**PROGRAMAR EN UNITY**

**Programar con componentes**

La filosofía de Unity es la composición por sobre la herencia. Esto quiere decir que para conseguir una funcionalidad compleja se prefiere crear las partes por separado y luego juntarlas a tener una clase gigantesca que englobe la funcionalidad completa ella sola.

Debido a su enfoque en esta filosofía, Unity tiene un sistema de GameObjects y componentes. Los GameObjects son simplemente contenedores de componentes, y lo que ellos representen estará dado por estos. Por ejemplo para representar una cámara que observe el mundo en Unity, creamos un GameObject que contenga un componente de posición y un componente de cámara. En conjunto estos dos componentes, agrupados bajo el mismo GameObject, representan una cámara en algún lugar del juego.

**Nota:** El término GameObject será utilizado de manera intercambiada con el término “objeto”. En el contexto de estos apuntes significan exactamente lo mismo.

**Componentes de Física**

Unity ofrece montones de componentes distintos, pero se puede argumentar que los más importantes son los relacionados con física. Los siguientes componentes forman la base del sistema de física de Unity, y casi todos los juegos los utilizan en algún grado.

Los 2 tipos de componentes que se mencionan a continuación existen en sus variantes para juegos 2D y 3D. Los componentes de una categoría no interactúan con los componentes de la otra categoría. Los nombres de estos componentes son los mismos con la única diferencia de que dicen 2D al final si son para juegos en 2 dimensiones. Por ejemplo, el Rigidbody tiene una versión llamada Rigidbody2D.

**Rigidbody**

El componente rigidbody, al ser añadido a un GameObject, hace que este sea capaz de recibir fuerzas y reaccionar a ellas. Estas fuerzas incluyen la fuerza de gravedad, y podemos notar inmediatamente que al añadirle este componente a un GameObject este va a caer al ponerle play al juego.

**Masa:** La masa puede ser vista como la inercia de un objeto físico. Mientras más masa tenga, más fuerza será necesaria para moverlo.

**Kinematic:** Si un Rigidbody es kinemático significa que no será afectado por las fuerzas que se apliquen a él, pero este si será capaz de golpear y desplazar a otros objetos.

**Drag/AngularDrag:** Puede ser entendido como la resistencia que el “aire” opondrá al movimiento del objeto. Si un objeto que está cayendo abre un paracaídas, por ejemplo, podemos aumentar el drag para simular el efecto. El angular drag es lo mismo pero aplicado al movimiento de rotación de un objeto.

**Collision Detection:** Define qué método utilizará Unity para detectar colisiones con este objeto. La colisión de tipo discreta es la más económica en términos de poder computacional, pero si la utilizamos en objetos que viajan a velocidades muy altas puede fallarnos y hacer que el objeto ignore ciertas colisiones. Las colisiones continuas evitan que esto ocurra pero necesitan un mayor gasto de recursos del computador.

**Colliders**

Un Rigidbody por sí solo no logra hacer que nuestros objetos colisionen con otros. Para que los objetos se afecten físicamente unos a otros deben tener colliders.

Los colliders son una gran familia y vienen en muchas formas. Todos representan algún tipo de volumen de colisión. Puede ser una caja, una esfera o una cápsula, por ejemplo.

**trigger:** Esta propiedad transforma al collider en un volumen que en vez de colisionar y afectar con una fuerza a los objetos que lo golpeen, los deja pasar a través de él. Sin embargo, la “colisión” entre los objetos sí es detectada, y por lo tanto son muy útiles para detectar cuando el jugador llega a un checkpoint o cruza una puerta. Se verá cómo detectar estos eventos de colisión más adelante en este mismo documento.

**Componentes de Renderizado**

Son una familia de componentes cuya función es dibujar algo en la pantalla, ya sea una imagen plana para juegos 2D o un modelo tridimensional para los juegos 3D.

**SpriteRenderer**

El SpriteRenderer es el componente de renderizado más importante para juegos 2D. Al poner un SpriteRenderer en un objeto y asignarle un sprite a este, inmediatamente podrá verse en la pantalla ese sprite en la posición del objeto.

