State Graph User Guide

by @darkrainbowsprinkles

⚠️ Unity Version Compatibility

This tool is designed for **Unity 2021.2 or newer**. It uses UI Toolkit and the experimental GraphView API.

Section 1: What is a state machine?

A **state machine** is a design pattern used to manage the different states an object can be in, along with the rules for transitioning between those states.

In simpler terms, it's like giving your GameObject a brain that knows:

- What it’s currently doing (Idle, Running, Attacking, etc.)
- When and how to switch to something else based on certain conditions

Each **state** represents a specific behavior, and the transitions between them define how and when those behaviors change.

Example: Enemy AI

Imagine an enemy with these behaviors:

- ***Patrol***: Walks around a path
- ***Chase***: Runs after the player when spotted
- ***Attack***: Tries to hit the player when close

Using a state machine:

- The enemy starts in the ***Patrol*** state
- When the player is seen, it **transitions** to the ***Chase*** state
- When close enough, it **transitions** to ***Attack***
- If the player escapes, it may return to ***Patrol***

This makes complex behavior easier to manage, extend, and debug.

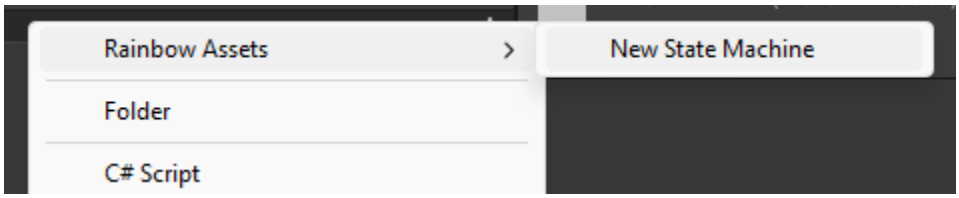
Section 2: Creating a custom state machine

To create a new state machine in your project:

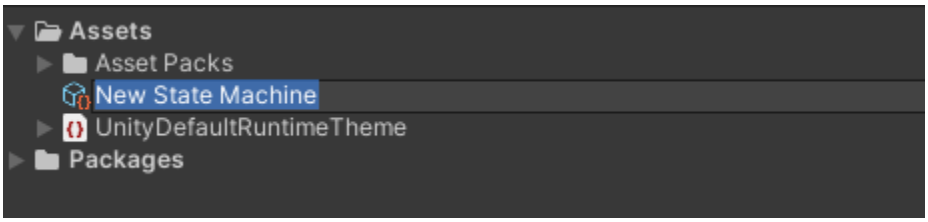
1. Open your Unity project.

2. In the **project window**, navigate to the folder where you want to create your state machine asset.

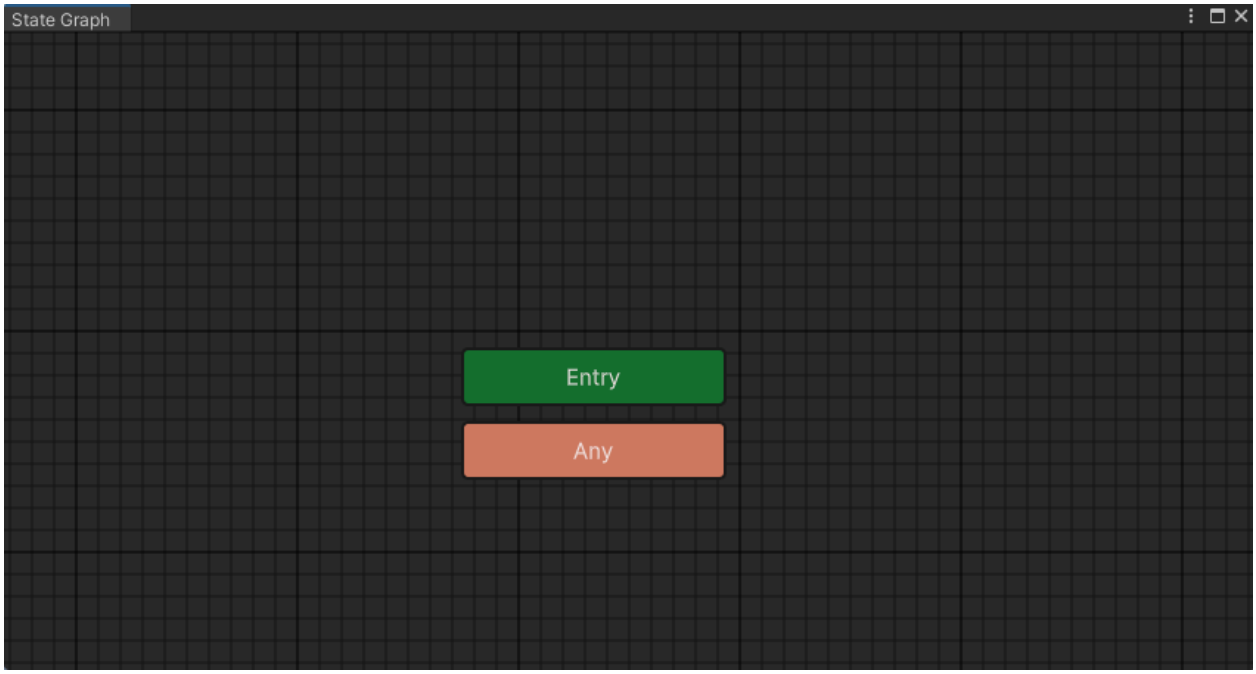
3. **Right-click** inside the folder, then select:
   `Create → Rainbow Assets → New State Machine`



4. Unity will generate a new ScriptableObject file representing your state machine. Give it a descriptive name (e.g., Enemy State Machine).



5. **Double click** on the generated state machine asset to open the State Graph Editor.



You will notice that there are **two default states** in a newly created state machine:

- **Entry State**
  This is the starting point of the state machine.
  Please connect it to the first state that should run when the state machine is activated (see how to connect states in the following sections).

