sus lados, específicamente comparando si el que esta su izquierda (en este caso, el valor de la propiedad cantidad) es menor que el que tiene a su derecha (cero esta vez). Si dicha condición es cierta, el operador se transforma en "true", sino en "false". El resultado del operador esta como parámetro del if, por lo que si el resultado es "true" se ejecuta el código entre las llaves del if, sino no. De esa forma se lee "si la cantidad es menor a 0, entonces destruirme". Fijense que esto lo podríamos haber hecho también en el componente que maneja la bala, siendo en este caso igual de válido, pero si tendríamos otro lugar que también quitase vida, por ejemplo, una bomba, tendríamos que volver a poner el mismo código. De esta forma evitamos tener que preocuparnos de que en cada parte donde quitamos vida de comprobar esto. Veamos otro ejemplo.

```
public class Disparador : MonoBehaviour
{
    public int balas;
    public GameObject prefab;

    public void Update()
    {
        if (Input.GetButtonDown("Disparar"))
        {
            if (balas > 0)
            {
                Instantiate(prefab, transform.position, transform.rotation);
                balas = balas - 1;
            }
        }
    }
}
```

Este es un caso similar, pero con una serie de diferencias. Primero veamos que tenemos una propiedad de tipo "int". Int es parecido a float, sirve para guardar números, la diferencia es que float puede guardar decimales (números con coma) e Int no, solamente números enteros (sin coma). En este caso hemos usado int ya que no podemos tener 2.5 balas, o 0.3 balas, o tenemos 0, o tenemos 1, o tenemos 2, pero ningún intermedio. En vez de usar un tipo de datos, que, si bien nos hubiese servido, permite poner un valor invalido, es preferible usar un tipo de datos que nos permite hacer lo que necesitamos, ni más ni menos.

La otra diferencia es que en vez de usar el operador "<", usamos ">", que claramente es el opuesto, o sea, ">" significa "mayor que". En este caso estamos comparando si el valor de

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)



nuestra propiedad balas es mayor a 0, en tal caso efectivamente estamos habilitados a disparar.

Otro detalle, y el más importante, es que pusimos un IF adentro de otro IF, lo que conlleva a que ambas condiciones tengan que ser ciertas. Por ejemplo, si presionamos la tecla de disparo efectivamente el primer IF es exitoso por lo que procede a ejecutar el código en su interior, pero lo primero que se encuentra es con otro if. En este caso procedería a ejecutar el segundo if, imponiendo la segunda condición, en este caso, si las balas son mayores a 0. El hecho de que un if esté adentro del otro, significa que el código que está adentro de todo (el de instanciar la bala) se ejecute solamente si ambas condiciones se dan. Por ejemplo, si tengo balas, el if de adentro es exitoso, pero como el de afuera no, ya que no apretamos el botón, nunca se llegó a ejecutar el if de las balas.

Esto se conoce como IFs anidados, o sea, un if dentro de otro. Si alguno tiene experiencia en programación sabe que hay una solución más adecuada para este problema, pero veremos más de ello en la siguiente unidad.

Un detalle extra es que podríamos haber usado el “>=”, que significa “mayor o igual”, y el “<=” que significa “menor o igual”. Fijense que en el caso de la vida preguntamos si la vida es menor a 0, pero si la vida es 0 el personaje queda vivo. Si usáramos el menor o igual, si la vida del personaje llega a 0 o es menor a cero muere, ya que, por ejemplo, si tenía 5 de vida y recibió 5 de daño, quedaba en 0, pero si tenía 5 de vida, y recibió 10 de daño, el daño quedaba en -5. En ambas situaciones debería tener que ejecutarse.



### Ejercicio N°5

- Agregar al componente “Vida” un evento Update que chequee si la vida llegó a ser menor o igual a 0 (<=), y en tal caso autodestruirse.