**MeshRenderer**

El MeshRenderer es el encargado de dibujar en la pantalla modelos 3D que no cambien de forma. Esto es cualquier cosa estática como un grifo, un árbol o un edificio. A diferencia del SpriteRenderer necesitamos asignarle 2 cosas a este componente para poder dibujar un modelo 3D: Un modelo de alambres (llamado mesh) y un material (una sábana que cubra este modelo de alambres para que sea visible).

**LineRenderer**

Solo para ilustrar el hecho de que hay más tipos de renderers mencionamos este. El LineRenderer se utiliza en juegos 2D y 3D. Es un componente encargado de dibujar líneas en la pantalla. Para definirlo necesitamos asignarle parámetros que definan cómo se verá esta línea, qué grosor tendrá y por qué puntos del mundo va a ser dibujada.

Durante el curso se estudiarán muchos componentes distintos, pero los mencionados incluyen los más utilizados y deberían darles una buena noción de qué es un componente y cómo se utilizan.

**Mono Behaviours y Funciones de Eventos**

Los componentes que nos ofrece Unity nos solucionan un montón de problemas, pero nunca un juego va a poder realizarse sin tener que crear algún componente de manera manual. Por ejemplo, Unity no ofrece un componente para que un personaje salte al apretar espacio, ni un componente de granada que cause una explosión. Esos tenemos que crearlos nosotros a mano, y es aquí donde tenemos que saber programar en C#.

Un componente creado por nosotros se conoce como un Script, y es técnicamente una clase que hereda de MonoBehaviour. Al heredar de MonoBehaviour estamos creando una clase cuyas instancias pueden añadirse como componentes a GameObjects.

Cuando creamos un script desde el editor de Unity automáticamente se nos generan unas cuantas líneas de código. En particular se nos genera una función Start y una función Update. Estas funciones pertenecen a una serie de métodos conocidos como funciones de eventos.

Las funciones de eventos se comportan como cualquier otra función, pero tienen la particularidad de que el motor de Unity las llama de manera automática cuando ciertos eventos ocurren. Por ejemplo la función Start es llamada una sola vez cuando el objeto se activa por primera vez durante el juego. A continuación se encuentra un listado y una descripción de las funciones de eventos más importantes de Unity. Basta con implementar cualquiera de estas funciones en un Script para que Unity las ejecute automáticamente cuando correspondan.

**Nota:** Las funciones de eventos relacionadas con colisiones existen en versiones 2D y 3D. Las versiones 2D se diferencian de las 3D porque tienen el sufijo “2D” al final del nombre de la función y al final de sus tipos de parámetros. Por ejemplo, la función “OnCollisionEnter (Collision collision)” en 2D sería: “OnCollisionEnter2D (Collision2D collision)”.

**Awake():** La función Awake es llamada en cada Script apenas se carga la escena. No importa si el script está desactivado en el inspector, la función Awake será llamada si o si. Esta es la primera función que es llamada en un script y, por lo tanto, es ideal para inicializar nuestras variables.

**OnEnable():** On enable es llamada cada vez que un objeto pasa de estar inactivo a activo. Es decir, si yo desactivo un objeto en el inspector de Unity y lo vuelvo a activar, se va a llamar la función OnEnable de todos los scripts que se encuentren en él. También se ejecuta este evento/función cuando un objeto es creado por primera vez (como cuando cargamos una escena). OnEnable es llamada después de la función Awake. No se utiliza OnEnable para inicializar los objetos porque esta función puede ser llamada más de una vez si el objeto se desactiva y activa en algún punto durante el juego. Si queremos que un objeto ejecute alguna lógica cada vez que se active este es el lugar para hacerlo.

**OnDisable():** Es la contraparte de OnEnable (ver arriba). Se ejecuta cada vez que el objeto pasa a estar desactivado. Si queremos que algo ocurra cuando un objeto deja de estar activo este es lugar para hacerlo.

**Start():** Se ejecuta cuando un objeto pasa a estar activo por primera vez durante el juego (por ejemplo cuando se carga una escena). A diferencia de Awake, si un GameObject está inactivo cuando una escena comienza, NO se llamará a su función Start hasta que este se active por primera vez. Una vez que se ejecutó el Start de un objeto no se volverá a ejecutar de nuevo. Start es llamado luego de Awake y OnEnable.

