IMPORTANT

The Spell Casting Motion pack requires the following:

Motion Controller v2.49 or higher

Mixamo's free Pro Magic Pack (using Y Bot)

Importing and running without these assets will generate errors!

SPELL CASTING MOTION PACK

5/5/2017

Overview

This document goes into detail about how to build your own spells. We'll talk about the flow, components, and my thought process.

With the spell editor, you can use the components that I've created or build your own to add even more functionality. Unfortunately, I can't possibly think of and implement every possible spell. So, I've provided a foundation for you to use and expand on.

For information about using spells, please see the Spell Casting User Guide.

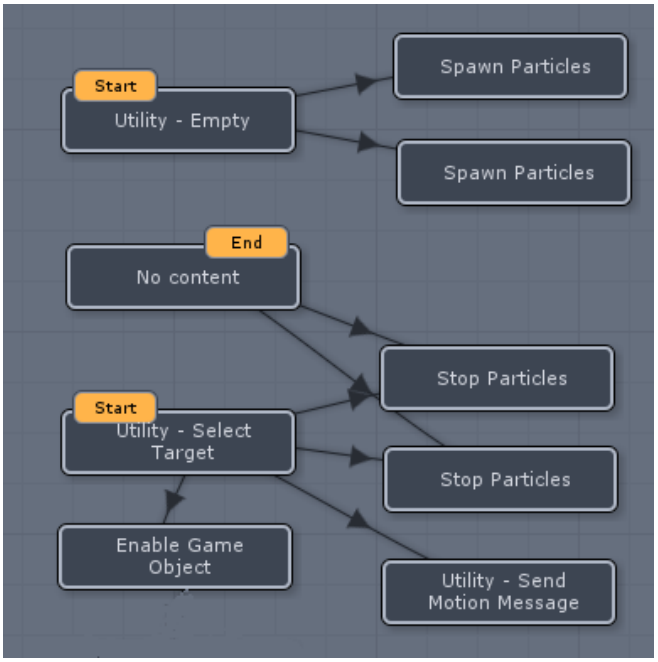
Stock Spells & Spell Actions

Your best resource for learning how to create spells and spell actions is to simply look at the stock spells and stock spell actions. Once you understand the flow, you should see that creating new items is fairly simple.

Spells

Spells are node-based graphs that create a flow of action.

As the life cycle of the spell goes on, each node performs a task; spawns particles, destroys objects, causes damage, etc. The links between the nodes define the path that a spell's life cycle will take.



Start Nodes

Unlike most graphs, the spell can have multiple starting points. This allows the spell to perform multiple actions at a time. Once there are no paths left to traverse, the spell begins to shut down.

End Nodes

Spells can also have multiple 'end' nodes. These end nodes are started when a spell has no paths left to travers and begins to shut down.

These end nodes are processed like regular nodes and give the spell an opportunity to clean up before it is destroyed.

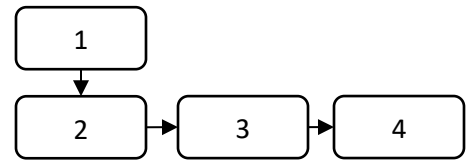
The end nodes are also called if the spell is cancelled.

Instances

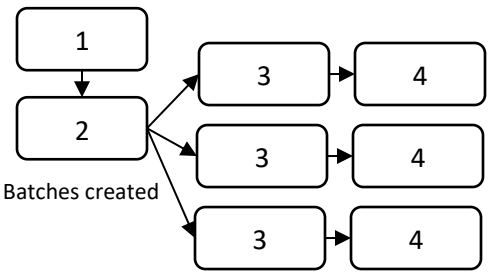
The spell assets themselves are templates. Each time a spell is cast, an instance of the graph is created. This way, we can cast a spell over and over as quickly as we want and the instances won't interfere with each other.

Think of each instance as running its own version of the graph.

More than that, as the nodes are processed new instances of the current graphs path will be created. Let's assume node #2 below gathers multiple targets and sends those targets out in batches. Each batch will spawn new instances of the path for that batch to process:



Batches created



Batches created

Components

Spells are made up of special parts that determine the flow and results of the spell. Understanding these components will help you control the flow.

Spell

Container that holds all the nodes and links. This represents the template that spell instances will be created from.

Node

Individual graph node in a spell that is a container. Its purpose is to hold a single action and help provide flow control within the spell.

Spell Action

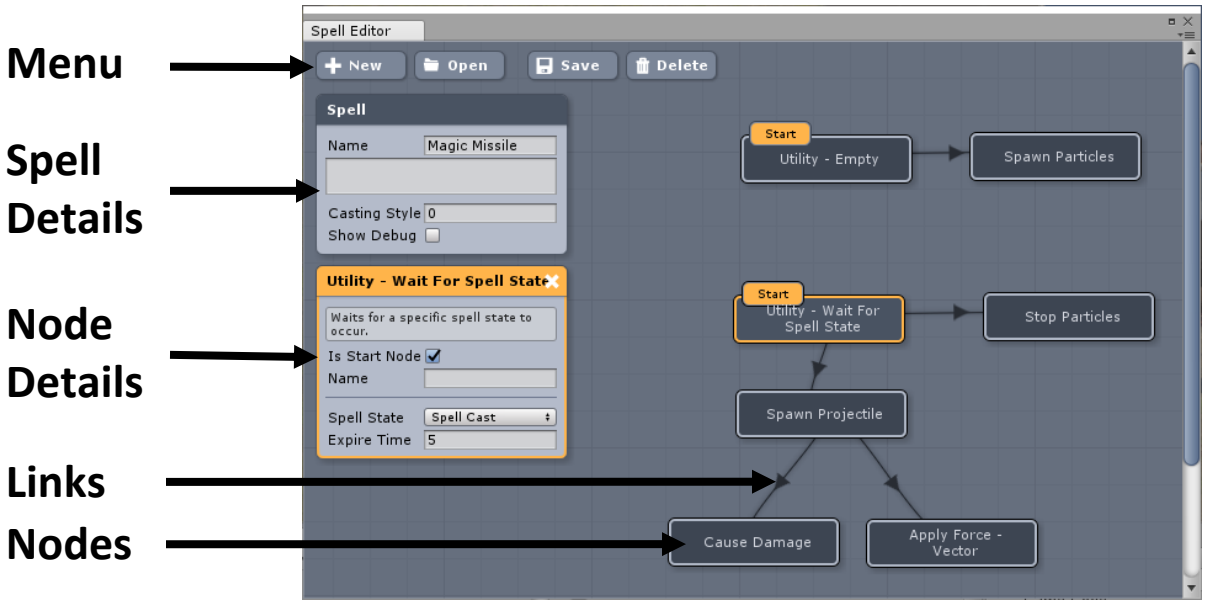
The spell action is what the node contains. During the life cycle, when a node is reached the action is activated. It's this action that will have any number of different implementations.

Link

A link is what ties two nodes together and creates the actual flow.

Spell Link Actions

Links can have different actions on them that help to control the flow. Think of them as gates that prevent a flow from continuing.

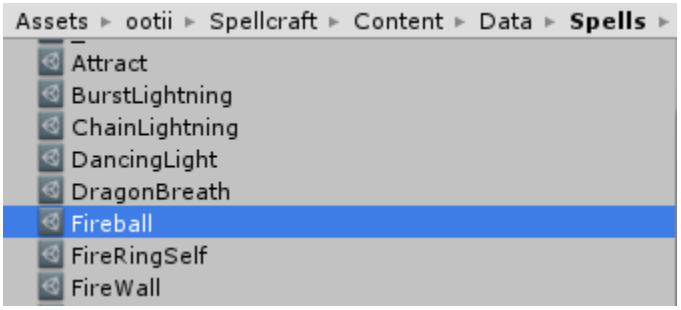


Spell Casting Motion Pack

5/5/2017

Assets

Each spell is an individual Unity asset. This means it's a file that Unity understands how to save, package, and load. All of the stock spells that are included with the Spell Casting Motion Pack can be found here:



As you build spells, you can place them where ever you want. I suggest using a separate folder so that you don't confuse your spells with mine.

