Dynamic Behaviour v1.0

By logicandchaos

Dynamic Behaviour

This is the Dynamic Behaviour asset for unity, it is for dynamic behaviours for your players and NPCs. It is great for designing Ai behaviours quickly and simply. It uses scriptable objects in a fuzzy pattern matching state machine. This means the acts your characters use are a best fit selection. The Acts are condition based like the bark system in left4dead. Conditions can be anything! They just need to return a bool with a Verify() method.

Conditions

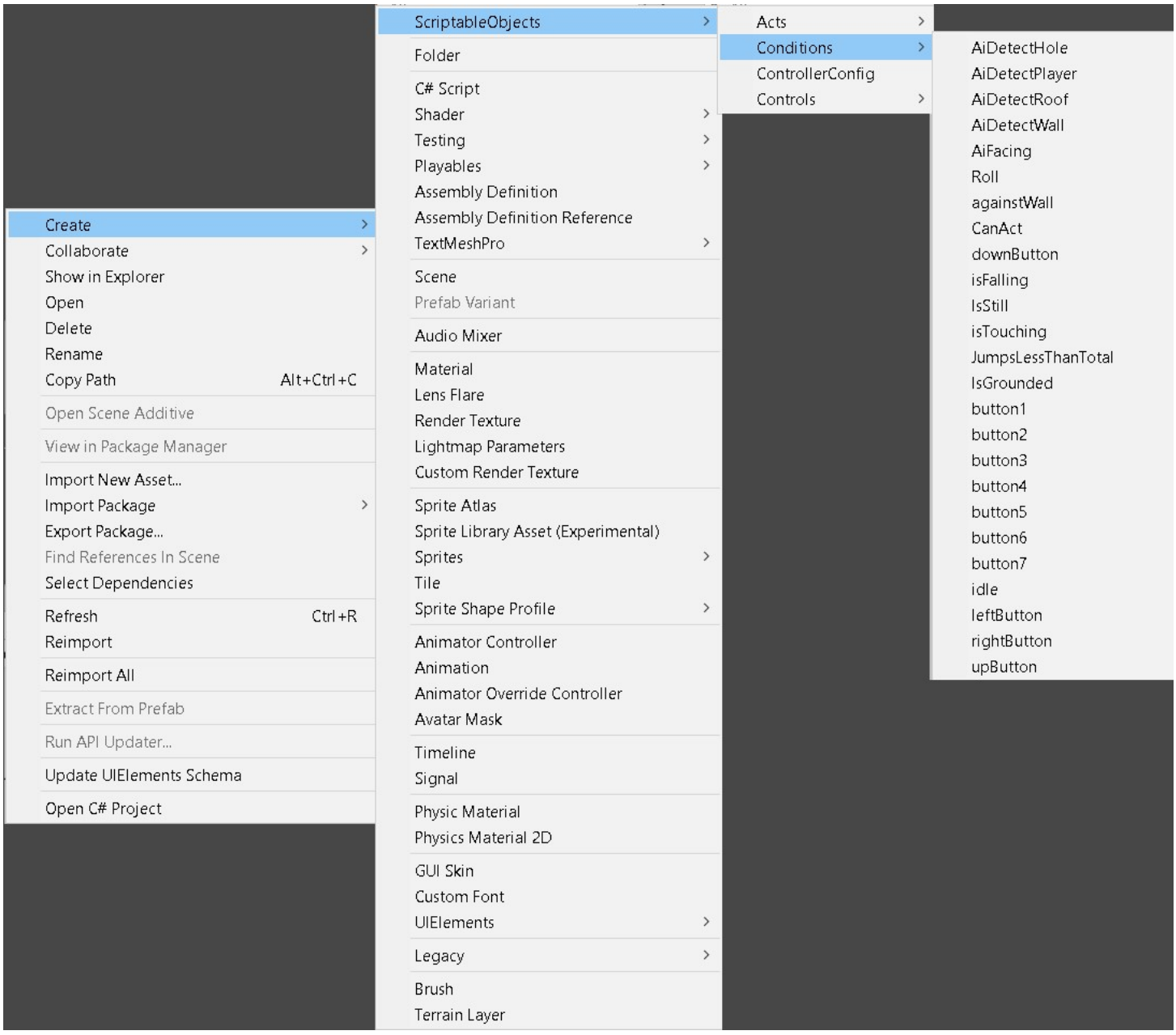
Conditions have a Verify method that takes in an actor and returns a bool, so they can be anything related to an Actor. There are a few conditions defined already to get you started and you can add in any condition you need. Condition is an abstract class inheriting from ScriptableObject this means that all conditions are scriptable objects. Conditions have two bool variables isTrue and inverted, and one method bool Verify(Actor p_actor).
You can make a condition for having low health or possessing a particular item, make one for the time of day or anything relevant to your game.

