Delegates and Events Lab

UNEVENTFUL CONVERSATIONS

Introduction

This lab will go over what C# delegates and events are, why to use them, where to use them, and how to use them. This is a relatively high level programming topic so please ask any question you may have and a facilitator will help. Now, Unity has something similar called UnityEvents. This lab will not cover them because: (1) knowing delegates and events will allow an easy transfer to working using UnityEvents, (2) UnityEvents are slower, and (3) UnityEvents are meant to help out designers by providing the ability to be edited from the editor while delegates and events are meant for the programmers to use. This information may change so if you know one of the points is incorrect, feel free to point it out. NOTE: C# delegates and events are a C# feature! They are NOT a Unity feature. UnityEvents ARE a Unity feature.

Throughout the course of this tutorial, you will add certain lines to some code in order to complete a game so that it becomes playable.

The Game

The goal of this game is to shoot down as many enemies as possible within the time limit. You move your character by holding down the left mouse button and dragging it around. The more ammunition you hold, the slower you move. You can only attack while stationary and in order to attack, use spacebar.

Definitions and Explanations

Before we begin, we need to understand what delegates and events are, why we should use them, and where might we use them. Keep in mind that this is by no means everything about delegates and events so not everything in this document is exactly right. The definitions are a little different from the actual definition so that they make more sense for the purpose we will use them.

What are delegates? Special types. Delegates are types just like your usual int, bool, string, etc. The difference is that these are custom types that you as the programmer create.

What are events? Variables that are declared using a delegate type.

Let's start by creating some delegates and events; delegates have specific signatures that must be followed so that nothing complains.

```
public delegate returnType delegateName(parameters);
```

Two potential delegates are (I like adding "Delegate" at the end of the name but it is not required)

```
public delegate void StartGameDelegate();
public delegate void IntFloatDelegate(int a, float b);
```

Given these delegates, potential events are

```
public static event StartGameDelegate StartingTheGameEvent;
public static event IntFloatDelegate UpdatePlayerScoreEvent;
public static event IntFloatDelegate DistanceFromEvent;
```

Now what are these things good for? I'm glad you asked!!

Up until now, certain things would require very weird and annoying solutions. For example, for our game, as soon as a round ends, we want our player gone, the score board updated, enemies and bullets deleted, and the UI showing up. What I assume you would try to do is have a reference to everything inside your game manager and when the game ends, you would enable/disable everything. This could get very messy, very quickly. With events, all we have to do is make a single call in the game manager.

Think about it this way. Our game manager is a guy with a megaphone. He has a crowd of anywhere from zero to a million people. He does not really care how many people are there because he will yell the same thing regardless. Now, anybody who is there, can hear him perfectly.

Let's assume we have three scripts: GameManager, PlayerGraphics, and EndGameUI. (Assume each class has more code that is not shown)

```
public class GameManager : MonoBehaviour
{
  public delegate void EndGameDelegate();
  public static event EndGameDelegate EndGameEvent;

  // The number of seconds left until the round ends
  private float m_TimeToRoundOver;

  // A boolean that lets us know when the round ended
  private bool m_RoundOver;

  private void Awake()
  {
    m_TimeToRoundOver = 60;
    m_RoundOver = false;
  }

  private void Update()
  {
    if (m_TimeToRoundOver > 0)
      m_TimeToRoundOver -= Time.deltaTime;
    else if (!m_RoundOver)
    {
      m_TimeToRoundOver = 0;
      m_RoundOver = true;
      if (EndGameEvent != null)
        EndGameEvent();
    }
  }
}
```

```
public class PlayerGraphics : MonoBehaviour
{
  private GameObject m_Graphics;

  private void OnEnable()
  {
    GameManager.EndGameEvent += TurnOffGraphcis;
  }

  private void OnDisable()
  {
    GameManager.EndGameEvent -= TurnOffGraphics;
  }

  private void TurnOffGraphics()
  {
    m_Graphics.SetActive(false);
  }
}
```

```
public class EndGameUI : MonoBehaviour
{
  private GameObject m_UI;

  private void OnEnable()
  {
    GameManager.EndGameEvent += TurnOnUI;
  }

  private void OnDisable()
  {
    GameManager.EndGameEvent -= TurnOnUI;
  }

  private void TurnOnUI()
  {
    m_UI.SetActive(true);
  }
}
```

Let's take note of a few things:

- PlayerGraphics and EndGameUI both have two Unity functions implemented (OnEnable and OnDisable)
- GameManager is the guy with the megaphone yelling; PlayerGraphics and EndGameUI are two separate people standing in the crowd, listening
- To add a listener, we do

```
ClassName.EventName += FunctionName;
```

- To remove a listener, we do

```
ClassName.EventName -= FunctionName;
```

- We want to always make sure to pair these. If you ever add an event, you MUST remove it as well. Why? Does it make sense for a dead person to be listening to someone? No because the person is dead so they literally cannot listen.
- Delegates have a particular signature. Any function that is added has to have the exact same signature. In our case, EndGameDelegate has a return type of void and zero parameters. Any function that will listen to it must also have a return type of void and zero parameters.
- Finally, notice how before calling EndGameEvent, I check to see if it is null. This is important because if you forget to do this check and it just so happens there are no listeners, Unity will complain that the guy with the megaphone is screaming at an empty crowd.