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)



- *Modificar el componente que sube la velocidad al objeto para que solamente suba la velocidad si la velocidad actual es menor a 5 (if(m.velocidad < 5)).*
- *Agregar el personaje el componente de disparo hecho la clase pasada agregándole la propiedad de tipo int llamada balas y comprobando aparte de si se presionó la tecla de disparo que queden balas disponibles (como el código arriba).*
- *Modificar el componente Vida agregándole la propiedad escudo de tipo float. Hacer que el componente que quita vida (Vida v = c.gameObject.GetComponent<Vida>();) pregunte si tiene escudo (if(v.escudo > 0)), en tal caso que le quite el daño al escudo (v.escudo = v.escudo - 5). En caso de que no tenga escudo (if(v.escudo <= 0)) quitarle a cantidad como hacía antes. Debería haber dos IF en el evento.*

### Filtrar colisiones

Los operadores vistos hasta ahora sirven para comparar si algo es mayor o menor a otra cosa. Ahora, hay ciertas variables que no se pueden comparar de esa forma. Por ejemplo, ¿Puede un booleano ser mayor o menor que otro? El criterio en ese caso es un poco difuso. O, por ejemplo, ¿Puede un GameObject ser mayor que otro? Esto no implica que esas variables no se puedan comparar, existen los operadores de igualdad.

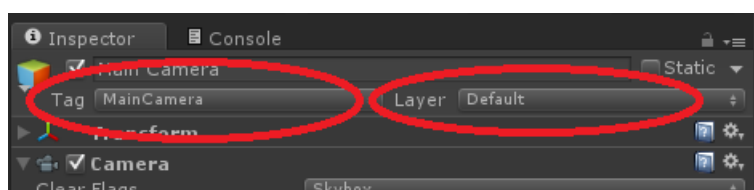
```
if(c.gameObject.name == "Jugador")
{
    print("Choque a un objeto con el nombre 'Jugador'");
}
```

En el ejemplo usamos el operador "==" (igual) para comprobar si chocamos contra un objeto llamado de determinada forma. Usamos la sintaxis de punto para acceder al nombre del objeto que chocamos y compararla contra un valor.

```
if(c.gameObject.name != "Jugador")
{
    print("Choque a un objeto que no es el 'Jugador'");
}
```

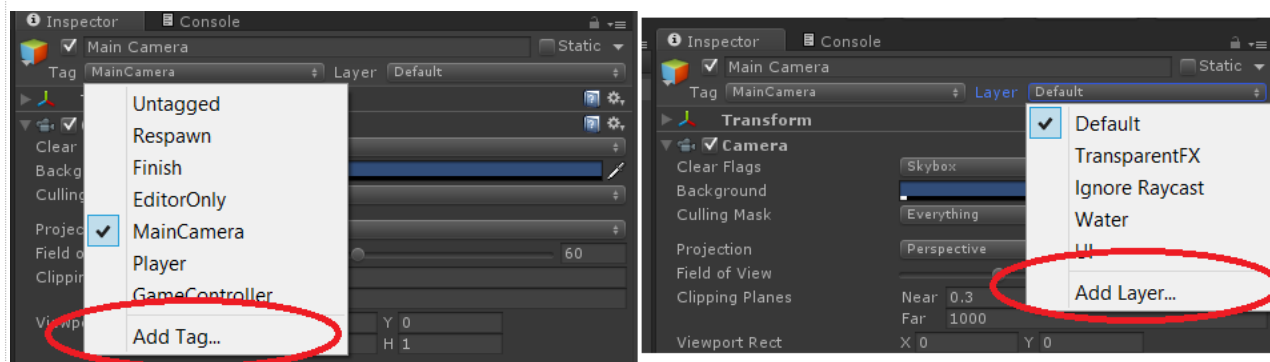
En este otro ejemplo hicimos lo contrario. El operador “!=” (no igual) sirve para comprobar si dos valores son diferentes. En este caso comprobamos que no hemos chocado contra el jugador.

