**TEMASEK POLYTECHNIC**

**SCHOOL OF INFORMATICS & IT**

**DIPLOMA IN IMMERSIVE MEDIA & GAME DEVELOPMENT**

**AY2025/2026 APRIL SEMESTER**

**GADV (CGE2C25)**

**Unity Programming**

**Coroutines**

To see the additional comments and resources, make sure you select **All Markup** in the **Review/Tracking** pane



**Overview**

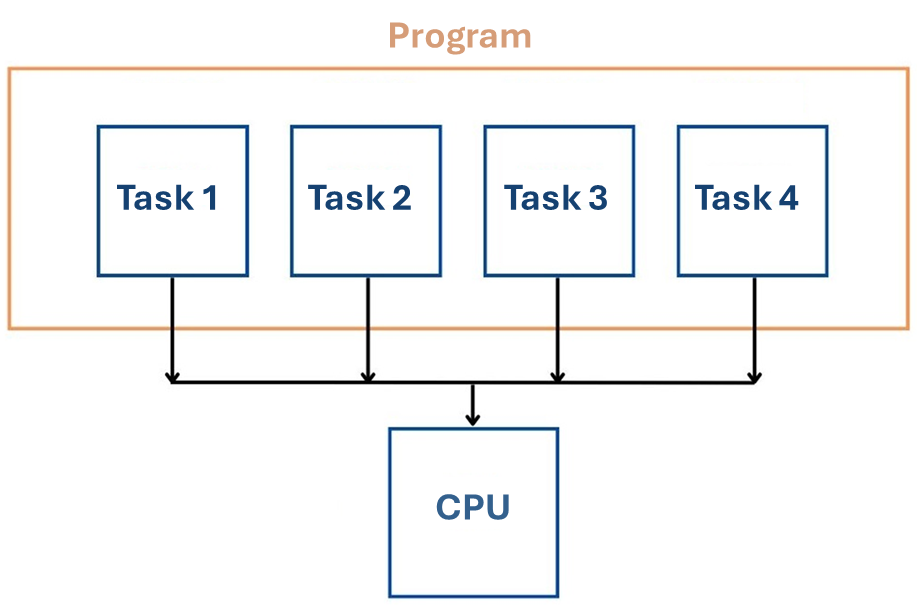
By the end of this worksheet, you should be able to:

* Explain what a coroutine is and why it's useful in Unity.
* Use StartCoroutine() and yield to delay actions or spread work over time.
* Convert Update-based logic into coroutine-based logic.
* Recognise common coroutine mistakes (like infinite loops).
* Compare Unity coroutines with C# async/await, and know when to use each.

Make sure you complete these pre-class activities!

**Part 1**

1. **What Is Multitasking in Programming?**



Imagine trying to walk and talk at the same time. Your brain switches quickly between different tasks to make it *seem* like both happen together. Computers do something similar.

In a computer:

* A **program** is a set of instructions.
* A **process** is a program that is currently running.
* A **thread** is a single line of execution within that process.

In Unity, all your scripts run on one main thread. This means they happen one after the other — not truly in parallel. But we often want actions to appear to happen *at the same time*, like:

* A player walking
* An enemy chasing
* A countdown running

**Questions**

1. Think about your game project. Write down FIVE parallel sub-tasks that Unity must perform while the game is running.
2. For modern computer systems, much of the processing for a game is done by the GPU (Graphics Processing Unit). Explain what this means.
3. When you buy a computer or a mobile phone, it’s probably a multi-core device.

What does multi-core mean? Are 8 cores generally better than a single core? Why or why not? Is multi-core better for Unity development?

So, if we don’t use DOTS to make use of all CPU cores, how can we run multiple tasks in Unity?

Coroutines!

1. **What’s wrong with just using Update?**

The Update() function runs every single frame — even if you don’t need it to. That can waste performance, especially if you have many objects using Update but doing nothing.

Let’s say you want something to move or fade *slowly*.