Create Condition Script

To make your own custom conditions you need to make a new script that inherits from Condition and the virtual Verify method must be overridden. To create a new condition script you make a new C# script and change it from inheriting from MonoBehaviour to Condition. This makes it a Condition and also a Scriptable Object. You must add a create asset menu header to allow you to create an scriptable object instance.
[CreateAssetMenu(fileName = "new Condition", menuName = "ScriptableObjects/Conditions/YourConditionName")]
where YourCondtionName is the name you want to use to select this condition from the create menu.
The Condition must override the the virtual method bool Verify(Actor p_actor) in this method is where you put your code for varifying the condition, setting the isTrue bool, and it must return isTrue.
To allow conditions to be inverted you must add in the lines:
if(inverted)
    isTrue=!isTrue;
right before you return isTrue. That will allow you to set the inverted bool in the inspector to invert your conditions so you do not need to make opposite conditions.
You can add any variables you may need.
There are many examples in the project and platformer example.

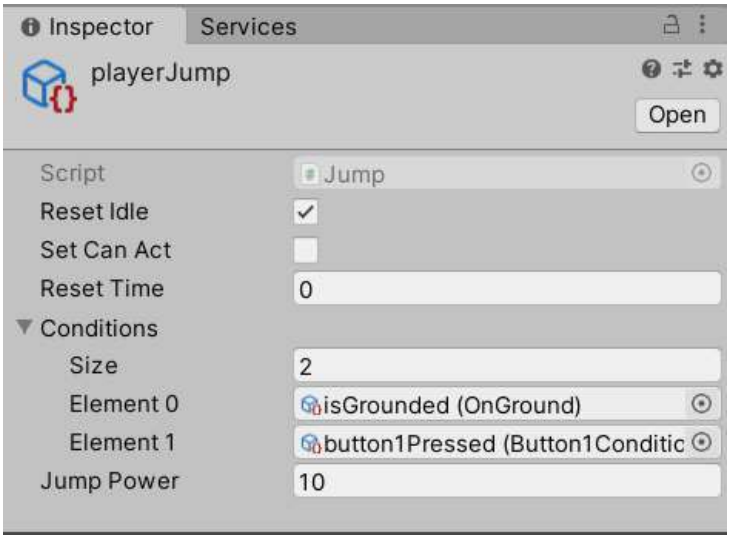
Create Condition ScriptableObject instance

To make a new condition from an existing script just navigate to the folder you want to keep your scriptable objects in, then right click right click in the project window and select Create, then ScriptableObject, then Condition, then pick the condition you want to create from the list,then you just need to name it. You can invert the condition by selecting inverted in the inspector.



