Fourth Workshop Notes

By Peter Mitchell

Contents

1 Foreword

You can find all the materials including this document at the GitHub URL seen below. In addition to this, I have made my UNO and TestTube games made in Java available in case you would like to browse patterns I have used in these games.

Fourth Workshop GitHub URL: https://github.com/Squirrelbear/FourthWorkshopDemoProject

UNO (Java): https://github.com/Squirrelbear/Uno

TestTube (Java): https://github.com/Squirrelbear/TestTube

2 Coroutines

Coroutines: https://docs.unity3d.com/Manual/Coroutines.html

You can write methods known as Coroutines to run asynchronously alongside your code. These methods do not have to finish their jobs right away. You can have them perform actions over time or have them perform actions until any arbitrary condition is met. There are a variety of these in the example project files. A single example is shown below. The important parts to look at here are that there is a method with the return type set to IEnumerator, it has a yield line in it, and there is a StartCoroutine() method call in Update.

- IEnumerator: You must return this from any method you want to use as something run as a coroutine.
- Yield: In the example code below you can see it has yield return new WaitForSeconds(1f);. This means that the method will exit for 1 second and then resume. If you want code to run without pauses asynchronously you can use yield return null;
- StartCoroutine(): This method is what starts the coroutine you need to call the method by passing it to this for it to be asynchronous.

```
public class ExampleCoroutine : MonoBehaviour
{
    private Vector3 origin;
    // Start is called before the first frame update
    void Start()
    {
        origin = transform.position;
    }

    // Update is called once per frame
    void Update()
    {
        if(Input.GetKeyDown(KeyCode.Space))
        {
            StartCoroutine(movePlaces());
        }
    }
```

```
    private IEnumerator movePlaces()
    {
        for(int i = 0; i < 10; i++)
        {
            Vector3 modifiedPosition = new Vector3(
                    Random.Range(origin.x - 4, origin.x + 4),
                    Random.Range(origin.y - 4, origin.y + 4),
                    Random.Range(origin.z - 4, origin.z + 4));
            transform.position = modifiedPosition;
            yield return new WaitForSeconds(1f);
        }
        transform.position = origin;
    }
}
```

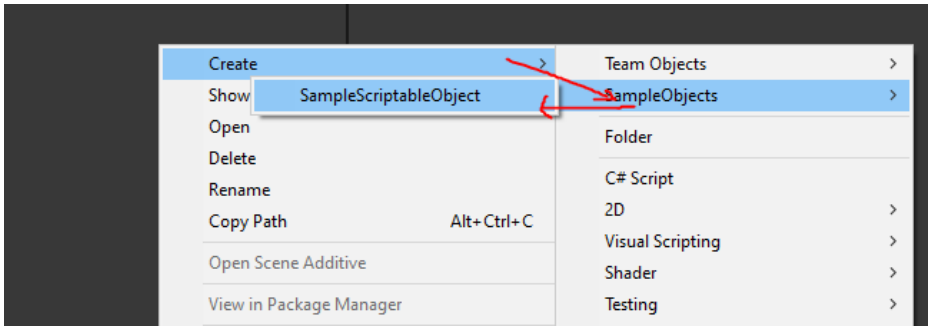
3 ScriptableObjects

ScriptableObjects let you define data as objects within Unity that can have behaviours/scripts as part of them have them be editable in the Inspector. You can find a variety of these examples spread across the core project files for defining the combat. A simple example though would be something like the following.