You don’t want to:

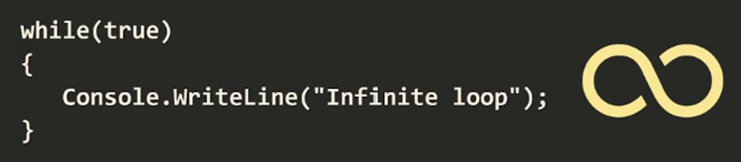
* Run the full task instantly
* Use Update every frame unnecessarily

Instead, you want the task to pause and resume later — like doing one small step per frame.

That’s where **coroutines** are used.

1. **Infinite Loops**

Let’s see what happens when we write code that never stops.



**Step 1:** Create a script **Coroutines.cs**

public class InfiniteLoop\_DANGER : MonoBehaviour

{

int x = 0;

void Start()

{

Infinite();

}

void Infinite()

{

while (x < 10)

{

Debug.Log("Still looping!");

}

}

}

**DANGER! You will have to ctrl-alt-delete Unity to stop the script. Save this scene and project, and any other work that you might have before running this code!**

**Step 2:** Attach the script to a GameObject and run the game.

Unity will freeze due to an infinite loop — a function that never ends. Since Unity runs scripts sequentially, one stuck script blocks everything else, causing the game to lock up.

1. **Using the Visual Studio Debugger to Stop an Infinite Loop**

This is a slight detour, but it’s worth knowing 😊

Make sure that Visual Studio is your preferred IDE. In Unity, select the menu option **Edit 🡪 Preferences**, and set Visual Studio as your script editor.

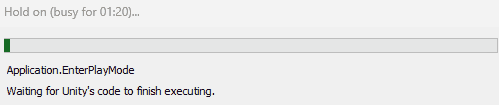
Before running, select **Attach to Unity** in Visual Studio to link the debugger. Pause and stop buttons will then appear.



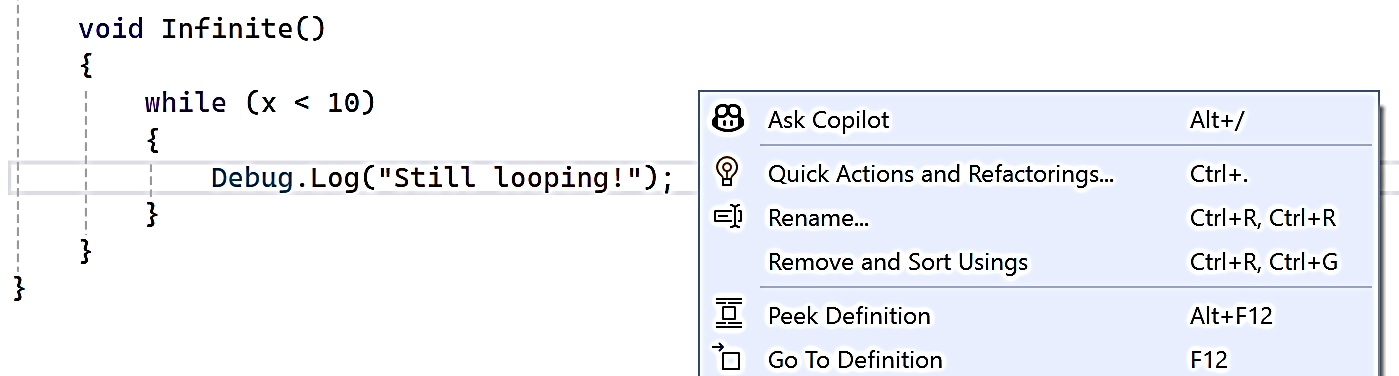
Click the **Attach to Unity** button.

Switch to Unity and run your project again (but first, save your work so far!).

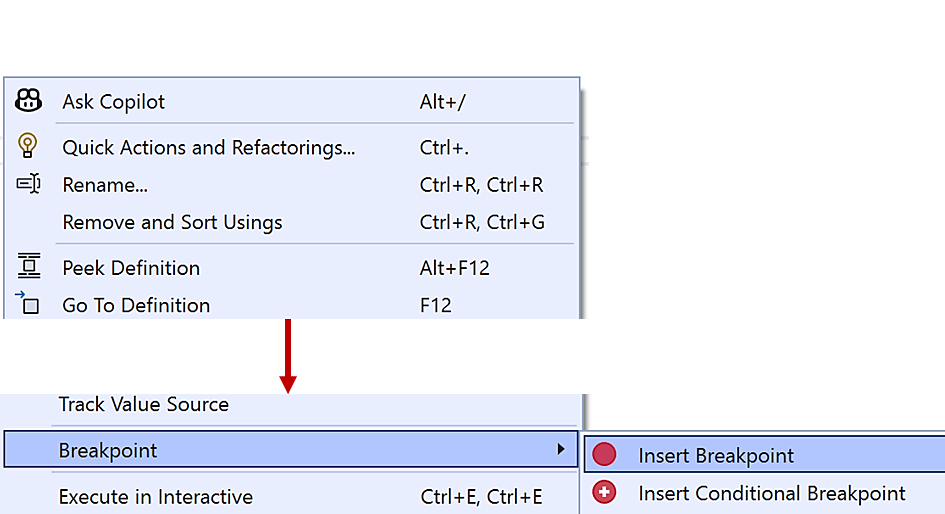
Unity will hang!



Go back to Visual Studio and select a line inside the while loop.



Right-click on the line, and select **Insert Breakpoint**.



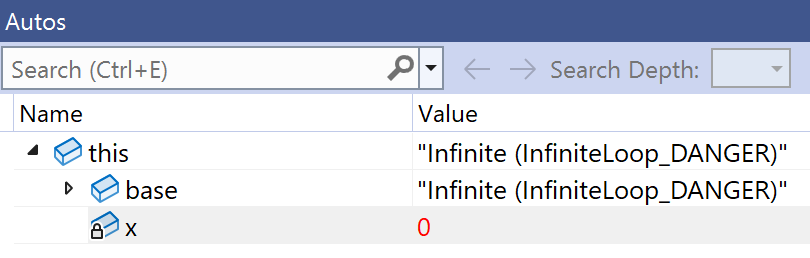
When Visual Studio hits a breakpoint, it pauses execution so you can inspect—and even modify—your variables.

Here, we want to change the value of x so the loop can end.

After setting the breakpoint, execution will pause at that line, which will be highlighted in the editor.

Follow these instructions carefully:

1. Look in the **Autos** panel at the bottom of Visual Studio



1. Set the value of x to something that will allow the loop to end, e.g. 10



1. Click the **Continue** button near the menu bar to continue execution



Unity should unfreeze and continue to run as expected!

Now read this web page: <https://docs.unity3d.com/Manual/Coroutines.html>

1. **Fixing It With a Coroutine**

Change your script like this:

using UnityEngine;

using System.Collections;

public class Coroutines : MonoBehaviour

{

void Start()

{

StartCoroutine(Infinite());

}

IEnumerator Infinite()

{

while (true)

{

Debug.Log("Now I yield");

yield return null;

}

}

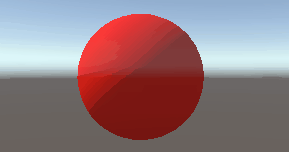
}

**Coroutine Vocabulary**

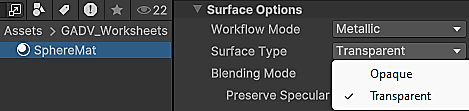
* **Coroutine:** A special function that can pause and continue later.
* **yield:** A keyword that tells Unity when to resume the coroutine.
* **StartCoroutine():** Runs a coroutine from another function.
* **IEnumerator:** The return type required for coroutines in Unity.

1. **Coroutine example: fading a sphere**

Let’s write a coroutine that makes a sphere slowly disappear.



1. Add a **Sphere** to your scene.
2. Create a new **Material** called **SphereMat** in a Materials folder. Choose a colour and set **Rendering Mode** to **Transparent**.
3. Set the material’s **Surface Type** to **Transparent**.



1. Apply the material to the sphere.
2. Add a **Collider** if the sphere doesn’t already have one.
3. Create a script called **MaterialFader.cs** and add this:

public class MaterialFader : MonoBehaviour

{

Material mat;

void Start()

{

mat = GetComponent<Renderer>().material;

}

void OnMouseDown()

{

StartCoroutine(FadeOut());

}

IEnumerator FadeOut()

{

for (float alpha = 1f; alpha >= 0; alpha -= 0.05f)

{

Color c = mat.color;

c.a = alpha;

mat.color = c;

yield return new WaitForSeconds(0.05f);

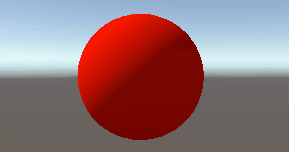
}

}

}

1. **Going Further: Oscillating Fade**

Let’s make it fade in and out repeatedly.



Create a new sphere with a new material. Add a script called **FadeOutPingPong** (see the code below). Complete the ??? parts of the code.

* When alpha reaches 0, start increasing it.
* When it reaches 1, start decreasing it again.
* Keep repeating.

Hints:

* Use a bool **fadingOut = true**;
* Use a **while (true)** loop
* Use **yield return** to delay each step

IEnumerator FadeOutPingPong()

{

bool fadingOut = **???**;

float alpha = 1f;

while (**???**)

{

Color c = mat.color;

c.a = alpha;

mat.color = c;

yield return new WaitForSeconds(0.05f);

if (**???**)

{

alpha -= 0.05f;

if (alpha <= 0f)

{

alpha = 0f;

fadingOut = **???**;

}

}

else

{

alpha += 0.05f;

if (alpha >= 1f)

{

alpha = 1f;

fadingOut = **???**;

}

}

}

}

1. **Coroutine vs Update: Which One Should I Use?**

Both Update() and coroutines run code over time. So when should you use each?

|  |  |
| --- | --- |
| **Use Update() when...** | **Use Coroutines when...** |
| You need something to respond **every frame** (e.g. checking input) | You need to do something **occasionally**, but over time (e.g. fade, wait, spawn) |
| You are working with **physics** in FixedUpdate() | You want to **pause** and continue code without blocking everything else |

Too many Update functions can hurt performance, especially if they’re mostly doing nothing. Coroutines are more efficient in those cases.

1. **Coroutine vs async/await in C#**

Unity supports coroutines using IEnumerator, but C# also has **async/await**.

Here’s a simple example (don’t add this code to your script, it’s just an illustration):

async void Start()

{

string result = await DownloadText();

Debug.Log(result);

}

async Task<string> DownloadText()

{

using (HttpClient client = new HttpClient())

{

return await client.GetStringAsync("https://example.com");

}

}

This downloads text from the internet, but doesn’t freeze the game while waiting.

Key differences:

|  |  |
| --- | --- |
| Coroutine (IEnumerator) | async/await (C#) |
| Built into Unity’s engine and runs in the game loop | Part of the C# language and better for async file or web I/O |
| Easy to use with yield return new WaitForSeconds() | Uses Task, await, and typically needs async methods |
| Great for game logic like movement, animation, fading | Best for downloading files, web requests, or waiting for results outside Unity |

In general:

* Use **coroutines** for gameplay, e.g. animations, fades, and in-game timing.
* Use **async/await** for downloading data, web APIs, saving files.

Unity objects are not thread-safe, so always update the game world using Unity’s main thread (coroutines are safer for this).

**Part 2**

1. **Introduction**

In Part 1, you learned how to use coroutines to delay actions and avoid overusing Update(). You also explored how to fade objects and loop actions over time. This worksheet goes further. You will learn how to:

* Run multiple coroutines
* Stop coroutines
* Pass parameters
* Chain coroutines
* Use them in simple game mechanics like timers, movement, and enemy spawning

This worksheet includes guided explanations, code examples, and challenge tasks.

1. **Running Multiple Coroutines on One Object**

You can start several coroutines on the same GameObject. Each one runs independently unless you choose to stop it.