Podemos filtrar las colisiones preguntando por el nombre del objeto que chocamos, pero hay un problema. Si nosotros creamos un enemigo con *Instantiate*, el mismo va a llamarse “Enemigo (Clone)”. Si bien podríamos preguntar si chocamos contra un “Enemigo (Clone)”, no es algo muy prolijo, para ello existen los layers y los tags.



Los layers y los tags son dos formas diferentes que tiene Unity de categorizar objetos. Si bien para diferenciar una colisión da lo mismo preguntar por cualquiera de ellos, el nombre es algo que puede llegar a variar mucho. Si tuviésemos dos tipos de enemigos, ambos tendrían diferente nombre. En tal caso podríamos ponerles a ambos objetos el mismo tag o layer, por ejemplo “Enemigo”, de esa forma preguntamos por todos los enemigos, independientemente de su tipo.

Para ponerle un tag o un layer a un objeto (o prefab) tenemos en la cabecera del inspector (la parte superior) dos propiedades para ello. En ambos casos cuando hacemos clic se nos despliega un menú de opciones donde se nos muestran los tags o layers que vienen por defecto en Unity, y debajo una opción para agregar más.



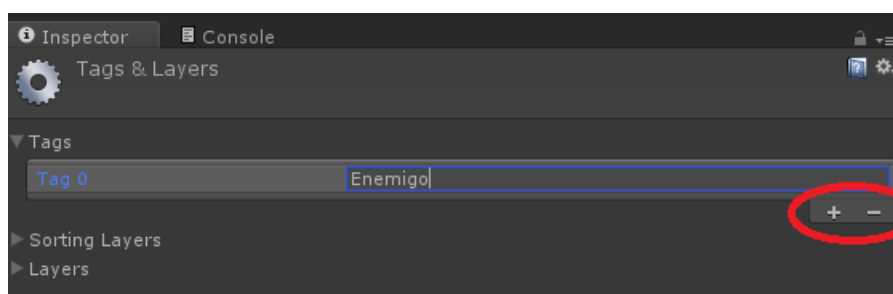
**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

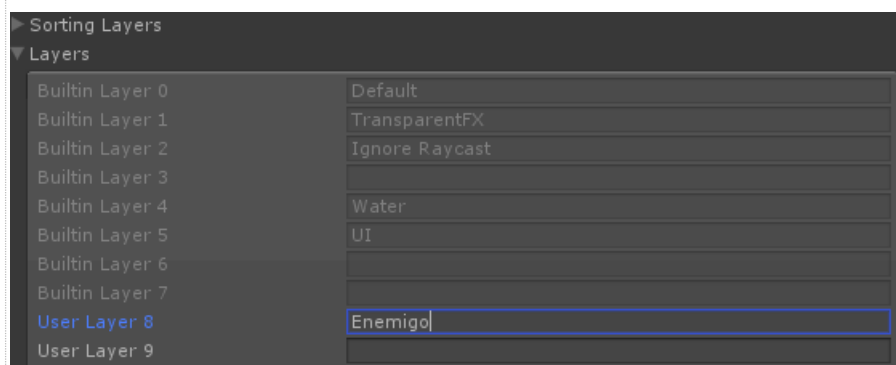
[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)

Dependiendo si queremos agregar uno o el otro, aparecen en el inspector diferentes menús para configurarlos, menú al cual podríamos acceder también yendo a Edit -> Project Settings -> Tags and Layers.

En el caso de los Tags aparece una lista vacía. Haciendo clic en el botón “+” podemos crear un nuevo tag y ponerle el nombre que queramos. Con el botón “-” eliminamos el tag seleccionado.



Para los layers es un poco diferente. Nos aparecen 32 casilleros pre-creados y no podemos agregar más ni quitar más. Para el que sabe programación, esto se debe a que un layer es en realidad un “bitmask”. Más allá de ese detalle, nosotros podemos llenar solamente del layer 9 en adelante, ya que Unity usa los primeros 8. Eso demuestra que un layer se representa con un número más allá de su nombre.