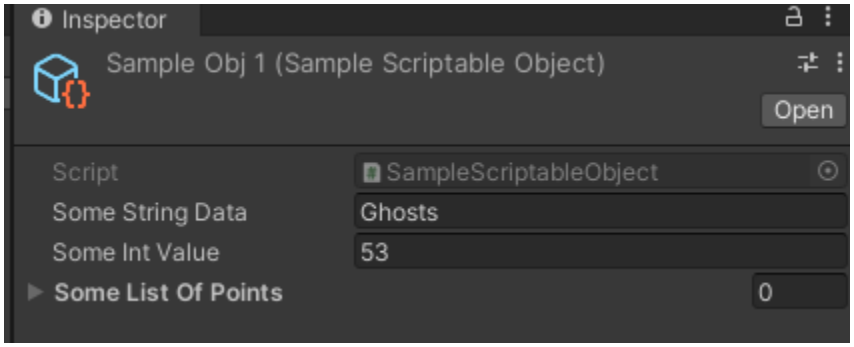
```
[CreateAssetMenu(fileName = "SampleScriptableObject", menuName =
"SampleObjects/SampleScriptableObject", order = 1)]
public class SampleScriptableObject : ScriptableObject
{
       public string someStringData;
       public int someIntValue;
       public Vector3[] someListOfPoints;
}
```

In this example script you can see that we have provided a CreateAssetMenu section above the class. This allows us to define generation of objects based on this script. Importantly it also extends from the ScriptableObject class. In this case it just has some data included, but you could very easily extend this with some methods if you wanted to as well.

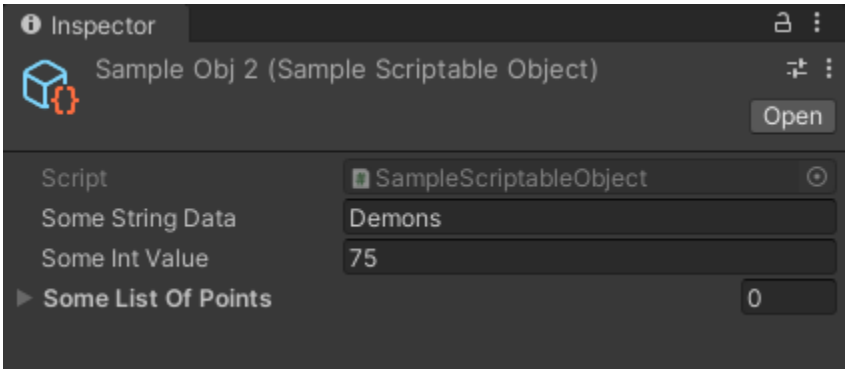
You can see that the menuName as set to “SampleObjects/SampleScriptableObject” this means that we will see the below when we right click to create objects.



After creating the object, you can give it some values like below.



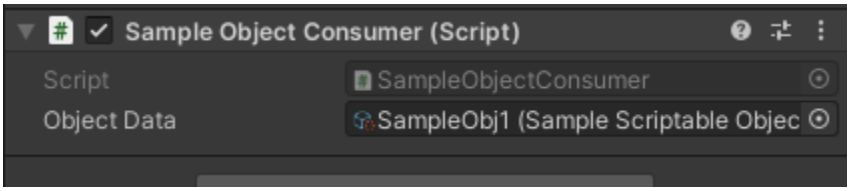
I also created a second object like below as well.



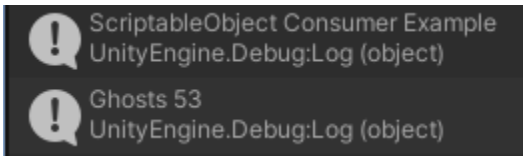
So now I have two objects that have data using the SampleScriptableObject class. To use those in a practical sense you can have some other script that takes an object of type SampleScriptableObject and this allows you to provide the objects above as the data source. A script to do this may look like below.

```
public class SampleObjectConsumer : MonoBehaviour
{
    [SerializeField] private SampleScriptableObject objectData;

    // Update is called once per frame
    void Update()
    {
        if(Input.GetKeyDown(KeyCode.Alpha1))
        {
            Debug.Log("ScriptableObject Consumer Example");
            Debug.Log(objectData.someStringData + " " + objectData.someIntValue);
        }
    }
}
```

You can attach this to an object and drag one of your generated SampleScritableObjects into the field. This would look something like below.