Create a new GameObject and attach this script:

using UnityEngine;

using System.Collections;

public class MultipleCoroutines : MonoBehaviour

{

void Start()

{

StartCoroutine(FlashRed());

StartCoroutine(MoveUp());

}

IEnumerator FlashRed()

{

while (true)

{

Debug.Log("Flashing!");

yield return new WaitForSeconds(1f);

}

}

IEnumerator MoveUp()

{

while (true)

{

transform.position += Vector3.up \* 0.1f;

yield return new WaitForSeconds(0.5f);

}

}

}

This example just prints out “Flashing!” without changing the object’s colour.

Write a new script called **MoveFlashCoroutines.cs** to make the colour change while the object is moving up.

A red ball in the air

AI-generated content may be incorrect.

Complete the ??? parts of the code.

using UnityEngine;

using System.Collections;

public class MoveFlashCoroutines : MonoBehaviour

{

Renderer renderer;

Color originalColor;

void Start()

{

??? = GetComponent<Renderer>();

??? = rend.material.color;

StartCoroutine(???);

StartCoroutine(???));

}

IEnumerator FlashRed()

{

while (???)

{

renderer.material.??? = Color.red;

yield return new ???(0.2f);

renderer.material.color = ???;

yield return new ???(0.2f);

}

}

IEnumerator MoveUp()

{

while (true)

{

transform.position += Vector3.up \* 0.1f;

yield return new WaitForSeconds(0.5f);

}

}

}

1. **Stopping Coroutines**

You can stop coroutines using **StopCoroutine()** or **StopAllCoroutines()**.

**StopCoroutine()** stops a particular coroutine instance stored in a **Coroutine** type variable.

using UnityEngine;

using System.Collections;

public class StopCoroutine : MonoBehaviour

{

Coroutine flashRoutine;

void Start()

{

flashRoutine = StartCoroutine(Flash());

StartCoroutine(StopAfterSeconds(5));

}

IEnumerator Flash()

{

while (true)

{

Debug.Log("Flashing...");

yield return new WaitForSeconds(0.5f);

}

}

IEnumerator StopAfterSeconds(float time)

{

yield return new WaitForSeconds(time);

**StopCoroutine(flashRoutine);**

Debug.Log("Stopped flashing");

}

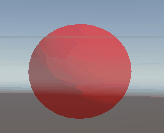
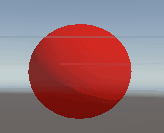
}

**Question 4:** Why is using a coroutine here more optimised than if you used the Update function?

**Hint:** How often does the repeated action occur, and does it continue forever?

1. **Passing Parameters into Coroutines**

You can pass arguments to coroutines just like normal functions. Here, we’ll pass a fade duration value to the FadeOut coroutine.



1s

5s

Create a new object and add this script:

using UnityEngine;

using System.Collections;

public class CoroutineParameters : MonoBehaviour

{

void Start()

{

StartCoroutine(FadeOut(5f));

}

IEnumerator FadeOut(float duration)

{

Material mat = GetComponent<Renderer>().material;

float elapsed = 0f;

while (elapsed < duration)

{

float newAlpha = 1f - (elapsed / duration);

Color c = mat.color;

c.a = newAlpha;

mat.color = c;

elapsed += Time.deltaTime;

yield return null;

}

// Ensure it's fully transparent at the end

Color final = mat.color;

final.a = 0f;

mat.color = final;

}

}

**Question:** What is **newAlpha = 1f - (elapsed / duration)** for?

**Hint:** Look at these graphs for linear and non-linear change. Is the fade out supposed to be linear over the specified duration in seconds? Ask Chat GPT to explain the math!

1. **Chaining Coroutines**

You can wait for one coroutine to finish before starting the next.

A screen shot of a computer

AI-generated content may be incorrect.

The **MoveUp** coroutine waits for the **FlashRed** coroutine to stop before it starts.

(In this case, FlashRed just prints out 5 messages and then stops.)

Create a new object and add this script:

using UnityEngine;

using System.Collections;

public class CoroutineChain : MonoBehaviour

{

void Start()

{

StartCoroutine(Sequence());

}

IEnumerator Sequence()

{

yield return StartCoroutine(FlashRed());

yield return StartCoroutine(MoveUp());

Debug.Log("Sequence complete!");

}

IEnumerator FlashRed()

{

for(int i = 0; i < 5; i++)

{

Debug.Log("Flashing red...");

yield return new WaitForSeconds(0.2f);

}

}

IEnumerator MoveUp()

{

while (true)

{

transform.position += Vector3.up \* 0.1f;

yield return new WaitForSeconds(0.5f);

}

}

}

Note that we now have a third coroutine: **Sequence**.

This runs the **FlashRed** and **MoveUp** coroutines. The **MoveUp** coroutine waits for **FlashRed** to finish before it starts.

**Question 5:** Why is **yield return StartCoroutine(...)** used instead of just **StartCoroutine(...)**?

**Hint:** Do we want both coroutines to run at the same time?

Note that you can simplify this code a bit:

yield return FlashRed();

yield return MoveUp();

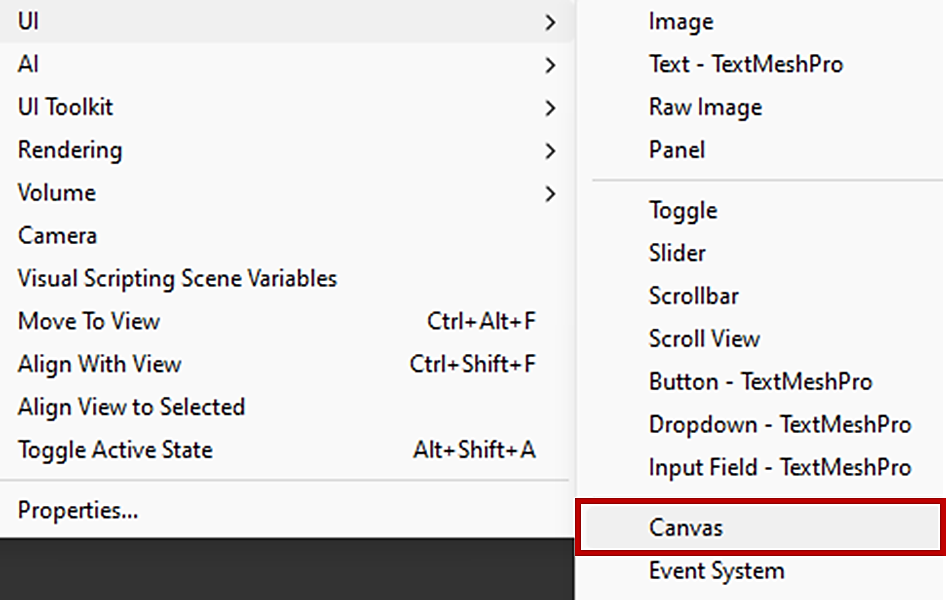
Debug.Log("Sequence complete!");

Unity automatically treats yield return **SomeCoroutine()** the same as yield return **StartCoroutine(SomeCoroutine ())**.

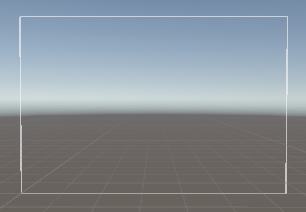
1. **Timer and Countdown**

Let’s implement a UI countdown timer.

* 1. Right-click in the Hierarchy to create a Canvas.

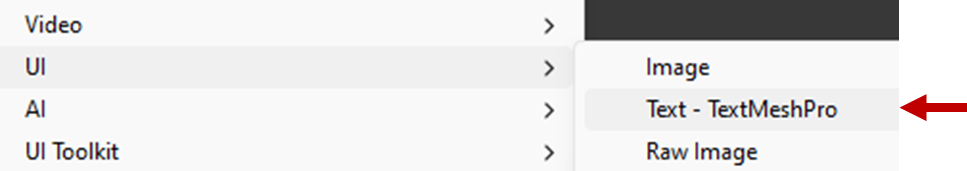


Select the Canvas in the Hierarchy and press the F key. This will frame, or center, the selected object in the Scene view.



