UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES
Centro de e-Learning

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

Módulo 1: Prototipado

Unidad 4: Ganar y Perder

Presentación:

De a poco el proyecto se va pareciendo a un juego. Podemos movernos, disparar y los enemigos también, tenemos PowerUps y cualquier otra cosa extra que le hayan querido agregar al juego. Con todo lo que vimos hasta ahora podemos hacer mucho, pero al proyecto hay que hacerle mucho pulido. Una de las cosas más importantes que faltan es ganar y perder.

En ésta unidad veremos cómo hacer la Condición de Victoria y de Derrota, o sea, verificar si el juego se ganó o se perdió. También veremos un par de detalles extra que nos faltó. Recuerden que hasta ahora todo lo que hemos hecho es la lógica del juego, no hemos contemplado ningún aspecto estético. En el próximo modulo comenzaremos con el apartado estético, viendo gráficos, audio y UI.

Objetivos:

Que los participantes aprendan:

- A detectar cuando se gana y se pierde.
- A corregir detalles finales del juego.

Bloques temáticos:

- Comunicación e Interacción II.
- Finalizado de Prototipo.

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.

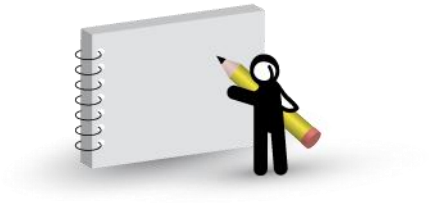
Los contextos de desempeño de los participantes.

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.***



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.

UTN.BA UNIVERSIDAD TECNOLÓGICA NACIONAL FACULTAD REGIONAL BUENOS AIRES | Centro de e-Learning

Comunicación e Interacción II

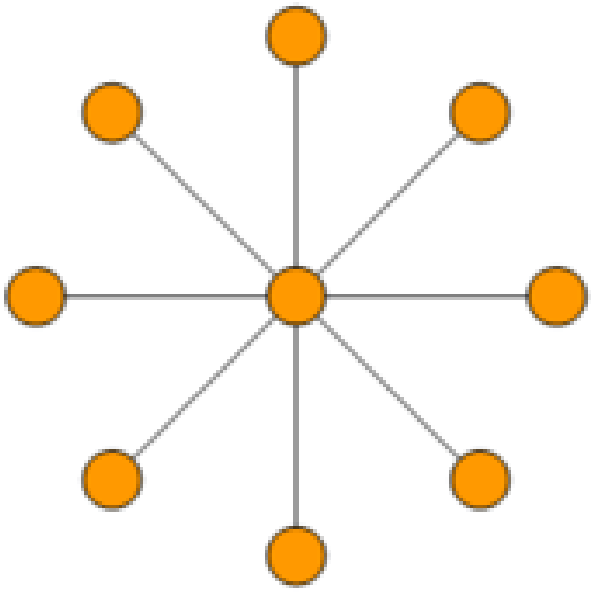
Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Managers

Un aviso antes de comenzar. En este bloque no hablaremos de alguna característica puntual de Unity, sino de una forma de utilizarlo. En programación hay muchas formas de resolver lo mismo, y veremos a continuación una forma posible de una situación, no siendo la única, ni la mejor, ni la peor, sino la que consideramos adecuada para el grado de conocimiento esperado en esta etapa.

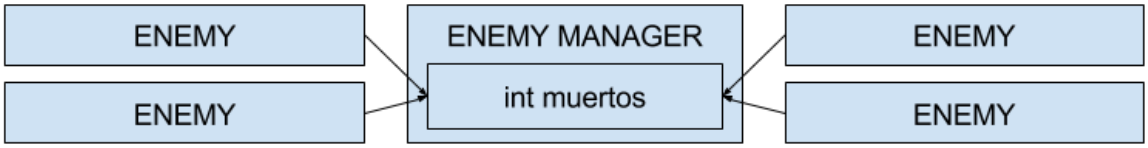
En la parte I de este bloque hablamos sobre cómo los componentes se comunican entre sí. Si tenemos acceso a un objeto, usando la instrucción GetComponent, podemos preguntarle por un componente y poder modificar sus diferentes aspectos. Esto es posible siempre y cuando tengamos de alguna forma acceso a un objeto, tanto como si hicimos una propiedad y le arrastramos el objeto, como si lo accedemos a través del objeto que chocamos. Ahora, si bien la mayoría de las situaciones se pueden resumir a eso, hay casos puntuales donde nosotros necesitamos acceder directamente a otro objeto por otros medios.

Hasta ahora tenemos muchos objetos comunicándose entre sí. Cada uno tiene sus datos, su vida, la cantidad de balas, etc. Hay aspectos del juego que requieren que haya un núcleo central donde se depositen datos en común entre todos los objetos, y que a veces tenga una cierta lógica sobre esos datos. Estos son los que normalmente se conoce como "Managers". Los Managers (Administradores) son objetos fácilmente accesibles y hay uno solo de cada tipo al mismo tiempo, ya que al ser el único punto de contacto entre diferentes objetos puede retener información de todos, y así todos acceder a los mismos datos compartidos.



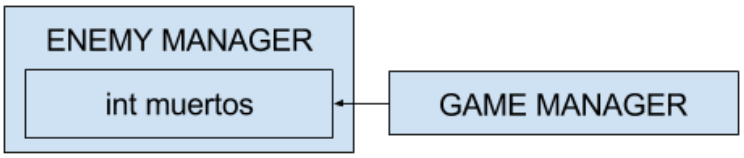
Managers de Juego

En el caso de los enemigos, éstos tendrían que compartir una propiedad que cuente la cantidad que han muerto. Si cada enemigo tuviera su propia propiedad para ello, no tendría sentido, ya que cada enemigo tendría una cuenta diferente, y no sólo ello, sino que cuando un enemigo muriera, la cuenta se iría con él. En este caso tendría sentido utilizar un manager que tuviera la cuenta de enemigos muertos. Se podría llamar "EnemyManager" por ejemplo.

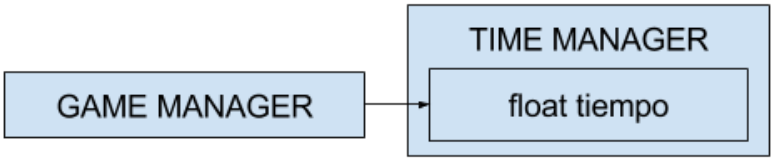


Nuestro juego va a requerir que en algún momento se pueda ganar o perder, para ello existe lo que normalmente se conoce como Condiciones de Victoria y Derrota. Como su nombre lo indica, son situaciones que se tienen que dar para que el juego se considere ganado o perdido. Dichas condiciones se pueden basar en información de diferentes fuentes.

Por ejemplo, tenemos un nivel donde matar 5 enemigos es una condición de victoria. Para éste caso, podríamos que hacer un "GameManager", encargado de leer información del "EnemyManager" y determinar si se ganó el juego.



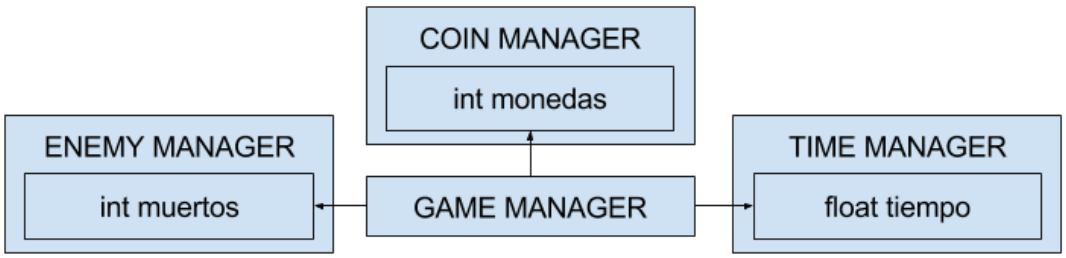
En otro nivel la condición de victoria podría ser aguantar 10 segundos vivo. Para ello podríamos tener un "TimeManager", que vaya contando la cantidad de segundos que pasaron desde que comenzamos el nivel. El GameManager en este caso podría chequear cuando llegaron a pasar 10 segundos, y así declarar la victoria del nivel.