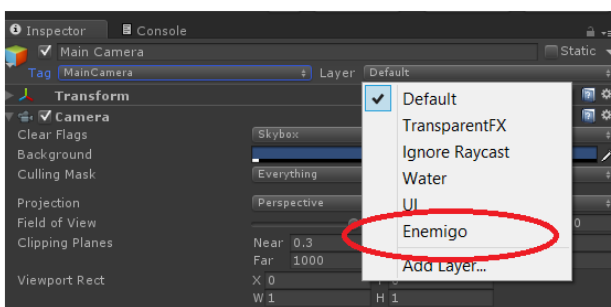
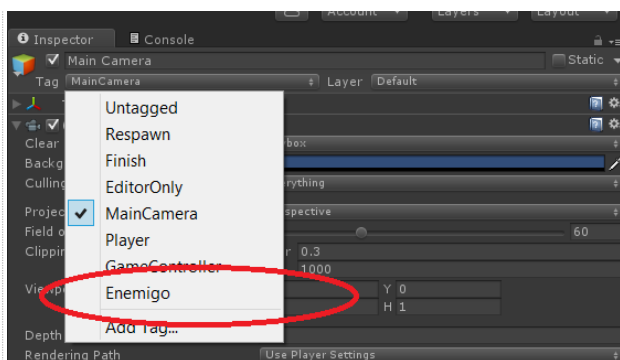


Una vez creado el tag o layer, simplemente podemos seleccionarlo desde las propiedades del objeto, ya que crearlo no significa que lo pusimos en el objeto.



**UTN.BA**  
UNIVERSIDAD TECNOLÓGICA NACIONAL  
FACULTAD REGIONAL BUENOS AIRES

**Centro de  
e-Learning**



De esta forma tenemos 3 formas de diferenciar objetos, con el nombre, el tag y el layer. Lo que no queda claro es exactamente la diferencia entre ellos, cuando se usa uno, y cuando el otro (los layers tienen un uso más específico en ciertas situaciones, pero lo veremos más adelante). Si bien no hay un estándar definido, se suele usar los Layers como grupo principal, los tags como subgrupo, y el nombre para diferenciar cada individuo. La realidad es que si queremos también podemos agregar un componente que le agregue alguna propiedad identificativa al objeto, como, por ejemplo, un componente con una propiedad numérica llamada "ID" para diferenciarlo por un número, como si fuese el Nro. de Documento. Esto depende del juego, no hay un general.



### Ejercicio N°6

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)

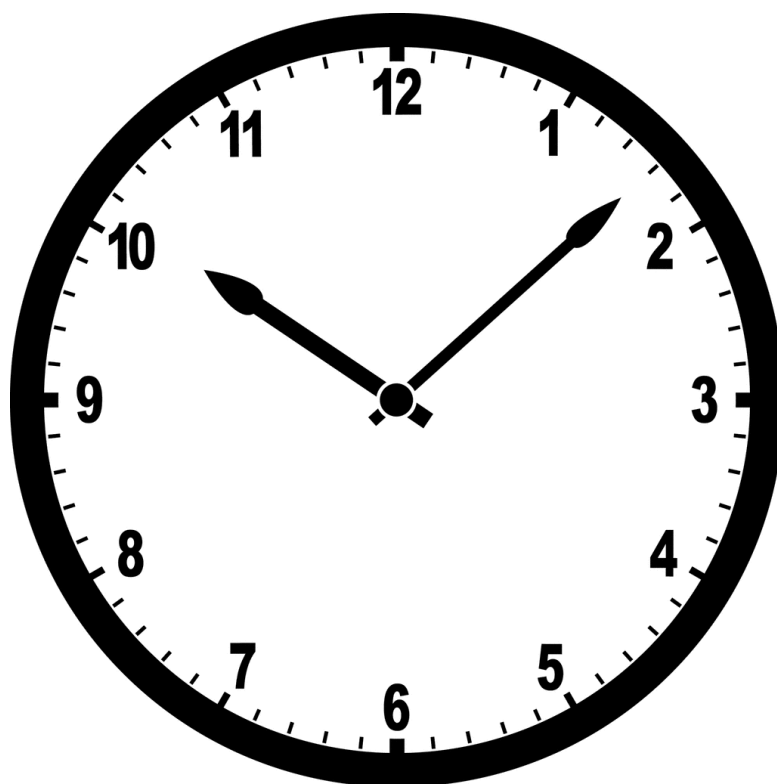
- *Hacer que las balas solamente choquen contra los enemigos preguntando por layer (si el layer de los enemigos es el número 8, `if(c.gameObject.layer == 8)`).*
- *Hacer que el componente que agrega fuerzas solamente empuje a objeto con el tag "Física" (`if(c.gameObject.tag == "Física")`).*
- *Hacer que el componente que sube la velocidad al personaje solamente le suba al objeto con el nombre "Jugador" (`if(c.gameObject.name == "Jugador")`).*

