«Talento Tech»

Desarrollo de Videojuegos

Unity 3D

Clase 05

Buenos Aires Ciudad

Clase N° 5 | Delegados y Eventos I

Temario:

- Introducción a Delegates: qué son y cómo funcionan.
- Uso básico de Delegates & Events en C#.
- Aplicaciones iniciales: llamados a funciones específicas desde eventos.

Objetivos de la clase

Comprender Delegates y Events en C#: qué son y cómo funcionan.

- Explicar qué son los Delegates y Events en C#.
- Entender las diferencias clave entre Delegates y Events en términos de funcionalidad y restricciones.

Implementar el uso básico de Delegates y Events en C#.

- Configurar scripts que utilicen Delegates y Events para comunicar cambios.
- Configurar y disparar Events para manejar acciones específicas en un proyecto de Unity.

¿Qué son los Delegates y los Events?

Son herramientas que usaremos para comunicar de manera más eficiente nuestro juego. En vez de estar chequeando constantemente si algo sucedió o realizar una cadena de acontecimientos uno detrás de otro, podremos crear un Evento que avisará a todos los Scripts necesarios si esto llega a suceder.

Delegates:

Un **delegate** es un tipo de dato que puede almacenar referencias a métodos que cumplen con una firma específica. Pensá en ellos como contenedores o "puentes" para métodos: puedes usar un delegate para invocar un método sin saber exactamente cuál es en tiempo de compilación. Esto los hace muy útiles para ejecutar métodos de manera flexible y dinámica.

- **Firma específica**: Los métodos que un delegate puede almacenar deben coincidir en el tipo de retorno y los parámetros.
- **Seguridad**: A diferencia de punteros a funciones en otros lenguajes como C/C++, los delegates son seguros porque el compilador verifica las firmas de los métodos asociados.
- **Usos comunes en juegos**:
    - Ejecutar diferentes acciones en respuesta a un evento del juego.
    - Configurar lógica personalizada para IA o enemigos.

Ejemplo simple (Explicado):

Un delegate que ejecuta diferentes comportamientos dependiendo del contexto:

```
   public class DelegateExample : MonoBehaviour
{
private delegate void ActionDelegate();
   private ActionDelegate OnAction;

   private void Start()
   {
      OnAction = Jump; // Asigna el método Jump al delegate
      OnAction(); // Invoca Jump
   }

   private void Jump()
   {
      Debug.Log("The player jumps!");
   }
}
```

Paso a paso:

1. Declaración del delegate

```
public delegate void ActionDelegate();
```

delegate: Esta palabra clave define un tipo que puede almacenar referencias a métodos.
ActionDelegate: Es el nombre del tipo de delegate.
void y (): El delegate está diseñado para almacenar métodos que no tienen parámetros ni devuelven un valor.

En otras palabras:
ActionDelegate es como un "contenedor" que podrá guardar referencias a métodos con la firma void NombreMetodo().

2. Declaración de una variable de tipo delegate

```
public ActionDelegate OnAction;
```

- Aquí se declara una variable pública llamada OnAction que es del tipo ActionDelegate.
- Esta variable puede almacenar métodos cuya firma coincida con la definida por el delegate.

Por ahora, OnAction no tiene ningún método asignado.

3. Método Start

```
private void Start()  {
      OnAction = Jump; // Asigna el método Jump al delegate
      OnAction(); // Invoca Jump
   }
```

3.1. Asignación del método al delegate

```
OnAction = Jump; // Asigna el método Jump al delegate
```

- **Jump**: Es un método que coincide con la firma de ActionDelegate (no tiene parámetros y devuelve void).
- Esta línea asigna el método Jump a la variable OnAction.
  Ahora, OnAction es una referencia al método Jump.

3.2. Invocación del delegate

```
OnAction(); // Invoca Jump
```

- Aquí se invoca el delegate como si fuera un método.

- Esto llama al método que fue asignado a OnAction, que en este caso es Jump.

4. Método Jump

```
private void Jump()
{
    Debug.Log("The player jumps!");
}
```

Este método imprime un mensaje en la consola que dice: "The player jumps!".
Dado que fue asignado al delegate OnAction, se ejecutará cada vez que el delegate sea invocado.

Funcionamiento completo del script:

1. **Inicio del script (Start)**:
    - Se asigna el método Jump al delegate OnAction.
2. **Invocación del delegate**:
    - OnAction() se ejecuta, lo que llama al método Jump.
3. **Resultado**:
    - Aparece en la consola el mensaje: "The player jumps!".

¿Por qué es útil este enfoque?

Este ejemplo puede parecer simple, pero la verdadera utilidad de los delegates radica en su flexibilidad:

1. **Asignar diferentes comportamientos dinámicamente**: Puedes cambiar el método asociado al delegate en tiempo de ejecución.

```
OnAction = Run; // Cambia el comportamiento a otro método llamado Run
```

- **Llamar a múltiples métodos**: Los delegates pueden almacenar referencias a varios métodos al mismo tiempo si usas += para agregar métodos:

```
OnAction += Jump;

OnAction += Run;

OnAction(); // Ejecuta Jump y luego Run
```

- **Desacoplamiento del código**: Permite que diferentes partes del sistema interactúen sin depender directamente unas de otras.