You can copy and paste these assets to use existing spells as a starting point for new spells.

Spell Data

Each instance of a spell also creates a 'Spell Data' object. Think of this a temporary storage that moves with the flow of the spell. This way, one node may select targets and another node can use those targets.

The spell data structure is really a simple list of GameObjects (targets), Vector3s (positions), and some other properties. Once the spell has finished running, it is destroyed.

Spell Prefabs

If you look at the stock spells, they are built up on a couple of layers of prefabs; effects and components.

The reason I separate the two is reusability. With this approach, I can reuse the same glow in multiple spell effects and I can use the same spell effect in multiple spells.

Effect Prefabs

Assets\ootii\Spellcraft\Content\Effects\Prefabs

These represents the core functionality of the spell. This is where colliders, core logic, and other functional aspects of the spell are put together.

Effect prefabs tend to reference component prefabs

Component Prefabs

Assets\ootii\Spellcraft\Content\Effects\Components

These represent the visual and audio parts of the spell. Here's where I have particle effects, projectors, and other imagery. Very rarely is there code or logic here.

Spell Editor

The spell editor is where you will create, edit, and save spells. You can open the editor a couple of ways:

1. From the Unity menu: Window | ootii Tools | Spell Editor
2. Double clicking a spell asset
3. Pressing 'open' in the Spell Inventory inspector

If you're creating a new spell, the spell editor will be blank. Simply press the 'New' button at the top and select where the spell will be stored.

Enter the name and a description.

Casting Style

The casting style is defined by the PMP_BasicSpellCastings motion that is part of the Spell Casting Motion Pack. There are 12 styles that currently exist and each with an 'instant' and 'paused' version:


Instant: 0 Paused: 1


Instant: 2 Paused: 3


Instant: 4 Paused: 5


Instant: 6 Paused: 7


Instant: 8 Paused: 9


Instant: 10 Paused: 11


Instant: 12 Paused: 13


Instant: 14 Paused: 15


Instant: 16 Paused: 17


Instant: 18 Paused: 19


Instant: 20 Paused: 21

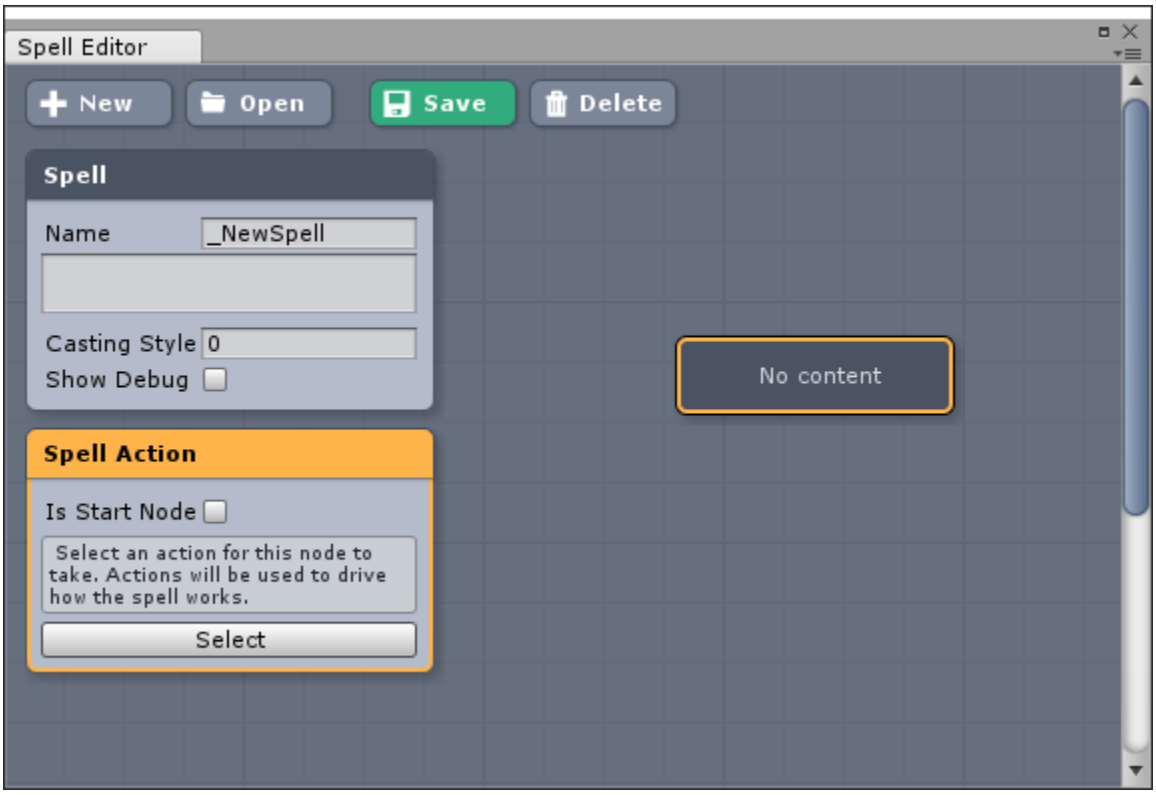

Instant: 22 Paused: 23

By 'paused', I mean that the character goes into a casting pose and waits for a spell action to continue. This is useful with something like the Teleport spell where a position needs to be chose.

The 'instant' styles don't have a pause and simply animate from start to stop in one flow.

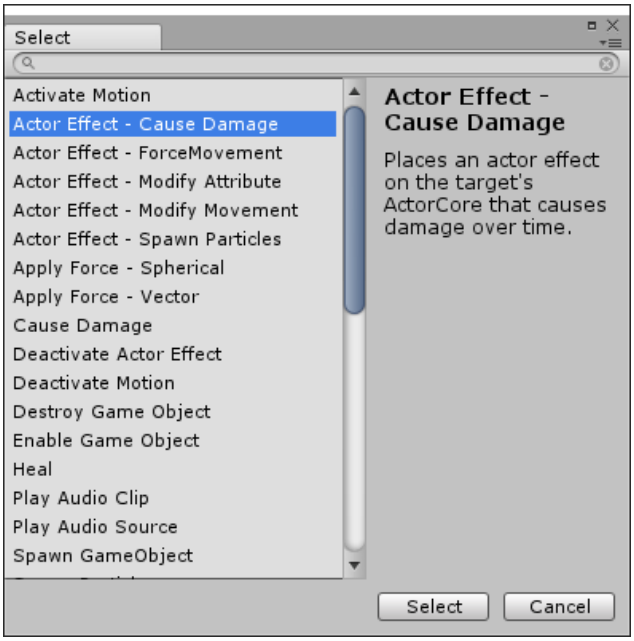
Adding nodes

With the spell editor open, just right-click in the editor to add a new node.