## Temporizadores



**UTN.BA**  
UNIVERSIDAD TECNOLÓGICA NACIONAL  
FACULTAD REGIONAL BUENOS AIRES

**Centro de  
e-Learning**



## *Midiendo el paso del tiempo*

*Todo no puede suceder instantáneamente en un juego, si el enemigo dispara balas, tienen que salir cada X cantidad de tiempo, porque sería muy difícil si saldrían todos los frames. Si hay un límite de tiempo para completar el nivel, tendríamos que saber cuándo pasó. Cuando perdemos tenemos que esperar a que termine una animación o sonido antes de reaparecer. Estas pausas servirán para dar un respiro al jugador, o para ajustar ciertas mecánicas, como la velocidad de disparo, o el tiempo que tiene que pasar antes de que empiece a recargar la vida. Unity tiene instrucciones que nos facilitan la tarea.*

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148  
[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)





### Retraso de ejecución

La primera instrucción se llama "Invoke". Esta instrucción lo que permite es pedirle a Unity que ejecute una instrucción luego de una cantidad X de segundos.

```
public void Awake()  
{  
    Invoke("Autodestruir", 2);  
}
```

En la imagen vemos que en el Awake estamos ejecutando la instrucción. Es importante aclarar que Unity comienza a contar el tiempo desde el momento que ejecutamos la instrucción. En este caso, al haber ejecutado la instrucción en el Awake, comenzamos a contar desde que sea crea el objeto. Analicemos los parámetros. El primero es un texto y seria el nombre de la instrucción que crearemos para que se ejecute (en este caso "Autodestruir"). El segundo parámetro es el tiempo que va a contar Unity antes de ejecutar la instrucción (en este caso 2). Entonces, el código de la imagen lo que hace es ejecutar la instrucción "Autodestruir" luego de 2 segundos desde la creación del objeto donde este el componente. Recordemos que cuando le damos play a la escena es cuando Unity crea los objetos.

Lo último que nos queda es crear la instrucción a ejecutar. Eso lo hacemos al igual que cuando creamos cualquier otra función o evento, pero poniéndole el nombre correspondiente (en este caso "void Autodestruir()").

```
public void Autodestruir()  
{  
    Destroy(gameObject);  
}
```



### **Ejercicio N°7**

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)



- *Agregar a las balas un script de autodestrucción luego de N cantidad de segundos (al igual que el código arriba, o sea, en el Awake haciendo Invoke de una función que ejecute el Destroy).*
- *Agregar a dicho script una propiedad de tipo float llamada "tiempo" que se use como segundo parámetro del Invoke (Invoke ("Autodestruir", tiempo)).*
- *Crear un script que cuando detecte la presiona de una tecla imprima el mensaje "Boom" luego de 2 segundos.*

