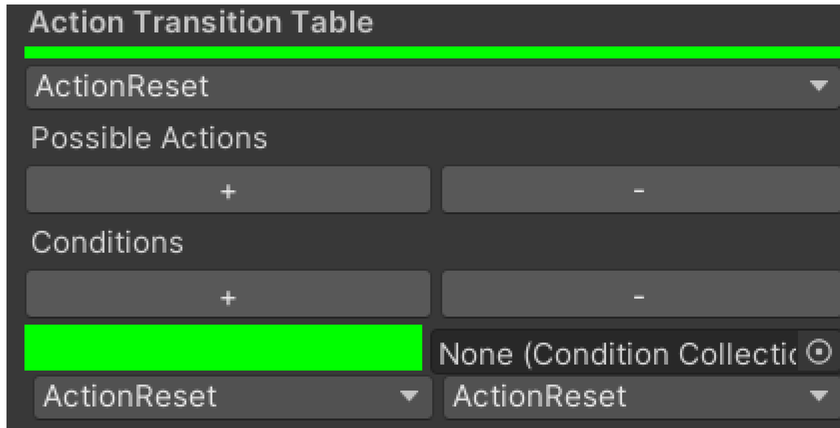
Documentation---!

<Inspector>

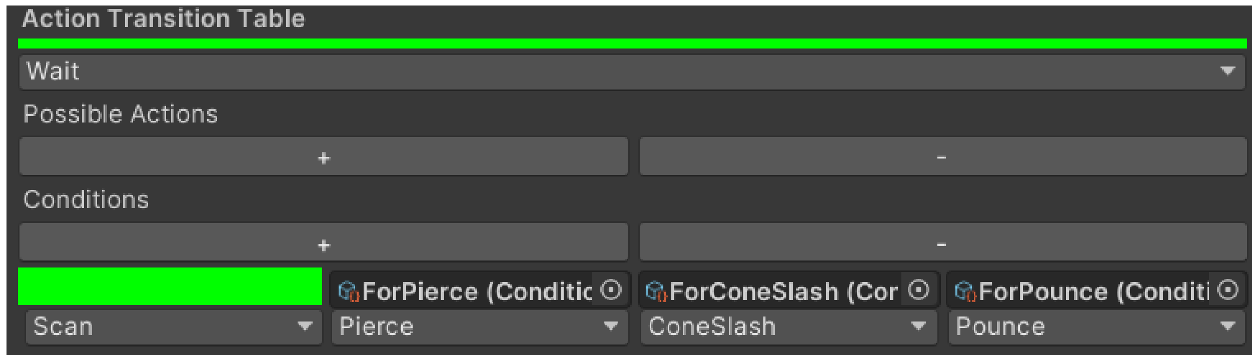


<- starting state

<- add or remove Action row

<- add or remove Condition column

This Character system utilizes a table layout for state transitions. The row labels contain states deriving the Action class and the column labels contain the conditions that need to be checked. The table lookup of state-row and conditions-column determines what state to transition to. For example:



In this character configuration, upon ending the Scan state, the character will check the conditions, left to right.

The first collection of conditions (an instance of ConditionCollection class) being checked is ForPierce. If this collection does not return false, the collection is deemed true and the corresponding state is chosen for the next state—Pierce in this case. If the first collection does not pass, the next one is checked, then so on.

For adding new states to the system, see **<Writing New States>**. For adding new conditions to the system, see **<Writing New Conditions>**.

<Writing New States>

While commonly referred to as states, this Character System refers to snippets of behavior code as FunctionClip. FunctionClip has three derivations, but only one of them is relevant for the Action Transition Table: Action.

The Action class has public virtual methods of Start, Update, FixedUpdate, and End. Writing a new Action is as simple as deriving the Action class and overriding these core functionalities. Among these, End is special as it also triggers the condition-checking stage of the character's state machine.

An example of a new Action, Pierce, is shown below:

```
Unity Script | 0 references
public class Pierce : Action
{
    List<CharacterHurtbox> alreadyHit = new();

    9 references
    public override void Start()
    {
        base.Start();
        character.StartCoroutine(PierceSequence());
    }

    1 reference
    public IEnumerator PierceSequence()...

    1 reference
    private IEnumerator VFX()...
}
```

In this case, End is called inside PierceSequence, which is a coroutine handling the sequence of events that represent a "piercing" attack (including moving the character and dealing damange). Also visible is that Action has a member character, which refers to the instance of the character class this FunctionClip belongs to. It is possible to access data about the Character instance this way.

When finished writing a class deriving Action, it will automatically populate the state selection dropdown of the inspector.

<Writing New Conditions>

Here is the script for InCone, a class deriving Condition:

```
[CreateAssetMenu(menuName = "Condition/InCone")]
Unity Script | 0 references
public class InCone : Condition
{
    public int minCharacters = 2;
    public float coneAngle = 60f;

    2 references
    public override bool Check()...
}
```

The only thing necessary for a condition class is to override the base Check method and return a bool somehow. This class returns whether there are other characters within a 60-degree cone with the radius of the character's scan area. The necessary information is accessed by the character member which references the Character instance this condition belongs to.

Multiple conditions can be combined into a ConditionCollection, which simply holds a List<Condition> to check. If any of these conditions return false, the entire collection is deemed false. Otherwise, it is deemed true and the corresponding state will be chosen for the next transition.

ConditionCollection and Condition inherit ScriptableObject, so you must add the CreateAssetMenu header above the class declaration. Then, the asset must be created in the Project Folder. These can be dragged into the columns of each character's action transition table. To understand how the table functions, see **<Inspector>**.