Events:

Un **event** es una extensión de los delegates que se utiliza para comunicar cambios o sucesos. Funcionan como un sistema de notificación: una clase puede "emitir" eventos y otras clases pueden "escuchar" y reaccionar a ellos.

- **Abstracción**: Los eventos se basan en delegates, pero con mayor control sobre quién puede invocarlos.
- **Desacoplamiento**: Los eventos permiten que las clases interactúen sin conocerse directamente. Esto mejora la modularidad y el mantenimiento del código.
- **Restricción**: Solo la clase que declara el evento puede invocarlo. Otros objetos solo pueden suscribirse o desuscribirse.

Ejemplo simple:
Un evento que notifica cuando un jugador recoge un ítem:

```
public class Player : MonoBehaviour

{

    public delegate void ItemCollectedDelegate(string itemName);

    public event ItemCollectedDelegate OnItemCollected;

    public void CollectItem(string itemName)

    {

        Debug.Log($"Collected: {itemName}");

        OnItemCollected?.Invoke(itemName); // Notifica a los suscriptores

    }

}
```

Explicación paso a paso

1. Declaración del delegate:

```
public delegate void ItemCollectedDelegate(string itemName);
```

- **delegate**: Define un nuevo tipo que puede referenciar métodos con una firma específica.
- **ItemCollectedDelegate**: Es el nombre del tipo del delegate.
- **Firma del método**:
    - void: No devuelve ningún valor.
    - (string itemName): Requiere un argumento de tipo string llamado itemName.

En resumen:
ItemCollectedDelegate puede almacenar métodos que acepten un parámetro de tipo string y no devuelvan nada.

2. Declaración del evento

```
public event ItemCollectedDelegate OnItemCollected;
```

- **event**: Define un evento basado en el delegate ItemCollectedDelegate.
- **OnItemCollected**: Es el nombre del evento.
- **Propósito**:
    - Este evento permitirá a otros scripts suscribirse para ser notificados cuando ocurra una acción (en este caso, cuando el jugador recoja un ítem).

Nota importante:
Un evento es más seguro que usar directamente un delegate, ya que **solo la clase que declara el evento puede invocarlo** (en este caso, solo la clase Player puede ejecutar OnItemCollected). Sin embargo, otros scripts pueden suscribirse y reaccionar cuando el evento se dispare.

Método CollectItem.

```
public void CollectItem(string itemName)  {
    Debug.Log($"Collected: {itemName}");
    OnItemCollected?.Invoke(itemName); // Notifica a los suscriptores  }
```

- **OnItemCollected**: Es el evento que hemos definido anteriormente.
- **?.**: Comprueba si el evento tiene suscriptores antes de invocarlo (esto evita errores si no hay nadie suscrito).
- Si nadie está suscrito, no hace nada.
- Si hay métodos suscritos, se ejecutan con el argumento proporcionado.
- **Invoke(itemName)**: Llama a todos los métodos suscritos al evento, pasando el parámetro itemName como argumento.

Funcionamiento del código

1. **El jugador recoge un ítem**:
    - Llamar al método CollectItem imprime el nombre del ítem en la consola y dispara el evento OnItemCollected.
2. **El evento notifica a los suscriptores**:
    - Todos los métodos suscritos al evento OnItemCollected se ejecutan con el nombre del ítem (itemName) como parámetro.

Ejemplo extendido: Cómo usar este evento

En otro script, suscribimos un método al evento.

```
private void Start()
{
    Player player = FindObjectOfType<Player>();
    player.OnItemCollected += HandleItemCollected; // Suscribirse al evento
}

private void HandleItemCollected(string itemName)
{
    Debug.Log($"GameManager: Player collected {itemName}");
}
```

Esto hará que cuando el evento “OnItemCollected” sea invocado, este llame todas las funciones inscriptas en él, como en este caso sería “HandleItemCollected”.

Para terminar de entender sus diferencias:

Característica	Delegates	Events
Invocación	Cualquiera puede invocar un delegate si lo tiene como referencia.	Solo la clase que declara el evento puede invocarlo.
Seguridad	Menos restrictivo; otros objetos pueden modificar la referencia.	Más seguro; restringe la invocación solo al propietario.
Uso principal	Ejecutar métodos dinámicamente.	Notificar cambios de estado o sucesos.
Modularidad	Útil, pero puede acoplar clases si no se controla.	Promueve el desacoplamiento entre clases.

Ejemplo Aplicado

Crearemos una situación de PickUp sencilla como ya venimos acostumbrados, pero le añadiremos la idea de eventos. Para que al agarrar el objeto, suceden distintas situaciones.

Haremos 3 Scripts:

1) El dueño del evento. EL Script del Item que detectara cuando es “agarrado”.

2) Un Script para el player que dirá un diálogo al agarrar un objeto.

3) Un Script de GameManager que cambiara una variable “Score”/Puntaje cuando agarra al objeto. No olviden colocarlo en un “EmptyObject” en la **escena**.