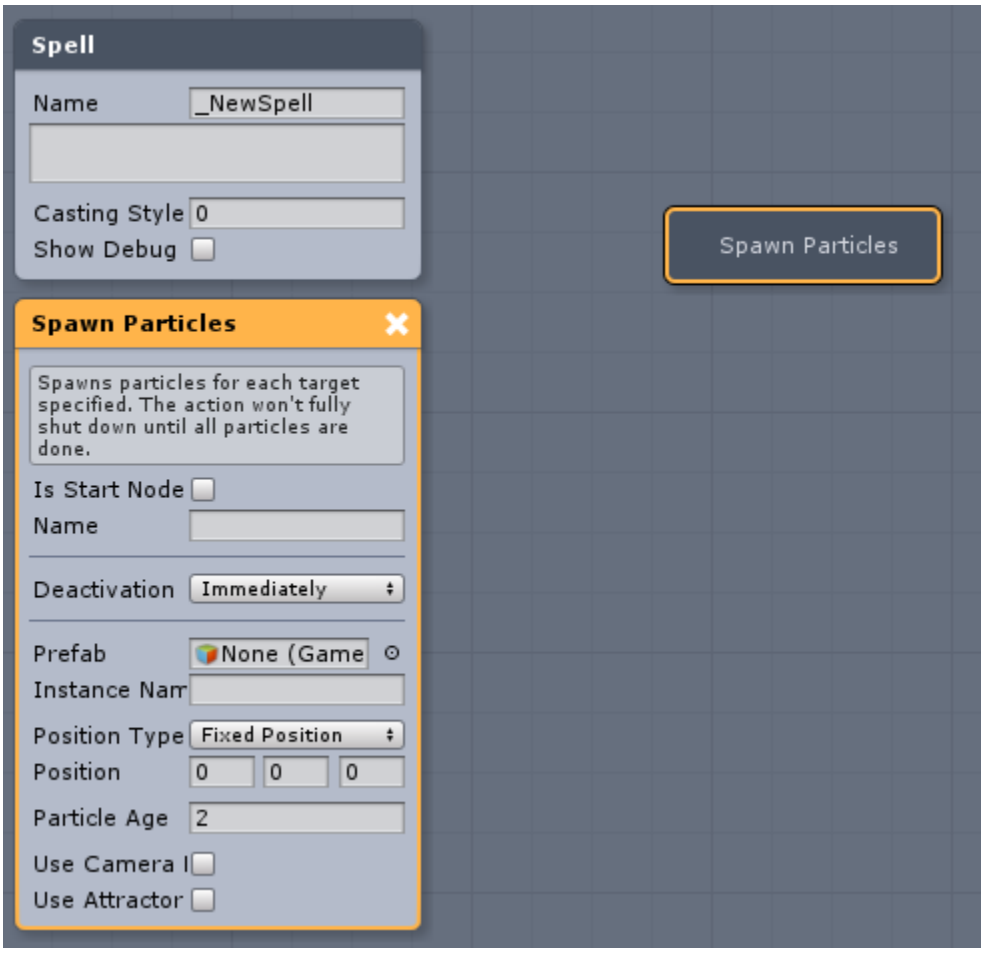


This will create a node that has no action. It will say 'No content'.

When the node is selected, it will be bordered in gold and you can press the 'Select' button in the node details window to select what action the node will perform.



Once selected, the node will change to show the action's title. The node details will also update in order to let you set properties.



Deleting Nodes

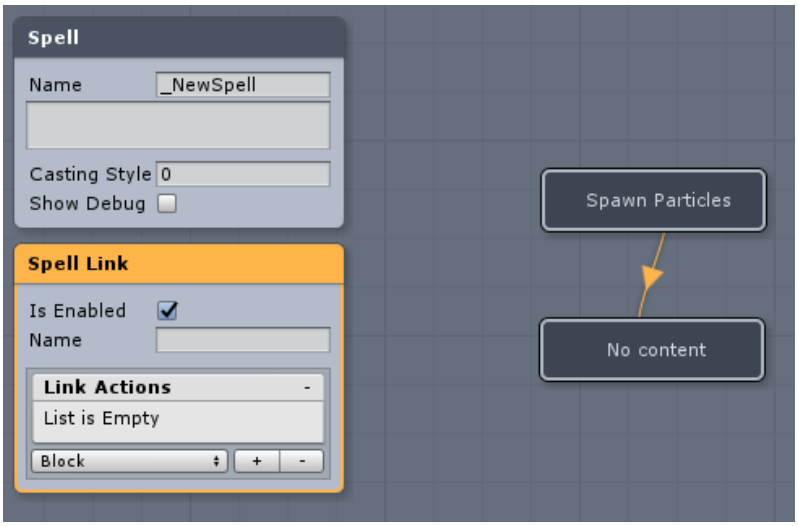
To delete a node, simply right-click the node and select "Delete Node".

Clearing Nodes

To clear an existing spell action from a node, simply press the 'X' at the top right of the node details window. The node will stay, but the spell action will be removed.

Adding Links

When two or more nodes exist, you can right-click a node and select “Add Link”. You can use the mouse to then select the child node that the link will end in.



In doing this, the link will turn gold (meaning its selected) and the node details window will turn into a link details window.

Here, you can add a link action if you want.

Deleting Links

To delete a link, simply right-click on the triangle and select “Delete Link”.

Menu

The menu at the top allows you to create a new spell, open an existing spell, save an edited spell, or delete a spell.

Note: Unity’s asset database actually stores some data during run-time. So even if you DON’T save spell changes, you can close the spell editor and re-open it and the changes will be there. However, if you close Unity without pressing “Save”, the changes will be lost.

Tips

When I create spells, I have a general idea of what I plan on doing. I start by laying out a bunch of empty nodes that follow what I think the flow will be. Then, I go back and fill in the spell action for each node.

You don’t have to do it this way, but I found it helps me to organize at a high level first.

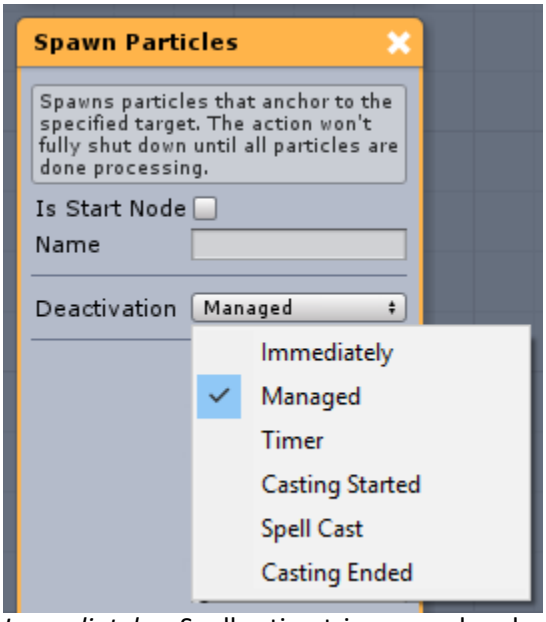
Spell Actions

I've included several dozen common spell actions for you to use. However, you can also create your own spell actions and they will show up in the list as well.

In the editor, the set spell action will provide a small description of what it's for. Hovering over the tool-tips will provide information about each property.

Common Properties

There are some common properties across most spell actions.



Is Start Node – If checked, the node will be a starting point for the spell. All start nodes have their flows started at the same time.

Is End Node – If checked, the node will be started once the spell finishes or is cancelled. This provides a way to clean-up.

Name – Descriptive name of the node that you can add. It will change the name in the graph as well.

Deactivation – This drop-down is used to determine how the node deactivates.

A node that is active typically won't have flow move beyond it. Once an action stops, the node becomes inactive and flow continues. So, deactivating can happen as needed.

Immediately – Spell action triggers and node deactivates right away.

Managed – Node won't deactivate until the spell action tells it to.

Timer – Node will automatically deactivate after the specified seconds.

Casting Started – Deactivation occurs when the casting animation starts.

Spell Cast – Deactivation occurs when the spell is actually cast.

Casting Ended – Deactivation occurs when the casting animation ends.

Prefab – The Unity prefab that will become the template for the action. An instance will be created when the spell action activates.