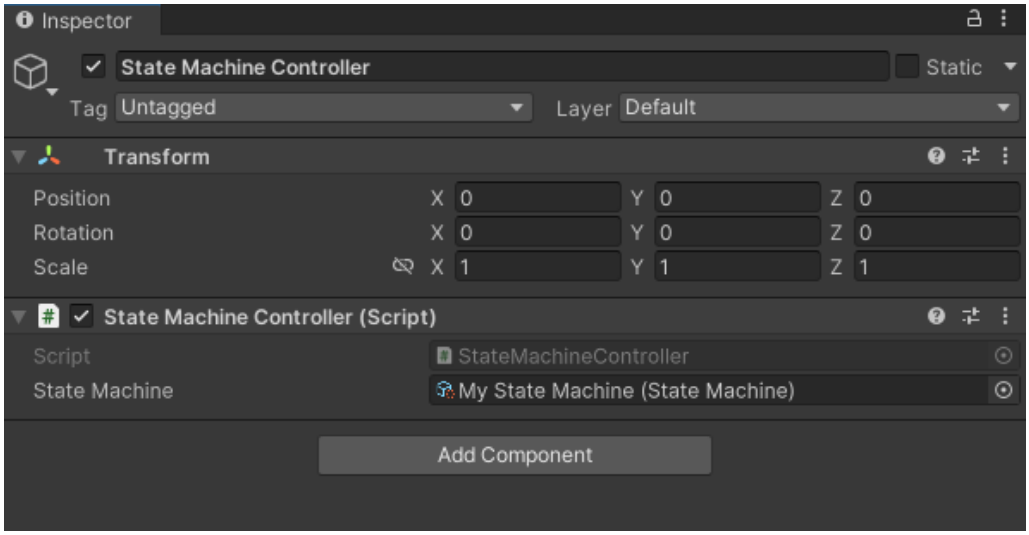
- **Any State**
  This special state can transition to *any other state* at any time.
  It evaluates its transitions **every frame**, regardless of the current active state.
  Use it for global transitions like interruptions, emergency exits, or reactions to high-priority events.

Assigning the State Machine to a GameObject

1. Select the GameObject where you want the state machine to run.

2. In the Inspector, click **Add Component** and choose `StateMachineController`.

3. Drag your State Machine asset into the **State Machine** field of the component.

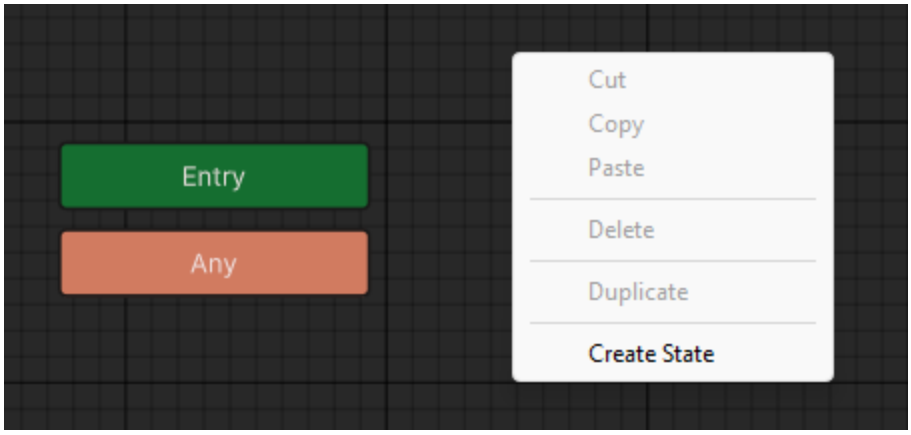


This links your state machine asset to the runtime system so it can start executing in Play Mode.

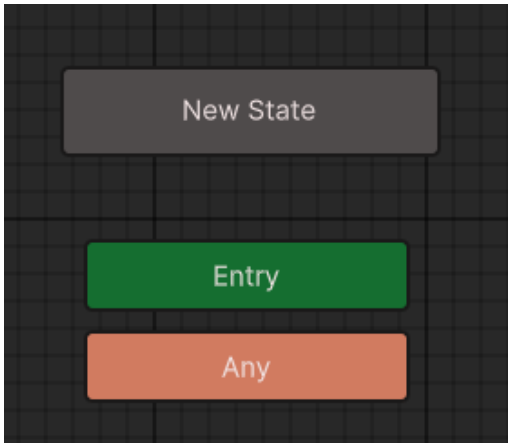
Section 3: Creating custom states

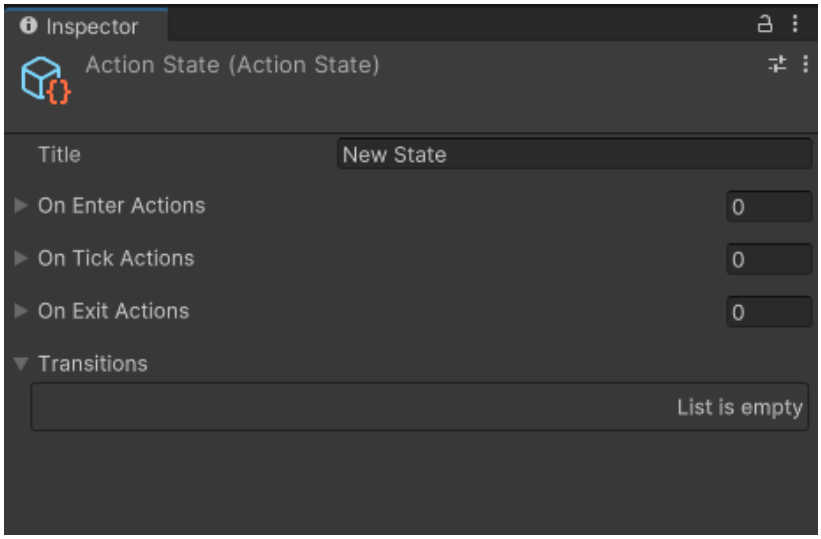
To create a custom state inside a state machine:

1. Right click inside the State Graph, this will open a pop-up menu, then click on `Create State`



You will see your newly created state and a custom inspector for it.





Understanding the state inspector

When you select a state in the State Graph, the inspector window will display all configurable elements for that state. Let's break down each component:

- **Title**

The **name of the state**, make it concise and descriptive.

- **On Enter Actions:**

These actions are **called once when the state becomes active.**

- **On Tick Actions:**

These actions are **called every frame** while the state is active.

- **On Exit Actions:**

These actions are **called once before leaving the state.**

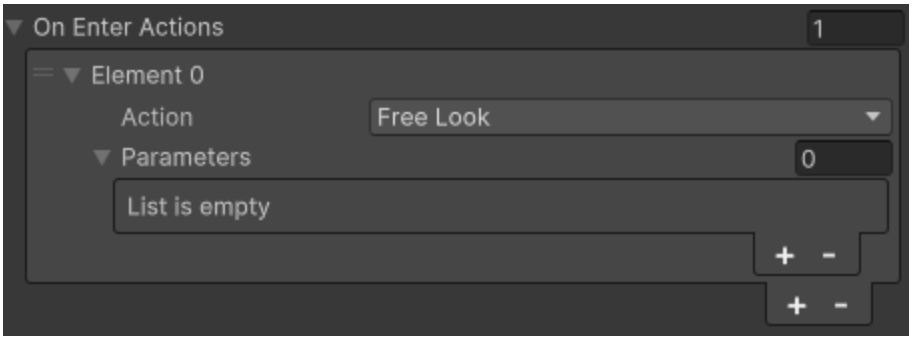
- **Transitions:**

A list of all the transitions from the state to other states with custom conditions.

Note: We will cover how to create actions, transitions and conditions in the following sections.