You can test running with this data injected by pressing “1”. You should see with the first data that it comes out with the following.



You can change the object data to the other data, and you should see the different output print out.

3.1 Resources to Learn More

Unity Introduction: https://learn.unity.com/tutorial/introduction-to-scriptable-objects#

Ability System: https://learn.unity.com/tutorial/create-an-ability-system-with-scriptable-objects

Character Select System: https://learn.unity.com/tutorial/create-a-character-select-system-with-scriptable-objects

4 Game Design Patterns

You can find additional examples of these patterns with good discussion at: https://www.youtube.com/watch?v=hQE8lQk9ikE

4.1 Singleton Pattern

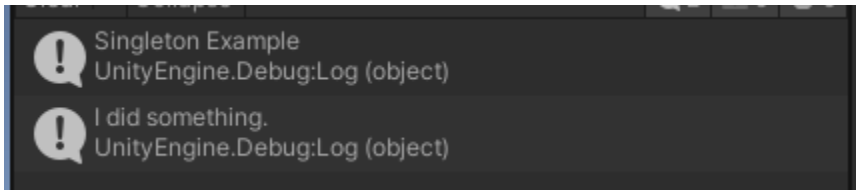
This is a pattern where you intend to only have a single instance of that class ever as part of your project. You can create a reference inside it to let the class share its own existence. This is useful for scripts that manage game wide events, or other state information that is not going to be duplicated for multiple object instances.

The example below shows the instance variable providing access to this class. It is assigned in the Awake method and then can be accessed like shown in the SingletonCaller class seen after.

```
public class SingletonExample : MonoBehaviour
{
	public static SingletonExample instance { get; private set; }

	void Awake()
```

```
        {
                instance = this;
        }

        public void doSomethingMethod()
        {
                Debug.Log("I did something.");
        }
}

public class SingletonCaller : MonoBehaviour
{
    void Update()
    {
        if(Input.GetKeyDown(KeyCode.Alpha2))
        {
            Debug.Log("Singleton Example");
            SingletonExample.instance.doSomethingMethod();
        }
    }
}
```

Pressing ‘2” on the keyboard should trigger output that appears like the following.



4.2 Observer Pattern

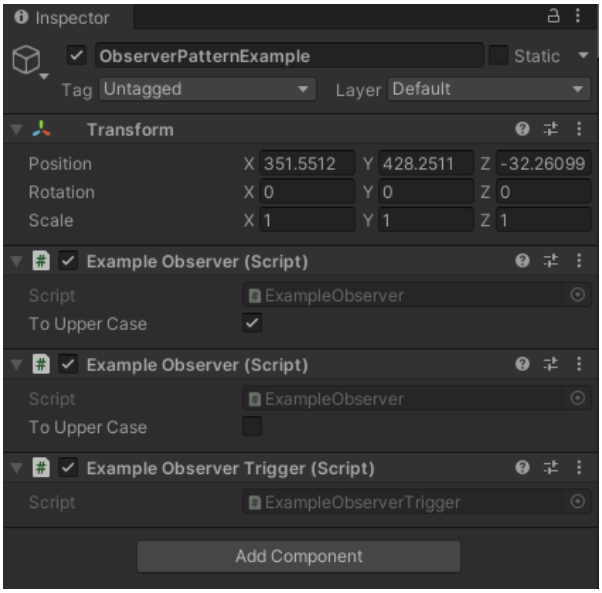
The observer pattern provides a simple way to separate objects to be self-sufficient. It can be messy to have an object need to manually call lots of other objects to trigger their ways of handling what happens when some trigger happens. By using events you can set up an event that sits waiting for a trigger to happen. The trigger does not care who is listening, just notifying that something has happened. Then any observers who were listening to the event can handle their own custom logic.