Ustedes se preguntarán, ¿Y si en vez de chequear eso en el GameManager lo chequeo directamente en el EnemyManager o en el TimeManager? Veamos el siguiente caso.

Imaginemos un nivel donde matar 5 enemigos puede ser una condición de victoria, y también, que pasen 30 segundos puede ser una condición de derrota. Para éste caso el GameManager podría leer información del "EnemyManager" y del "TimeManager" y poder así determinar si se ganó o si se perdió.

Si nosotros teníamos los chequeos en cada manager, no podríamos haber logrado este tercer nivel, ya que, por ejemplo, lo que antes era una condición de victoria (el tiempo) se convirtió en una condición de derrota. Pongamos un último caso para dejar el concepto claro. Tenemos un nivel donde para ganar se tienen juntar 10 monedas, pero se pierde si pasaron 30 segundos o se mataron 10 enemigo (en un juego de espionaje puede considerarse negativo matar enemigos). Como se imaginarán, todo esto lo resolvemos con un EnemyManager, un TimeManager, un CoinManager y un GameManager.



Esto se conoce como Separación de Responsabilidades. Ahora también se preguntarán si quizás en vez de tener todo separado en muchos managers, lo mejor hubiera sido haber hecho uno único y controlar todo desde ahí. Esto se podría haber hecho, pero habría un problema.

En el modelo que planteamos el componente del enemigo se comunica con el EnemyManager para contar la muerte. También el GameManager en cada nivel es uno diferente ya que cada uno chequea diferentes cosas. Si nosotros hubiésemos hecho todo en un solo Manager, y queremos tener diferentes condiciones tendríamos que haber hecho de todas formas diferentes Managers para cada nivel. Pero el componente del Enemigo tendría que comunicarse con cada Manager, por lo que dicho componente

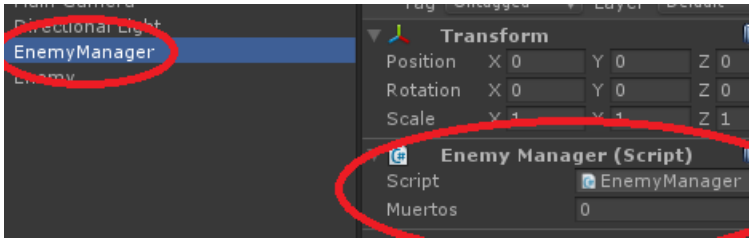
también tendría que ser diferente en cada nivel, complicando todo, ya que tendríamos que tener muchos Prefabs de enemigo, con ese componente siendo la única diferencia.

Programando Managers

Como vemos, crear managers en nuestro juego no es más que crear componentes con propiedades y preguntar cosas en el Update (como en el caso del componente Vida, donde chequeábamos si la cantidad era menor a 0), pero lo que no está claro es como comunicar los componentes entre sí.

Anteriormente vimos que podemos tener propiedades que se conecten con objetos (de tipo GameObject), siendo el editor el lugar donde hacemos dichas conexiones arrastrando objetos. También vimos que podemos usar GetComponent para acceder a un componente de un objeto guardado en una propiedad, parámetro o variable local (de ahora en más por cuestiones de facilidad les llamaremos variables, ya que son lugares cuyos datos pueden variar), y así poder acceder a sus datos.

Siguiendo el ejemplo, el enemigo tiene que comunicarse con el EnemyManager. Recordemos que este manager es simplemente un componente llamado EnemyManager en un Objeto en la escena (preferentemente un objeto vacío).



Podríamos hacer que los enemigos tengan una propiedad de tipo GameObject al cuál le arrastramos el objeto que tiene el manager, y luego cuando necesitemos acceder a él, hacemos un GetComponent.

```
public class Enemy : MonoBehaviour
{
    public float vida;
    public GameObject manager;

    public void Update()
    {
        if(vida <= 0)
        {
            EnemyManager enemyManager = manager.GetComponent<EnemyManager>();
            enemyManager.muertos += 1;
            Destroy(gameObject);
        }
    }
}
```

En el código de la imagen vemos como cuando la vida llega a ser menor o igual a 0, hacemos GetComponent sobre la propiedad que conectamos al Manager desde el editor, para así finalmente sumarle a la propiedad muertos una unidad. Esta vez para sumarle a una variable una cantidad usamos el operador "+=" (sumar a), que suma la cantidad especificada a la derecha del operador, con la variable a la izquierda del operador.

Si bien esta forma funciona y es correcta tiene un problema. ¿Qué sucede si en vez de arrastrar el manager nos equivocamos de objeto y le arrastramos, por ejemplo, la cámara? En este caso el código fallaría y tiraría error, ya que, si conectamos a esta propiedad un objeto que no tiene el componente que buscamos (EnemyManager), la instrucción GetComponent fallaría (más sobre ello adelante).

Esto se debe a que una propiedad de tipo GameObject no es para nada específica, o sea, se puede arrastrar cualquier objeto, cosa que en algunos casos es deseable, por ejemplo, con los Prefabs sería algo necesario así arrastramos cualquier prefab. Nuestro caso actual se soluciona simplemente asegurándose de arrastrar el objeto correcto, pero esto tiene un problema. Si nosotros programamos los componentes de una forma en la cual configurarlo puede llegar a causar problemas, a medida que nuestro código crezca y haya muchos más componentes, va a ser un caos configurarlos todos correctamente. Hay que asegurarse de simplificar lo más posible esta tarea, ya que en Unity vamos a estar configurando componentes una gran parte del tiempo. Podemos solucionar este problema usando propiedades de tipo Componente.

Ejercicio N°1

- *Hacer un componente "EjemploManager1" que tenga una propiedad numérica entera llamada "numero" (int). Hacer que el mismo imprima el mensaje "Hola" si esa propiedad vale 10 (un if en el Update).*
- *Hacer un componente "EjemploObjeto1" que tenga una referencia a un objeto llamada "manager" (una propiedad llamada "manager" de tipo "GameObject") y que cuando detecte la presión de una tecla, obtener el componente "EjemploManager1" de la propiedad manager y aumentar una unidad a la propiedad "numero" del manager.*
- *Crear un objeto llamado "Manager" y poner el componente "EjemploManager1".*
- *Crear un objeto llamado "Objeto" y poner el componente "EjemploObjeto1", arrastrándole el objeto "Manager" en la propiedad "manager".*

Referencias a Componentes

Hemos visto anteriormente, en el caso de las colisiones, que podemos tener variables de tipo componente, o sea, una variable que se conecte con un componente en vez de un objeto (como hacíamos con el parámetro Collider c). Esto se lograba simplemente creando una variable cuyo tipo se llame como el componente deseado. Para resolver el problema formulado anteriormente podemos usar esta habilidad, ya que nada nos impide crear una propiedad de tipo componente.