Section 3: Creating custom actions

Open an action dropdown and hit the plus button, you will notice that each action contains an enum dropdown and parameters.



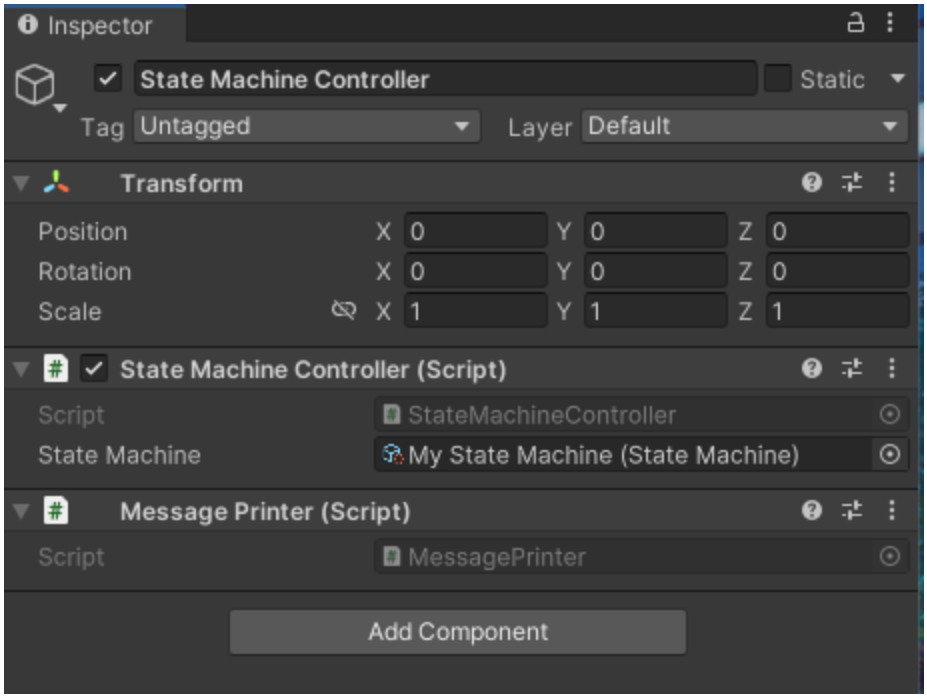
Note: the current actions in the Action dropdown are for demo purposes, we'll see how to edit them later.

Let's create a custom action, in this case for printing a message, follow this steps:

1. Create a **MonoBehaviour C# script**, this will be the provider for the actions. In this case we'll name it "MessagePrinter"

```
using UnityEngine;

0 references
public class MessagePrinter : MonoBehaviour
{

}

```

Important note: The scripts providing the actions need to be components of the same GameObject that holds the StateMachineController, in this case it will look like this:



2. Implement the interface called "**IAction**" included in the namespace `RainbowAssets.Utils`

```
using RainbowAssets.Utils;
using UnityEngine;

0 references
public class MessagePrinter : MonoBehaviour, IAction
{
    2 references
    void IAction.DoAction(EAction action, string[] parameters)
    {

    }
}

```

IAction Interface Structure

Each action class implements a method defined by the `IAction` interface.

This method includes two important inputs:

- **EAction action**

An **enum value** that acts as an identifier or keyword for the action.
It tells the system which specific behavior should be executed.This enum is also used by the editor to reference and select available actions.

- **string[] parameters**:

An **array of strings** containing additional information needed for the action.These parameters are provided through the editor and allow you to customize the action's behavior without writing extra code.

For example:

- Setting a movement speed
- Defining an animation name
- In this case, setting the message we want to print.

Note: When using non-string values like float, int, double, etc. the parameter needs to be parsed into the desired type

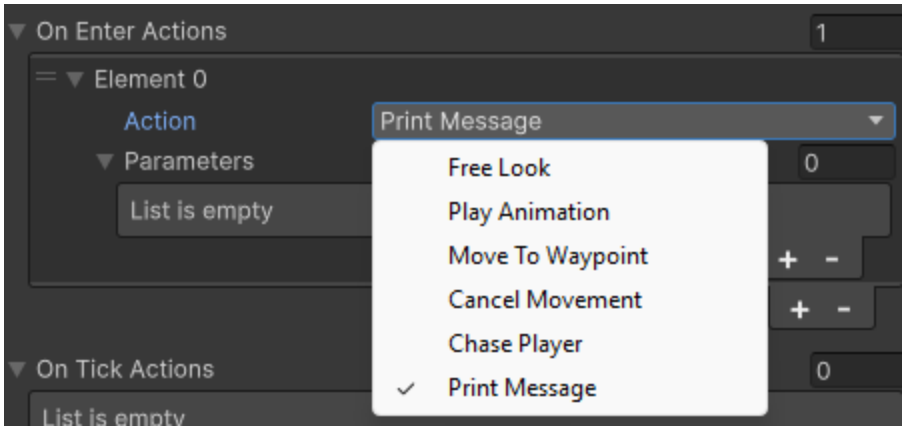
3. Access the EAction reference or go to Assets/Asset Packs/Rainbow Assets/Scripts/State Machine/Utils/EAction.cs

```
namespace RainbowAssets.Utils
{
    /// <summary>
    /// Enum that defines specific actions for gameplay or logic implementation.
    /// </summary>
    13 references
    public enum EAction
    {
        // Following actions are used for Demo purposes, remove them if desired
        1 reference
        FreeLook,
        1 reference
        PlayAnimation,
        1 reference
        MoveToWaypoint,
        1 reference
        CancelMovement,
        1 reference
        ChasePlayer
    }
}
```

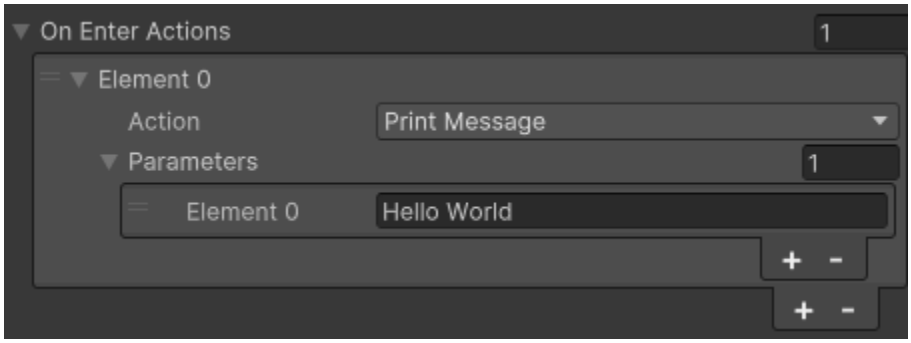
In this case I won't remove the demo actions but feel free to do so, now let's add a new EAction named "PrintMessage"

```
namespace RainbowAssets.Utils
{
    /// <summary>
    /// Enum that defines specific actions for gameplay or logic implementation.
    /// </summary>
    13 references
    public enum EAction
    {
        // Following actions are used for Demo purposes, remove them if desired
        1 reference
        FreeLook,
        1 reference
        PlayAnimation,
        1 reference
        MoveToWaypoint,
        1 reference
        CancelMovement,
        1 reference
        ChasePlayer,
        0 references
        PrintMessage
    }
}
```

Now compile your project, you shall see that our custom EAction is now available in the state editor.