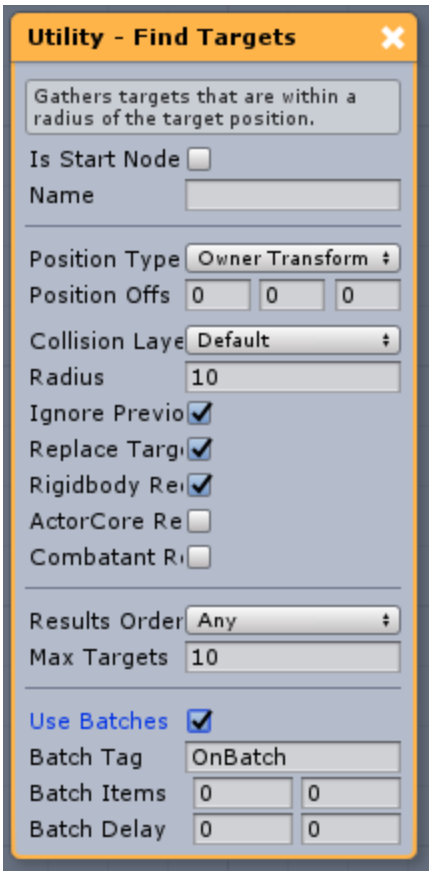
Stock Spell Actions

I won't go into detail on all the spell actions since there is a description in the editor. However, I'll cover some of the more important ones.

I also won't re-define properties over and over. So, if I define a property once, I won't redefine it for each action.

Utility – Find Targets

This action is used to gather a group of GameObjects that are within a radius and then send those targets to the follow-on nodes.



Most of the properties will be self-explanatory, but here's some key ones:

Position Type – Allows you to define the center of the radius. This can be the owner, a fixed position, or even the target that was gathered from another node.

Layers – Determines which layers the GameObjects must be on to be considered a target.

Ignore Previous – If a target was selected by a previous node, don't select it this time.

Replace Targets – If targets were previously stored in the spell data, clear it and then store these new targets.

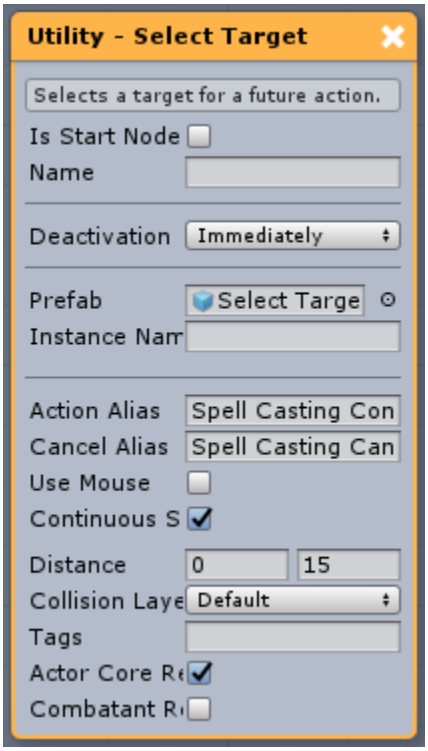
Use Batches – Determines if we'll chunk the targets into small batches and return those batches as different times.

Batch Items – Min and max number of targets per batch.

Batch Delay – Min and max number of seconds between batches.

Utility – Select Targets

This action allows you to select a GameObject either by looking at it with the camera view or clicking it with the mouse.



Prefab – Used to identify the visual aspect of the select. For example, a glow that occurs when a potential target is hovered over.

Action Alias – Input alias used to do the selection (ie the left-mouse-button)

Cancel Alias – Input alias used to cancel the selection (ie the esc key)

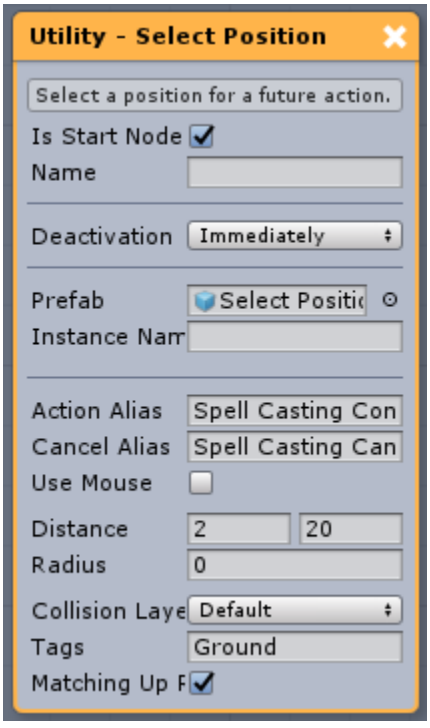
Use Mouse – Determines if we use the mouse to select instead of the camera.

Continuous Select – Determines if we'll automatically move the prefab instance as the camera scans over potential targets.

Tags – Tags on an Attribute Source that must exist for the target to be viable.

Utility – Select Position

Selects a position on the ground based on the specified criteria.



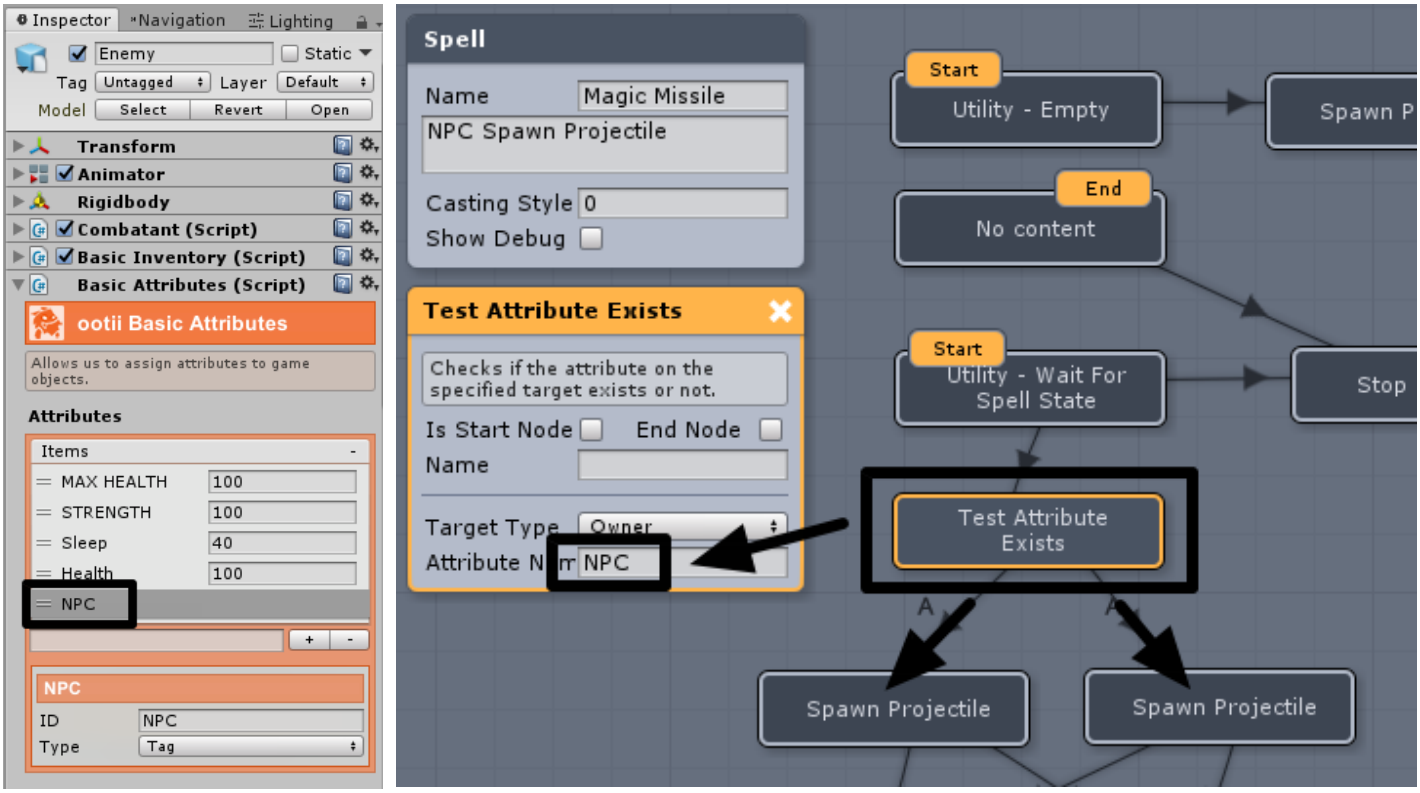
For property descriptions, see previous Spell Action descriptions.

