Delegates and Events

Table of Contents

Lab Overview

In this lab you'll learn about delegates and events, which is a powerful tool in sending messages to several scripts without needing to have references to each script. Imagine that in your game, when the player dies, you want the score board updated, enemies and bullets deleted, and the UI showing up. One way to do this is to have a GameManager that, when the game ends, enable/disable everything --but this can get messy very quickly. With events, you only need to make a single call in the GameManager.

Delegates are special types just like ints, bools, strings, etc. However, delegates are custom types that you as the programmer create.
Events are variables declared using a declared type.

Delegates have a special signature:

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

Two potential delegates are:

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

And given these delegates, potential events are:

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

Now 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();
    }
  }
}
```

Let's take note of a few things:

- PlayerGraphics and EndGameUI both have two Unity functions implemented (OnEnable and OnDisable)
    - OnEnable is called when the GameObject is set to be active, whether it's when the game starts or when gameObject.SetActive(true) is called
    - OneDisable, likewise, is called when the GameObject is disabled, whether it's when the object is destroyed or when gameObject.SetActive(false) is called
- GameManager will broadcast the message, while PlayerGraphics and EndGameUI will be listening for the message

To add a listener:

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

To remove a listener:

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

If you ever add an event, you **must** remove it as well.

```
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);
  }
}
```

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.

Notice how, before calling EndGameEvent in GameManager, we check to see if it is null. This is important because if you don't check it and there happens to be no listeners, Unity will throw an error.

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 to know when the game ends. To do that, in GameManager, we call:

```
EndGameEvent();
```

When this is called, PlayerGraphic's TurnOffGraphics and EndGameUI's TurnOnUI will both be called and their code will run.

Lab Instructions

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 carry, the slower you move. You attack by pressing the spacebar, and you can only attack while stationary.

1. Let's start by making the game playable when the "Play" button is pressed
    a. Open up *Scripts/Countdowns/Countdown.cs*
    b. There is a region called *Delegates and Events*; inside it, add the following lines:

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

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

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

        i. The countdown timer script will send out a message to the GameManager once the timer reaches zero. In this case, although we know ahead of time that the GameManager will listen in to the timer, it doesn't make sense for the timer to store a reference to GameManager.
2. Let's go ahead and choose a function in the GameManager script to listen for this event
    a. Open up *Scripts/GameManager.cs*
    b. In the *OnEnable, Setups, and Resetters* region, tell our GameManager's StartGame function to listen to Countdown's StartGameEvent method

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

c. Remember to tell the function to stop listening to the event in the OnDisable method as well.

3. Finally, let's implement a score counter so that when enemies are killed, it adds points
   a. Open *Scripts/Player/Bullet.cs*
   b. Add this to the *Delegates and Events* region:

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

   c. Next, look at the *Collision Methods* region and inside the OnParticleCollision method, add:

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

4. All that's left is to make GameManager's AddPoint function listen. We need to add two lines; one in OnEnable and one in OnDisable.

Here's a slightly more complex example. This time, we'll use the following scripts: AllDelegates, PointOrb, ScoreManager, ScoreText, ScoreEffects, GameManager, and WinnerText (like before, assume that each class has more code that is not shown)

The goal of the game is to gather 21 point orbs before any of your opponents.

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

Here, we created 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);
    }
  }
}
```

Here, we made a small check each time something collides with the orb. If the GameObject 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 ScoreManager runs its increase score method to increase the score of the appropriate player. If a player starts nearing the end mark, the ScoreManager 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 the 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 ScoreManager's event and only updates the UI when it receives the message that the score has changed.

```
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()
  }
}
```

Similar to the previous script, when the score changes, a particle system plays right in front of the player 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 a player's score reaches 21, the GameManager 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!!";
  }
}
```

WinnerText is one of the many scripts listening to see when the game ends. This script will print out who won the game once the game ends.

Lab Summary

Now that you've learned about delegates and events, you can use them to send messages to several different scripts without having to clutter up your code with unnecessary references. They're easy to extend and debug, and will be useful as your game gets bigger, and there are more scripts and information to keep track of.

Checkoff

1. Play one game and get a few kills before crashing into an enemy
2. Play a second game and get a better score, and stay alive all the way
3. Show that the scoreboard changes appropriately

Challenges (Optional)

1. Make it so that whenever an enemy is killed, two are spawned
2. Add a powerup that, when picked up, increases the movement speed of all enemies by some scalar
    a. When picked up, the powerup sends a message to all enemies that are alive
    b. The message also passes a float
    c. The velocity of each enemy is scaled by the float
    d. After a certain amount of time, the velocity of the enemies returns to normal