Coroutines

Table of Contents

Lab Overview

In this lab you will be learning all about coroutines, which are Unity's method of performing **asynchronous updates**, allowing you to make things happen outside of Unity's update cycle. By default, any code you put inside your Update function, as well as any physics or collider interactions, will execute **every frame**. But if you want to provide the player with a smooth transition from one state to another (e.g. changing colors or positions), it will need to take place over multiple frames; and for that, you will need coroutines.

Key terms

In order to make a function coroutine ready, you will need to use a special header and a yield statement.

Header: IEnumerator FunctionName([Parameters])

- Declaring the function as an IEnumerator tells Unity that it is a coroutine

Yield Statement:

- Once your coroutine is called, it will run until it reaches a yield statement and then stop executing. After a given period of time, execution will continue, starting with the next line. If you remember generators from 61A, coroutines function in exactly the same way!
- There are quite a few different yield statements, but these are some of the most useful:
    - yield; or yield return null;
        - Execution resumes at the start of the next frame (when Update() is called).
    - yield return new WaitForSeconds(waitTime);
        - Execution resumes after waitTime has elapsed.
    - yield return new WaitUntil(() => [condition])
        - Execution resumes after the declared condition evaluates to true.

        - This is a bit of a weird yield statement and less commonly used than the first two, but it can be useful if you only want something to occur under certain conditions.

StartCoroutine:

- When you want to start your coroutine for the first time, you call it with the command StartCoroutine(CoroutineName());
- If your coroutine takes in any parameters, you can pass those in by calling StartCoroutine(CoroutineName(Parameters));

Lerps and Time.DeltaTime:

- A very handy function to use in conjunction with coroutines is Lerp (short for linear interpolation)
- Lerp(start, end, x);
    - Imagine these three parameters as a number line with the start state at 0, the end state at 1, and x being a decimal number of how far along the number line you want to travel
    - The Lerp will return the state "x" of the way between the starting and ending state
    - This works for numbers, positions, colors, pretty much anything with clearly defined states!
- If you want a smooth transition from your starting state to ending state, then you will need to gradually increase your "x" value through multiple calls to your Lerp function
    - Time.DeltaTime can be used to do this; it tells you the amount of time that has elapsed since the last Update call

Putting it together:

```
IEnumerator SampleCoroutine() {
      float elapsedTime = 0;
      while (startState != endState) {
            Lerp(startState, endState, elapsedTime / totalTransitionTime);
            elapsedTime += Time.DeltaTime;
```

```
            yield return null;
        }
    }
    void Start() {
        StartCoroutine(SampleCoroutine());
    }
```

Lab Instructions

Task 1: Create a "death animation" for the player; when you press the "F" key, your character sprite should fade to black

- To change the player's color, you will have to modify the Color component of the player's Sprite Renderer
    - Use GetComponent
- All code should be written in the PlayerController script
- Your implementation should use both a Coroutine and Color.Lerp

Task 2: Use a coroutine to spawn in enemies at regular intervals

- All code should be written in the SpawnManager script
- Use *yield return new WaitForSeconds*
- Use Instantiate to spawn enemies
- Multiple spawn points have been created for your convenience (feel free to create your own or modify the existing ones)
    - Use a loop or multiple *yield* statements to cycle through them so that enemies spawn in new locations each time

Task 3: Give enemies coroutine based movement and turn them into actual sentries

- All code should be written in the EnemyMovement script
- You'll notice that the sentires all have a Trigger Collider Component attached to them. Whenever the player enters a sentry's collider zone, the sentry should record the player's entry point and investigate by Lerping towards it
    - The sentries should not continuously follow the player --just move them to the point the player was first detected
    - You might have to pass in parameters to your coroutine to successfully complete this
    - You'll have to keep track of whether a sentry is currently moving

- Since coroutines operate asynchronously to Unity's normal update cycle, it's possible for multiple coroutines to be running at once, which can lead to some weird teleporting bugs

Lab Summary

Now that you've learned about Coroutines, you can use them to create smooth transitions between different states, whether that be for an object's position, color, or something else. Because Coroutines are asynchronous, Unity can run multiple Coroutines at once, making it a powerful tool for handling multiple changing states at once.

Coroutines can also be used to handle larger functions. For example, functions that take more than a frame to execute (you won't experience any of these in class, but you certainly might in industry) might need to be staggered across multiple frames to prevent your game from lagging.

Checkoff

1. Explain why you might want to use a Coroutine
2. Explain what a Lerp is
3. Show that you have successfully completed the three tasks by:
    a. Starting the game and showing that enemies spawn in multiple locations at regular intervals
    b. Moving the player into multiple trigger zones and showing that the sentries move towards the player smoothly
    c. Pressing F and watching the player fade to black

Challenges (Optional)

1. Add a health system and call your death coroutine when you get hit and run out of lives
2. Give the player a sword that he can swing and defend himself with!
    a. This will involve Lerping rotations and colliders, and will be more challenging than the rest of the lab
3. Return to previous labs and see what you can rework using Coroutines and Lerps