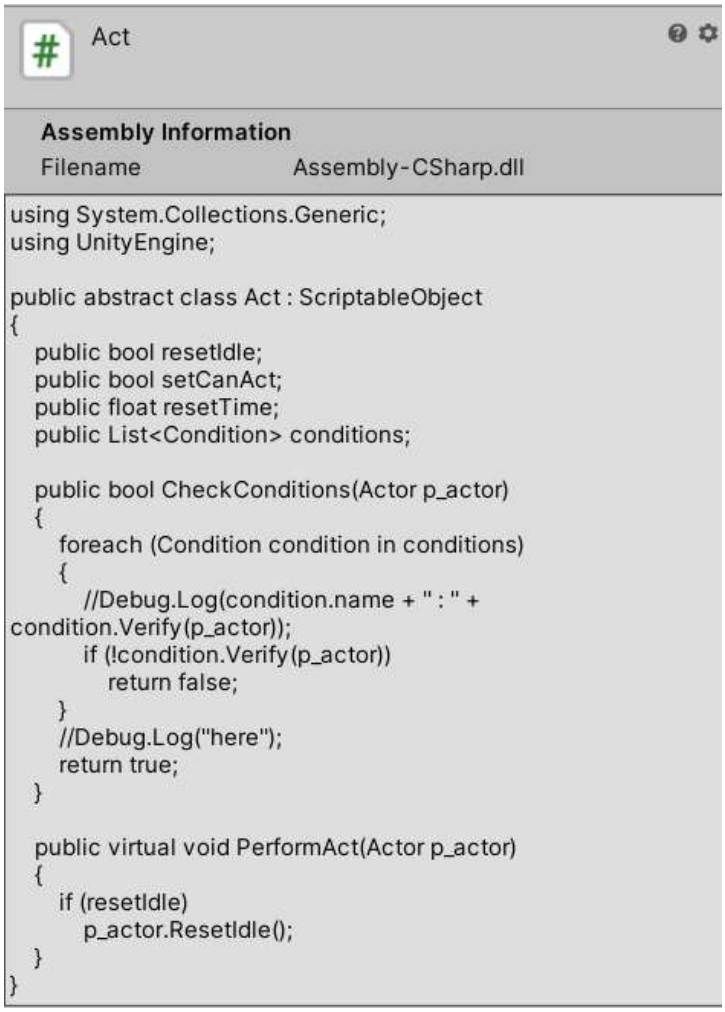
Drag Condition into Condition List

Acts all have condition lists attached to them. Go in the inspector and expand Conditions if needed you select how many conditions you want then drag and drop the conditions in the slots, or click the small target ot the right to assign.
In the image the Condition list is expanded and you can see the size and the elsements. That is where you assign the conditions.



Acts

Act is an abstract class that inherits from ScriptableObject and contains some variables, a CheckConditions(Actor p_actor) method and a virtual PerformAct(Actor p_actor) method.
Acts contain a list of conditions, the CheckConditions method checks to see is all the conditions are met, it does this by calling all the verify methods for the conditions in the conditions list.
The PerformAct method is virtual and must be overridden, this is where you code the act, what you want your actor to do.
There are many examples in the project and platformer example.

