Coroutines Lab

Overview

In this lab you will be learning all about coroutines! Coroutines 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*.

For many functions, this works perfectly fine, but some functions require a different update cycle. This may be necessary when running particularly large functions that will take more than a frame to execute. The function might need to be staggered across multiple frames to prevent your game from lagging. For example, if you had a 1000 x 1000 grid and needed to perform some computation for each cell you would want to spread out computation over multiple frames.

The more common usage you'll see in this class, is when you want to provide the player with a smooth transition from one state to another (e.g. changing colors or positions). For this transition to happen smoothly, as opposed to just instantly completing, it needs to take place over multiple frames which we can accomplish using coroutines.

Nitty-Gritty

In this section we'll go over the actual code required to make a coroutine. 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.

Ultimately, your coroutine should look something like this:

```
IEnumerator CoroutineName() {
    while (condition)
    {
        // Do something here!
        yield return null;
    }
}
```

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). Linear interpolation is a mathematical method of interpolating between 2 points a and b, using an interpolant t which updates the value of something according to the formula `(b - a) * t` as time moves between [0, 1]. When you use a `Lerp`, you pass in three parameters: a starting state, an ending state, and a float between 0 and 1. You can think of these three parameters as a number line, with the start state at 0, the end state at 1, and the float being how far along the number line you want to travel. The `Lerp` will then return the state

“X” (the third parameter) 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. A common and easy way to do this is through the use of `Time.deltaTime`. `Time.deltaTime` tells you the amount of time that has elapsed since the last `Update()` call, which allows you to keep track of how long your transition has been occuring and choose “X” values appropriately. Your `Lerp` would look like `Lerp(Start State, End State, elapsedTime / totalTransitionTime)`. As long as `elapsedTime` starts at 0 and you increase it by `Time.deltaTime` each time the function is called, this will provide you with extremely smooth looking transitions! With that out of the way, it’s time to begin the actual tasks of this lab!

Task 1:

- For this task you will be creating a “death animation” for the player. When you press the “F” key, your character sprite should fade to black.
- All code should be written in the `PlayerController` script.
- To be checked off for this task, your implementation will have to use both a Coroutine and `Color.Lerp` (try to use `yield return null` and `Time.deltaTime`).
- To change the player’s color, you will have to modify the `Color` component of the player’s Sprite Renderer (use `GetComponent`).

Task 2:

- For this task, you will be utilizing a coroutine to spawn in enemies (with the Instantiate command) at regular intervals.
- All code should be written in the `SpawnManage`r script.
- Use `yield return new WaitForSeconds`.
- 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:

- These enemies are pretty boring! Let's give them a coroutine based movement script and turn them into actual sentries.
- All code should be written in the `EnemyMovement` script.
- You'll notice that the sentries 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 (you can use `Vector3.Lerp()`). Note: the sentries should not continuously follow the player - just move them to the point the player was first detected.
- You will likely have to pass in parameters to your coroutine to successfully complete this!
- You'll also have to make sure 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 glitches.

Checkoff:

To be checked off for this lab, you will have to complete the following:

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

Challenges/Extensions:

1) Turn this into an actual game! Make something happen when the sentries hit the player! You could add a health system, and call your death coroutine when you run out of lives!
2) Give the player a sword swing to defend himself! Note: This will involve Lerping rotations and colliders, and will be more challenging than the rest of the lab.
3) Go back to previous labs and see what you can rework using coroutines and Lerps!