The below code shows the simple event that sends out “Hello” when the “4” key is pressed.

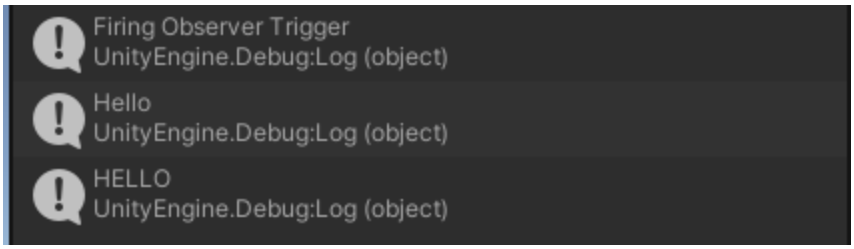
```
public class ExampleObserverTrigger : MonoBehaviour
{
    public static event Action<string> OnNewMessageEvent;

    private void Update()
    {
        if(Input.GetKeyDown(KeyCode.Alpha4))
        {
            Debug.Log("Firing Observer Trigger");
            OnNewMessageEvent?.Invoke("Hello");
        }
    }
}
```

To listen to this event there is a simple observer seen below. The observer registers to listen for the events when it is enabled and stops listening when it is disabled. When the event comes through it performs logic based on the value of the event. In this case, it either transforms the string to become upper case or leaves the string as it was before printing it out.

```
public class ExampleObserver : MonoBehaviour
{
    public bool toUpperCase;

    private void OnEnable() => ExampleObserverTrigger.OnNewMessageEvent += printMessage;
    private void OnDisable() => ExampleObserverTrigger.OnNewMessageEvent -= printMessage;

    private void printMessage(string value)
    {
        if(toUpperCase)
        {
            Debug.Log(value.ToUpper());
        }
        else
        {
            Debug.Log(value);
        }
    }
}
```



You can see above that there are two observers watching for the trigger. One of the observers is set to change the message to upper case and the other is set to not modify it at all. In practice you would likely have these spread over multiple objects, but for the example here it is all shown on the single object. By pressing “4” on the keyboard you should see output that looks like the following.



4.3 Command Pattern

The command pattern allows you to queue up a sequence of actions that can occur in a specific sequence. Depending on what you are wanting to do with it, this could include commands that can be applied as a "do" or "undo". Another option is to only execute commands after the previous one is finished. The important part though is that by creating a base abstract command and then many command types that inherit from that base command you can create a diverse set of different powerful commands.

You can find a Java based example below where I replicated a colour sorting game with a Command pattern included to allow undo.

https://github.com/Squirrelbear/TestTube

You can find a far more advanced version of the command pattern with tree branching execution in my UNO implementation at the following.

https://github.com/Squirrelbear/UNO

For an example in Unity first you need to define a command. This could be something like as follows.

```
public abstract class SampleAbstractCommand
{
    public abstract void executeCommand();
    public abstract void undoCommand();
}
```

Then using this as a base class you can define a couple of different commands. These could be something more relevant like moving to a position or building an object in the context of something like an RTS game, but in this case, they are just going to output messages when they are executed.

```
public class SampleDoSomethingCommand : SampleAbstractCommand
{
    private int variable;

    public SampleDoSomethingCommand(int variable)
    {
        this.variable = variable;
    }
```

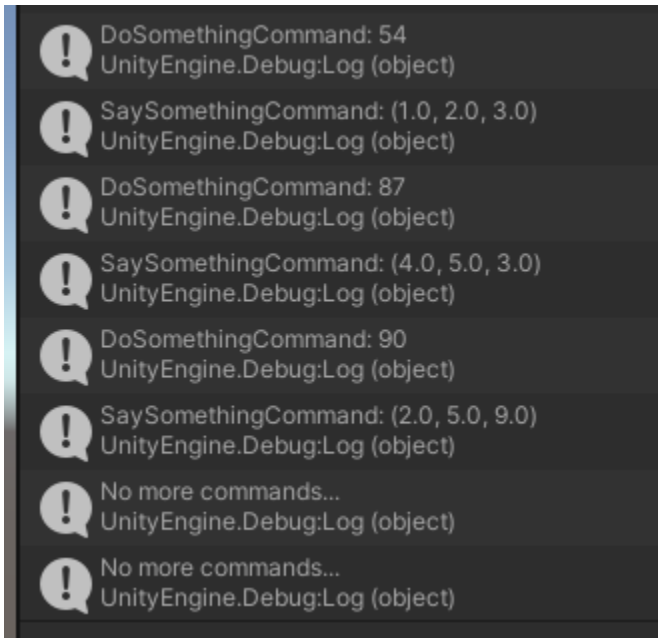
```
    public override void executeCommand()
    {
        Debug.Log("DoSomethingCommand: " + variable);
    }

    public override void undoCommand()
    {
        // Could reverse based on stored variables
    }
}

public class SampleSaySomethingCommand : SampleAbstractCommand
{
    private Vector3 variable;

    public SampleSaySomethingCommand(Vector3 variable)
    {
        this.variable = variable;
    }

    public override void executeCommand()
    {
        Debug.Log("SaySomethingCommand: " + variable);
    }

    public override void undoCommand()
    {
        // Could reverse based on stored variables
    }
}
```

Then to use these commands you would have some form a queue that sequences the commands into a structured order. This could be like below.

```
public class SampleCommandPlayer : MonoBehaviour
{
    public Queue<SampleAbstractCommand> commandQueue;

    // Start is called before the first frame update
    void Start()
    {
        commandQueue = new Queue<SampleAbstractCommand>();
        commandQueue.Enqueue(new SampleDoSomethingCommand(54));
        commandQueue.Enqueue(new SampleSaySomethingCommand(new Vector3(1, 2, 3)));
        commandQueue.Enqueue(new SampleDoSomethingCommand(87));
        commandQueue.Enqueue(new SampleSaySomethingCommand(new Vector3(4, 5, 3)));
        commandQueue.Enqueue(new SampleDoSomethingCommand(90));
        commandQueue.Enqueue(new SampleSaySomethingCommand(new Vector3(2, 5, 9)));
    }

    // Update is called once per frame
    void Update()
    {
        if(Input.GetKeyDown(KeyCode.Alpha3))
        {
            if (commandQueue.Count > 0)
            {
```

```
                SampleAbstractCommand command = commandQueue.Dequeue();
                command.executeCommand();
            }
            else
            {
                Debug.Log("No more commands...");
            }
        }
    }
}
```

When running this attached to an object by pressing “3” it will print out the following. Where it executes the commands in order as defined and then tells us there are no more commands when we try and execute more.



4.4 Component Pattern

This is prevalent in all of what is being done in Unity. Instead of all objects inheriting from each other, you can have objects that exist as components within something else. Those objects can be mostly self-sufficient, and objects may be able to communicate between each other. When you have multiple components on objects this is essentially what is happening. You can do the same thing in your scripts where you create multiple objects and let them manage themselves.

4.5 Flyweight Pattern

In some cases, there are protections in place to prevent data being directly modified. There may still be a way provided to update that information while keeping it immutable to the outside except through a specific interface. An example of this is how the Renderer works in Unity. When you try and change properties directly of a material it will go and create a new material instance

instead of modifying the existing instance. This is significant because it means the one that is thrown away goes to garbage collection. If you are doing this constantly (like every update) it will add up a lot of extra work.

First an example of the wrong way to do it where you are directly setting changes to the material.

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

public class FlyweightWrong : MonoBehaviour
{
    private Renderer _renderer;
    void Awake()
    {
        _renderer = GetComponent<Renderer>();
    }

    private Color randomColor()
    {
        return new Color(Random.Range(0f, 1f), Random.Range(0f, 1f), Random.Range(0f,
1f));
    }

    void Update()
    {
        // This is bad because it creates a copy of the material causing a lot of memory
use if you do it frequently.
        _renderer.material.color = randomColor();
    }
}
```

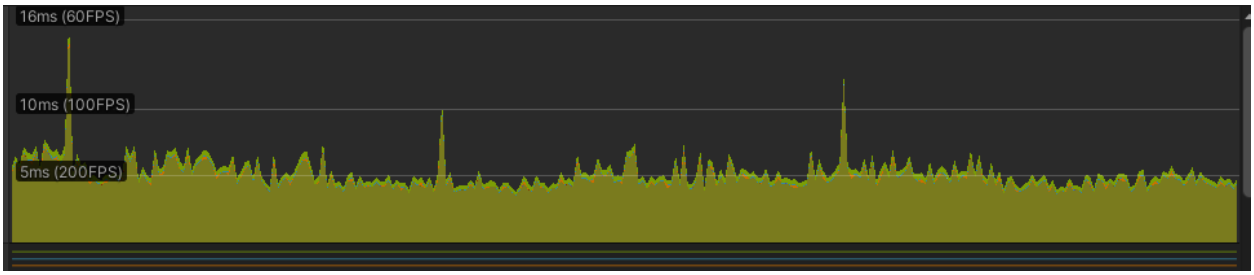
And now the correct way where instead you retrieve the properties and pass back the modified set of properties.

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

public class FlyweightCorrect : MonoBehaviour
{
    private Renderer _renderer;
    private MaterialPropertyBlock _propBlock;

    void Awake()
    {
        _renderer = GetComponent<Renderer>();
        _propBlock = new MaterialPropertyBlock();
    }

    private Color randomColor()
    {
        return new Color(Random.Range(0f, 1f), Random.Range(0f, 1f), Random.Range(0f,
1f));
    }
```

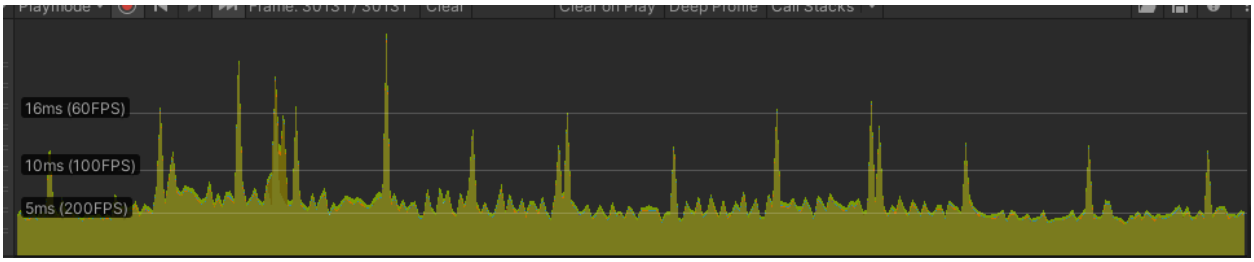
```
    void Update()
    {
        // Get current properties
        _renderer.GetPropertyBlock(_propBlock);
        // Change the properties (does not apply to the renderer yet)
        _propBlock.SetColor("_Color", randomColor());
        // Apply changed properties back to renderer
        _renderer.SetPropertyBlock(_propBlock);
    }
}
```

You can view the profile by going to Window->Analysis->Profiler.

If you do this with the correct object enabled, but the wrong object disabled you will see a consistent low pattern like below.



With the wrong object enabled, and the correct object disabled it will look like below. Where the spikes indicate that the CPU is having to do lots more in bursts of work dealing with the garbage collection.