As you can see, this is very easy to extend and debug because everything is clean and separated. The GameManager has only one task and that is to let everybody who is listening know when the game ends. And to do that, we just use the line

```
EndGameEvent();
```

In this case, once that function is called, PlayerGraphics' TurnOffGraphics and EndGameUI's TurnOnUI will both be called and their code will run.

Now, I would like to present you with another, slightly more complex example. This time we will use the following scripts: AllDelegates, PointOrb, ScoreManager, ScoreText, ScoreEffects, GameManager, and WinnerText. (like before, assume each class has more code that is not shown)
The goal of this particular game is to gather 21 point orbs before any of your opponents.
Unlike the previous example, I will walk you through this example with short descriptions below each code segment.

```
public class AllDelegates
{
  public delegate void IntStringDelegate(int num, string str);
  public delegate void StringDelegate(string str);
}
```

All we did here is create a script that can hold all of our general delegates in one spot. This allows us to use the same delegate signature for multiple events without having to recreate the delegate each time.

```
public class PointOrb : MonoBehaviour
{
  public static event AllDelegates.IntStringDelegate PointAcquiredEvent;

  // How many points an orb is worth
  private int m_Points = 3;

  private void OnCollisionEnter(Collision collision)
  {
    GameObject other = collision.gameObject;
    // Make sure this is a player
    if (other.CompareTag("Player") && PointAcquiredEvent != null)
    {
      PointAcquiredEvent(m_Points, other.name);

      // Make sure this point is removed from the game for now
      gameObject.SetActive(false);
    }
  }
}
```

Over here, we made a small check each time something collides with the orb. If the game object that collided with the orb is a player, we send a message saying which player collided and how many points they should receive.

```
public class ScoreManager : MonoBehaviour
{
  public static event AllDelegates.IntStringDelegate ScoreChangedEvent;

  private Dictionary<string, float> m_PlayersScoresDict;

  private int m_NearEndMark = 15;

  private void Awake()
  {
    m_PlayersScoresDict = new Dictionary<string, float>();
    AddAllPlayersToDict(); // Code for this function not here
  }

  private void OnEnable()
  {
    PointOrb.PointAcquiredEvent += IncreaseScore;
  }

  private void OnDisable()
  {
    PointOrb.PointAcquiredEvent -= IncreaseScore;
  }

  private void IncreaseScore(int amt, string playerName)
  {
    int newScore = m_PlayersScoresDict[playerName] + amt;
    m_PlayersScoresDict[playerName] = newScore;

    if (newScore >= m_NearEndMark && ScoreChangedEvent != null)
      ScoreChangedEvent(newScore, str);
  }
}
```

Each time a player scores a point, the score manager fires its increase score method to increase the score of the appropriate player. If a player starts nearing the end mark, the score manager will start sending messages that the score has changed (it only sends a message *after* receiving a message that a player acquired more points). The idea here is that players know who is winning and by how much as the game reaches its end.

```
public class ScoreText : MonoBehaviour
{
  private string m_TextToDisplay = " just got a score of ";

  private Text m_ScoreText;

  private void OnEnable()
  {
    ScoreManager.ScoreChangedEvent += DisplayScoreText;
  }

  private void OnDisable()
  {
    ScoreManager.ScoreChangedEvent -= DisplayScoreText;
  }

  private void DisplayScoreText(int score, string playerName)
  {
    StartCoroutine(DisplayAndHideText(score, playerName));
  }

  /* If you do not know what this is, then here is what it does:
   * First it changes the text of score text to the appropriate message
   * Then it waits 3 seconds so all of the players have a chance to read
   * Finally, it resets the text to nothing so it doesn't bother anyone
   */
  private IEnumerator DisplayAndHideText(int score, string playerName)
  {
    m_ScoreText.text = playerName + m_TextToDisplay + score;
    yield return new WaitForSeconds(3);
    m_ScoreText.text = "";
  }
}
```