NPCs

Not all spell actions can be used with NPCs. Take the “Utility – Select Position” action that we just talked about. Typically it allows the player to select a position using the camera. An NPC can’t do that. Instead, we need to select a position for him.

When creating a spell, you need to think about how parts of that spell would be activated and use the appropriate actions.

In spells like Magic Missile and Chain Lighting I use an attribute to define if the caster is an NPC:



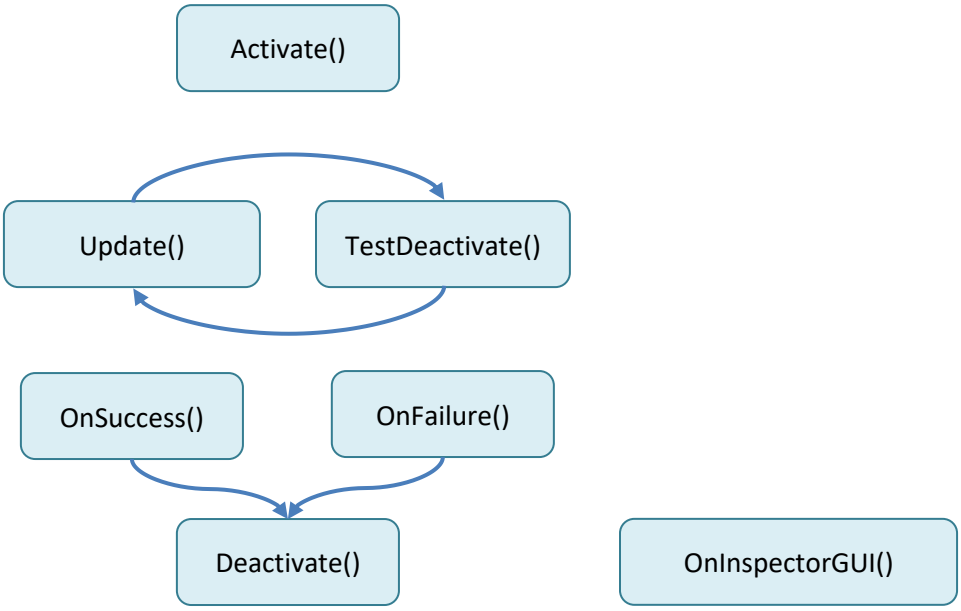
Within the spell, the “Test Attribute Exits” node is used to change the flow of the spell depending if it is cast by an NPC or the player.

Custom Spell Actions

In order to write custom spell actions, you need to write some code. The basic format of the spell action is pretty simple. We’re simply activating the spell action and waiting for it to deactivate. When it deactivates depends on the spell action itself.

Spell Action Flow

Each time a spell action is entered, the Activate() function is called. This allows you to initialize the action at each run. Then, the Update() function runs each frame and calls TestDeactivate(). When the action has finished, it typically calls OnSuccess() or OnFailure() which then tells the action to end using Deactivate().



Your action will inherit from SpellAction and override these functions in order to create your unique action. In some cases, you may simply call OnSuccess() in the Activate() function if your action does an "instant" task and is done. Other times, you may want to use TestDeactivate() to determine if it's time to end.

The majority of your work will be done in Update() in order to process each frame.

Remember to remove the keyword '**abstract**' from your class or it won't show in the list of Spell Actions!

Spell Action Template

```
using System;
using com.ootii.Base;

namespace com.ootii.Actors.Magic
{
    [Serializable]
    [BaseName("Friendly Name")]
    [BaseDescription("Friendly Description")]
    public abstract class SpellActionTemplate : SpellAction
    {
        /// <summary>
        /// Used to initialize any actions prior to them being activated
        /// </summary>
        public override void Awake()
        {
            base.Awake();
        }

        /// <summary>
        /// Called when the action is activated
        /// </summary>
        public override void Activate(int rPreviousSpellActionState = -1, object rData = null)
        {
            base.Activate(rPreviousSpellActionState, rData);
        }

        /// <summary>
        /// Called when the action is meant to be deactivated
        /// </summary>
        public override void Deactivate()
        {
            base.Deactivate();
        }

        /// <summary>
        /// Called each frame that the action is active
        /// </summary>
        public override void Update()
        {
            base.Update();
            if (mAge >= 5f) { OnSuccess(); }
        }

#if UNITY_EDITOR
        /// <summary>
        /// Called when the inspector needs to draw
        /// </summary>
        public override bool OnInspectorGUI(UnityEngine.Object rTarget)
        {
            mEditorShowDeactivationField = true;
            bool lIsDirty = base.OnInspectorGUI(rTarget);

            return lIsDirty;
        }
#endif

    }
}
```

In the above example, you can see that I overrode several of the functions. To keep the code sample small, I just customized it in the Update() function. Once the action has been active for more than 5 seconds, I end it successfully.

Steps

1. Copy the above code to a new file and rename that file to match your spell action
2. Line 7 – Change the friendly name that will display in the selection list
3. Line 8 – Change the friendly description that will display in the selection list
4. Line 9 – Remove the keyword 'abstract'
5. Line 9 – Change the class name so it matches the file name (without the .cs)
6. Line 16 – Add any initialization code needed
7. Line 24 – Add any activation code needed
8. Line 32 – Add any deactivation code needed
9. Line 41 – Remove my example and add any update code needed
10. Line 52 – Add any inspector code needed

OnInspectorGUI()

This function is an edit-time only function that is used to render the spell action in the node details window. It follows the same flow and logic as Unity's standard inspector.

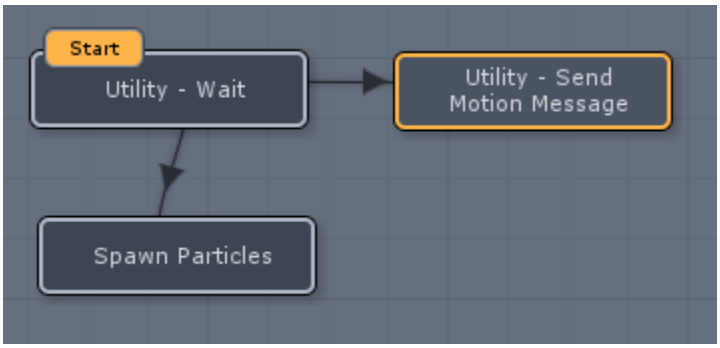
To see how I use it, just look at the stock spell actions as an example.

Custom Spell FAQ

When you think of spells in role-playing games like Dungeons & Dragons, there are some standard concepts that exits. Part of this FAQ will be to show how to implement those concepts.

How do I implement activation time?

Use one of the 'paused' casting styles (1 or 3) and use the 'Wait' action.

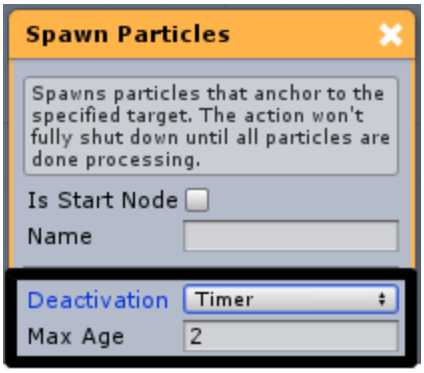


When cast, the 'start' node will have the caster held in the casting pose for as long as you set.