1) Item

```
private string name1;
private int value = 5;
private delegate void Format1(string item, int value);

public static event Format1 myPickEvent;

void Start()  {
   name1 = gameObject.name;
}

private void OnTriggerEnter(Collider other)  {
   if (other.CompareTag("Player"))  {

         FuiAgarrado(name1);
   }
}

private void FuiAgarrado(string a)
{
  if (myPickEvent != null)
  {
    myPickEvent.Invoke(name1, value);
    Debug.Log($"Soy {name1} y me agarraron");
    Destroy(gameObject, 0.1f);
   }
}
```

Con este código estamos armando un ítem sencillo que posee el evento “myPickEvent” con el formato del delegate “Format1”. Este será **Static** para poder acceder a él sin importar si nuestros “pickUps” desaparecen.
El **ítem** también tendrá una variable de “nombre” y una de puntaje para, al ser llamado el evento, transmitir qué objeto es y cuanto vale.

1. **Start**: Asignamos el nombre de nuestro objeto.
2. **OntriggerEnter:** Planteamos la situación: “Si toco al player”, llamo al método.
3. **FuiAgarrado:** Verificar si el evento es **distinto** de null, es decir si NO está vacío. Si es así, invoco al evento, envio un mensaje por consola y me destruyo.

2) Player:

```
void Start()  {
    Clase5PickEvent.myPickEvent += PickedUp;
}
private void PickedUp(string item, int value)  {
    Debug.Log($"Agarre {item} chaval. Junte {value} de oro");
}
```

1. En el player solamente crearemos una función que “diga” el objeto que agarre y cuanto vale, como si fuera un diálogo.
2. Lo más importante, en el Start() **añadimos** el método a nuestro Event.

3) GameManager

```
void Start()  {
    Clase5PickEvent.myPickEvent += AddScore;
}
private void AddScore(string item, int value) {
    score += value;
    Debug.Log($"Mi puntaje ahora es de {score}");
}
```

Este código estará colocado en un EmptyObject en la Scene, cuya función será empezar a manejar partes del juego. En este caso, el puntaje.

Siendo similar al del Player, crearemos una función/método que sume a nuestra variable “Score” y muestre su valor en consola. Recuerden implementar estas opciones con el UI correspondiente.

De esta manera, terminaremos teniendo un evento de “PickUp” que puede alertar a varios objetos. Podemos imaginar hacerlo con el UI, alguna Quest de recolección, municiones o lo que se nos ocurra.

Otro día en TalentoLab:

El cliente está cada vez más emocionado con los avances de Nexus, pero han identificado un problema crítico: los eventos dentro del juego carecen de interconexión. Las acciones del jugador no desencadenan reacciones significativas en el mundo del juego. Por ejemplo, al recolectar monedas o alcanzar un objetivo, no hay un sistema que informe al resto del juego sobre lo que sucede.

Para resolver esto, el cliente quiere que TalentoLab implemente un sistema de **Events** y **Delegates**, una herramienta poderosa para conectar los elementos del juego de manera eficiente. Este sistema será clave para asegurar que las acciones del jugador tengan un impacto tangible y para garantizar que el juego se sienta cohesionado.

Ejercicios prácticos:

La solicitud del cliente:
El cliente necesita ver ejemplos claros de cómo los eventos del juego se comunican entre sí, cómo recolectar objetos, desbloquear zonas o responder a la derrota del jugador. ¡Es momento de convertir Nexus en un mundo lleno de reacciones dinámicas!

(*Tengan en cuenta que esta situación será resuelta entre esta clase y la siguiente*)

¡Roberta puede ver tu gran potencial! Así que te asignará la siguiente tarea para probar tus capacidades!

Sistema de "Game Over" con Eventos:
En Nexus, la muerte del personaje debe ser un momento dramático que refuerce la conexión emocional del jugador con el mundo del juego. TalentoLab quiere que al morir el personaje, se active un evento que:

- Muestre una pantalla de "Game Over" que detenga el flujo del juego. *(Tip: LoadScreen o crear un Screen(GameObject) desactivado durante el juego*)
- Reproduzca efectos visuales o sonoros que subrayen la gravedad del momento.

Desbloqueo de una zona secreta al recolectar monedas:
En Nexus, las zonas secretas son un elemento clave que recompensa la exploración y la habilidad del jugador. TalentoLab quiere que, al recolectar un número determinado de monedas, se active un evento que nos teletransporte a un lugar desconocido.
(*Tip: Tranform.Position = lugarsecreto.transform.position*)

Materiales y recursos adicionales.

Delegates:
https://learn.unity.com/tutorial/delegados#

Events:
https://learn.unity.com/tutorial/eventos-w#5e419557edbc2a0a62170fe6

Preguntas para reflexionar.

1. ¿Sería cómodo crear reacciones múltiples que dependan de una situación sin usar Events?
2. ¿Qué eventos podríamos crear en nuestro juego?

Próximos pasos.

En la próxima clase seguiremos aprendiendo sobre Eventos creando un Event Manager que nos permita coordinar los distintos momentos de nuestro proyecto.

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad