50.033 Game Design and Development

Unity for Adults

JANUARY 06, 2021

Learning Objectives: More Advanced Management

- Implementing a Combo Manager
- Dealing with heavy computations:
  - Unity Async/Await,
  - Task,
  - Comparisons with Coroutines
- C# Basics:
  - Dictionaries
  - Abstract Class

Introduction

It is not uncommon for video games to have some kind of combo system and maybe in some cases, needing to perform heavy computations while keeping the system responsive. This is the last tutorial where we use our starter Mario project, before we expand our knowledge in 3D games with another starter project. Hopefully by the end of this tutorial, you have more than enough knowledge to implement the basic Super Mario Bros game and beyond.

The Combo System

This suggested combo system works with Scriptable Object and a Manager Script meant to be attached on the player (that can cast the regular skill or the combo skill). As usual, we will make **each** skill and **each** combo a **Scriptable Object**, and **keep track** which Combo may potentially be casted given a current **key press**.

Skill Keys

The first step to do is to decide the keys that need to be pressed to activate *regular skills*, something that the character can keep casting on a regular basis.

Create a new script called `SkillKeys.cs` :

```
using UnityEngine;

[CreateAssetMenu(fileName = "SkillKeys", menuName = "ScriptableObjects/SkillKeys", c
public class SkillKeys : ScriptableObject
{
    [Header("Inputs")]
    public KeyCode key;

    public bool isSameAs(SkillKeys k)
    {
        return key == k.key;
    }
}
```

Regular Skills

Then, we also need to define the types of regular skill (the visual effect, sound effect, etc) that a character can cast. Create a new script called `RegularSkill.cs` :

```
using UnityEngine;

public enum AttackType
{
    heavy = 0,
    light = 1,
    kick = 2,
    //test

}
```

```
[System.Serializable]
public class Effect
{
    // change this to your own data structure defining visual or audio effect of thi
    public GameObject particleEffect;
    // o
}

[System.Serializable]
public class Attack
{
    public float length;

    // change this to your own data structure defining visual or audio effect of thi
    public Effect effect;
}

[CreateAssetMenu(fileName = "RegularSkill", menuName = "ScriptableObjects/RegularSki
public class RegularSkill : ScriptableObject
{
    public AttackType skillType;
    public SkillKeys key;
    public Attack attack;
}
```

Combo Skill

Finally, we need another script to describe our Combo: its type, effect, skill keys to trigger, and a simple *check* on whether the current given input is the correct key (**on track** to trigger the Combo):

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;

[System.Serializable]
public enum ComboType
{
    electric = 0,
```

```
    explosion = 1,
    shine = 2
};

[CreateAssetMenu(fileName = "ComboSkill", menuName = "ScriptableObjects/ComboSkill",
public class ComboSkill : ScriptableObject
{
    // a list of combo inputs that we will cycle through
    public List<SkillKeys> inputs;
    // public ComboAttack comboAttack; // once we got through all inputs, then we su
    public Attack attack;
    public UnityEvent onInputted;
    public ComboType comboType;

    int curInput = 0;

    public bool continueCombo(SkillKeys i)
    {
        if (currentComboInput().isSameAs(i))
        {
            curInput++;
            if (curInput >= inputs.Count) // finished the inputs and we should cast
            {
                onInputted.Invoke();
                //restart the combo
                curInput = 0;
            }
            return true;
        }
        else
        {
            //reset combo
            ResetCombo();
            return false;
        }
    }

    public SkillKeys currentComboInput()
    {
        if (curInput > inputs.Count) return null;
        else return inputs[curInput];
    }

    public void ResetCombo()
```

```
        {
            curInput = 0;
        }
    }
```

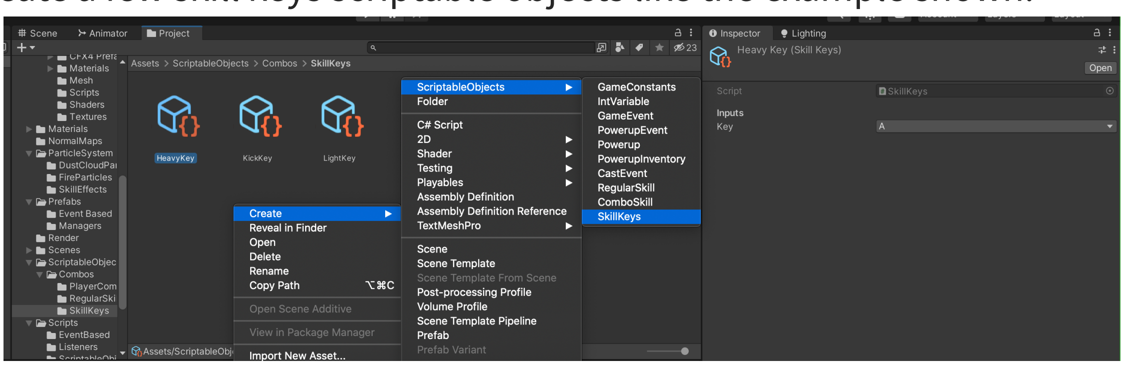
Note the instruction `onInputted.Invoke();` , means that in this design we expect some other callback method subscribed to this combo's `onInputted` event, which triggers whatever animation or effects necessary when the combo is successfully launched.

Also, the method `continueCombo` keeps track of the input entered *so far*. Thus the instantiation of `ComboSkill` is *character dependent*, i.e: you need to instantiate one `ComboSkill` per character utilising it.

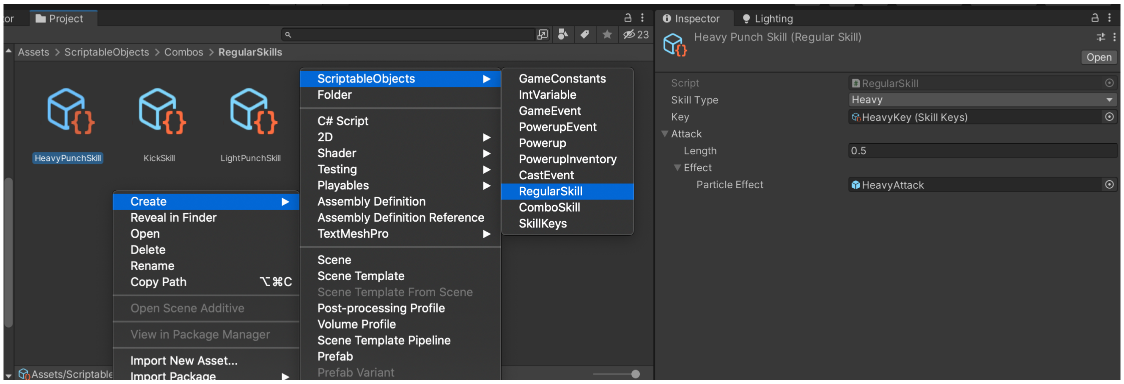
> We aren't saying that this is the *best* solution. If you have other designs you deem better, you're free to use it in your Project.

Instantiation: SkillKey, RegularSkill, ComboSkill

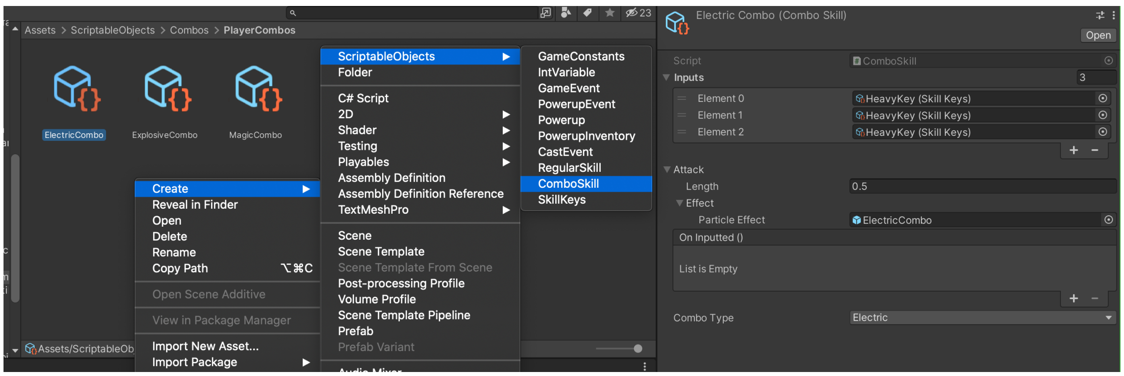
Now create a few skill keys scriptable objects like the example shown:



And a few regular skills scriptable objects that utilise a skill key:



Finally, a few combos utilising sequences of skill keys:



For this demo purpose, what we created was 3 different skill keys named:

- HeavyKey (A)
- KickKey (S)
- LightKey (D)

Each key will trigger a regular skill called:

- HeavyPunchSkill (utilising HeavyKey)
- KickSkill (utilising KickKey)
- LightPunchSkill (utilising LightKey)

A combination of a few keys will trigger a combo skill instead:

- HeavyKey x3: ElectricCombo
- KickKey x3: ExplosiveCombo
- Heavy-Light-KickKey: MagicCombo

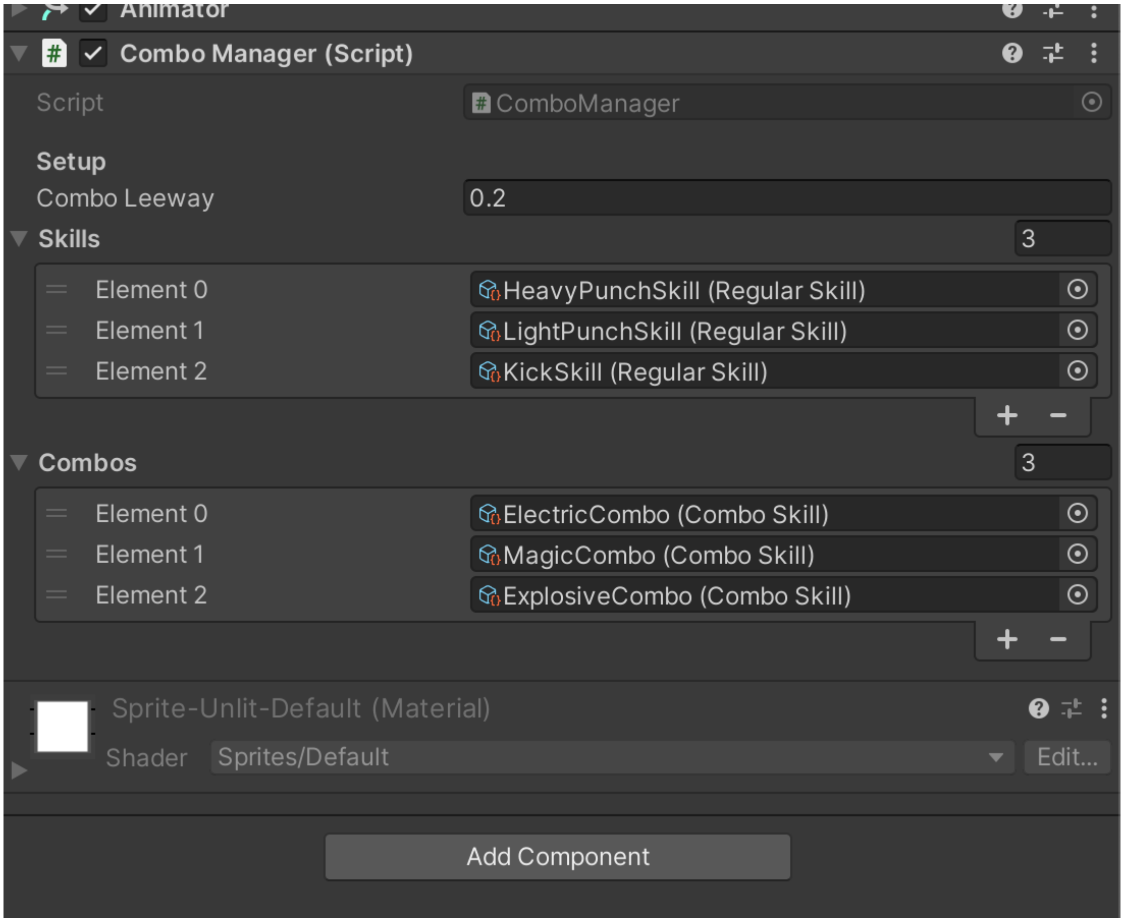
You can name and set your own SkillKeys, RegularSkills, and ComboSkills as you like. You may also change the structure of the `effect` instance defined under `public class Attack` in `RegularSkill.cs`. Simply add more fields in the `Effect` class. For example, it is common to trigger animations too and you may add relevant Animator parameter names that need to be triggered upon casting this skill or combo, and use them later on using `<AttackInstance>.effect.<ParameterName>`.

The ComboManager

Now we need a script (to be attached to the Player gameobject) to utilize these RegularSkills and ComboSkills. Create a new script `ComboManager.cs` declaring the following setup variables:

```
using System.Collections.Generic;
using UnityEngine;

public class ComboManager : ComboManagerBase
{
    [Header("Setup")]
    public float comboLeeway = 0.2f; // how fast should the delay between each key p
    public List<RegularSkill> skills; // a list of ALL basic skills player can do
    public List<ComboSkill> combos; // a list of ALL combo we can do, each combo has
}
```

In the inspector, we need to input the skills and combos that this player can do. Drag the respective scriptable objects we created earlier:



Now on to the implementation of `ComboManager.cs` , Declare the following private variables:

```
    // for visual effects
private Animator animator;
private Dictionary<AttackType, ParticleSystem> skillDictionary =
new Dictionary<AttackType, ParticleSystem>();
private Dictionary<ComboType, ParticleSystem> comboDictionary =
new Dictionary<ComboType, ParticleSystem>();

    // logic
private Attack curAttack = null; // currently executed attack, can be regular at
private RegularSkill lastInput;
private List<int> currentPossibleCombosID = new List<int>(); // keep track of ic
private float timer = 0; // to keep track how long current combo will play
private bool skipFrame = false;
private float currentComboLeeway; //to keep track time passed between each skil]
```

C#: Dictionaries

Variables under `visual effects` will be instantiated under `Start()`. These variables hold the **references** to the relevant triggers so that we can indicate to the player that a combo or regular skill is currently happening. We chose to use *Dictionaries* in this example. Like in any other similar programming language, we can initialise Dictionaries in C# using:

`Dictionary<Key, Value> dictionary = new Dictionary<Key,Value>();`

And then we can modify items in it or obtain `Value` given the `Key`:

```
dictionary.Add(newKey, newValue);
Value v = dictionary[inputKey];
bool success = dictionary.Remove(existingKey);
```

Our dictionary `skillDictionary` and `comboDictionary` `Value` field contains *references* to the particle system **component** that has to be triggered (`.Play()`) during runtime whenever this skill or combo is casted. The key to each of these references is the respective `AttackType` or `ComboType` we defined earlier in `RegularSkill.cs` and `ComboSkill.cs`.

```
    void Start()
    {
        animator = GetComponent<Animator>();
        InitializeCombosEffects();
        InitializeRegularSkillsEffects();
    }

    void InitializeCombosEffects()
    {
        // loop through the combos
        for (int i = 0; i < combos.Count; i++)
        {
            ComboSkill c = combos[i];

            // register callback for this combo on this manager
```

```
            c.onInputted.AddListener(() =>
            {
                // Call attack function with the combo's attack
                skipFrame = true; // skip a frame before we attack
                doComboAttack(c);
                ResetCurrentCombos();
            });
            // instantiate
            GameObject comboEffect = Instantiate(c.attack.effect.particleEffect, Vec
            comboEffect.transform.parent = this.transform; // make mario the parent
            // reset its local transform
            comboEffect.transform.localPosition = new Vector3(1, 0, 0);
            // add to particle system list
            comboDictionary.Add(c.comboType, comboEffect.GetComponent<ParticleSystem
        }
    }

    void InitializeRegularSkillsEffects()
    {
        // loop through regular skills effect
        for (int i = 0; i < skills.Count; i++)
        {
            RegularSkill r = skills[i];
            GameObject skillEffect = Instantiate(r.attack.effect.particleEffect, Vec
            skillEffect.transform.parent = this.transform; // make mario the parent
            // reset its local transform
            skillEffect.transform.localPosition = new Vector3(1, 0, 0);
            // add to particle system list
            skillDictionary.Add(r.skillType, skillEffect.GetComponent<ParticleSystem
        }
    }
```

We need three other helper function in `ComboManager.cs` before we can code its logic under `Update()`. First, is the callback when regular attack is casted:

```
    void doRegularAttack(RegularSkill r)
    {
        animator.SetTrigger("CastBasic");
        // Attack(r.attack);
        curAttack = r.attack;
        timer = r.attack.length;
```

```
        // particle cast
        skillDictionary[r.skillType].Play();
    }
```

Second, is the callback when combo attack is casted:

```
    void doComboAttack(ComboSkill c)
    {
        animator.SetTrigger("CastCombo");
        curAttack = c.attack;
        timer = c.attack.length;
        // particle cast
        comboDictionary[c.comboType].Play();
    }
```

Third, is a method to reset current *possible* combo list tracked by `ComboManager` and its combos:

```
    void ResetCurrentCombos()
    {
        currentComboLeeway = 0;
        //loop through all current combos and reset each of them
        for (int i = 0; i < currentPossibleCombosID.Count; i++)
        {
            ComboSkill c = combos[currentPossibleCombosID[i]];
            c.ResetCombo();
        }
        currentPossibleCombosID.Clear();
    }
```

C#: Abstract Class

Since it is imperative for the ComboManager to implement these three helper functions, and that these three functions will only by used by ComboManager and no one else, we can create an **abstract class** (instead of interface which requires the methods to be public because that's what *interface* is for).

> Abstraction in C# is the process to **hide** the internal details and showing only the functionality. The **abstract modifier** indicates the incomplete implementation.

Create a new script, `ComboManagerBase.cs` :

```
using UnityEngine;

public abstract class ComboManagerBase : MonoBehaviour
{
    protected abstract void doRegularAttack(RegularSkill r);
    protected abstract void doComboAttack(ComboSkill c);
    protected abstract void ResetCurrentCombos();
}
```

And inherit this in `ComboManager.cs` :

```
public class ComboManager : ComboManagerBase
```

> Note that you can write **implementations** inside abstract classes (like regular methods, but not shown in this example). If we just want to declare the method, we can add the `abstract` keyword which means that it has no *body* or *implementation* and declared inside the abstract class only. Eventually, an abstract method **must** be implemented in all non-abstract classes inheriting it using the `override` keyword.

Add the keywords: `protected override` in front of these three methods in `ComboManager.cs` to remove the errors.

Combo Manager Logic

Under `Update`, we need to constantly check if there's any last key press, or if the previous effect is playing, and determine if there's any possible combos with the current key press (if any).

Firstly, we need to check if there's current attack that's being casted or playing. We quit immediately and this will be repeatedly call for as many frames that can be fit in `timer`, which contains the length of the currently playing attack.

```
// if current attack is playing, we dont want to disturb it
if (curAttack != null)
{
    if (timer > 0) timer -= Time.deltaTime;
    else curAttack = null;
    return; // end it right here if there's a current attack playing
}
```

Next, we need to check if there's already combos registered in `currentPossibleCombosID`. This array contains the **indexes** of combos that can possibly happen given input in the previous frames:

```
// if current combo is not empty, increase leeway count
if (currentPossibleCombosID.Count > 0)
{
    // increase leeway, this means we are waiting for the next sequence
    currentComboLeeway += Time.deltaTime;
    if (currentComboLeeway >= comboLeeway)
    {
        // if time's up, combo is not happening
        // cast last input if any
        if (lastInput != null)
        {
            doRegularAttack(lastInput);
            lastInput = null;
        }
        ResetCurrentCombos();
    }
}
else
{
```

```
    // no combos currently registered, reset leeway to ensure we don't have
    currentComboLeeway = 0;
}
```

The variable `currentComboLeeway` works as follows:

- Suppose you have a Combo X that requires you to press key A, then D, then C.
- **If you've pressed A in the previous frame**, the **id** of Combo X would've existed inside `currentPossibleCombosID` (will implement this later). Now you *need to press D* within `comboLeeway` which value i set in the inspector.
    - Currently, it is set at 0.2 seconds as example.
- We continuously **increase** `currentComboLeeway` value and *if* it has passed 0.2s, no combo will be happening (means key D isn't pressed at all and time's up).
- We **cast** the **last** known regular attack (the attack invoked when we press A), and **reset** all combo-tracking variables in this script.

Now let's take care of the current input (if any),

```
RegularSkill input = null;
// loop through current skills and see if the key pressed matches
foreach (RegularSkill r in skills)
{
    if (Input.GetKeyDown(r.key.key))
    {
        input = r;
        break;
    }
}

        // return if there's no input currently that matches any skill
if (input == null)
{
    return;
}

        // set current input as last known input
lastInput = input;
```

If there's new input, loop through our current combos to see if it matches the *next input* of current possible combos. We also take note of the combo ID that will *never* happen because current input doesn't trigger that combo anymore (not the supposed next key to press).

```
        List<int> remove = new List<int>();
    // loop through our current combos to see if it continues existing combos
    for (int i = 0; i < currentPossibleCombosID.Count; i++)
    {
        // get the actual combo from the combo ids stored in currentCombos
        ComboSkill c = combos[currentPossibleCombosID[i]];
        // if this input is the next thing to press
        if (c.continueCombo(lastInput.key))
        {
            currentComboLeeway = 0;
        }
        else
        {
            // this combo isn't happening, we need to remove it
            // take note of the combo id
            remove.Add(currentPossibleCombosID[i]);
        }
    }
```

Then finally check if `skipFrame` turns `true` because any combo above is **activated**. This is possible because of the callback earlier in `InitializeCombosEffect`. This callback will happen if `continueCombo()` invokes it, means the current input is the last input i the Combo and will activate the combo.

```
    if (skipFrame)
    {
        skipFrame = false;
        return;
    }
```

If it doesn't return right above, we need to check if there's **new combos** that can be added to the `currentPossibleCombosID` due to current input key (if it is not already tracked):

```
// adding new combos to the currentCombo list with this current last known i
for (int i = 0; i < combos.Count; i++)
{
    if (currentPossibleCombosID.Contains(i)) continue;

    // if it's not being checked already, attempt to add combos into current
    if (combos[i].continueCombo(lastInput.key))
    {
        currentPossibleCombosID.Add(i);
        currentComboLeeway = 0;
    }
}
```

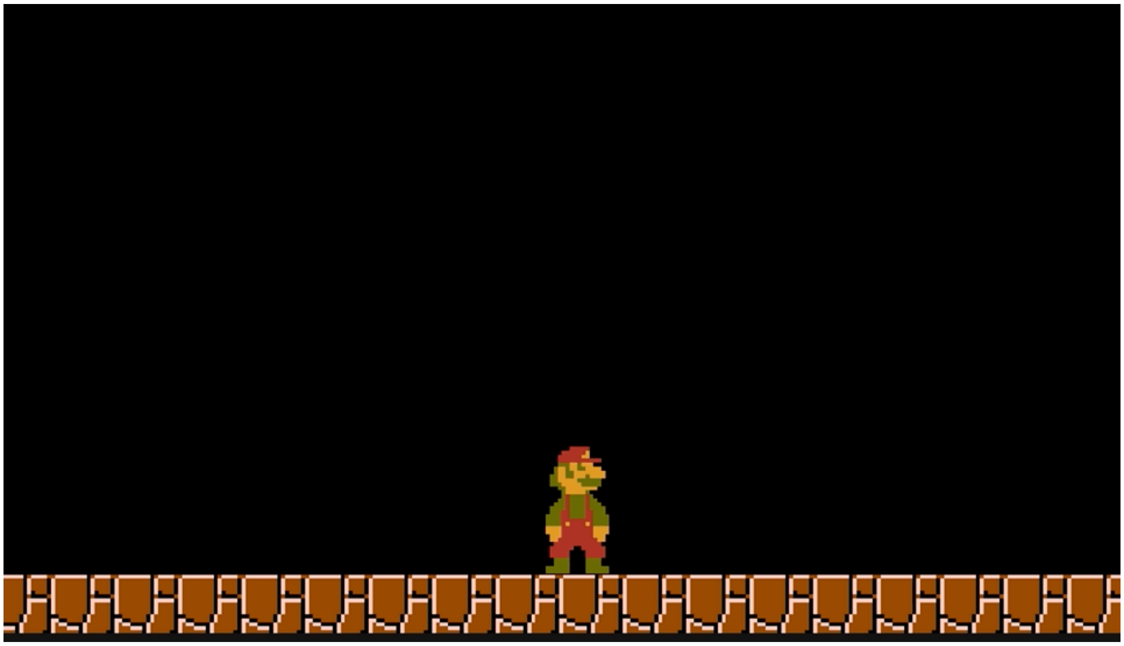
The last two parts of the logic removes any existing combo in `currentPossibleCombosID` that will never happen because the sequence isn't obeyed anymore with current last known input:

```
// remove stale combos from current combos
// recall 'remove' contains combo IDs to remove
foreach (int i in remove)
{
    currentPossibleCombosID.Remove(i);
}
```

… and do basic attack if there's no possible combo that can happen anymore (`currentComboLeeway` expired or the current key doesn't start any combo):

```
// do basic attack if there's no combo
if (currentPossibleCombosID.Count <= 0)
{
    doRegularAttack(lastInput);
}
```

The gif below shows an example of three regular skills, and three combos invoked. It might not be very clear in a gif, so refer to the lab recording if you are stuck.



C#: Async and Await

We have learned about Coroutines before, which is ideal to perform asynchronous operation since we can easily `yield` execution and resume only in the next frame or after a brief period amount of time using `WaitForSeconds`. The documentation for that can be found here.

But what if you need to perform intensive computation, for example something that requires tens of thousand of clock cycles *while still keeping your game responsive*?

To test this, create a new script called `ComputationManager.cs` with the following instructions:

```
using System.Collections;
using System.Threading;
using UnityEngine;
```

```
using System.Threading.Tasks;

public enum method
{
    useVanilla = 0,
    useCoroutine = 1,
    useAsync = 2
}

public class ComputationManager : MonoBehaviour
{
        public  method method;
    public int size;
    private bool calculationState = false;

    void Update()
    {
        if (Input.GetKeyDown("c"))
        {
            if (!calculationState)
            {
                Debug.Log("c is pressed.");
                switch (method)
                {
                    case (method.useVanilla):
                        PerformCalculations();
                        break;
                    case (method.useCoroutine):
                        StartCoroutine(PerformCalculationsCoroutine());
                        break;
                    default:
                        break;
                }
                Debug.Log("Perform calculations dispatch done");
            }
        }

        if (Input.GetKeyDown("q"))
        {
            Destroy(this.gameObject);
        }
    }
}
```

Depending on the value of `method` we set at the inspector later on, the `Update` function is going to call the respective function. Each function performs the same amount of "work", albeit in different manners.

Here's the implementation of the vanilla method. Nothing asynchronous here, therefore depending on the value of `size` , the game will **lag** (render unresponsive) when this method is called.

```
void PerformCalculations()
{
    System.Diagnostics.Stopwatch stopwatch = new System.Diagnostics.Stopwatch();
    stopwatch.Start();
    calculationState = true;
    float[,] mapValues = new float[size, size];
    for (int x = 0; x < size; x++)
    {
        for (int y = 0; y < size; y++)
        {
            mapValues[x, y] = Mathf.PerlinNoise(x * 0.01f, y * 0.01f);
        }
    }
    calculationState = false;
    stopwatch.Stop();
    UnityEngine.Debug.Log("Time taken: " + (stopwatch.Elapsed));
    stopwatch.Reset();
}
```

Here's the same implementation using Coroutine:

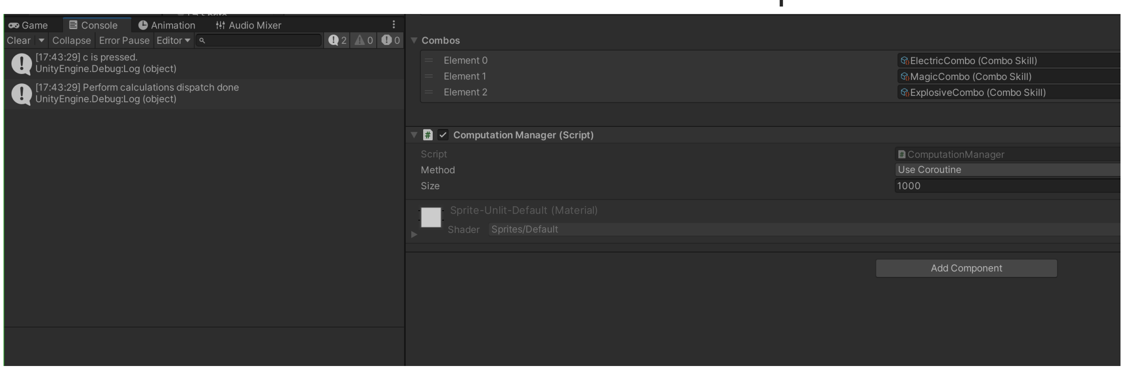
```
IEnumerator PerformCalculationsCoroutine()
{
    System.Diagnostics.Stopwatch stopwatch = new System.Diagnostics.Stopwatch();
    stopwatch.Start();

    calculationState = true;
    float[,] mapValues = new float[size, size];
    for (int x = 0; x < size; x++)
    {
```

```
            for (int y = 0; y < size; y++)
            {
                mapValues[x, y] = Mathf.PerlinNoise(x * 0.01f, y * 0.01f);
                yield return null; // takes super long, only called at 60 times a se
            }
        }
        calculationState = false;
        stopwatch.Stop();
        UnityEngine.Debug.Log("Time taken: " + (stopwatch.Elapsed));
        stopwatch.Reset();
        yield return null;
    }
```

Now attach this to the player (the player in the same scene you use for the combo above), and set `size = 1000` . Set Method to be `useVanilla` , and press the key 'c'. Observe in the console the time taken to execute the computation:

vanilla

Now change the method to `useCoroutine` , notice the output in the console doesn't include the time taken for the function to complete:



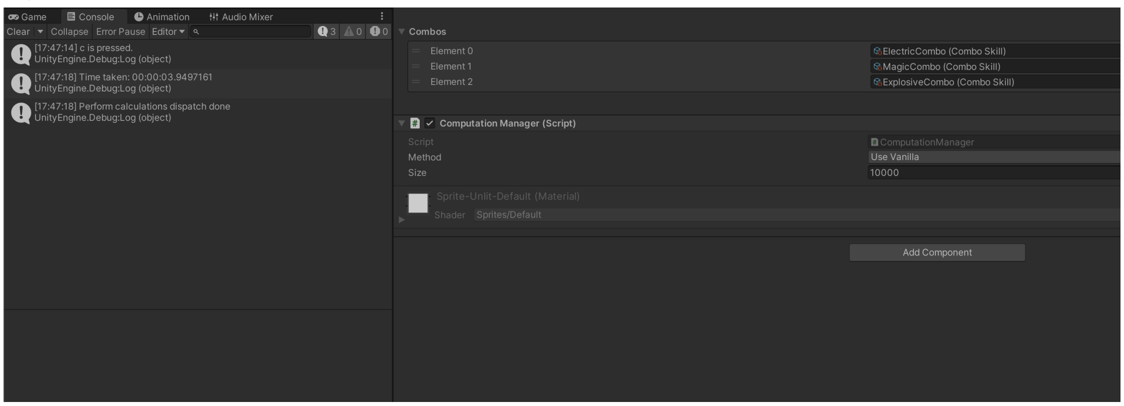
In fact, you have to wait for: $\frac{1000 \times 1000}{60} = 16667\ seconds$ ..because the coroutine `yield` at each inner loop (means the next loop is only resumed at the next frame). This is **way too long** of a wait even though the system stays responsive in the meantime.

We can place `yield return null` in the outer loop as such,

```
        for (int x = 0; x < size; x++)
        {
            for (int y = 0; y < size; y++)
            {
                mapValues[x, y] = Mathf.PerlinNoise(x * 0.01f, y * 0.01f);
            }
            yield return null; // takes super long, only called at 60 times a seconc
        }
```

But even this needs 16.7 seconds to complete although the system will stay responsive.

A bigger problem: what if now `size` is set to `10000` with `useVanilla` method?



In our 2019 16" MBP (8 core 2.3GHz), it takes 3.9 seconds to perform this computation. During this period, the game is **unresponsive**, and this is certainly **not tolerable**. If we were to use coroutine, well, we must wait for approximately 19 days for it to complete if `yield return null` is placed in the inner loop, and 167 seconds if `yield return null` is placed in the outer loop.

With Coroutine, **we cannot utilise the power of our CPU** while staying responsive, and without Coroutine our game wont even stay responsive because all resources is dedicated to execute this hefty computation until completion.

In order to **utilise our CPU** while staying responsive, we can use `async` function and `await` for `Task` completion. Remember that `Update` is only called 60 times a second (where we check for inputs, execute basic game logic, etc), so there's plenty of leftover time that we can use to complete this `calculation` function.

We can declare an `async` function with the `async` keyword:

```
async void PerformCalculationsAsync()
{
}
```

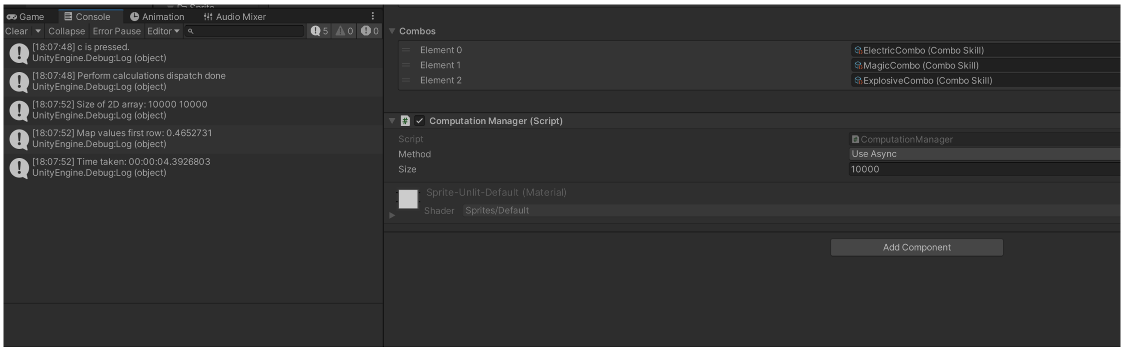
Calling an async function **without any await** results in synchronous execution. We need to `await` some `Task` as such:

```
async void PerformCalculationsAsync()
{
    System.Diagnostics.Stopwatch stopwatch = new System.Diagnostics.Stopwatch();
    stopwatch.Start();
    var result = await Task.Run(() =>
    {
        calculationState = true;
        float[,] mapValues = new float[size, size];
        for (int x = 0; x < size; x++)
        {
            for (int y = 0; y < size; y++)
            {
                mapValues[x, y] = Mathf.PerlinNoise(x * 0.01f, y * 0.01f);
            }
        }
        return mapValues;
    });

    calculationState = false;
    Debug.Log("Size of 2D array: " + result.GetLength(0) + " " + result.GetLengt
    Debug.Log("Map values first row: " + result[0, 0]);
    stopwatch.Stop();
    UnityEngine.Debug.Log("Time taken: " + (stopwatch.Elapsed));
    stopwatch.Reset();

}
```

Now simply add another case in `Update` to test this function:

```
case (method.useAsync):
    PerformCalculationsAsync();
    break;
```

Testing the `async` method with `size=10000` results in the following output:



This shows that using `async` function **does not** (necessarily) make any computation time faster than the `vanilla` method, but in the meantime, the system is still **responsive**.

> In short: the `async` function, very similar to JavaScript, is not executed in another thread. Instead, the function executes on the main thread, and only when the `await` keyword appears, the function executed may or **MAY NOT** on the main thread.

Comparisons with Coroutine

This section briefly covers the comparison between the two. There's no *better or worse solution*, and you can simply choose the solution that suits your project best. The content for this section is distilled from this video.

Async Functions Always Complete

Async functions **always** runs into completion, while Coroutines are run on the GameObject. Therefore, **disabling** the gameobject will cause any coroutine running on it to **stop** but *doesn't exit naturally*. This can potentially result in memory leak.

For example, suppose we instantiate `RenderTexture` in a Couroutine:

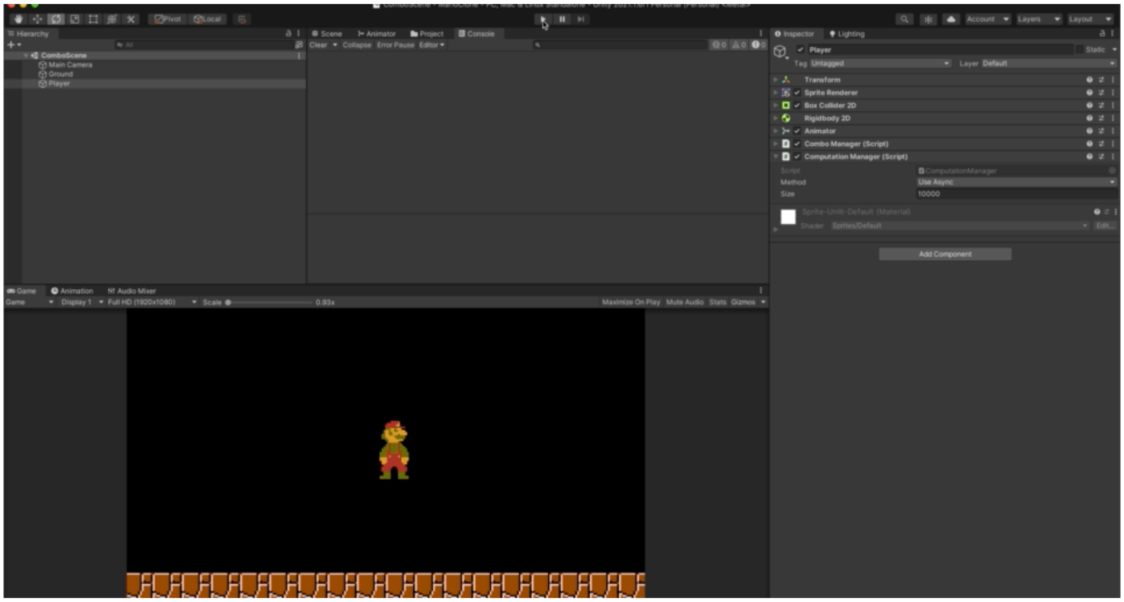
```
IEnumerator RenderEffect(RawImage r){
        var texture = new RenderTexture(1024, 1024, 0);
        try{
                for (int i = 0; i<1000; i++){
                        // do something with r and texture
                        yield return null;
                }
        }
        finally{
                texture.Release();
        }
}
```

As per Unity Official Documentation, `RenderTexture` is **not automatically managed**, therefore it is important to call `Release()` after we are done with it.

> As with other "native engine object" types, it is important to pay attention to the lifetime of any render textures and `release` them when you are finished using them, as they will not be garbage collected like normal managed types.

Now if the MonoBehavior (the gameobject) running this script is disabled, the `finally` clause never called, thus resulting in **memory leak**.

On the other hand, `async` function continues to run **even after the MonoBehavior is destroyed**. You can test this very easily by setting `Size = 10000`, and method: `useAsync`, run the the project quickly and press `c` then quickly stop it. The output will still appear at Console even after the program exits. This shows that `async` function **always exits**.



Cancelling Async Functions

To be more sure that Coroutines always exit, we need to be mindful to `StopCoroutine(...)` during `onDisable` the gameobject. Likewise, we can also *cancel* async functions using *cancellation tokens*.

You simply just need to declare it beforehand,

```
CancellationTokenSource token;
```

and initialise it at `Start()` :

```
token = new CancellationTokenSource();
```

Then simply pass this token when defining `Task` , and check it wherever you want *inside the Task*,

```
        var result = await Task.Run(() =>
        {
                calculationState = true;
            float[,] mapValues = new float[size, size];
            // ... implementation
            for (.....){
```

```
                // ... implementation
                // periodically check for cancellation token request
            if (token.IsCancellationRequested)
            {
               Debug.Log("Task Stop Requested");
               return mapValues;
             }
        }
        return result;
    }, token.Token);
```

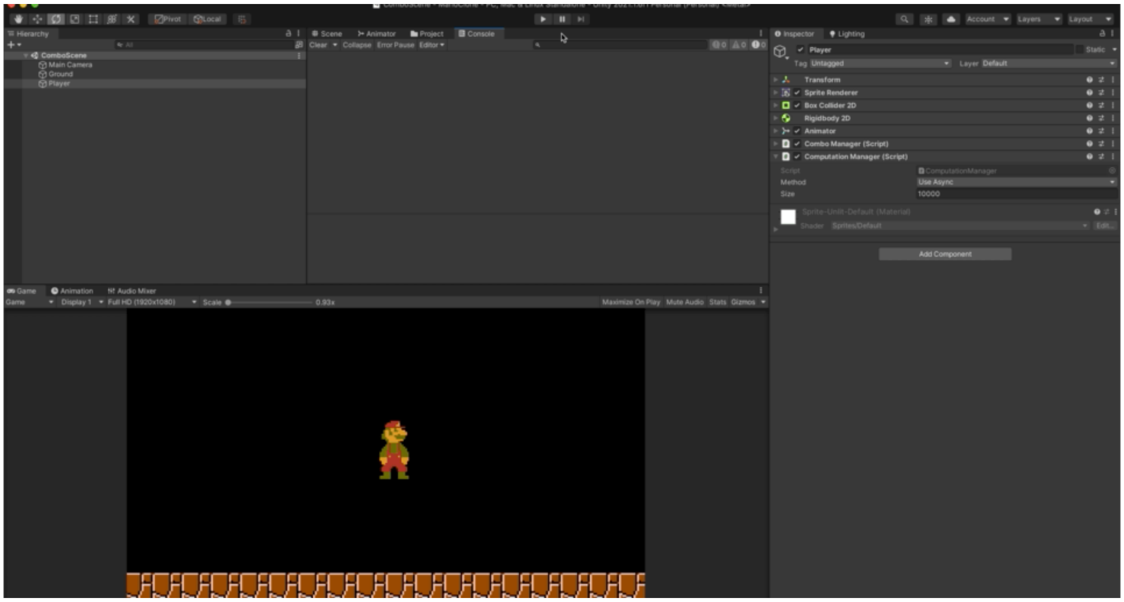
… or inside the `async` function that awaits that task:

```
    if (token.IsCancellationRequested)
    {
        Debug.Log("Task Stopped");
        return;
    }
```

We can create cancellation request as follows, for example:

```
private void OnDisable()
{
    Debug.Log("itemDisabled");
    token.Cancel();
}
```

This way, the async function will stop once the gameObject is disabled:



Return Values in Async Functions

We cannot return anything in a Coroutine, but async functions can the following return types:

- `Task` , for an async method that performs an operation but returns no value.
- `Task<TResult>` , for an async method that returns a value.
- `void` , for an event handler.

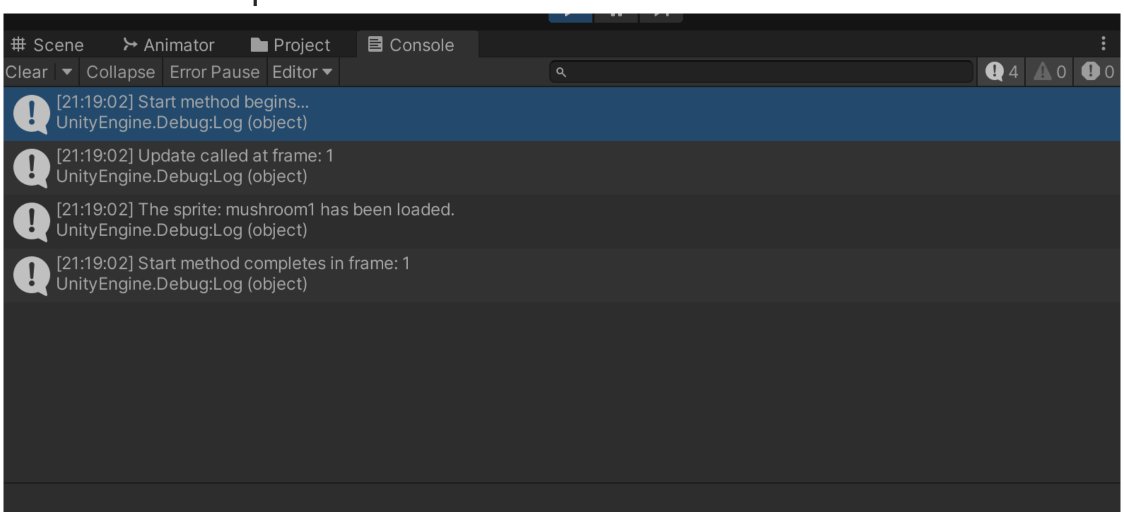
For example, the following async function returns a `sprite` :