* 1. Right-click again, and create a TextMeshPro text object.

Name this **CountdownText**.

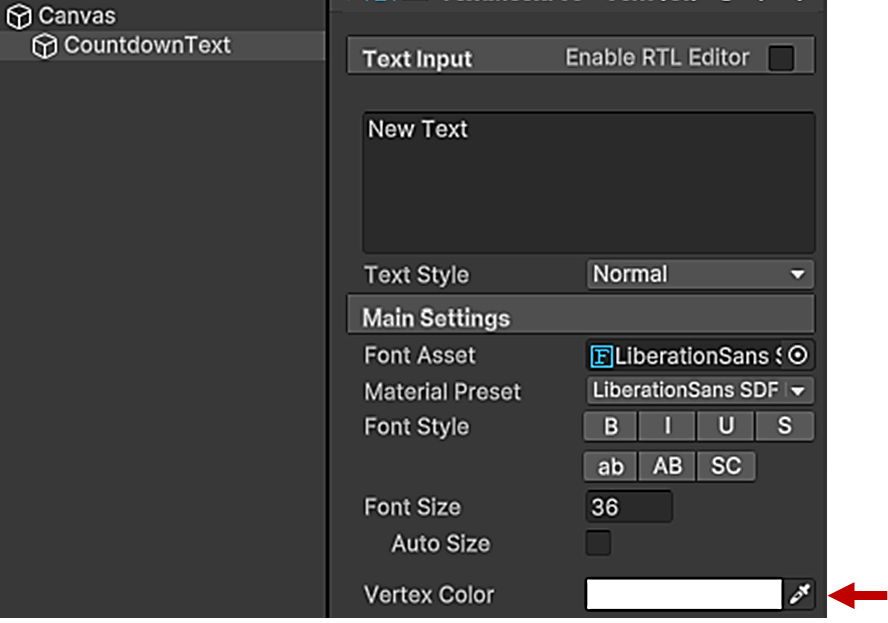


When prompted, click **Import TMP Essentials**.



TextMeshPro is Unity's advanced text rendering system. It provides better text quality, dynamic styling, rich text tags, and layout control compared to the default UI Text.

Select **CountdownText** in the Hierarchy and change the **Vertex Color** property to your preferred colour.





* 1. Create an empty GameObject called **Timer**. This will be used to contain the countdown timer script.
     1. Create this script and add it to **Timer**.

using UnityEngine;

using UnityEngine.UI;

using System.Collections;

public class CountdownTimer : MonoBehaviour

{

public TextMeshProUGUI timerText;

public int seconds = 10;

void Start()

{

StartCoroutine(Countdown());

}

IEnumerator Countdown()

{

while (seconds > 0)

{

timerText.text = seconds.ToString();

yield return new WaitForSeconds(1f);

seconds--;

}

timerText.text = "GO!";

}

}

The script needs a reference to the **CountdownText** TMP object. This is stored in **timerText**.

The duration of the countdown is stored in **seconds**.

These are both defined as public variables, which means that they can be set in the Inspector.

* + 1. Drag the **Timer** from the Hierarchy into the **Timer Text** field.



The script now has a reference to the TMP text object, which can be set to display the current countdown value:

timerText.text = seconds.ToString();

Run the project.