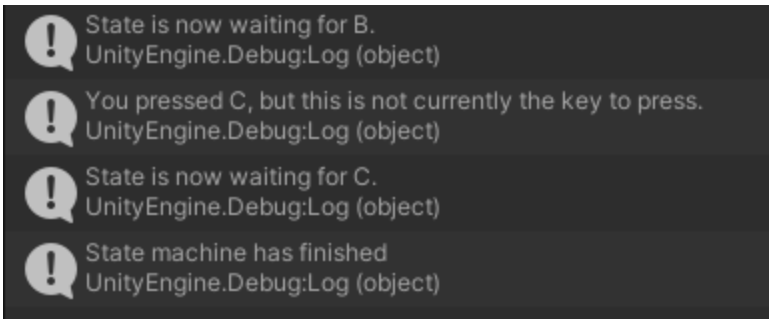
4.6 State Pattern

States can be defined with many different types of variables. One way to do it is to have an enum type object to define clearly what your states are with rules of how those states are transitioned between. The code below defines a set of states, the initial state (in Start()), then a method to check for transitions between the states with messages to display context about the attempted state transitions.

```
public class StateMachineExample : MonoBehaviour
{
    public enum ExampleState { WaitForPressA, WaitForPressB, WaitForPressC, Done }

    public ExampleState currentState;

    // Start is called before the first frame update
    void Start()
```

```
    {
        currentState = ExampleState.WaitForPressA;
    }

    // Update is called once per frame
    void Update()
    {
        if(Input.GetKeyDown(KeyCode.A) || Input.GetKeyDown(KeyCode.B) ||
Input.GetKeyDown(KeyCode.C))
        {
            moveToNextStateCheck();
        }
    }

    private void moveToNextStateCheck()
    {
        if(Input.GetKeyDown(KeyCode.A))
        {
            if(currentState == ExampleState.WaitForPressA)
            {
                currentState = ExampleState.WaitForPressB;
                Debug.Log("State is now waiting for B.");
            }
            else
            {
                Debug.Log("You pressed A, but this is not currently the key to press.");
            }
        }
        else if (Input.GetKeyDown(KeyCode.B))
        {
            if (currentState == ExampleState.WaitForPressB)
            {
                currentState = ExampleState.WaitForPressC;
                Debug.Log("State is now waiting for C.");
            }
            else
            {
                Debug.Log("You pressed B, but this is not currently the key to press.");
            }
        }
        else if (Input.GetKeyDown(KeyCode.C))
        {
            if (currentState == ExampleState.WaitForPressC)
            {
                currentState = ExampleState.Done;
                Debug.Log("State machine has finished");
            }
            else
            {
                Debug.Log("You pressed C, but this is not currently the key to press.");
            }
        }
    }
}
```

Running this code, you would expect when pressing the buttons in the order A, C, B, C, you would see the following.



5 SOLID Principles

For each of these if you want to know more about how they apply specifically to Unity, watch the linked videos I have provided below.

5.1 Single Responsibility Principle

"There should never be more than one reason for a class to change. In other words, every class should have only one responsibility."

Video: https://www.youtube.com/watch?v=Eyr7_l5NMds

5.2 Open Closed Principle

“Software entities ... should be open for extension but closed for modification.”

Video: https://www.youtube.com/watch?v=wYkzeKghjsI

5.3 Liskov Substitution Principle

“Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.”

Video: https://www.youtube.com/watch?v=eXPBR3WlRGY

5.4 Interface Segregation Principle

“Many client-specific interfaces are better than one general-purpose interface.”

Video: https://www.youtube.com/watch?v=ll6bxQGkyCk

5.5 Dependency Inversion Principle

“Depend upon abstractions, not concretions.”

Video:https://www.youtube.com/watch?v=fGshe3ILKnA

6 Other Information

Variety of Other Patterns If Interested

https://www.youtube.com/playlist?list=PLB5_EOMkLx_VOmnIytx37lFMiajPHppmj

Intro to using Mixamo with Unity

https://www.youtube.com/watch?v=-FhvQDqmgmU