**Update():** Esta función se ejecuta repetidamente, una y otra vez, lo más rápido que el computador lo permita, en cada Script que creemos. Si queremos que ocurra un cambio en el tiempo necesitamos escribir lógica en este tipo de funciones. Por ejemplo, si quiero que un objeto comience a brillar cada vez más, lo que hago es aumentarle el brillo en Update. El objeto se irá haciendo así cada vez más brillante. Esta función es especialmente importante para leer input del jugador. Puedo consultar en cada Update si el jugador está apretando una tecla, por ejemplo, y hacer que el script reaccione a este input de alguna manera (saltando, caminando, explotando, etc). No podemos estar seguros de cuántas veces por segundo esta función será llamada, eso dependerá de la potencia de cada computador.

**FixedUpdate():** A diferencia de Update (ver arriba), FixedUpdate se llama cada intervalos regulares de tiempo, independiente de la máquina que esté corriendo nuestro juego. Para entender esta función hay que mencionar que la física de Unity también se resuelve en intervalos regulares. Esto permite que los objetos se comporten físicamente igual en todos los computadores que corran nuestro juego. Si resolvieramos la física en intervalos irregulares, cada vez que jugáramos el juego las piedras rebotarían distinto, por ejemplo. FixedUpdate se ejecuta a intervalos regulares sincronizados con los update de física del juego, y por lo tanto es el perfecto lugar para hacer cambios en el movimiento y las fuerzas que le aplicamos a los objetos.

**OnCollisionEnter(Collision collision):** Esta función es llamada cuando un objeto entra en colisión con otro. El parámetro collision contendrá información relativa al golpe que se acaba de efectuar, e información del objeto que fue golpeado. Si queremos que un objeto reaccione de alguna forma a un golpe (por ejemplo aplastandose o cambiando de color), escribimos esa lógica dentro de esta función.

**OnCollisionStay (Collision collision):** Una vez que un objeto colisiona con otro es posible que se mantengan en contacto un tiempo. Esta función será llamada repetidamente mientras los objetos estén tocándose. Por ejemplo, si queremos que un personaje baile mientras esté sobre una pista de baile, podríamos escribir la lógica del baile en esta función.

**OnCollisionExit (Collision collision):** Esta función se ejecuta cuando un objeto deja de estar en contacto con otro. Hacer que un personaje deje de bailar cuando salga de la pista de baile es un ejemplo.

**OnTriggerEnter (Collider other):** Los colliders, como se menciona más arriba en este documento, tienen una propiedad booleana llamada “isTrigger”. Cuando en una colisión entre dos objetos está involucrado al menos un collider que sea un trigger, esta función es llamada en lugar de OnCollisionEnter. Por ejemplo, si queremos que el juego se guarde cuando el jugador atraviese una puerta, este es un buen lugar para hacerlo. Es muy importante notar que el tipo de parámetro no es Collision, sino Collider. Este parámetro Collider llamado “other” contiene una referencia al otro collider involucrado (además del collider de este mismo objeto).

**OnTriggerStay (Collider other):** Similar a OnCollisionStay, pero cuando un trigger está involucrado en la colisión. Esta función será llamada repetidamente mientras los colliders involucrados estén solapados entre ellos.

**OnTriggerExit (Collider other):** Similar a OnCollisionExit, pero cuando un trigger está involucrado en la colisión. Esta función será llamada cuando los colliders involucrados dejan de estar solapándose.

**OnMouseDown():** Si el objeto en cuestión tiene un collider (sea trigger o no) y el jugador aprieta el mouse sobre él, esta función será llamada. Esto puede ser utilizado para seleccionar o activar objetos con el mouse.

**Raycasts**

En un videojuego es muy común querer saber qué cosas se encuentran en una dirección en particular. En otras palabras, es muy común preguntarse: “Si lanzara un rayo en esa dirección, ¿Que objetos golpearía con él?”. Por ejemplo, esto puede utilizarse para saber si un enemigo es capaz de ver al jugador: lanzo un rayo desde sus ojos hacia el jugador y, si el rayo golpea una pared, entonces considero que el jugador está oculto para el enemigo.