En la imagen de la siguiente página vemos que reemplazamos el tipo de la propiedad manager, en vez de ser GameObject, pusimos EnemyManager. Esto implica que cuando desde el editor, intentemos arrastrar un objeto a esa propiedad, solamente va a permitírnoslo cuando el objeto que arrastremos tenga un componente EnemyManager. En vez de conectarse al objeto, la propiedad se conecta directamente al componente, facilitándonos el trabajo y asegurándonos de que nada más pueda conectarse. Fíjense como en el Update sacamos el llamado a GetComponent y directamente accedimos a la propiedad muertos, ya que la propiedad está conectada directamente al componente.

```
public class Enemy : MonoBehaviour
{
    public float vida;
    public EnemyManager manager;

    public void Update()
    {
        if(vida <= 0)
        {
            manager.muertos += 1;
            Destroy(gameObject);
        }
    }
}
```

Ejercicio N°2

- *Hacer un componente "EjemploManager1" que tenga una propiedad numérica entera llamada "numero" (int). Hacer que el mismo imprima el mensaje "Hola" si esa propiedad vale 10 (un if en el Update).*
- *Hacer un componente "EjemploObjeto1" que tenga una referencia a "EjemeploManager1" llamada "manager" (una propiedad de tipo "EjemploManager1") y que cuando detecte la presión de una tecla, aumentar una unidad a la propiedad "numero" del manager.*
- *Crear un objeto llamado "Manager" y poner el componente "EjemploManager1".*
- *Crear un objeto llamado "Objeto" y poner el componente "EjemploObjeto1", arrastrándole el objeto "Manager" en la propiedad "manager".*

Ejercicio N°3

- *Hacer un componente "EjemploMover1" que tenga una referencia a un Rigidbody (una propiedad llamada "objetivo" de tipo Rigidbody) que cuando detecte la presión de una tecla ejecute la instrucción AddForce del Rigidbody conecto a la propiedad "objetivo".*
- *Crear una esfera llamada "Pelota" de tipo "Dinámica Bloqueadora Realista".*
- *Crear un objeto vacío con el componente "EjemploMover1" y arrastrar la esfera a la propiedad objetivo.*

Referencias entre Contexto

La forma en que resolvimos el tema de los managers hasta ahora es parcial, porque no nos sirve para todas las situaciones. Acá nosotros tuvimos que arrastrarle a cada enemigo que ya está en la escena el manager, que es otro objeto que también está en la escena. Ahora, si nosotros queremos que los enemigos aparezcan en medio del juego, o sea, instanciados con Instantiate una vez que el juego ya está corriendo, vamos a necesitar que los enemigos no estén en la escena, sino en un prefab. El problema acá viene cuando nosotros queramos arrastrarle el Manager al prefab. Si lo intentamos veremos que no vamos a poder. Esto se debe al contexto del objeto.

El contexto de un objeto que está en la escena es la escena en sí, o sea, el objeto es parte de una escena. En cambio, como vimos anteriormente, un prefab no es parte de una escena, sino es parte del proyecto en sí, cosa que le permite estar presente en varias escenas. Por ende, el contexto de un prefab es el proyecto. No se pueden conectar objetos de cualquier contexto a cualquier contexto. Una propiedad de un objeto en una escena se puede conectar a un objeto tanto de una escena como del proyecto. En cambio, un objeto que está en el proyecto se puede conectar a objetos de proyecto (un prefab instancie otro prefab, como el caso de los enemigos que disparan balas), pero no de escena. Esto se debe a que los Prefabs al poder estar instanciados en varias escenas, no puede conectarse a algo particular de una escena, a diferencia de una instancia de un prefab, que ellas si residen en una escena.

Conexión entre contextos		
	Proyecto	Escena
Proyecto	Si	No
Escena	Si	Si

Entonces cuando queramos conectar el manager, que es un objeto de una escena particular, a un prefab, esto no va a ser posible. Para solucionar este problema necesitaremos buscar otra forma.

Buscar objetos por nombre

Una de las formas para solucionar este problema es hacer uso de la instrucción "GameObject.Find". La instrucción es muy simple, se encarga de buscar un objeto por su nombre y obtenerlo.

```
public class Enemy : MonoBehaviour
{
    public float vida;

    public void Update()
    {
        if(vida <= 0)
        {
            GameObject manager = GameObject.Find("EnemyManager");
            EnemyManager enemyManager = manager.GetComponent<EnemyManager>();
            enemyManager.muertos += 1;
            Destroy(gameObject);
        }
    }
}
```

En el código vemos como esta versión de Enemy no tiene ninguna propiedad que sirva para conectar el componente al manager. En vez de ello, lo que hicimos fue hacer uso de la instrucción Find. Recordemos que, en nuestra escena, el objeto que posee el componente "EnemyManager" se llama de igual manera, o sea, "EnemyManager". Nosotros al haberle pasado "EnemyManager" por parámetros a la instrucción Find, lo que hicimos fue pedirle que busque en la escena en la cual está el juego actualmente un objeto llamado así. Recalco que va a buscar el objeto "EnemyManager", no el componente, ya que al llamarse iguales presta a la confusión, cosa que va a suceder mucho en programación, ya que todo tiene que tener un nombre y no siempre es fácil buscar uno que se diferencie del otro.

De ésta forma finalmente tenemos un código que funciona en todas las situaciones, ya que por más que los objetos estén o no en la escena, no hace falta configurarlos previamente, simplemente busca el objeto necesario en el momento que lo necesita. Sin embargo, esta forma de solucionar el problema no es perfecta. Hay que tener cuidado de que exista un objeto en la escena llamado "EnemyManager", y que se llame exactamente igual, respetando mayúsculas y minúsculas a rajatabla, ya que, por ejemplo, si el objeto se llama "enemyManager" no lo va a encontrar. El otro tema con esta forma es que hay que recordar que tiene que estar ese objeto. Quizás hagamos el nivel 20, y nos olvidamos de poner el manager, y puede suceder que nunca nos demos cuenta ya que no probamos esa escena a menudo (cosa que se soluciona haciendo testeos periódicos de nuestro juego).

Ejercicio N°4

- *Hacer un componente "EjemploObjeto2" que cuando detecte la presión de una tecla busque un objeto llamado "Manager" (con la instrucción Find), acceda al componente "EjemploManager1" del mismo (usando GetComponent sobre el objeto encontrado con Find) y le aumente en 1 la propiedad "numero" del manager.*
- *Crear un objeto llamado "Objeto2" y ponerle el componente "EjemploObjeto2".*

Ejercicio N°5

- *Hacer un componente "EjemploMover2" que cuando detecte la presión de una tecla busque un objeto llamado "Pelota" (con la instrucción Find), acceda al componente "Rigidbody" del mismo (usando GetComponent sobre el objeto encontrado con Find) y le aplique una fuerza con la instrucción AddForce.*
- *Crear un objeto llamado "Objeto3" y ponerle el componente "EjemploMover2".*

Optimizando GetComponent

Un último detalle a tener en cuenta, que si bien no afecta la funcionalidad es una buena práctica tenerlo en cuenta, es que la función GetComponent hay que tratar de ejecutarla la menor cantidad de veces posible, esto se debe a que ejecutarla cuesta mucho proceso. Obviamente "mucho proceso" es algo relativo, estamos hablando de mucho menos de un milisegundo, pero no es para nada raro que quizás en un solo frame estemos ejecutando esa instrucción 100 veces, y a la larga todo suma.

Una pequeña optimización que podemos hacer a veces es simplemente hacer el GetComponent en el Awake, guardar el resultado en una propiedad, y luego utilizar esa propiedad sin tener que ejecutar nuevamente el GetComponent.

```
public class Enemy : MonoBehaviour
{
    public float vida;
    private EnemyManager enemyManager;

    public void Awake()
    {
        GameObject manager = GameObject.Find("EnemyManager");
        enemyManager = manager.GetComponent<EnemyManager>();
    }

    public void Update()
    {
        if(vida <= 0)
        {
            enemyManager.muertos += 1;
            Destroy(gameObject);
        }
    }
}
```

En el código vemos como el componente tiene nuevamente una propiedad de tipo "EnemyManager", pero veamos que en vez de tener la palabra "public" delante, tiene la palabra "private". Si bien esto significa otras cosas, por ahora la usamos simplemente para que esa propiedad no se vea desde el editor, ya que no es algo a configurar, es simplemente una variable donde guardamos la conexión al componente. Luego el resto es simplemente buscar el objeto deseado en el Awake, hacerle GetComponent y guardar el resultado en dicha propiedad, para luego, en el Update usarla sin tener que hacer GetComponent más de una vez si se diese el caso. Esto también puede servir para acceder a un componente de nuestro propio objeto y no tener que conectarlo.