Once that wait has finished, the 'Send Motion Message' will have the caster continue the animation. At the same time, the other spell nodes will continue.

How do I implement spell duration?

This can be a little different based on the flow you create. You could use a 'Wait' action or (in some cases), you can use the timer on the other actions.



In the case of 'Spawn Particles', the particles will last the specified amount of time (Max Age). Once that happens, the particles will be shut down and the flow will continue.

How do I set ranges?

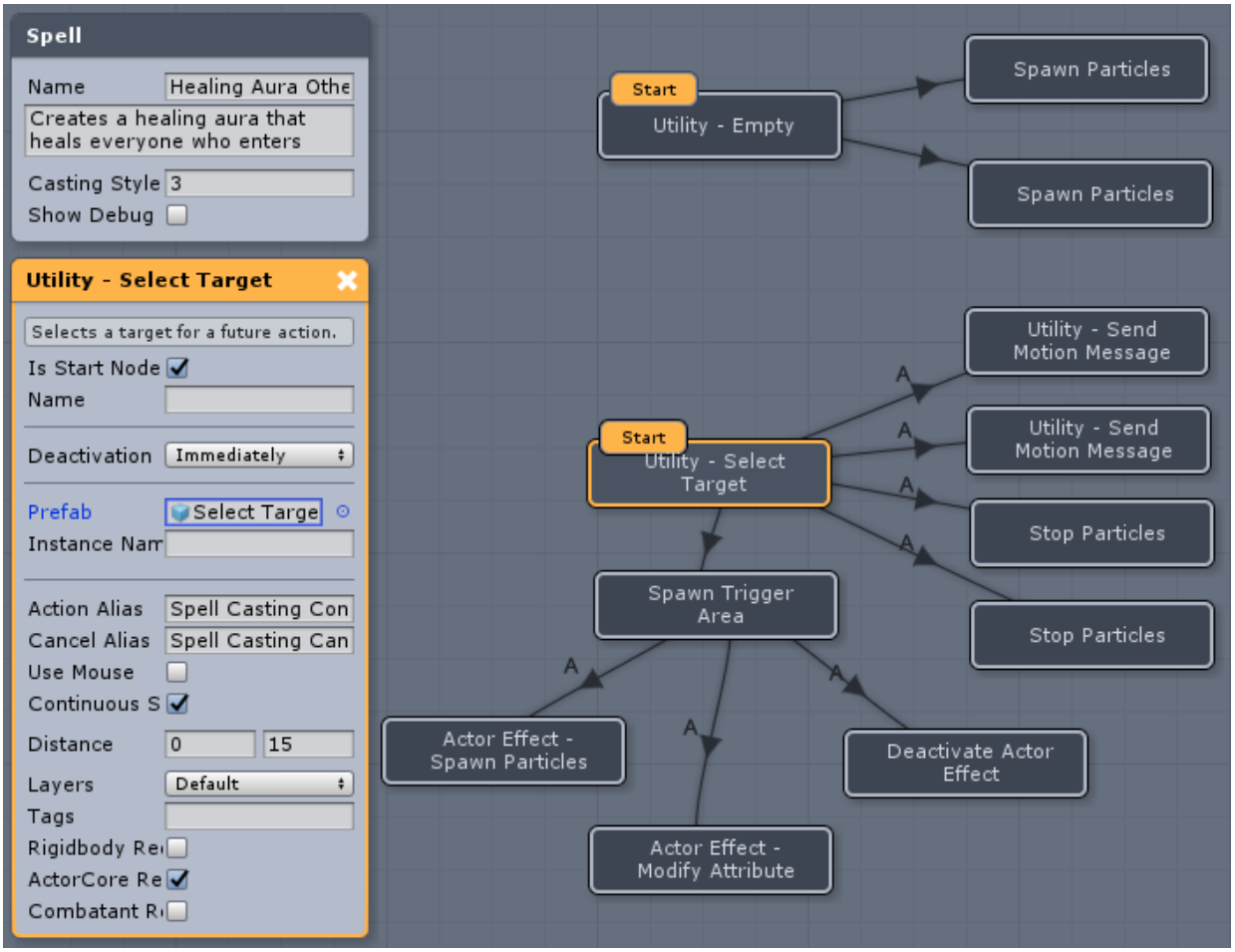
Spell range is typically set on the spell actions themselves. For example, the 'Spawn Projectile' action has a min and max distance that can be set. The 'Find Targets' action has a radius you can set.

How do I select a target?

This is a pretty common thing to do. To do this, it will take a couple of steps. Let's look at the 'Healing Aura Other' spell.

The first thing we'll do is set the spell's Casting Style to a 'paused' style (3). This will keep the caster in a casting animation until we tell it to continue.

The Utility – Select Target is what really matters. This action will cause the selection and return results.



Utility – Spell Target

The "Utility – Select Target" spell action is important here as it's what we'll use to pause the flow and select a target.

The following properties will be important:

Prefab – This is the GameObject that will be instantiated and do the actual selection. It's an effect prefab that you can replace if you want. The important on that prefab is the **Select Target Core** component (we'll talk about that later).

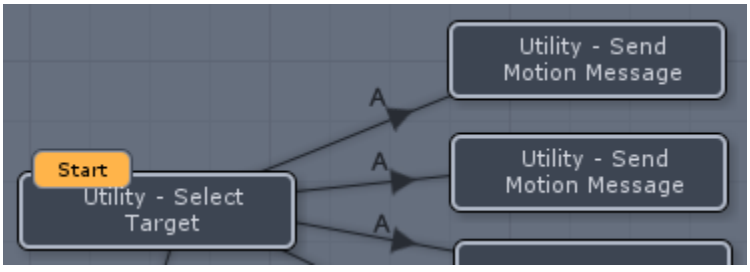
Action Alias & Cancel Alias – The input aliases used for selecting the target or canceling the selection.

Continuous Selection – Determines if the raycast is constantly looking for targets or only when the Action Alias is pressed.

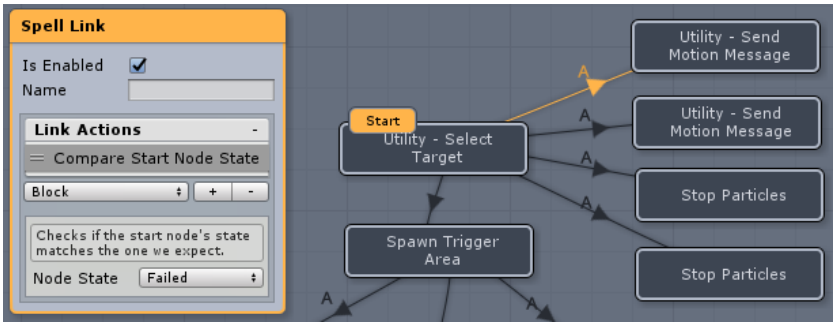
The rest of the properties are all conditions that can be placed to determine what can be selected. The property tooltips will explain more.

Coming out of the Utility – Select Target node, we have two options... a target was selected or the selection was cancelled. In the case of a selection, the node is a success and flow continues. In the case of a cancel, the node fails and (typically) no child node is processed.

