🔍

Coroutines

For more information go here

Differences between a normal Method and a Coroutine Method

- Coroutines act as normal functions with a return type of IEnumerator
- Allows pausing its execution and resuming from the same point after a condition is met
- Requires at least 1 Yield Return statement
- Yield statements gives us control of time and allow us to run our code asynchronously
    - Asynchronous - occurring at the same time
- Coroutines are not multithreaded, they are Asynchronous multitasking methods.

Put simply, this allows you to pause a function and tell it to wait for a condition or action to occur before continuing. This allows you to split up its functionality into a number of steps that can be executed in order.

▼ Start Coroutines in Unity

- To Start or Invoke a Coroutine we use the **StartCoroutine**(nameof(coroutine)) or **StartCoroutine**(YourMethod())

▼ Ending Coroutines

- To end a coroutine we use **StopCoroutine**(nameof(Coroutine)) or **StopCoroutine**(YourMethod())
    - This will stop a Specific Coroutine
- To end all coroutines **StopAllCoroutines**()
- **yield return break** - rather like a normal "return", breaks out of the iterator function early - any subsequent code will not be executed

- Disabling or Destroying the GameObject that started or invoked a Coroutine will result in all the coroutines being stopped.

Pausing a Coroutine

The Yield statement marks the point at which the coroutine "pauses".

- **yield return null, yield return 1, and yield return new WaitForEndofFrame()** are similar, but subtly different. They all return control to the calling function, and schedule the coroutine to continue execution in the next frame. The difference is in at *what point* in the next frame the coroutine will next be executed
    - Any other yield value, including 1, null, true, false, or "bananas" will continue the coroutine on the next frame as part of the standard game logic Update step
- **yield return WaitForFixedUpdate()** - WaitForFixedUpdate will schedule the coroutine to run as part of the physics update step
- **yield return WaitForSeconds(**time in seconds**)** - WaitForSeconds schedules the coroutine to continue after the given number of seconds has elapsed
- **yield return WaitForEndOfFrame(**) will schedule the coroutine to be run at the end of the frame just before rendering
- **yield return WaitUntil**(bool delegate) - WaitUntil will suspend the coroutine until the supplied delegate evaluates to True
- **yield return WaitWhile**(bool delegate) - WaitWhile will suspend the coroutine until the supplied delegate evaluates to False

Make up of a Coroutine

- I like to break up coroutines into 3 main sections.
    - Pre-Coroutine Phase
        - This is where you setup the information that will be used by the coroutine
            - Setting up default values
    - Main Coroutine Phase
        - This is where the coroutine does its work
            - Manipulates the default values of the Pre-Coroutine Phase

    - Post-Coroutine
        - This is where you would do any work after the Main Coroutine Phase has been executed.
        - Setting the default values to the target values

Why and When to Use Coroutines

Why Use Coroutines

- Coroutines are ideal for setting up game logic that needs to takes place over time

When to Use Coroutines

It’s worth considering using a Coroutine whenever you want to create an action that needs to pause, perform a series of steps in sequence or if you want to run a task that you know will take longer than a single frame.

Examples

- Moving and object to a position or Rotating an Object to a specific angle
- Perform a Sequence of Tasks
- Fading Visuals such as UI, Colours, Transition between on Audio clip to another

Code Examples

- Coroutines with no Parameters

```
private void Start()
{
  StartCoroutine(RotateTileRoutine()); // Starting a Coroutine
  StopCoroutine(RotateTileRoutine()); // Stopping a Coroutine
}

private IEnumerator RotateTileRoutine()
{
  yield return null; // Must have at least 1 yield return statement!!!
}
```

- Coroutines with Parameters

```
private void Start()
{
  float a = 0f, b = 10f, delay = 3f;
  StartCoroutine(RotateTileRoutine(a, b, delay));
}

private IEnumerator LerpValueRoutine(float a, float b, float delay)
{
  // Pre-Coroutine Phase
  float elapsedTime = 0f;
  float lerpValue = 0f;

  while(elapsedTime < delay)
  {
    // Main Coroutine Phase
    lerpValue = Mathf.Lerp(a, b, elapsedTimed / TimeToLerp);
    elapsedTime += Time.deltaTime;
    yield return null; // Must have at least 1 yield return statement!!!
  }

  // Post-Coroutine Phase
  lerpValue = b;
}
```

- Coroutine with WaitForSeconds (Scaled Time) and WaitForSecondsRealtime (Unscaled Time)
  - Scaled Time - Affect by Time.timeScale (Speed up or Slow down your game)
  - Unscaled Time - Not Affected by Time.timeScale
  - Note that WaitForSeconds and WaitForSecondsRealtime create objects

```
public float SimpleDelay = 3f;
public WaitForSeconds Delay;

private void Start()
{
  Delay = new WaitForSeconds(SimpleDelay);
  StartCoroutine(RotateTileRoutine());
}

private IEnumerator WaitForSecondsRoutine()
{
  // Affected by Time.timeScale
  yield return new WaitForSeconds(1f); // Waits for 1 second before continuing.
}

private IEnumerator WaitForSecondsRealtimeRoutine()
{
  // Not Affected by Time.timeScale
  yield return new WaitForSecondsRealtime(1f); // Waits for 1 second before continuing.
}
```