Ejercicio N°6

- *Modificar los componentes que usen GetComponent hechos en los ejercicios anteriores y hacer que lo hagan en el Awake, guardándose la referencia en una propiedad para utilizarla posteriormente.*

Ejercicio N°7

- *Hacer un componente llamado "EjemploMover3", que en el Awake se haga GetComponent a si mismo buscando un Rigidbody (simplemente poner GetComponent sin nada adelante) y que guarde en una propiedad dicho componente, así guardando en la misma su propio Rigidbody. Cuando detecte la presión de una tecla agregar una fuerza a su propio Rigidbody guardado en dicha propiedad.*
- *Crear otra esfera llamada "Pelota2" de tipo "Dinámica Bloqueadora Realista" y que tenga el componente "EjemploMover4".*

Previniendo Errores

El detalle del cual nos vamos a encargar ahora son los posibles errores de buscar objetos y componente. Si tenemos una propiedad para conectar un objeto o un componente simplemente lo hacemos desde el editor, sabiendo de que lo que arrastramos existe. Cuando nosotros le preguntamos a Unity que nos dé un objeto por nombre, o un componente en un objeto pueden suceder una serie de errores.

¿Qué pasaría si el objeto que buscamos no existe? Más allá de si tipiamos mal o bien el nombre quizás el objeto no está en la escena. Y si encontramos el objeto, ¿Qué pasa si no tiene el componente?, ya sea porque nos olvidamos de poner el componente en el manager o porque el objeto al que chocamos no tiene ese componente Vida, porque chocamos contra una pared. Estaríamos buscando algo que no está. GetComponent y Find no tiran errores en esas situaciones, simplemente el valor que devuelven vale "null".

Null es un valor especial que, como su nombre lo indica, significa nulo. Las variables que se conectan a objetos, a diferencia de las que no (como int, float, bool, etc.) pueden valer null. Cuando esas variables valen null significa que no están conectadas a nada. Desde el editor cuando ven una propiedad de este estilo (referencias), cuando no está conectada a nada pone la leyenda "None". Entonces, si nosotros queremos asegurarnos de que el código funcione correctamente tenemos que chequear estas cosas usando if.

```
public void Awake()
{
    GameObject manager = GameObject.Find("EnemyManager");

    if (manager != null) //Si el objeto es diferente de null
    {
        Debug.LogError("No existe el objeto EnemyManager");
    }

    enemyManager = manager.GetComponent<EnemyManager>();

    if(enemyManager != null) //Si el componente es diferente de null
    {
        Debug.LogError("No existe el componente EnemyManager");
    }
}
```

Usando el if y el operador "!=" (no igual) podemos preguntar primero si el objeto existe y luego si el componente existe en ese objeto. Lo que usamos nuevo es "Debug.LogError". Esta instrucción es similar a print, imprime un mensaje en la consola, pero con un ícono de error que permite diferenciar el mensaje.

Ejercicio N°8

- *Modificar los componentes que usen GetComponent y Find para que sean seguros, o sea que, si no encuentran el objeto o componente que buscan, informen mostrando un error en la consola.*
- *Probar esto borrando el manager.*
- *Asegurarse de volver a crearlo después de probar.*

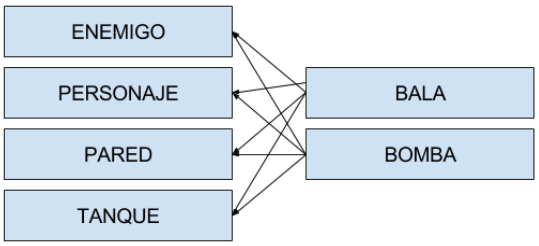
Eventos Personalizados

Hasta ahora tenemos el manager funcionando, pero como siempre, hay algunos detalles de los cuales tenemos que encargarnos. En los ejemplos que tenemos hasta ahora, la vida del enemigo la tenemos en el componente "Enemy", sin embargo, en lo que vinimos programando hasta ahora la teníamos en un componente "Vida" que se encargaba de darle vida a los objetos. De esta forma las balas no se preocupaban por saber si chocaron contra un personaje o contra un enemigo, simplemente preguntaban por si el objeto que chocaba tenia vida, y en tal caso, quitarle.

La diferencia entre ambas versiones es que claramente el enemigo no va a ser el único objeto que tenga vida. Pueden tener vida los enemigos, el personaje, paredes destruibles, puertas destruibles, autos, tanques de gas, básicamente cualquier cosa que a lo que se le pueda disparar en un juego. En nuestro juego tiene tanto vida el Enemigo como el Personaje, por lo que, si la vida de cada uno estaría en los componentes Enemy y Player, tendríamos el problema de que la bala tendría que preguntar si el objeto es Enemy o Player, o sea, preguntar si tiene un componente u otro.

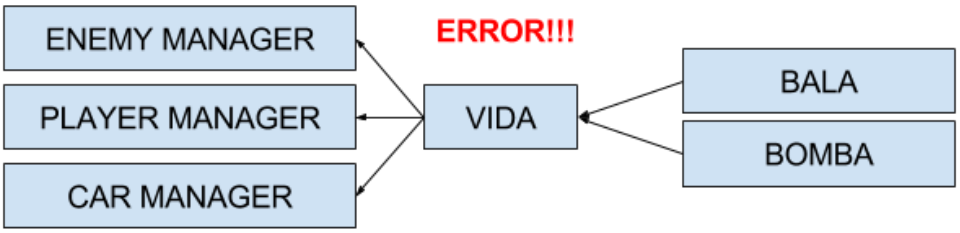
```
public void OnCollisionEnter(Collision c)
{
    Enemy enemy = c.gameObject.GetComponent<Enemy>();

    if(enemy != null)
    {
        enemy.vida -= 10;
    }

    Player player = c.gameObject.GetComponent<Player>();

    if (player != null)
    {
        player.vida -= 10;
    }
}
```

El problema de hacer esto se va haciendo evidente a medida que vamos agregando más cosas destruibles. Si agregásemos paredes y tanques, tendríamos que agregar 2 ifs más. Ni hablar de tener otros objetos que quiten vida, como bombas con daño radial y lasers (se hacen de otra forma que veremos más adelante), en esos casos tendríamos esos 4 ifs en 3 componentes de daño. Un juego real generalmente tiene muchos más tipos de armas y objetos dañables, por lo que el código, si bien no es imposible de hacer, realmente se va a complicar.



El camino que tomamos en el módulo anterior era el correcto, tener un componente Vida separado que sólo se encargue de eso, cosa de que todos los objetos destruibles tengan el mismo componente vida. Los objetos que dañan simplemente preguntan por el componente Vida al objeto que quieren dañar en vez de preguntar por cada tipo de objeto, y simplifica el código enormemente. Pero, siempre hay una trampa.