1. **Spawning asteroids over time**

Let’s implement one more example.

using UnityEngine;

using System.Collections;

public class AsteroidSpawner : MonoBehaviour

{

public GameObject asteroidPrefab;

void Start()

{

StartCoroutine(SpawnAsteroids());

}

IEnumerator SpawnAsteroids()

{

while (true)

{

Vector2 screenPosition = new Vector2(

Random.Range(0f, Screen.width),

Random.Range(0f, Screen.height));

Vector3 worldPosition = Camera.main.ScreenToWorldPoint(

new Vector3(screenPosition.x, screenPosition.y, 10f));

Instantiate(asteroidPrefab, worldPosition, Quaternion.identity);

yield return new WaitForSeconds(0.5f);

}

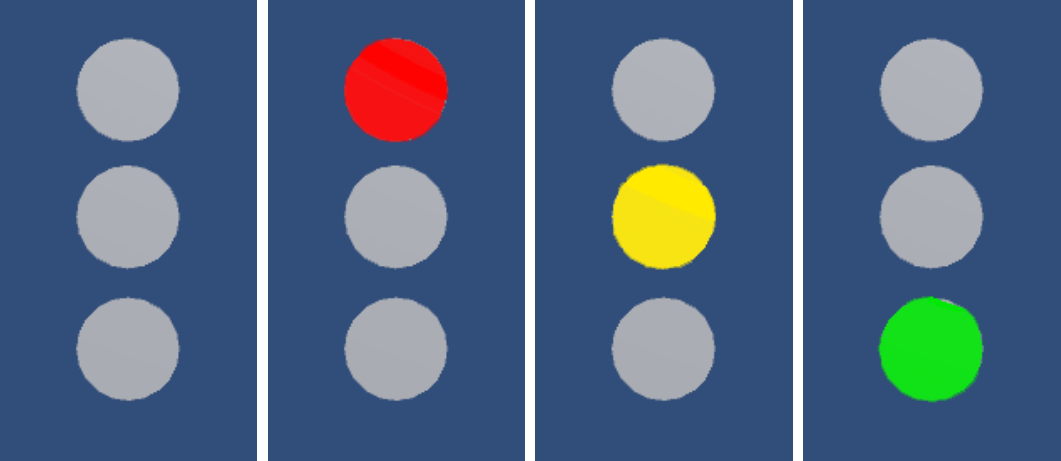
}

}

**Challenge:** Add a public variable **int maxAsteroids** to stop spawning asteroids after that number.

**Exercise: Part 2**

1. Implement a set of traffic lights. The lights should change from green to amber to red then back to green, with an interval of 2 seconds. Here are some hints.



1. Implement using only the Update function and boolean variables.   
   You will need to implement the time interval in some way.
2. Implement using coroutines.

Which is simpler? Explain why.

Coroutines, since there is much less code needed to handle waiting, unlike the update function, where you have to keep track of the frames that have passed.

**List of Coroutine return types**

Corotuines have a number of different yield conditions, listed below. More detailed explanations are given here.

/\* Coroutine Return Types \*/

// Waits until the next Update() call

yield return null;

// Waits until the next FixedUpdate() call

yield return new WaitForFixedUpdate();

// Waits until everything for this frame has executed

yield return new WaitForEndOfFrame();

// Waits for game time in seconds

yield return new WaitForSeconds(float seconds);

// Waits until a custom condition is met

yield return new WaitUntil(() => MY\_CONDITION);

// Waits for a web request

yield return new WWW("MY/WEB/REQUEST");

// Waits until another Coroutine is completed

yield return StartCoroutine("MY\_COROUTINE");

These all specify different **wait** conditions for the coroutine.

You will probably use these the most:

* Null
* WaitForSeconds

**Optional Challenge Tasks**

1. **Ping-Pong Move**
   * Make a cube move back and forth between two points.
2. **Typewriter Effect**
   * Show a sentence one character at a time using: **yield return new WaitForSeconds(0.1f);**
3. **Temporary Power-Up**
   * Double player speed for 5 seconds, then return to normal.

**Reflection Questions**

These questions are meant to help you identify where coroutines could be useful in your game project.

You can discuss these with your tutor.

* Are there any actions in your game that happen too quickly and would look better if they happened gradually?
* Does your game have timed events (e.g. spawning enemies, countdowns, delayed effects) that could use coroutines instead of Update()?
* Have you ever used Update() when nothing happens most of the time? Would a coroutine be more efficient?
* Are there moments in your game where one action should wait for another to finish before starting?
* How might coroutines help make your game logic cleaner, easier to read, or easier to test?