I say typically, because we can use Link Actions to change that. Let's focus on these two top links:

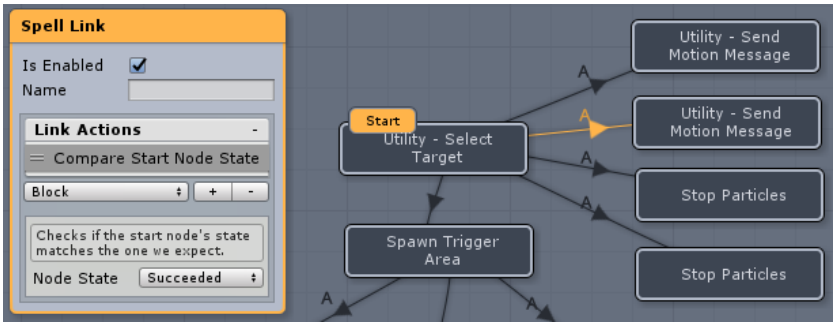


Both links have Link Actions on them (as indicated by the 'A').



The first one will look at the previous (or starting) node's state. If the state is 'Failed', we'll actually traverse this node.

That means, when the selection is cancelled.



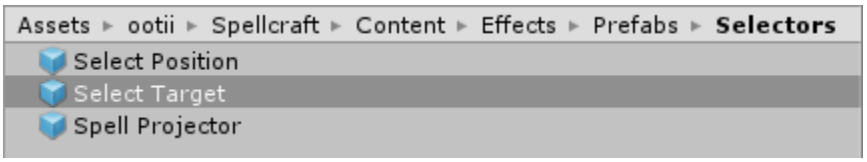
The second one will also look at the previous (or starting) node's state. If the state is 'Succeeded', then we'll traverse this node.

That means, when a selection was made.

The Utility – Send Motion Message is then used to tell the caster's animation to deactivate or to continue and finish gracefully.

Select Target Core

So, let's go back to the Prefab property. In this spell, we're using the effect prefab called Select Target:

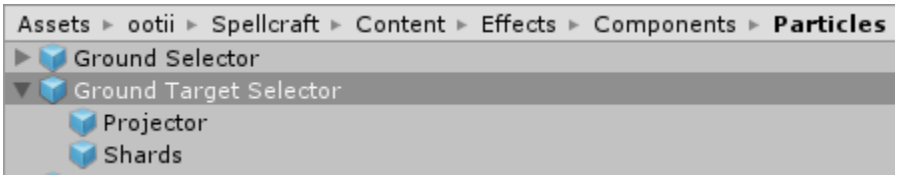


We break it out like this because I use Select Target everywhere. If we look at that prefab, we see it uses the SelectTargetCore component.

Notice how the component has lots of properties similar to the action. The action actually sets these. That means we override this prefab based on the properties of the action.

This effect prefab then uses the component prefab called "Ground Target Selector". As discussed earlier, this is just the actual particle effects.

The Ground Target Selector prefab is found here:



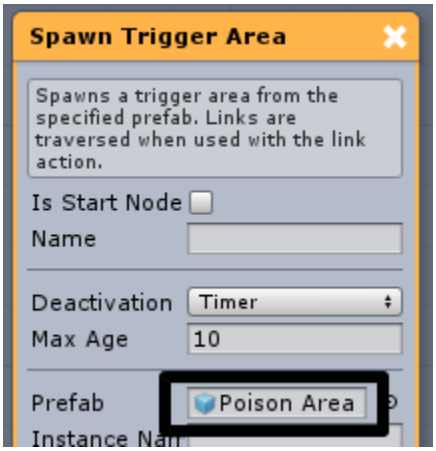
I do it in these layers so that you can simply replace the particle effects if you want, replace the actual selector core, or replace the spell action itself. All of these pieces are also reusable.

How do I do area-of-effect spells?

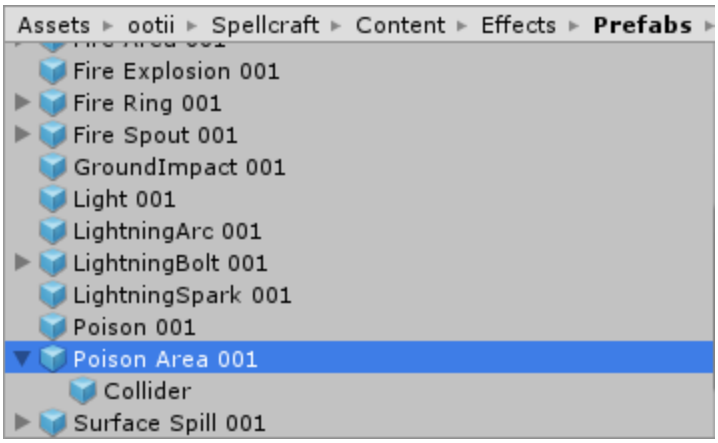
Several of my spells use this ability. The trick is to simply use the features Unity gives us... that means triggers.

Let's look at the Poisonous Fog spell. It uses a custom collider that is a column to define the area.

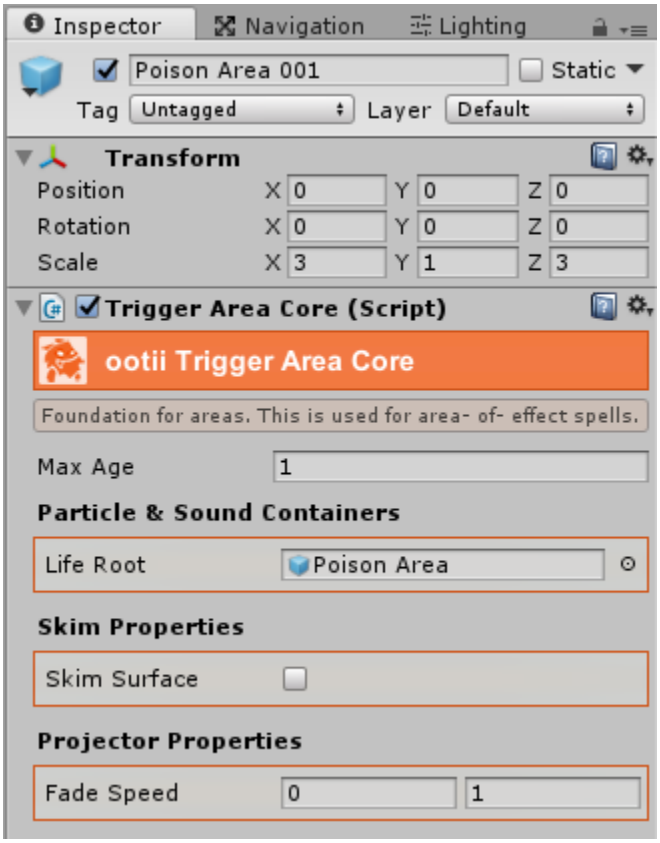
The collider itself exists on the effect prefab. Think of this as the low-level functionality of the spell:



Let's look at "Poison Area 001" (you can't see the 001 in the image above):



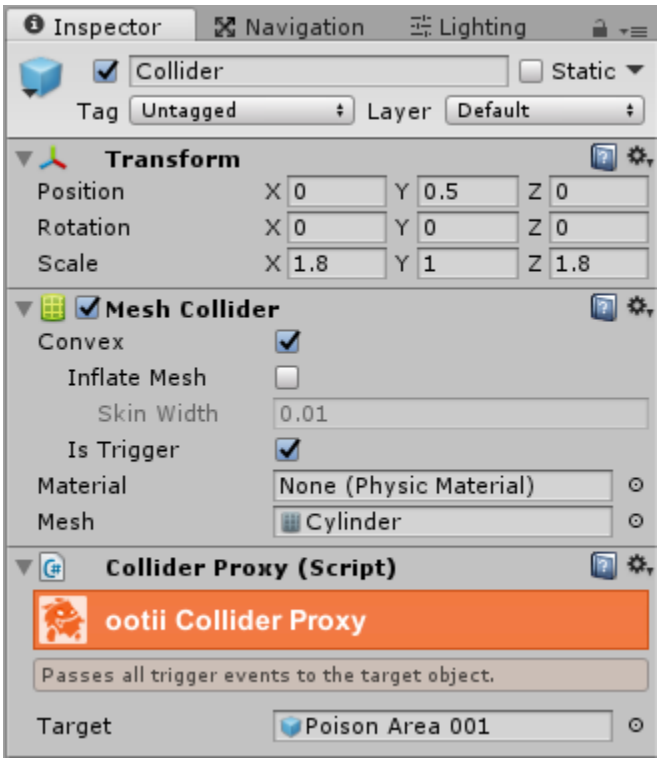
We use these separate effect prefabs because it allows us to reuse them in different spells. In this case, the "Poison Area 001" prefab has a couple of parts:



First is the main GameObject that is the prefab.