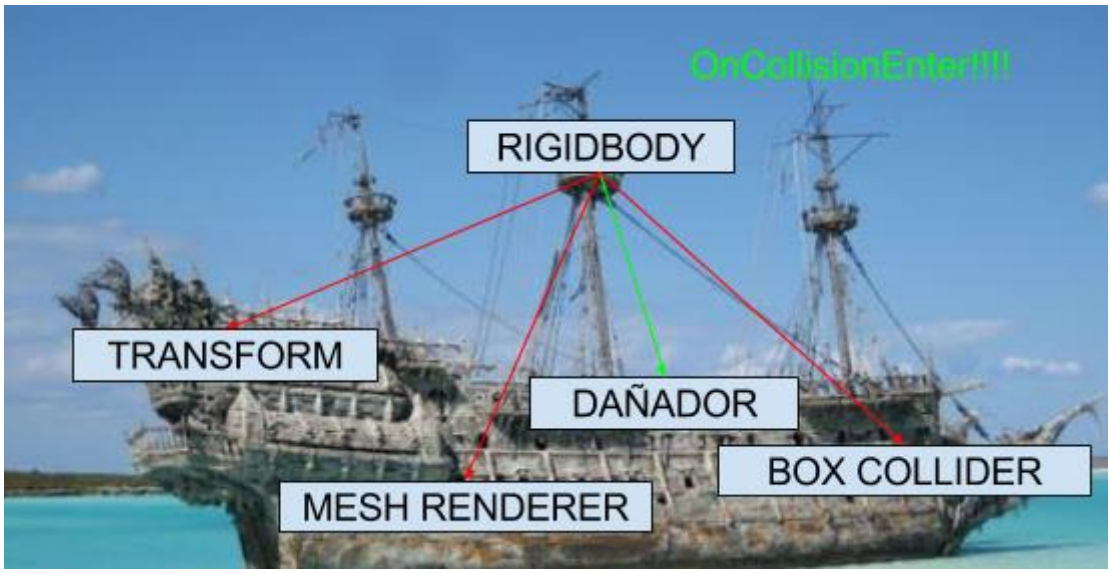
Si vamos por ese camino se complica el caso de los managers. Si nosotros en el componente vida chequeamos si la vida llega a 0, y en tal caso destruimos al objeto. Más allá de que a veces eso puede llegar a variar, el problema viene cuando queremos comunicarnos con el manager. Al tener en todos los objetos el mismo componente, no podemos poner ahí el llamado a sumar una muerte en el "EnemyManager", ya que cualquier cosa que se destruya sumaría una muerte al enemigo, y claramente cuando destruimos una pared no matamos a un enemigo. Otro caso que se puede dar, es que quizás matamos a 9 de 10 enemigos, pero cuando nos matan, al tener esto contaría como que ganamos, porque matar al personaje contaría como una muerte del enemigo, cosa que no queremos que suceda.



Ahora estamos en una encrucijada, o complicamos el código poniendo muchos ifs por todos lados (error muy común a la hora de comenzar a programar), o tenemos errores en los managers. La lógica diría complicar el código ya que por lo menos quedarían los managers funcionando, pero cada vez que en programación nos encontramos con estos problemas, siempre hay una solución mejor que no conocemos. En este caso lo que necesitamos son los "Eventos Personalizados".

Pensemos en el componente "Rigidbody". Este componente como sabemos, si lo ponemos en nuestro objeto, hace que nuestro objeto sea capaz de detectar colisiones. Eso lo hace en un "Update" propio (no es el mismo que usamos), y tiene código como el que escribimos nosotros, pero para detectar colisiones (mucho más complejo). Ahora, cuando detecta una colisión nosotros sabemos que de alguna forma nuestros componentes reciben el evento "OnCollisionEnter" o "OnTriggerEnter". Eso pasa porque Rigidbody usa la instrucción "SendMessage", la cual se encarga de comunicarle a los componentes del objeto donde está puesto que sucedió algo.

Para visualizar esto pensemos que un objeto es un barco, y los componentes sus tripulantes. Cada tripulante tiene su responsabilidad y solamente se encarga de eso. En nuestro caso, el Rigidbody sería el tripulante en el carajo (una zona en la parte superior del barco que permite ver en la lejanía, de ahí el insulto) que está verificando que no estemos por chocar con algo, siendo el caso del barco un iceberg, una costa, u otro barco. En dicho caso, el tripulante gritaría al resto de la tripulación un mensaje de que está sucediendo algo, por ejemplo "¡ICEBERG!". En nuestro caso, el Rigidbody cuando detecta una colisión grita "¡OnCollisionEnter!" o "¡OnTriggerEnter!". Al gritar el tripulante en el barco, sus compañeros se enteran, pero no todos reaccionan ante ello, si el cocinero o el que trapea el piso lo escuchase no le tendría que importar y seguirían su labor. En cambio, el timonel tendría que comenzar a virar el barco para prevenir la colisión, y quizás los cañones se preparan para el ataque. En nuestro caso, el Rigidbody al gritar, no todos los componentes reaccionan, por ejemplo, el Transform y el MeshRenderer no hacen nada en caso de colisión, pero el componente que quita vida en las balas sí, ya que las colisiones son de su interés. Lo mismo sucede con el "Awake" y el "Update".



Usando ese concepto, podemos usar la instrucción SendMessage en el componente Vida. El componente Vida cuando detecta que la vida llego a ser menor o igual a 0 aparte de destruir al objeto puede enviar el evento "¡Morí!" y no va a hacer nada más. Usando el concepto de separación de responsabilidades, el componente vida no puede hacer más que eso ya que su único trabajo es ése, al igual que el Rigidbody no hace más que separar objetos y aplicar física cuando detecta la colisión y luego nos avisa que sucedió una colisión para que nosotros hagamos lo que queramos, como destruir cosas, mover cosas, etc.

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

    public void Update()
    {
        if(vida <= 0)
        {
            Destroy(gameObject);
            SendMessage("Mori");
        }
    }
}
```

La instrucción es bastante simple de utilizar, simplemente le pasamos como primer argumento el mensaje que deseamos enviar (en el caso del Rigidbody, allí pondría "OnCollisionEnter" por ejemplo). Un detalle a tener en cuenta es que, si se envía un mensaje y no hay ningún componente que le interese, se va a ver un error en la consola diciendo que nadie recibe el mensaje. Si bien hay una forma de hacer que eso no pase, por ahora no nos preocupemos de ello ya que en nuestro proyecto siempre va a haber alguien que lo reciba, y este mensaje nos ayuda a detectar que nos olvidamos algo.

Ahora, al hacer esto el componente vida ya no se comunica más con el Manager, simplemente envía el mensaje "Morí" (en el código no puse acento porque suele causar errores) y nada más. El encargado de comunicarse con el manager sería otro componente que lo hace cuando escucha el evento "Morí" (al igual que el Dañador escucha el OnCollisionEnter que avisa el Rigidbody para quitar vida), este pudiendo ser el componente "Enemy".

```
public class Enemy : MonoBehaviour
{
    private EnemyManager enemyManager;

    public void Awake()
    {
        GameObject manager = GameObject.Find("EnemyManager");
        enemyManager = manager.GetComponent<EnemyManager>();
    }