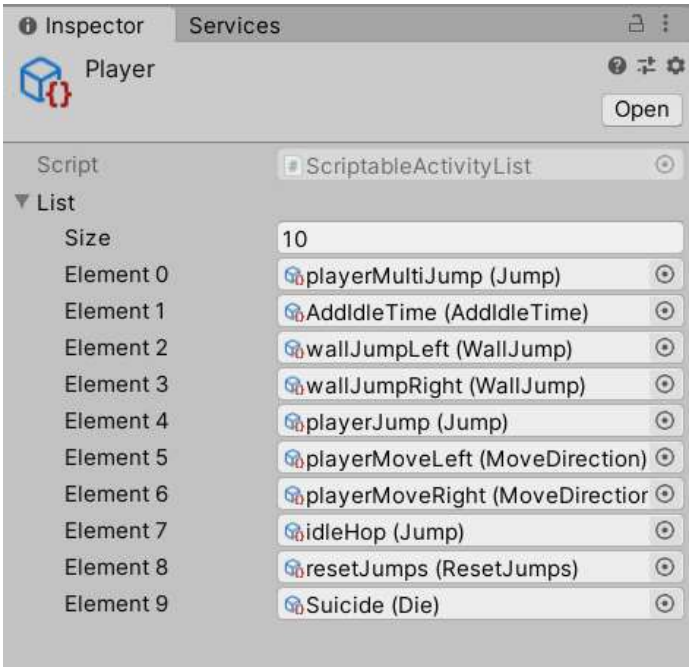


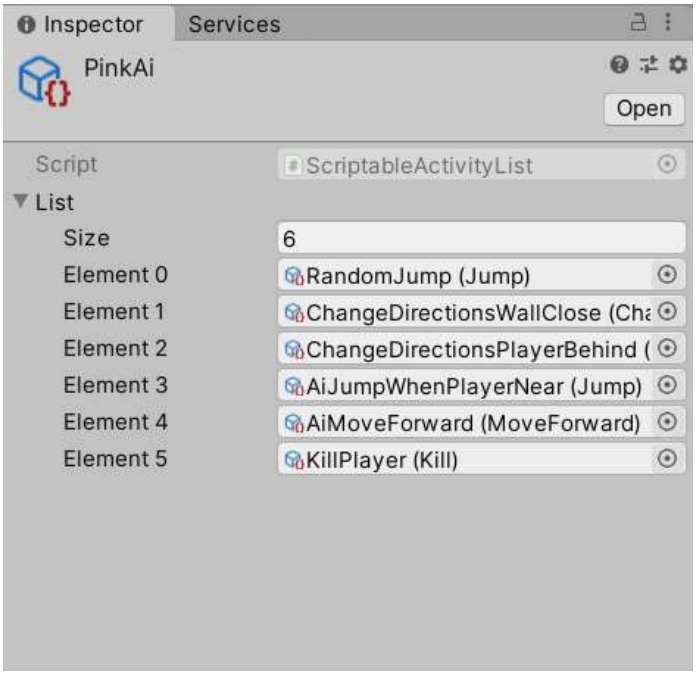
Create Act Script

To make your own custom acts you need to make a new script that inherits from Act and the virtual PerformAct method must be overridden. To create a new Act script you make a new C# script and change it from inheriting from MonoBehaviour to Act. This makes it an Act and also a Scriptable Object. You must add a create asset menu header to allow you to create an scriptable object instance.
[CreateAssetMenu(fileName = "new Act", menuName = "ScriptableObjects/Acts/YourActName")]
where YourActName is the name you want to use to select this act from the create menu. You can create any variables you may need.

Drag Act into ScriptableActivities List

Actors all have a scriptable activities list reference attached to them which contain all the Acts they can perform. The acts are on scriptable lists so you can share behaviours among Actors and adjust the lists in play mode. Create a new scriptable activities list from the create menu, input how many Acts you want then drag and drop the Acts in the slots, or click the small target to the right to assign.
In the images the activities list is expanded and you can see the size and the elsements. That is where you assign the conditions. There are 2 eamples of scripts inheriting from Actor, NPC and Player, they bother have an Activities list with Acts filled in.





Actors

Actors are a container class and also house the informal fuzzy pattern matching state machine, which will select the appropriate Act from the list using a best fit method. Actor contains references to all these variables:

```
ScriptableActivityList activities; //list of acts you can perform
private float idleTime; //used to control isIdle variable
private float idleDelay; //used to control isIdle variable
public bool isIdle; //used to tell if an actor is idle or not, great for idle animations
public bool isAlive; //keeps track of if actor is alive or dead
public Rigidbody2D rb; //standard Rigidbody2D, required component
public Collider2D actorCollider; //standard Collider2D, required component
private int collisionLayer; //set collision layer of actor in inspector
private int collisionMask; //creates mask from collisionLayer
public Vector2 newVelocity; //used to apply velocity in FixedUpdate()
public Vector2 maxAccelleration; //limits an actor's acceleration
public Vector2 collisionSize; //size for checking collisions
```

These variables are used for conditions, for instance isAlive keeps track of if the actor is alive.
To use the system first you must create a new script that inherits from Actor.
This will contain all the variables for your character that can be used for conditions.
There is an example of inheriting from Actor in the platformer folder. There is the PlatformerActor class which inherits from Actor and contains all the variables needed for a platformer game character.
There is also the Player and NPC classes which inherit from PlatformerActor, Player contains the reference to controllerConfig so that it can be controlled by input.

Platformer Demo

In the project there is a platformer folder that contains actors, acts and conditions for a face paced platformer game. The scene contains a small game using all these scripts and scriptable objects.
In my platformer example I create the class PlatformerActor which inherits from Actor, it contains the variables for multi-jumps and facing and the Player class inherits from PlatformerActor to have the references for the Controller Configuration. Within the Conditions for a player act there is a cast of the input parameter p_actor to a Player object use the as keyword to prevent errors:

```
Player player = p_actor as Player;
```

Then I perform a null check to make sure the cast was successful, if not I return false:

```
if (player == null)
    return false;
```

Otherwise, the cast was successful and you can access all the variables in the Player class.

Going through the conditions and acts for the platformer demo will help show you how to make your own conditions and acts.

The platformer example contains everything you need to create a great platformer game with just some great level design. I made my levels with the sprite shape tool, but there are many other options as well. Just make sure to put it on a collision layer and input the value where needed for ground collisions.