This method simply sets a UI text component so that everyone can read and know immediately who's ahead and how many points they have. It listens to the score manager's event and only updates the UI when it sees that the score manager claims someone had their score change.

```
public class ScoreEffects : MonoBehaviour
{
  private ParticleSystem m_Effects;

  private void OnEnable()
  {
    ScoreManager.ScoreChangedEvent += PlayEffects;
  }

  private void OnDisable()
  {
    ScoreManager.ScoreChangedEvent -= PlayEffects;
  }

  private void PlayEffects(int score, string playerName)
  {
    m_Effects.Play(); // These don't loop so no reason to call Stop()
  }
}
```

Very similar to the previous script. When the score changes, a particle system plays right in front of every player as a way to notify them that someone is getting closer to winning.

```
public class GameManager : MonoBehaviour
{
  public static event AllDelegates.StringDelegate EndGameEvent;

  private void OnEnable()
  {
    ScoreManager.ScoreChangedEvent += DisplayScoreText;
  }

  private void OnDisable()
  {
    ScoreManager.ScoreChangedEvent -= DisplayScoreText;
  }

  private void CheckScore(int score, string playerName)
  {
    if (score >= 21 && EndGameEvent!= null)
      EndGameEvent(playerName);
  }
}
```

When the score of a player reaches 21, the game manager sends out a message that the game is over. There are a lot of scripts listening to this message in preparation to end the game.

```
public class WinnerText : MonoBehaviour
{
  private Text m_Text;

  private void OnEnable()
  {
    GameManager.EndGameEvent += DisplayWinnerText;
  }

  private void OnDisable()
  {
    GameManager.EndGameEvent -= DisplayWinnerText;
  }

  private void DisplayWinnerText(string playerName)
  {
    m_Text.text = playerName + " has won the game!!";
  }
}
```

One of the many scripts listening to see when the game ends. This script will print out to everyone who won the game once the game ends.

Tutorial

Now that we answered the what, the why, and the where, it is finally time to answer the how! And we will begin by trying to make the game actually playable when the “Play” button is pressed.

1. Open up ***Scripts/Countdowns/Countdown.cs***
2. There is a region called Delegates and Events, inside it, add the following lines:

```
public delegate void StartGameDelegate();
public static event StartGameDelegate StartGameEvent;
```

3. Now open up the region called OnDisable and Other Enders. Inside the OnDisable method, add the following lines:

```
if (StartGameEvent != null)
  StartGameEvent();
```

This countdown timer script will send out a message to the game manager once the timer reaches zero. In this case, we do know ahead of time that we only plan to have the GameManager listen in but it does not make sense for a timer to store a reference to the GameManager.

Before we forget, let’s go ahead and actually choose a function in the GameManager script to listen for this event. It just so happens that there is a function named StartGame that has all of the functionality we want.

1. Please open up ***Scripts/GameManager.cs*** if you have not already.
2. In the OnEnable, Set Ups, and Resetters region, tell our GameManager’s StartGame function to listen to Countdown’s StartGameEvent in the OnEnable method. (Incase you forgot how:

```
Countdown.StartGameEvent += StartGame;
```

)

3. The OnDisable and Other Enders has an OnDisable method where you will need to tell the function to stop listening. All you have to do is copy-paste the line right above and replace the plus with a minus.

The final little thing we need to implement is a score counter so that when enemies are killed, it adds points.

1. Begin by opening ***Scripts/Player/Bullet.cs***
2. You will need to add:

```
public delegate void EnemyHitDelegate();
public static event EnemyHitDelegate EnemyDestroyedEvent;
```

In the Bullet's Delegates and Events region.

3. Next up, look at the Collision Methods region and inside the OnParticleCollision method, paste:

```
if (other.CompareTag("Enemy") && EnemyDestroyedEvent != null)
{
  EnemyDestroyedEvent();
  other.SetActive(false);
}
```

4. We are nearly done!! All that's left is to make GameManager's AddPoint function listen. We need to add two lines; one at line at OnEnable and the other at OnDisable. I hope at this point you know what the line is but if you forgot, just scroll up a little bit and make the appropriate replacements. If you still are not sure what to do, feel free to ask one of the facilitators for help.

Reminder for how to play: You can only attack while stationary and in order to attack, use spacebar. In order to move, left click.

Checkoff

- Create a pickup from scratch. Requirements:
    - When picked up, it sends a message to all enemies (that are alive)
    - The message sends a float
    - The velocity of each enemy is scaled by the float
    - After a certain amount of time, the velocity of the enemies returns to normal
- Complete the full tutorial and prove it works by:
    - Play one game and get a few kills before crashing into an enemy
    - Play a second game and get a better score as well as stay alive all the way
    - Show that the scoreboard changes appropriately

- Optional Challenges (you are expected to use delegates and events for these; you can use existing ones if you figure out a way):
    - Make it so whenever an enemy is killed, two are spawned
    - Add a power up (it can be anything you want from a new weapon to a stat modifier that affects you, the enemy, or both)