    public void Mori()
    {
        enemyManager.muertos += 1;
    }
}
```

Como vemos, para escuchar el evento "Morí" hacemos lo mismo que hacíamos con el "OnCollisionEnter", el "Awake" o el "Update", simplemente creamos una función llamada como el evento que queremos escuchar y listo. De esta forma creamos nuestro propio evento.

Ejercicio N°9

- *Hacer un componente llamado "EjemploEnvioMensaje1" que envíe el mensaje "Arriba" cuando se presione la tecla "W" y el mensaje "Abajo" con la tecla "S".*
- *Hacer un componente llamado "EjemploRecepcionMensaje1" que tenga los eventos Arriba y Abajo y que impriman un mensaje cuando se ejecuten.*
- *Hacer un objeto llamado "PruebaEvento1" que tenga ambos componentes.*

Ejercicio N°10

- *Hacer un componente llamado "EjemploEnvioMensaje1" que tenga una propiedad entera (int) llamada "cantidad". Cuando el objeto detecte una colisión restar una unidad a dicha propiedad, y chequear en el mismo evento de colisión si después de restar la cantidad es menor o igual a 0. En dicho caso enviar el mensaje "Boom" y destruirse a sí mismo.*
- *Hacer un componente llamado "EjemploRecepcionMensaje2" que tenga el evento Boom y que cuando lo reciba imprimir el mensaje "¡Boom!".*
- *Crear una caja llamada "PruebaEvento2" de tipo "Estática Bloqueadora" y ponerle ambos componentes.*
- *Crear un objeto "Personaje" que sea una cápsula "Dinámica Bloqueadora" que tenga un componente de movimiento con teclas.*
- *Hacer que choquen ambos objetos.*

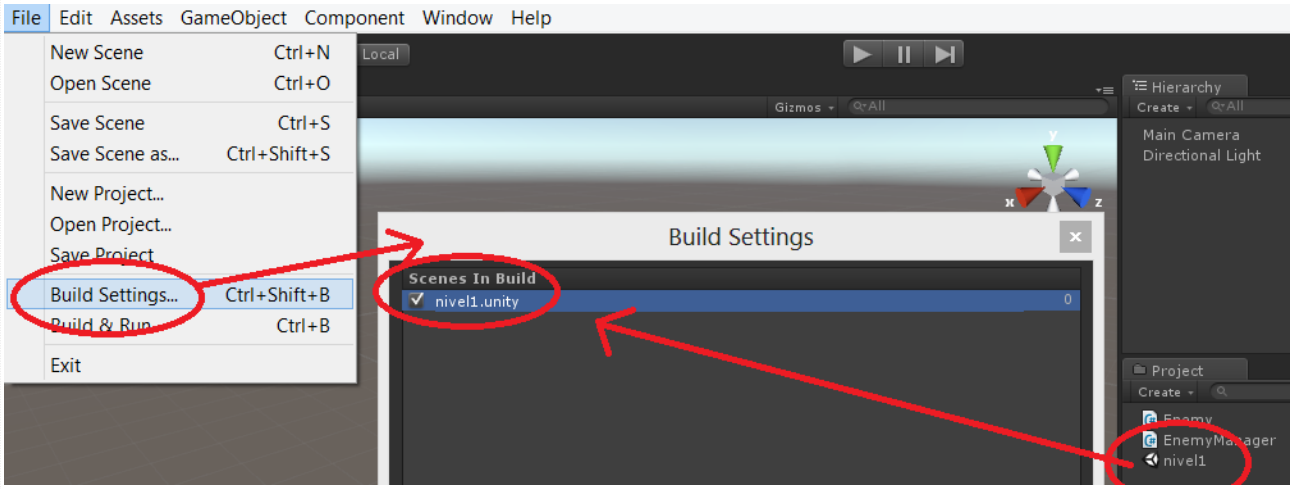
Comunicación entre Escenas

Ahora que ya tenemos una forma de detectar si ganamos o no, lo último que nos queda es finalizar el nivel. Si bien, generalmente cuando ganas o pierdes hay un tiempito con alguna animación, etc., en este caso vamos a hacer algo muy simple, cambiar la escena. En nuestro caso podríamos ir al nivel 2, o a alguna escena intermedia con un mensaje de ganaste y un botón de continuar (veremos interfaz gráfica más adelante). Más allá de a dónde vamos, tenemos que saber cómo. Para ello está la instrucción LoadLevel.

```
public class GameManagerNivel1
{
    private EnemyManager enemyManager;

    public void Awake()
    {
        GameObject manager = GameObject.Find("EnemyManager");
        enemyManager = manager.GetComponent<EnemyManager>();
    }

    public void Update()
    {
        if (enemyManager.muertos >= 10)
        {
            Application.LoadLevel("Nivel2");
        }
    }
}
```

Como vemos la instrucción es bastante simple, con pasarle el nombre del nivel al cual queremos ir alcanza. Cuando hacemos esto primero Unity destruye todos los objetos de la escena actual avisándonos con el evento OnDestroy y luego crea los objetos de la otra escena. Recalco lo del evento OnDestroy porque muchas veces uno para reemplazar al mensaje "Morí" usan "OnDestroy" olvidándose de que dicho evento se ejecuta también cuando cambiamos la escena, cosa que podría provocar errores. Por ejemplo, si matamos a 9 enemigos de 10 pero nos mataron antes de que pudiésemos matar al último, cambiamos de escena a la escena de perder. Al contar una muerte de enemigo en el OnDestroy, los enemigos restantes contarían la muerte, y ganaríamos.

Un detalle a tener en cuenta de esta instrucción es que necesitamos decirle a Unity que escenas van a ser las que van a estar en el juego. Uno normalmente pensaría que estarían todas, pero no siempre es cierto. Muchas veces tenemos escenas para probar cosas, como movimientos de personaje o enemigos. Esas escenas conviene dejarlas para de vez en cuando verificar que todo siga andando bien, porque probar cosas puntuales en escenas donde hay muchas cosas juntas es difícil, entonces, esas escenas no tienen que ir en el juego. Para ahorrar peso en el juego Unity nos pide que le indiquemos que escenas van, y lo hacemos yendo al menú que está en File -> Build Settings y arrastrándole la escena desde el panel de Project a ese menú.



Ejercicio N°11

- *Hacer un componente llamado "EjemploCambioEscena1" que cuando presione una tecla cambie a una escena indicada con una propiedad llamada "nivel" (una propiedad de tipo string).*
- *Crear una escena llamada "Escena1" que tenga una caja con el componente y que en la propiedad "nivel" tenga el valor "Escena 2".*
- *Crear una escena llamada "Escena2" que tenga una esfera con el componente y que en la propiedad "nivel" tenga el valor "Escena 1".*
- *Asegurarse de que ambas escenas estén en las Build Settings.*

Ejercicio N°12

- *Hacer un componente llamado "EjemploCambioEscena2" con una propiedad entera "numero" que cuando presione una tecla reste una unidad a la propiedad y si la propiedad llego a 0 cambie a una escena indicada con una propiedad llamada "nivel" (una propiedad de tipo string).*
- *Crear una escena llamada "Escena1" que tenga una caja con el componente y que en la propiedad "nivel" tenga el valor "Escena 2" y en la propiedad "cantidad" 2.*
- *Crear una escena llamada "Escena2" que tenga una esfera con el componente y que en la propiedad "nivel" tenga el valor "Escena 1" y en la propiedad "cantidad" 2.*
- *Asegurarse de que ambas escenas estén en las Build Settings.*

Finalizado del Prototipo

Detalles Finales

En este bloque veremos una serie de detalles aislados para dar por terminado nuestro primer prototipo y poder pasar en los siguientes bloques al apartado gráfico. Recordemos que hicimos dentro de todo algo relativamente simple y que a partir de la segunda parte del curso comenzaremos a desarrollar un nuevo juego que, si bien usará todo lo que vimos hasta ahora, tendrá requisitos que requieren nuevos conocimientos.

Todos los tips que veremos en este bloque requieren de poco o ningún conocimiento adicional del que ya adquirimos hasta el momento. La idea es poner en práctica todo lo visto y hacer uso creativo de ello.

Feedback de Objetos

En los videojuegos no solo importa que las cosas sucedan, sino que el usuario se dé cuenta de que sucedieron. Actualmente cuando destruimos a un enemigo el mismo desaparece sin más. Si bien nosotros sabemos porque, ya que nosotros hicimos el juego, cuando le mostremos a nuestros amigos el juego, veremos que probablemente se confundan. ¿Qué pasó? ¿Por qué desapareció el enemigo sin más? Este tipo de preguntas se debe a que uno normalmente está acostumbrado a que pase algo cuando un enemigo muere. Quizás se ve una explosión, quizás se ve la nave destruida, etc.

Si bien, todavía no hemos visto mucho acerca de efectos visuales (cosa que veremos en las siguientes unidades) estamos en condiciones de dejar el código preparado para contemplar estos casos, ya que eventualmente veremos cómo hacer Prefabs con explosiones y modelos tridimensionales de nuestro objeto destruido. La idea sería que cuando el enemigo sea destruido, al desaparecer deje en su lugar otro objeto que solamente sea un efecto gráfico, como una explosión, llamas, sonidos, restos del enemigo, o todos ellos juntos.

Esto se logra simplemente Instanciando un prefab en el lugar donde estaba el enemigo.

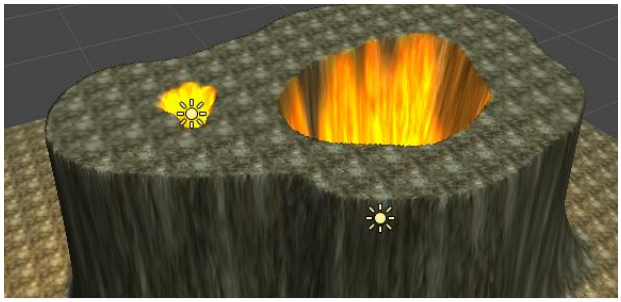
Ejercicio N°13

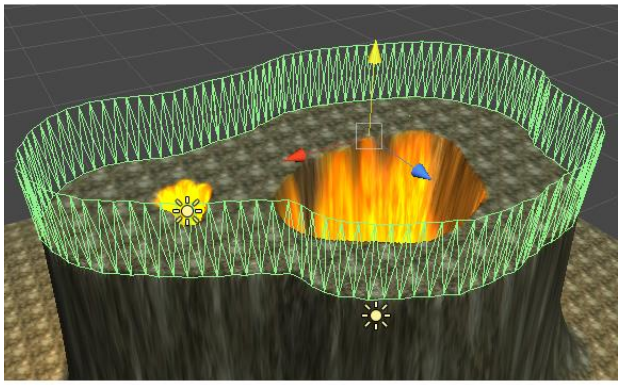
- *Hacer un componente llamado "EjemploDestruccion1" que cuando detecte la presión de una tecla se destruya a sí mismo e instancie un prefab en su lugar.*
- *Crear un prefab de una esfera llamada "Prefab1".*
- *Crear un cubo llamado "Destruible1" con el componente "EjemploDestruccion1" y arrastrarle la esfera en la propiedad prefab.*

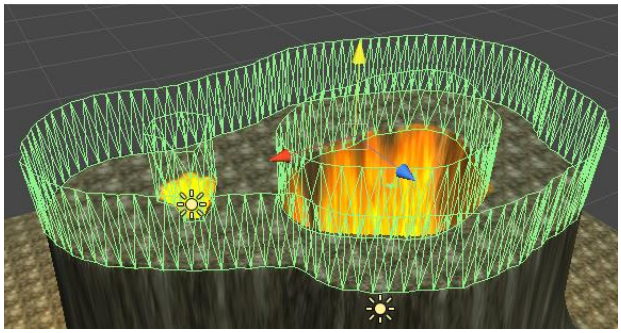
Límites de escenario

Nuestro personaje actualmente se mueve y no puede traspasar las paredes. Un pequeño detalle a tener en cuenta es que a veces hay juegos en donde el escenario tiene una figura diferente a lo que se ve, o sea, que quizás veamos un montón de piedras bloqueando el camino, cuando realmente hay una caja gigante invisible que nos impide el camino. También tenemos juegos donde los límites no tienen representación visual, simplemente llegamos al borde de la pantalla y no nos podemos ver más. Esto se debe a la misma razón por la cual no pasábamos por las piedras en el ejemplo anterior, hay una caja gigante invisible.

Para lograr eso simplemente creamos un cubo, le eliminamos el componente MeshRenderer y MeshFilter (clic derecho -> Remove Component) y dejamos el BoxCollider, dejando el objeto configurado como "Estático Bloqueador".







Ejercicio N°14

- *Crear un personaje que se mueva con las teclas.*
- *Ponerlo entre 4 cajas a modo de pared y sacarles los MeshRenderer y MeshFilter.*

Juegos Scroller

Nosotros si queremos podemos hacer de nuestro juego un Vertical Top Down Scroller Shooter, o sea, los juegos donde vemos una nave de arriba que se mueve constantemente hacia adelante y los enemigos van apareciendo desde arriba. Para lograr esto, podemos usar un truco, que sería hacer que la nave no se mueva para arriba, simplemente que la nave se pueda mover a los costados, limitada por las cajas anteriormente mencionadas, y hacer que los enemigos aparezcan arriba moviéndose hacia abajo constantemente. Eso sumado a que el escenario mismo se puede mover enteramente, nos ahorraría el trabajo de tener que hacer componentes de cámara, por lo que con nuestro conocimiento actual podremos lograr tranquilamente (más adelante veremos sobre este tema).

Ejercicio N°15

- *Crear un componente de movimiento llamado "EjemploMovimiento1" que solo se mueva horizontalmente.*
- *Hacer un personaje que se tenga dicho componente.*
- *Ponerlo entre 4 cajas a modo de pared y sacarles los MeshRenderer y MeshFilter.*

Ejercicio N°16

- *Crear un componente de movimiento llamado "EjemploMovimiento21" que solo se mueva verticalmente.*
- *Hacer un personaje que se tenga dicho componente.*
- *Ponerlo entre 4 cajas a modo de pared y sacarles los MeshRenderer y MeshFilter.*

Feedback al Ganar y Perder

Otra cosa a tener en cuenta que hemos mencionado anteriormente es que normalmente en un juego al ganar o perder no vamos directamente a la pantalla de ganar o perder. Si hacemos eso sería muy brusco y el jugador no entendería que paso. Por más que no parezca, dejar tiempos prudentes para que el usuario procese lo que acabo de suceder es necesario, ya que sino el juego sería muy confuso al ser tan brusco. Esto mezclado con el Feedback de objetos serviría para, por ejemplo, ver la explosión de nuestro personaje antes de cambiar de escena a la de perder, así nos damos cuenta que acabo de suceder.

Esto se logra usando un Invoke. A la hora de detectar la condición, en vez de cambiar el nivel, ejecutamos un Invoke para que luego de 2 segundos cambie la escena. Si hacemos esto hay que tener en cuenta que verificamos la condición en el Update, por lo que los Invoke se empezarían a acumular ya que estamos ejecutándolos todos los frames. En este caso particular no es un problema, ya que el primer Invoke haría que cambiemos de escena, haciendo que los Invoke acumulados no se ejecuten. De todas formas, no sería algo prolijo ya que en otros casos esto puede ser un problema. Es conveniente hacer que el if de la condición esté adentro de otro if que chequee si el juego está ganado o perdido, con un bool que lo encendamos cuando ganamos.

```
public class GameManagerNivel1 : MonoBehaviour
{
    private EnemyManager enemyManager;
    private bool gano;

    public void Awake()
    {
        GameObject manager = GameObject.Find("EnemyManager");
        enemyManager = manager.GetComponent<EnemyManager>();
    }

    public void Update()
    {
        if (gano != true) //Todavía no ganamos
        {
            if (enemyManager.muertos >= 10)
            {
                Invoke("CambiarNivel", 2);
                gano = true;
            }
        }
    }

    public void CambiarNivel()
    {
        Application.LoadLevel("Nivel2");
    }
}
```

Ejercicio N°17

- *Crear un componente llamado "EjemploCambioEscena2" que cuando detecte la presión de una tecla imprima el mensaje "Cambiando…" y espere 2 segundos y que cuando pase el tiempo cambie a la escena "Escena2".*

Ejercicio N°18

- *Crear un componente llamado "EjemploCambioEscena2" que cuando detecte la presión de una tecla imprima el mensaje "Cambiando…" y espere 2 segundos y que cuando pase el tiempo cambie a la escena "Escena2". Asegurarse de que el código de detección de la tecla se ejecuta una sola vez, o sea, que no vuelva a ejecutar el Invoke ni el Print. Para ello crear una propiedad booleana y encenderla también cuando se presiona una tecla, y poner el código de presión de una tecla adentro de un if que compruebe que dicha propiedad sea false (== false).*

Vidas

Algo muy común en un juego es que no perdamos instantáneamente, sino que tengamos varios intentos antes de considerarnos derrotados. Lo que podemos hacer es que cuando la vida llegue a 0, restar uno a una propiedad "cantidadDeVidas" que agregamos al componente vida y chequear si no llego a 0. Si llegó a 0 podemos destruir al objeto, sino, podemos simplemente volver a poner la vida en 100 y poniendo al objeto en su posición inicial usando el siguiente código.