Add the desired message to be printed in the parameters array, for example “Hello World”



Finally let’s go back to our MessagePrinter class, and compare if the EAction passed in the DoAction() method is the one we care about, in that case we’ll print the message passed through the parameters.

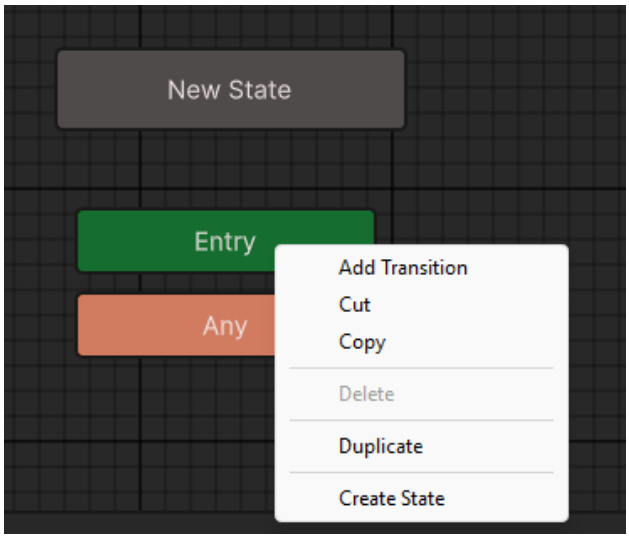
```
using RainbowAssets.Utils;
using UnityEngine;

0 references
public class MessagePrinter : MonoBehaviour, IAction
{
    2 references
    public void DoAction(EAction action, string[] parameters)
    {
        if(action == EAction.PrintMessage)
        {
            string message = parameters[0];

            print(message);
        }
    }
}

```

Section 4: Making transitions

To create a transition between two states:

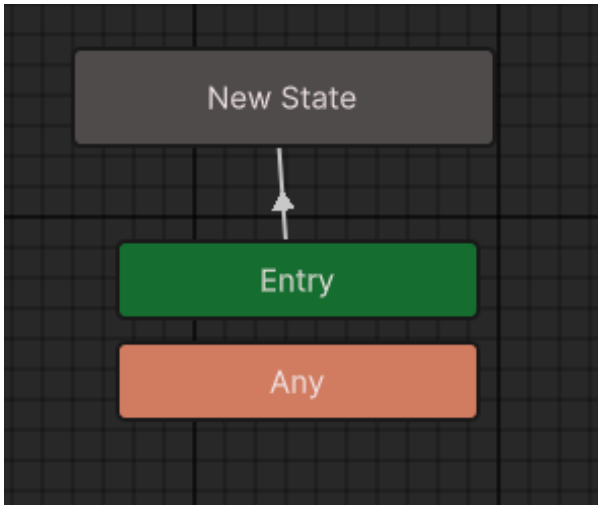
1. **Right-click** on the state you want to transition *from*.

2. Select **"Add Transition"** from the context menu.



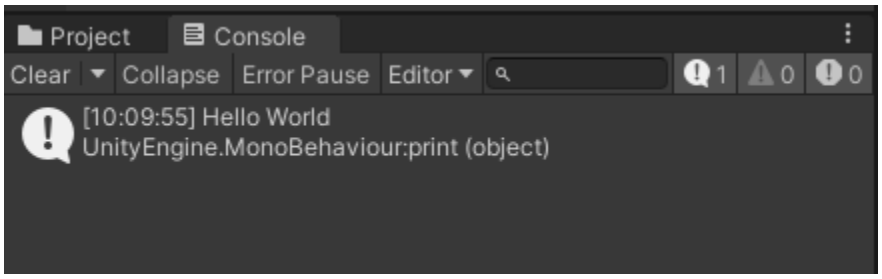
3. Then, **click on the target state** you want to transition *to*.

Let's add a transition from the ***Entry State*** to our custom state so that we can start seeing our state machine in action.



You should now see an **arrow** connecting both states, indicating a transition path.

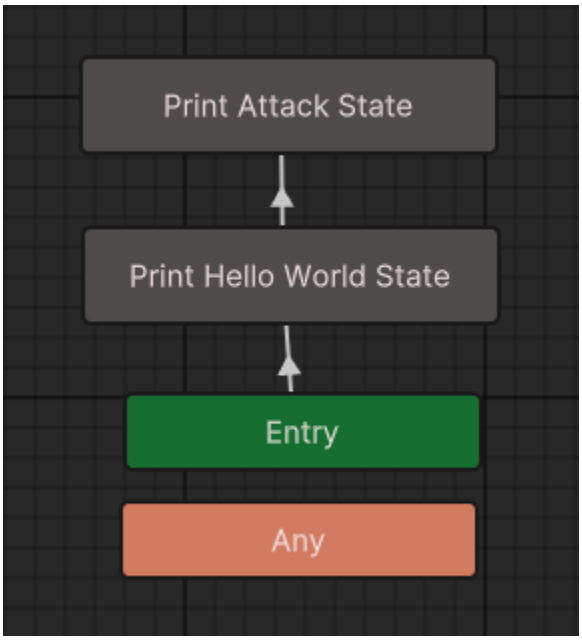
Now, if you enter Play Mode, the state will print "Hello World". This happens because we added the Print Message action to the **On Enter** section of the state. That means the message is triggered **once**, when the state is first entered.



Feel free to add the **Print Message** action to the **On Tick** or **On Exit** sections as well, and provide different messages to observe when each one is triggered.

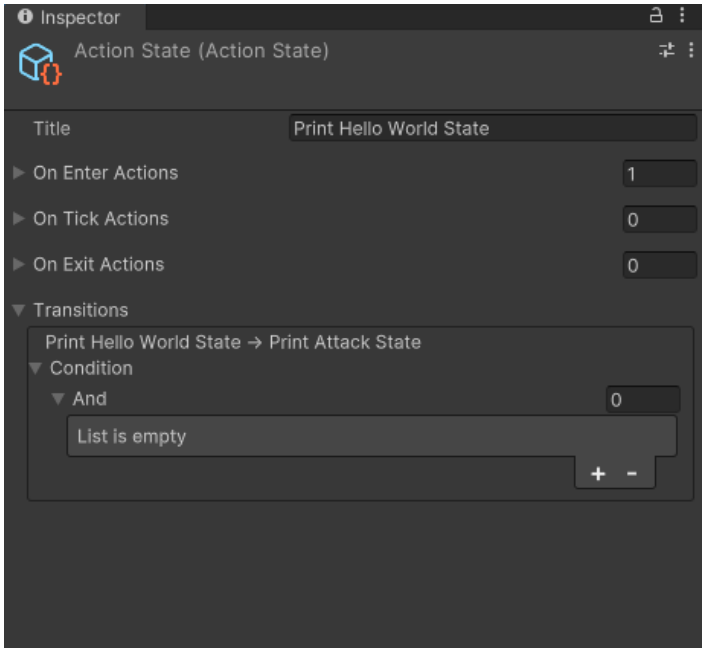
Section 5: Transition conditions

In order to transition from one custom state to another a condition is required, so first let's create another state and create a transition to it.