```
async Task<Sprite> LoadAsSprite_Task(string path)
{
    // getting sprite inside Assets/Resources/ folder
    var resource = await Resources.LoadAsync<Sprite>(path);
    return (resource as Sprite);
}
```

We can call them as such in `Start()` (notice how the `Start` method has to be `async` now to `await` this `Task`), and print some quick test in `Update()` to **confirm** if `Update()` is run at least for one frame before `Start()` is continued, and we can obtain some information about the **return value** of `LoadAsSprite_Task` *async* function:

```
    private bool testTask = false;
    private int frame = 0;
    async void Start()
    {
        Debug.Log("Start method begins...");
        token = new CancellationTokenSource();
        var sprite = await LoadAsSprite_Task("mushroom1");
        Debug.Log("The sprite: " + ((Sprite)sprite).name + " has been loaded.");
        Debug.Log("Start method completes in frame: " + frame.ToString());
        testTask = true;
    }
    void Update()
    {
        frame++;
        if (!testTask)
            Debug.Log("Update called at frame: " + frame);
     }
```

Here's the console output:



It shows that `Start` is called first as usual, but **asynchronously**, allowing `Update` to advance and increase the frame value. When the sprite has been loaded, the `Start` method resumes and print the `Start method completes...` message.

UniTask

Finally before we conclude, we'd like to introduce you to an alternative called `UniTask`. Not only a nicer replacement for Unity Coroutine and C# async-await, UniTask also provides a nicer background *thread management*. The complete documentation and installation details can be obtained here.

You can download it as UnityAsset and import it to your project. We won't be going into details on how to utilise UniTask in your project, only to quickly introduce to you because it is a good and popular alternative. Among other things, UniTask is capable of:

- Making all Unity AsyncOperations and Coroutines **awaitable**
- Running completely on Unity's PlayerLoop so doesn't use threads and runs on WebGL, wasm, etc
- Summoning TaskTracker window to prevent memory leaks

After importing the asset, you can declare the namespace as such:

```
using Cysharp.Threading.Tasks;
```

You can implement an async function as usual that will return `UniTask` this time round:

```
async UniTask<Sprite> LoadAsSprite(string path)
{
    // getting sprite inside Assets/Resources/ folder
    var resource = await Resources.LoadAsync<Sprite>(path);
    return (resource as Sprite);
}
```

Then `await` that in the caller:

```
private async void TestUniTask()
{
        // parallel load, and will complete when all of the supplied tasks have c
    var (a, b) = await UniTask.WhenAll(
```

```
            LoadAsSprite("gomba1"),
            LoadAsSprite("gomba2"));
        Debug.Log("The sprite: " + ((Sprite)a).name + " has been loaded.");
        Debug.Log("The sprite: " + ((Sprite)b).name + " has been loaded.");
        await  UniTask.Delay(2000); // introduce delay purposely for learning purpos
        Debug.Log("TestUniTask completed at frame: "  +  frame);
    }
```

You can simply test this in `Update()` using some flag `bool testUniTask=false` instantiated in the beginning, and then call:

```
                frame++;
        if (Input.GetKeyDown("t") && !testUniTask)
        {
                Debug.Log("TestUniTask called at frame: "  +  frame);
            TestUniTask();
            testUniTask = true;
        }
```

You should see the log message in this exact sequence:

- `TestUniTask called...`
- `The Sprite ... has been loaded`
- `The Sprite ... has been loaded`
- `TestUniTask completed...` but `frame` should've advanced by a few values.

Switching Between Thread Pool and Main Thread

Another cool feature of UniTask is that you can switch the current context execution to the thread pool instead of the main thread easily. For example, try out this function:

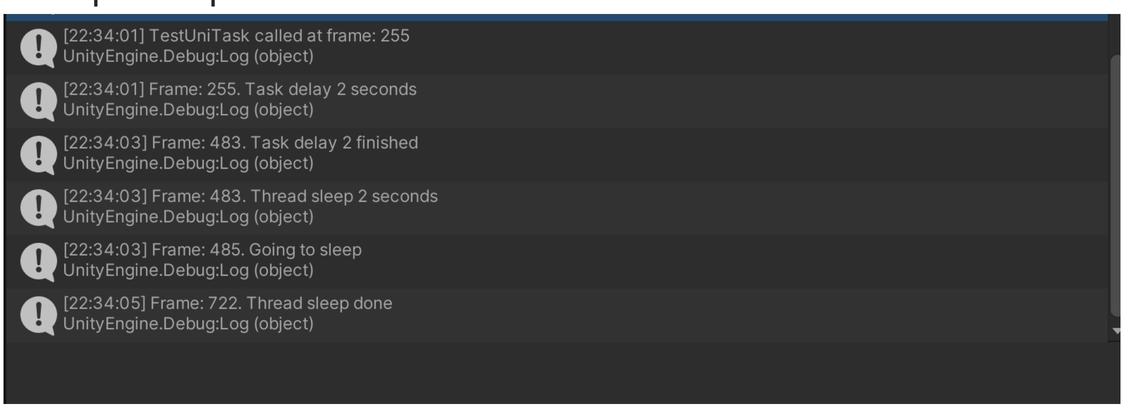
```
    private async void TestUniTask()
    {
        Debug.Log("Frame: " + frame.ToString() + ". Task delay 2 seconds");
        await UniTask.Delay(2000);
```

```
Debug.Log("Frame: " + frame.ToString() + ". Task delay 2 finished");
Debug.Log("Frame: " + frame.ToString() + ". Thread sleep 2 seconds");
await UniTask.SwitchToThreadPool();
Debug.Log("Frame: " + frame.ToString() + ". Going to sleep");
Thread.Sleep(2000);
await UniTask.SwitchToMainThread();
Debug.Log("Frame: " + frame.ToString() + ". Thread sleep done");
}
```

And just call it at will in `Update` to test:

```
        frame++;
if (Input.GetKeyDown("t") && !testUniTask)
{
    Debug.Log("TestUniTask called at frame: " + frame);
    TestUniTask();
    testUniTask = true;
}
```

Here's a sample output:



Observe that the `Frame` **increases** at each even though we call `Thread.Sleep(2000)` because **we have switched to thread pool** before calling that instruction. Otherwise, `Thread.Sleep(2000)` will **block** on the **main thread** instead (because we aren't `await` -ing anything in that line) and cause the main thread to block, rendering the game unresponsive for two seconds.

Summary

There’s no checkoff associated with this tutorial, but the contents of this tutorial will be tested for our midterms.

Next

In the next tutorial, we will learn about basics in 3D Unity Projects, and also learn about *pathfinding* in that environment.

PREVIOUS POST
Unity for Midlifes

NEXT POST
Unity for Teens