```
public int vidas;
public Transform posicionOriginal;

public void Update()
{
    if (vidas <= 0)
    {
        transform.position = posicionOriginal.position;
    }
}
```

Lo que hicimos fue crear una propiedad de tipo Transform, por lo que podremos conectarla con otro transform, en este caso, arrastrando un objeto vacío de la escena a esa propiedad. Este objeto vacío (GameObject -> Create Empty) estaría posicionado en el punto inicial del personaje, y al copiar su posición en la posición de nuestro transform, estaríamos moviéndonos a ese punto. Opcionalmente podemos crear una explosión antes de esa línea para dejar la explosión en el lugar donde estábamos.

También vale la pena destacar que, si podemos cambiar la posición, también podemos cambiar la rotación, usando la propiedad "rotation" de los Transform

Ejercicio N°19

- *Crear un componente llamado "EjemploReset1" que tenga una propiedad "objetivo" de tipo Transform y que cuando detecte la presión de una tecla ponga su posición y rotación en la misma de la propiedad objetivo.*
- *Crear un personaje que se mueva con las teclas y poner el componente "EjemploReset1".*
- *Crear un objeto vacío y arrastrarlo a la propiedad "objetivo" del personaje.*
- *Moverse y apretar la tecla de necesaria para ir a la posición del objetivo.*

Ejercicio N°20

- *Crear un componente llamado "EjemploReset2" que tenga una propiedad "objetivo" de tipo MeshFilter y que cuando detecte la presión de una tecla ponga su la propiedad mesh del MeshFilter de la propiedad en nuestro propio MeshFilter (buscándolo con GetComponent).*
- *Crear un cubo y una esfera.*
- *Poner el componente en el cubo y arrastrar la esfera a la propiedad.*

Dificultad creciente

Hay juegos los cuales no tienen niveles, sino son un nivel el cual no termina jamás, ya que los enemigos van apareciendo y cada vez más difíciles. Ese tipo de juegos generalmente o se juega hasta perder, o tienes un tiempo el cual tienes que estar vivo. De una forma u otra los enemigos tienen que crecer su dificultad. Nosotros anteriormente hicimos un ejercicio donde hacíamos que los enemigos se vayan creando por código (instanciando Prefabs), pero los enemigos eran siempre los mismos. Lo que podríamos hacer es tomar la instancia y modificarle la velocidad haciendo un GetComponent al objeto recién instanciado, sumándole un valor según la cantidad de enemigos creados.

```
public class Generador : MonoBehaviour
{
    private int cantidadCreada;
    public GameObject prefabEnemigo;
    public float tiempo;

    public void Awake()
    {
        InvokeRepeating("CrearEnemigo", 5, tiempo);
    }

    public void CrearEnemigo()
    {
        GameObject enemigo = (GameObject)Instantiate(prefabEnemigo, transform.position, transform.rotation);
        MovimientoEnemigo movimientoEnemigo = enemigo.GetComponent<MovimientoEnemigo>();
        movimientoEnemigo.velocidad += cantidadCreada * 2;
        cantidadCreada += 1;
    }
}
```

Aquí lo que es nuevo es lo que está delante de la instrucción Instantiate. Ese código tiene una razón de ser, pero la misma es muy avanzada para esta etapa, así que por ahora vamos a decir que si ponemos "(GameObject)" delante del Instantiate estamos guardando en una variable (en este caso "enemigo") el objeto que acabamos de instanciar, si no ponemos eso el código va a tirar error. Si desean saber más al respecto del porqué recomiendo que busquen "Type Casting" en internet, pero insisto en que todavía es algo avanzado que puede llegar a confundir más de lo que clarifica, y es el único momento que lo vamos a necesitar en este curso.

Una vez que accedimos al objeto que instanciamos, accedemos al componente que mueve al objeto para aumentarle la velocidad, cosa que hace que los enemigos nuevos se muevan más rápido. También hay que fijarse como la variable "cantidadCreada" es aumentada en 1 cada vez que creamos un enemigo, haciendo que los enemigos siguientes al usar esa fórmula especificada en el código vayan creciendo su velocidad linealmente. Obviamente podemos cambiar esto por la fórmula de nuestra preferencia.

Ejercicio N°21

- *Crear un componente "EjemploOleadas1" que instancie un prefab cada 2 segundos y que cada vez que instancie uno, acceda a él y ejecute la instrucción Translate del transform del objeto instanciado. Tener en cuenta que la cantidad de unidades a mover depende de la cantidad de objetos que instanció el componente.*

Ejercicio N°22

- *Crear un componente "EjemploOleadas1" que instancie un prefab cada 2 segundos y que cada vez que instancie uno, acceda a él y ejecute la instrucción Translate y Rotate del transform del objeto instanciado. Tener en cuenta que la cantidad de unidades a mover y rotar depende de la cantidad de objetos que instanció el componente.*

Otra forma en la cual podemos subir la dificultad en vez de subir la velocidad del enemigo, seria generar enemigos cada vez más rápido usando el siguiente código.

```
public class Generador : MonoBehaviour
{
    private int cantidadCreada;
    public GameObject prefabEnemigo;
    public float tiempo;

    public void Awake()
    {
        Invoke("CrearEnemigo", 5);
    }

    public void CrearEnemigo()
    {
        GameObject enemigo = (GameObject)Instantiate(prefabEnemigo, transform.position, transform.rotation);
        MovimientoEnemigo movimientoEnemigo = enemigo.GetComponent<MovimientoEnemigo>();
        if (tiempo > 1)
        {
            tiempo -= 1;
        }
        Invoke("CrearEnemigo", tiempo);
    }
}
```

Esta vez, en vez de usar InvokeRepeating, el cual repite a una frecuencia constante, nosotros la primera vez usamos un Invoke que tarda 5 segundos, dándole tiempo al jugador para que tenga un momento de paz antes de comenzar la pelea. Luego de 5 segundos creamos el primer enemigo y volvemos a ejecutar el Invoke, esta vez pasándole por parámetros una variable de tiempo. El código va decrementando dicha variable en 1 cada vez que se crea un enemigo, siempre y cuando el tiempo sea mayor a un mínimo, en este caso 1, así evitando que se empiecen a juntar demasiado los enemigos, o que la variable sea menor a 0, o sea, negativa. De esta forma hacemos que el tiempo entre creación de enemigo y enemigo vaya decrementando progresivamente.

Un último detalle es que podríamos poner varios generadores en diferentes lugares con diferentes tiempos de repetición y de inicio, haciendo que el tiempo que tarda en generar el primer enemigo sea una propiedad. Al variar eso y con muchos generadores podemos dar una sensación de aleatoriedad.

Ejercicio N°23

- *Crear un componente "EjemploOleadas2" que instancie un prefab cada X cantidad de segundos. Dicha cantidad debe decrementar a medida que se instancien más y más enemigos*

Ejercicio N°24

- *Crear un componente "EjemploOleadas3" que instancie un prefab cada X cantidad de segundos. Dicha cantidad debe decrementar a medida que se instancien más y más enemigos. También los objetos que instancia deben aparecer en posiciones diferentes como en el ejercicio anterior.*

Propiedades

El último detalle a recalcar es que todo lo que vean que se puede convertir en una propiedad, conviértanlo. Si ven un numero en el código que tranquilamente podría cambiarse por otro y hacer que el componente sirve en otra situación, háganlo una propiedad.

Por ejemplo, tiempos de instanciación de enemigos, velocidad de enemigos, cantidad de daño que ejerce una bala, cantidad de vida inicial de un enemigo, el nombre del botón a presionar, etc. Esto hace que los componentes se puedan reutilizar en otras situaciones, haciendo un buen uso de sus configuraciones y evitando complicar las cosas con muchos componentes cuyo código hace cosas similares.

Ejercicio N°25

- *Modificar los componentes hechos en los ejercicios anteriores para que tengan las propiedades necesarias para hacerlos completamente configurables.*

Bibliografía utilizada y sugerida

Recursos Online

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 crear managers.*
2. *Cómo detectar la condición de victoria y derrota.*
3. *Cómo simplificar el código.*
4. *Cómo mejorar el Feedback de nuestro juego.*

Lo que viene:

Comenzaremos a ver el apartado gráfico del juego.