In this newly created state the message "Attacking!" gets printed in the On Tick section, you can add any message and change the title as you want though.

You will notice in the ***Print Hello World State*** that a new transition was added, this transition includes the names of its root and true state, it also contains a condition.



Understanding conditions

The conditions use something known as CNF(**Conjunctive Normal Form)**, a logical structure that allows for expressive and flexible evaluations.

In CNF, a condition is composed of:

- One or more **AND** groups (**Conjunctions).**
- Each group contains one or more **OR** (**Disjunctions**).

So we can define a CNF as a **Conjunction of Disjunctions of Predicates.**

Why use Conjunctive Normal Form?

The Conjunctive Normal Form allows us to build **any logical condition**, no matter how complex, by combining simple, modular predicates.

Instead of writing hard coded logic like:

```
if (isGrounded && (isRunning || isJumping)) { ... }
```

You can construct that same logic visually in the editor using:

- One **AND** group:
  ➤ `isGrounded`
- One **OR** group:
  ➤ `isRunning OR isJumping`

This structure gives you:

- **Flexibility**: Any logic can be expressed using this format.
- **Modularity**: Reuse and combine basic predicates easily.
- **Scalability**: Add complex conditions without changing core logic.

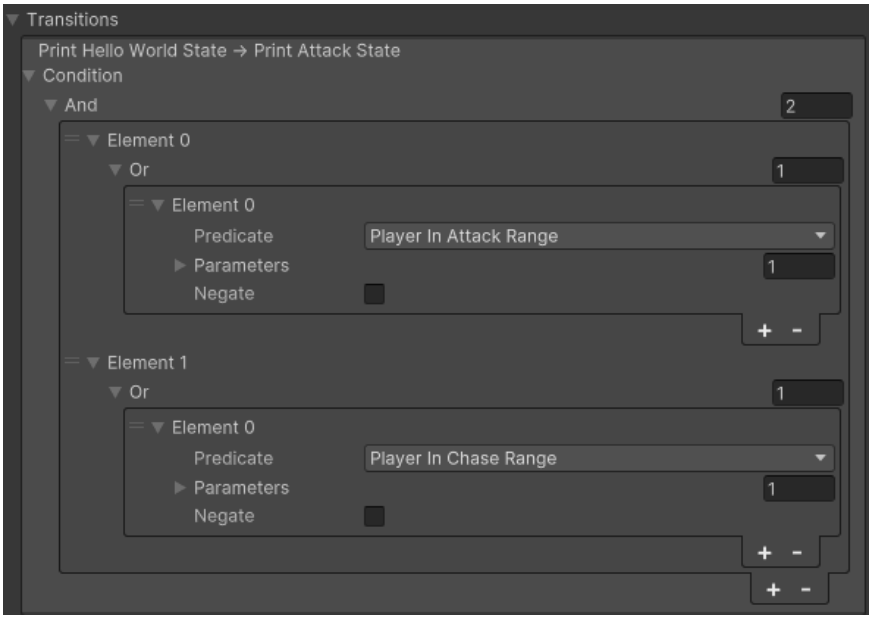
In short, **CNF makes our transitions powerful and editor-friendly**.

In the condition editor the **ANDs** and **ORs** are represented through nested arrays, where the outer array represents the ANDs (Conjunctions) and the inner arrays represent the ORs (Disjunctions).

For example, a simple CNF condition like:

```
playerInAttackRange && playerInChaseRange
```

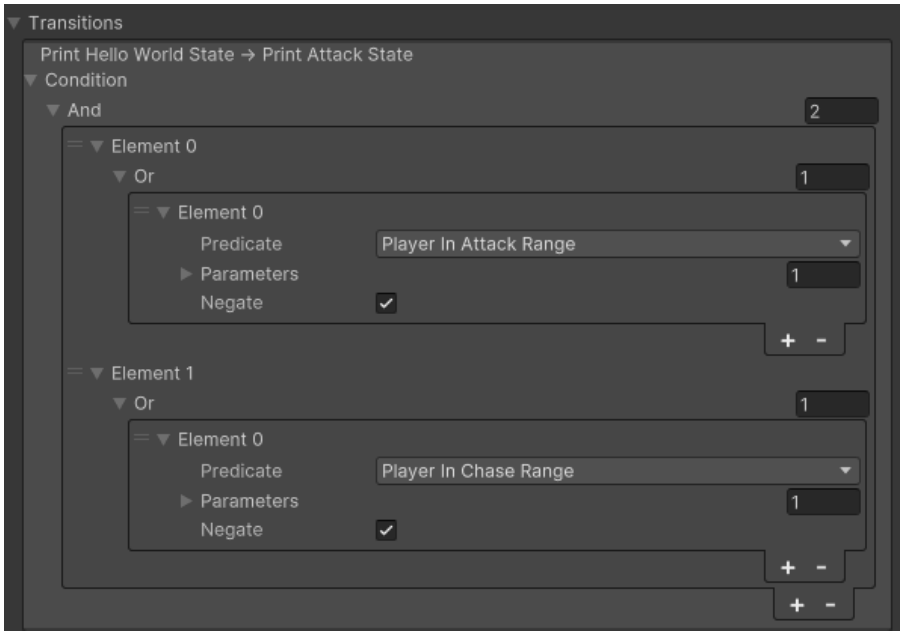
Will look like this in the editor:



Notice **we can negate the predicates** as well, so if we wanted to change the condition to:

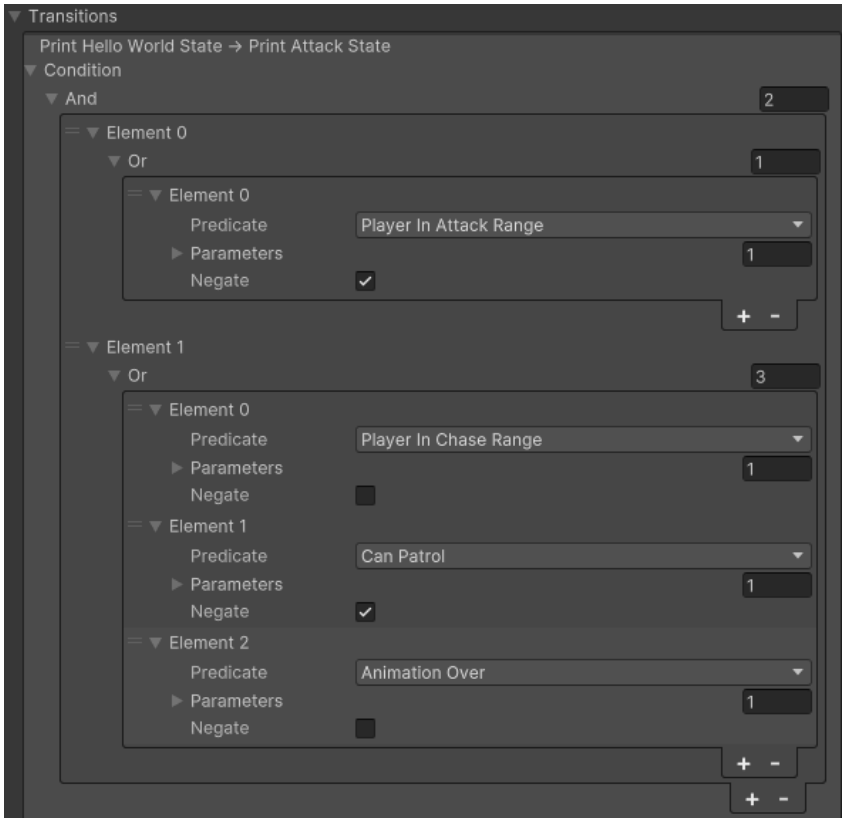
```
!playerInAttackRange && !playerInChaseRange
```

Will look like:



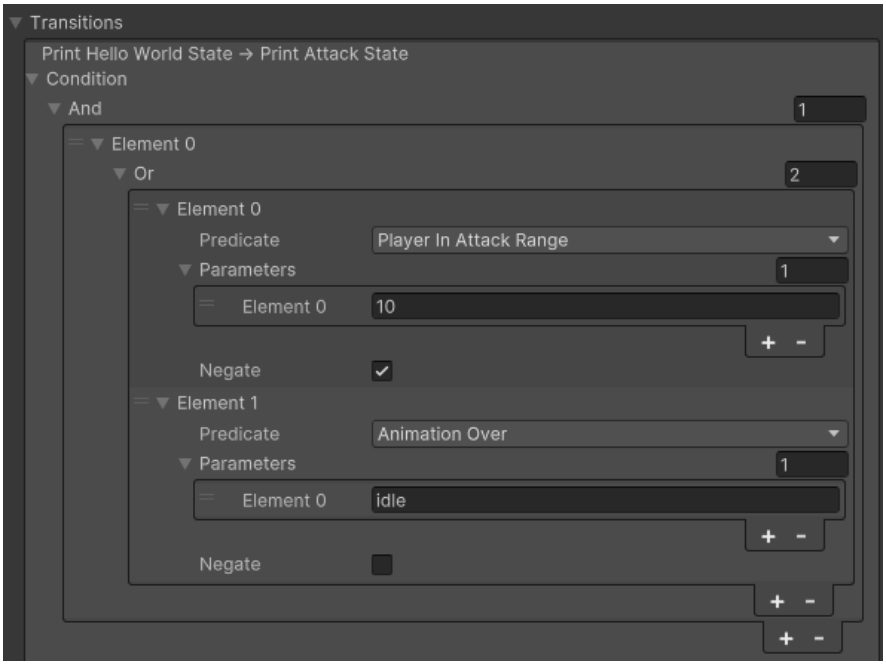
A more complex condition like:

```
!playerInAttackRange && (playerInChaseRange ||
!canPatrol || animationOver)
```



If we wanted to add parameters to the predicates like:

```
!playerInAttackRange(10) || animationOver(“idle”)
```



Let’s see how to handle predicates to create our custom conditions:

1. In the MessagePrinter class Implement the interface called “**IPredicateEvaluator**” included in the namespace `RainbowAssets.Utils`

```
using RainbowAssets.Utils;
using UnityEngine;

0 references
public class MessagePrinter : MonoBehaviour, IAction, IPredicateEvaluator
{
    2 references
    bool? IPredicateEvaluator.Evaluate(EPredicate predicate, string[] parameters)
    {

    }

```

Note: It doesn’t matter where you implement this interface as long as it is a component attached to the same GameObject as your StateMachineController, same goes for the IAction interface.

IPredicateEvaluator Interface Structure

Each predicate check is handled by a class that implements the `IPredicateEvaluator` interface.
This interface defines a method that receives:

- **EPredicate predicate**
  An **enum value** that identifies the condition being evaluated.
  It tells the system **what logic to check**, such as "IsGrounded", "HasTarget", or any other condition you define.

- **string[] parameters**
  An **array of strings** used to **provide extra data** required for the predicate.
  These parameters come from the editor and allow the condition to be dynamic without changing code.

For example:

- Defining a health value threshold.
- Defining an animation name to see if the animation is over.
- In this case, checking if an input was pressed.

Note: When using non-string values like float, int, double, etc. the parameter needs to be parsed into the desired type

2. Access the EPredicate reference or go to Assets/Asset Packs/Rainbow Assets/Scripts/State Machine/Utils/EPredicate.cs

```
namespace RainbowAssets.Utils
{
    /// <summary>
    /// Enum that define specific predicates to represent different condition checks.
    /// </summary>
    17 references
    public enum EPredicate
    {
        // Following predicates are used for Demo purposes, remove them if desired
        1 reference
        KeyCodePressed,
        1 reference
        AnimationOver,
        1 reference
        AtWaypoint,
        1 reference
        CanPatrol,
        1 reference
        DamageTakenEvent,
        1 reference
        DieEvent,
        1 reference
        PlayerInChaseRange,
        1 reference
        PlayerInAttackRange,
        1 reference
        SuspicionFinished
    }
}
```

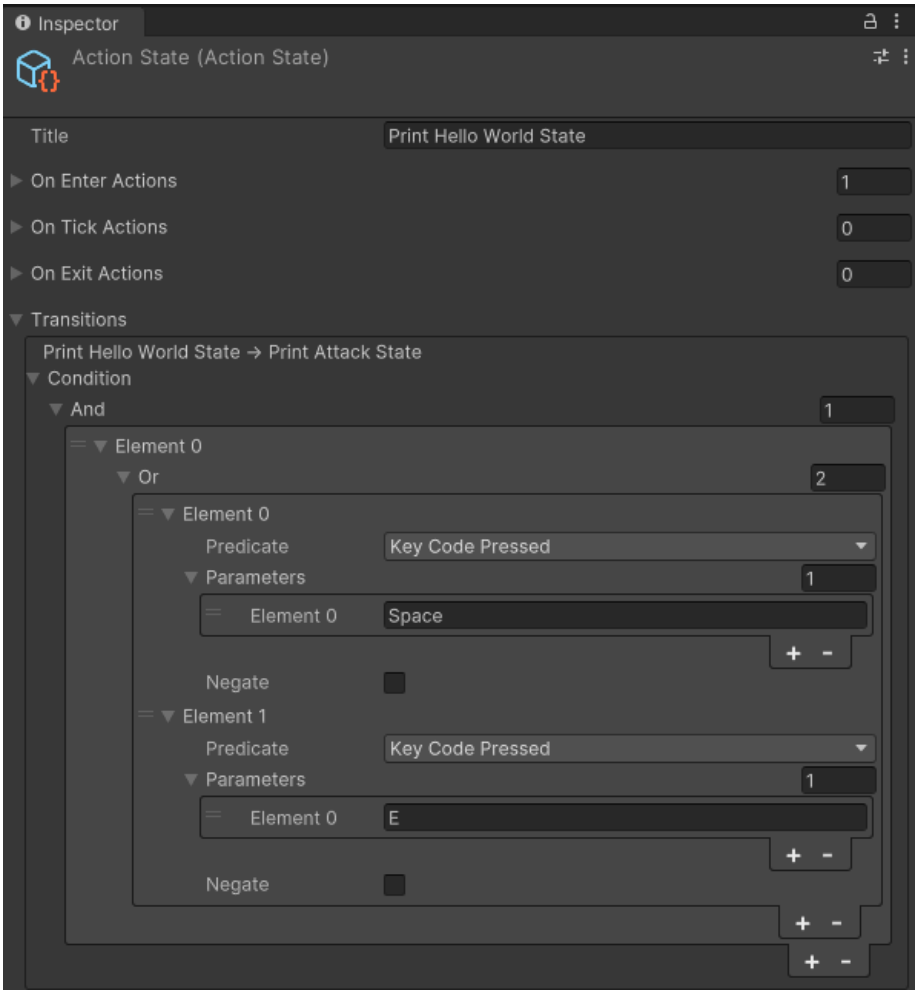
We'll use the one called "KeyCodePressed" already created for the demo, but feel free to create your own or to remove all of the demo ones.

3. Go back to the MessagePrinter class, check if the predicate passed through the Evaluate() method is the one we care about, in that case we'll parse the parameter into a KeyCode enum and check if its associated key was pressed.

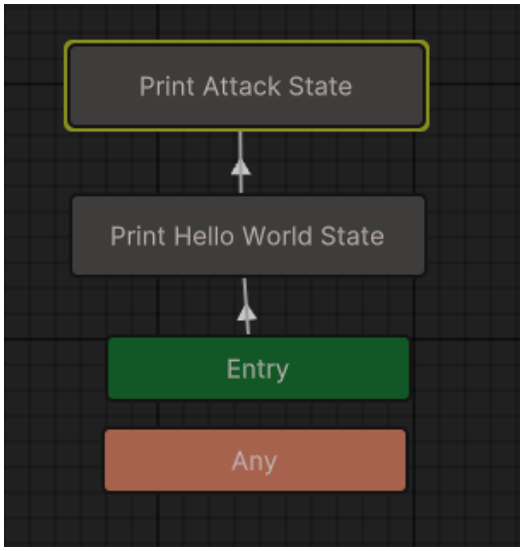
```
using System;
using RainbowAssets.Utils;
using UnityEngine;

0 references
public class MessagePrinter : MonoBehaviour, IAction, IPredicateEvaluator
{
    2 references
    bool? IPredicateEvaluator.Evaluate(EPredicate predicate, string[] parameters)
    {
        if(predicate == EPredicate.KeyCodePressed)
        {
            string enumString = parameters[0];

            if(Enum.TryParse(enumString, out KeyCode keyCode))
            {
                return Input.GetKeyDown(keyCode);
            }

            return false;
        }

        return null;
    }

```

Important note: if the desired predicate is not met in the evaluation, **we recommend that the Evaluate() method returns null**, this way the method is kept neutral.

4. Back in the state editor, set the transition from the ***Print Hello World State*** to the ***Print Attack State*** to happen when we press "Space" or when we press "E"



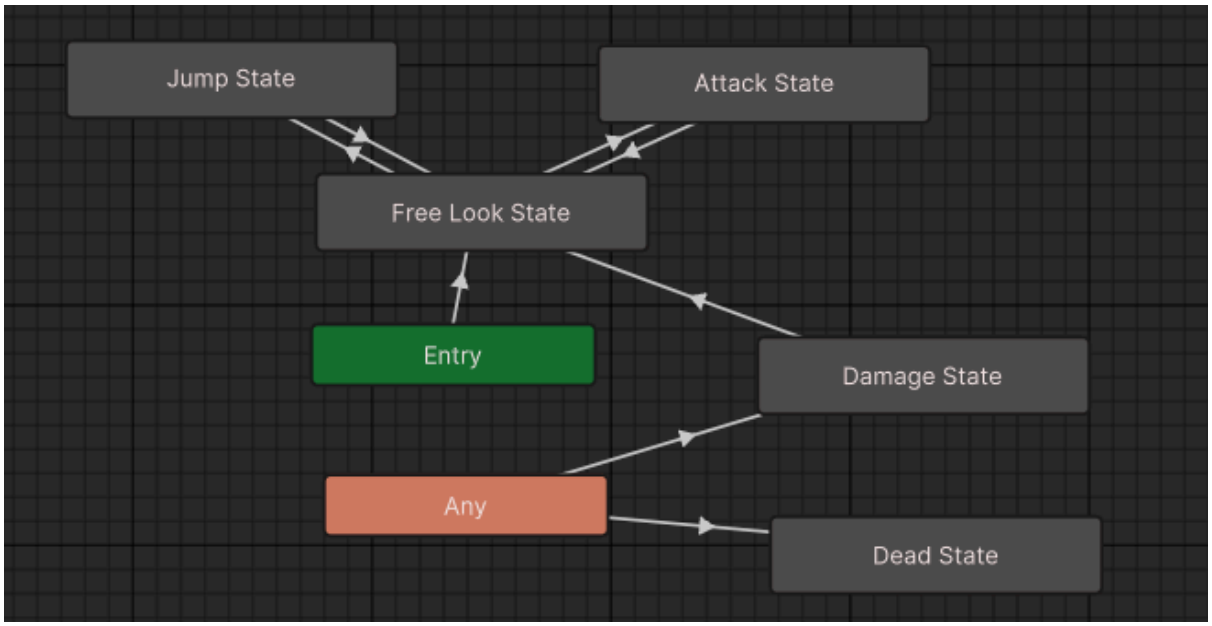
5. Enter play mode and press either “Space” or “E” to see the transition happen:



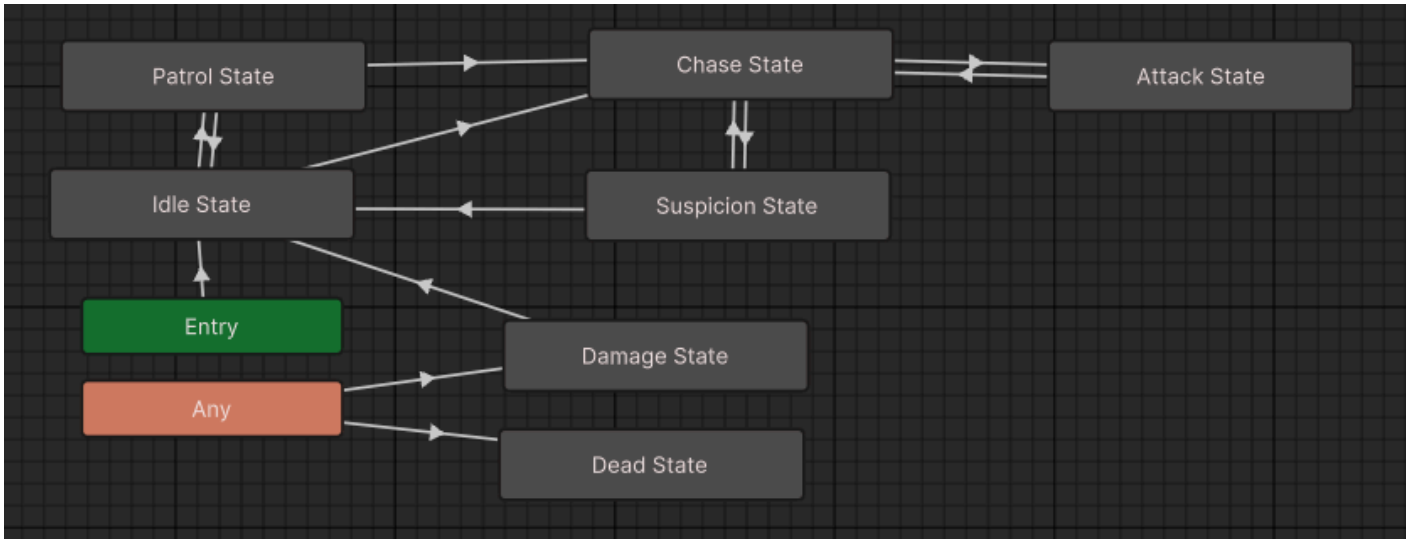
Section 6: Demo Assets

If you want to see more complex State Graphs in action, please check the demo section of the tool, we provide the following:

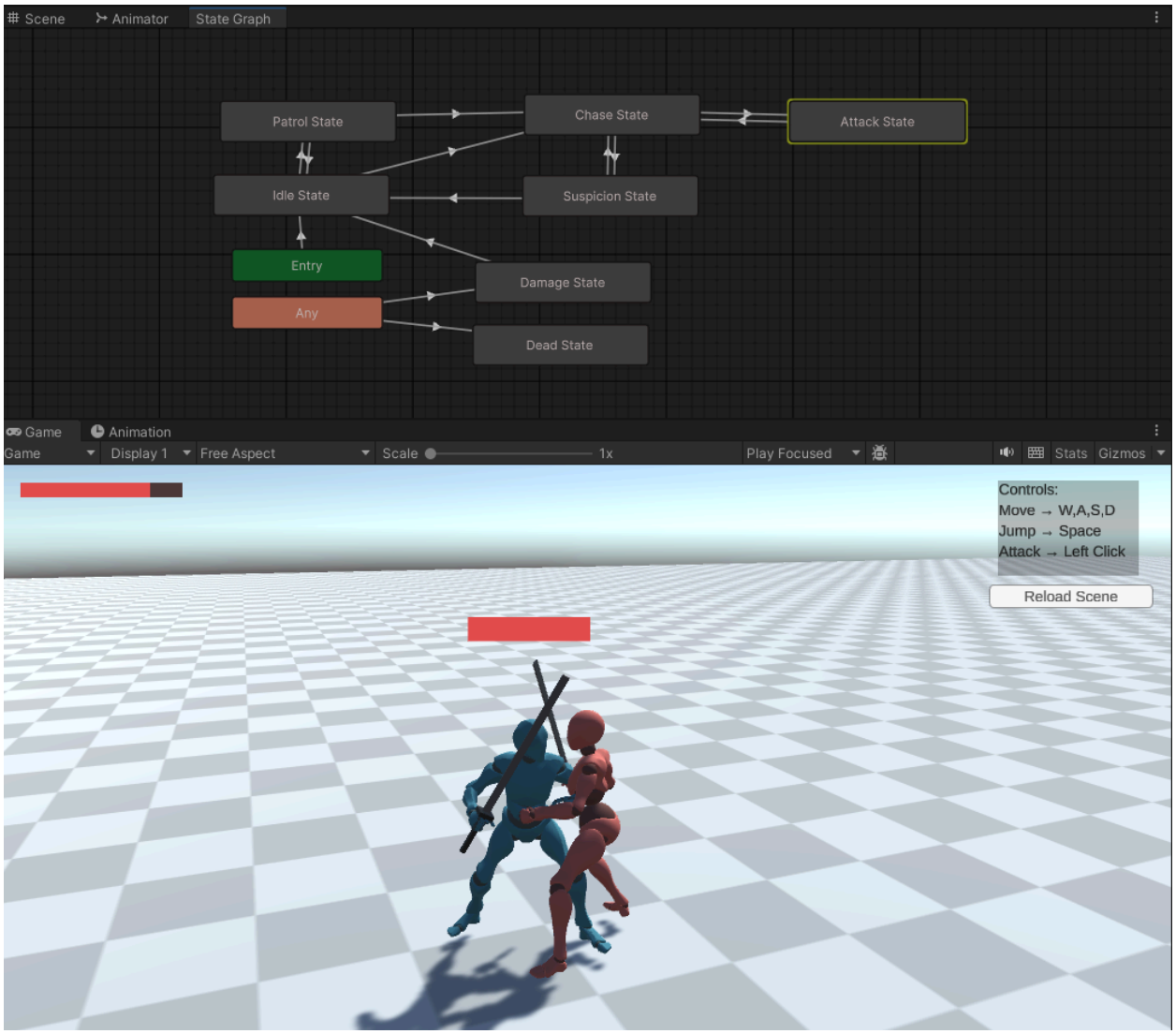
1. Complex player state machine.



2. Complex AI state machine with combat, patrolling, suspicion, etc.



3. Real time combat system, feel free to use this combat system in your projects



Final notes

You now have a complete overview of how to use the State Graph tool, from creating new states and transitions, to implementing actions and conditions with full flexibility.

If something doesn’t work, always make sure:

- Your interfaces are implemented correctly.
- The parameters are parsed as expected.

Developer contact:

octaviortegan@gmail.com

www.linkedin.com/in/octavio-ortega-novoa