### Repetición de ejecución

*En el caso de que deseemos no solo retrasar la ejecución de algo, sino también que se repita la ejecución cada cierto tiempo, tenemos la instrucción "InvokeRepeating". Como su nombre lo sugiere, es similar al Invoke, pero esta Repite.*

```
public void Update()
{
    if (Input.GetButtonDown("Disparar"))
    {
        InvokeRepeating("Disparo", 0, 0.2f);
    }
}
```

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148  
[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)



En este caso el código es un poco diferente. Lo que hacemos es en el Update es preguntar si se acabó de presionar una tecla (GetButtonDown) y en tal caso ejecutamos el InvokeRepeating. Esta función recibe 3 parámetros, los primeros dos significando lo mismo que en el Invoke, el primer parámetro es el nombre de la ejecución y el segundo el tiempo que va a pasar antes de que empiece a ejecutar. El tercer parámetro establece cuanto tiempo va a pasar entre repetición y repetición. En este caso pusimos 0 segundos hasta que empiece a repetir, por lo que va a empezar a repetir apenas apretamos la tecla, y luego 0.2 segundos entre repetición y repetición, lo que implica que desde que apretamos la tecla va a disparar 10 balas por segundo ( $10 \times 0.2 = 1$ ). De esta forma evitamos tener soltar y presionar la tecla de disparo cada vez que queramos disparar. Tengamos en cuenta que le pusimos "0.2f". La f adelante del decimal es necesaria.

### Acumular y Cancelar las ejecuciones

En el código anterior hay unos problemas. Primero, nosotros apretamos la tecla y comienza a disparar, pero si soltamos no deja de disparar, ya que nunca se lo indicamos a Unity, él nunca va a asumir nada por nosotros. Segundo, después de soltar, si volvemos a presionar, comienza a disparar más balas, ya que se vuelve a ejecutar el InvokeRepeating, acumulándose con el anterior. Esto se debe a que nunca paramos la ejecución del InvokeRepeating. Para ello necesitamos ejecutar la instrucción "CancelInvoke" cuando se suelta la tecla, usando el "GetButtonUp".

```
if (Input.GetButtonUp("Disparar"))  
{  
    CancelInvoke("Disparo");  
}
```



### **Ejercicio N°8**

- Modificar el componente de disparo para que repita el disparo cada X cantidad de segundos (usando el código de arriba).
- Crear componente que instancie Prefabs cada una x cantidad de segundos y agregárselo a los enemigos para que disparen.

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148  
[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)

- Usar ese mismo componente poniéndolo en un objeto vacío (*GameObject -> Create Empty*), pero que esta vez instancie un prefab de enemigos que se muevan hacia adelante.



## *Bibliografía utilizada y sugerida*

### *Recursos Online*

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

**[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)**

*Manual y referencia del componente Rigidbody*

<http://docs.unity3d.com/ScriptReference/Rigidbody.html>

<http://docs.unity3d.com/es/current/Manual/class-Rigidbody.html>

*Videotutoriales de la interfaz de Unity.*

<http://unity3d.com/es/learn/tutorials/topics/interface-essentials>

*Videotutoriales de programación de Unity.*

<http://unity3d.com/es/learn/tutorials/topics/scripting>

### **Libros y otros manuscritos**

Creighton, Ryan Henson. *Unity 3D Game Development by Example Beginner's Guide*. 1° Edición. UK. Packt Publishing. 2010.

## **Lo que vimos:**



*En esta unidad hemos visto:*

1. *Cómo configurar el sistema de física.*
2. *Cómo detectar colisiones.*
3. *Cómo filtrar las colisiones.*
4. *Cómo modificar a otros objetos.*
5. *Cómo temporizar acciones.*

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148  
[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)

## Lo que viene:

*En la siguiente unidad veremos cómo hacer para detectar cuándo se ganó o perdió el juego y cómo finalizar el prototipo.*



**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

**[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)**