La manera en que hacemos esto por código es llamar a una función que lanza el rayo, dándole como parámetros desde donde y hacia donde queremos lanzarlo. De una u otra manera obtenemos información acerca del golpe del rayo y la almacenamos en un objeto especial. A través de este objeto podemos consultar qué objeto golpeamos, a qué distancia y en qué punto, y actuar en consecuencia.

Cuando lanzamos un rayo lo hacemos de manera instantánea, es decir, obtenemos inmediatamente la información acerca de lo que golpeamos. Es fácil pensar que desde el momento en que lanzamos el rayo va a pasar un tiempo antes de que el rayo intersecte algún objeto. Cuando decimos “instantáneo” queremos decir que nada va a ocurrir en nuestro juego hasta que el rayo sea resuelto.

Lanzar rayos en 2D y en 3D es lo suficientemente distinto como para ameritar una sección para cada uno de ellos.

**Raycasts en 3D**

El siguiente código describe cómo se lanza un rayo en 3 dimensiones. Se describe a continuación en detalle qué es lo que estamos haciendo.

RaycastHit hitInfo;

if (Physics.Raycast (puntoInicio, vectorDireccion, out hitInfo))

{

// Aca ejecutamos la lógica que ocurre cuando el rayo golpea algo.

}

Lo primero que hacemos es crear un objeto de tipo RaycastHit, llamado hitInfo. Este objeto guardará dentro de él toda la información relativa a qué ocurrió con nuestro rayo, en caso de que haya habido alguna colisión.

Luego de esto tenemos la llamada a la función Raycast. Podemos darnos cuanta de que, dado que se encuentra dentro de un “if”, el método retorna un booleano. Raycast retornará falso si el rayo no golpeó nada y verdadero si hubo alguna intersección con algo.

La función Raycast necesita saber desde donde y hacia donde lanzar el rayo. Esta información se la damos con parámetros. Le decimos cual es el punto de origen del rayo y qué vector representa la dirección hacia la que tiene que ser lanzado.

El último parámetro de la función es un objeto de tipo RaycastHit. El objeto que pongamos acá va a ser poblado automáticamente con la información relativa a la intersección del rayo. La palabra clave “out” se escribe en este caso para indicar que el objeto RaycastHit que entrará a esta función habrá cambiado luego de que la función se ejecute. No se preocupen demasiado por entender “out” en este momento.

Finalmente, dentro del if podemos escribir qué es lo que queremos que ocurra cuando el rayo golpee algo, y podemos hacer consultas a hitInfo para decidir qué hacemos. Recordemos que si estamos dentro del if significa que el rayo intersectó con algo, pues la función Raycast retornó true.

**Raycasts en 2D**

Partamos viendo la estructura del código de un Raycast en 2D

RaycastHit2D hitInfo = Physics2D.Raycast (puntoInicio, vectorDireccion);

if (hitInfo.collider != null)

{

// Aca ejecutamos la lógica que ocurre cuando el rayo golpea algo.

}

La principal diferencia con los Raycasts en 3D es que la función Raycast en 2D no retorna un booleano, sino que retorna directamente el objeto que contiene la información del golpe del rayo. Es por esto que no tenemos un parámetro out como en el caso anterior.

Por supuesto, dado que no retorna un booleano, necesitamos otra forma de saber si el rayo hizo contacto con algo. La manera más típica de verificar esto es consultar si la información del rayo (hitInfo) contiene algo en su variable de collider. Esta variable guarda el collider con el que el rayo intersectó, en caso de que haya golpeado algo, y es nula (null) si el rayo no intersectó nada. Por lo tanto si collider es distinto de null significa que sí tuvimos un contacto.

**Notas adicionales sobre Raycasts**

En los ejemplos anteriores se llamó a las funciones de raycast con 2 o 3 parámetros para definir el rayo lanzado, pero se pueden añadir más parámetros para especificar algunas cosas adicionales acerca del rayo. Los métodos de raycast aceptan parámetros opcionales para definir el largo del rayo (que en su defecto es inifinito), los layers que el rayo intersectará y si el rayo considerará las intersecciones con triggers o no.