Notice the **Trigger Area Core** component. It's the component that manages the life cycle of the trigger area.

The "Life Root" is just the particle system prefab (component prefab) that will play... the green fog. It runs as long as the area is active.



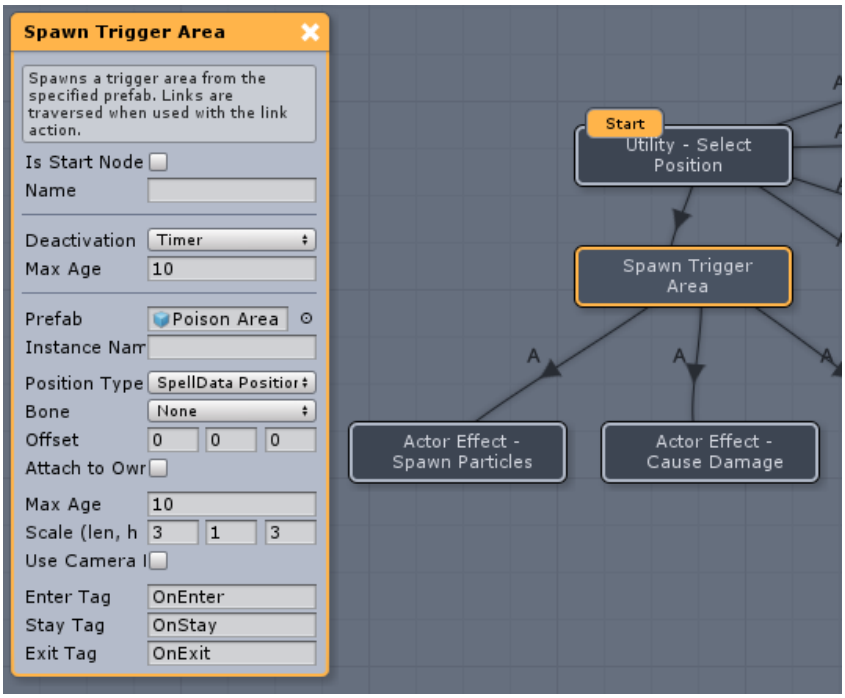
Second is the child GameObject of “Poison Area 001”. This is the actual collider/trigger.

The reason I place it separate is so that we can control its position, rotation, and size relative to the actual spell’s center.

We use the **Collider Proxy** to send the actual collision/trigger events back to the parent GameObject.

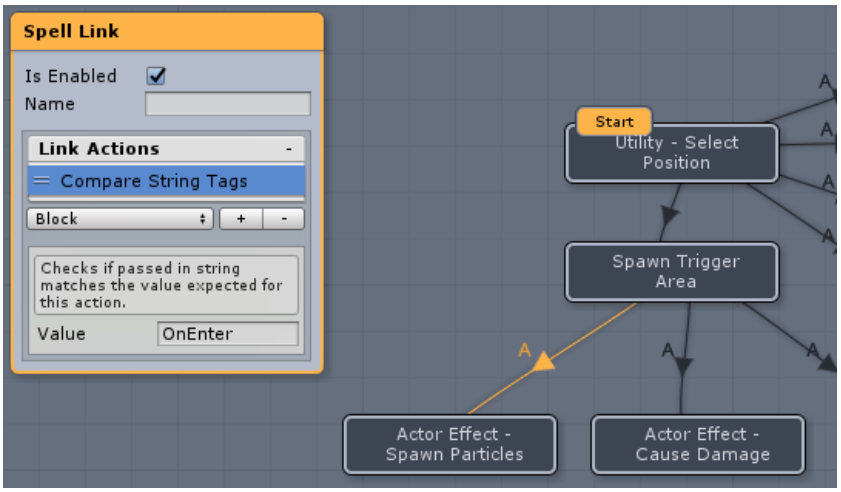
The main purpose of the Trigger Area Core is to return the TriggerEnter, TriggerStay, and TriggerExit events from the collider back into the spell graph. The flow is like this:

1. The Collider child gets the Unity events
2. The Collider Proxy sends them back to the parent (Poison Area 001)
3. The Trigger Area Core gets these events and sends them back to the graph node (Spawn Trigger Area)
4. Spawn Trigger Area allows child nodes to continue



Notice the "Enter Tag" property on Spawn Trigger Area?

That's a string that we'll use as a Link Action to cause an effect to happen on someone who enters the trigger area.



The Spawn Trigger Area will force the child links to process with the specified tag. If there's a 'Compare String Tags' link action, that will filter to ensure the right child node is processed.

In this way, our spell can do things as objects enter and leave the trigger area.

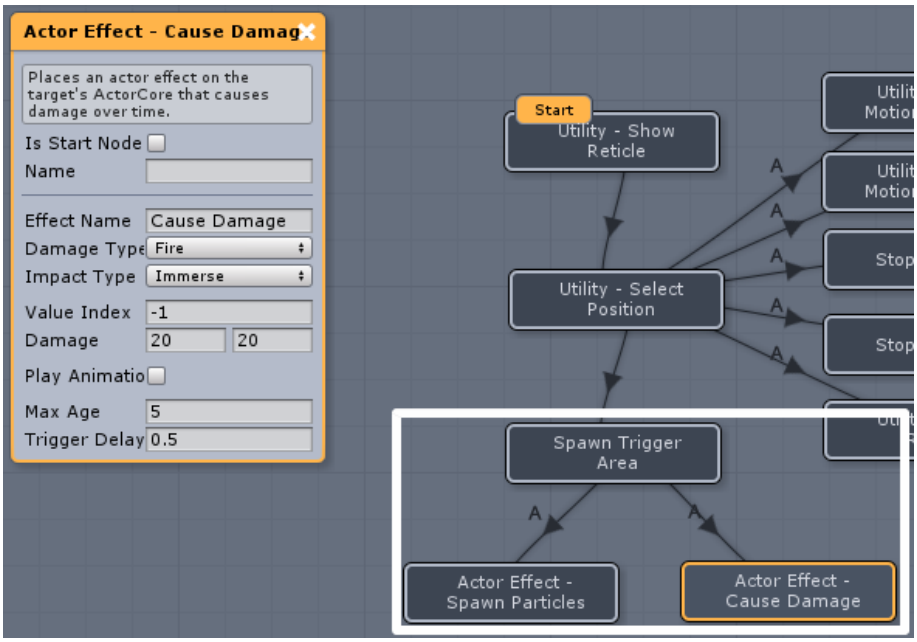
How do I do damage-over-time spells?

Most spells that affect the target over time are done using ActorCoreEffect. The reason is that it move the life cycle of the effect out of the spell and onto the character.

Let's take a character who walk into a wall of fire as an example.

The Fire Wall spell is cast and an enemy walks into it. When he does, the spell really isn't burning the character... it's the fact that he walked into fire and his clothes catch fire. If the spell caster dies, the enemy's clothes are still on fire and still burn him.

So, with this mindset we use the 'Actor Effect – XXX' actions:



Following from the previous question, the enemy walks into the area and the child nodes at the bottom right of the image fire.

We spawn a particle effect that is managed by the ActorCore and then we spawn a damage effect that is also managed by the ActorCore.

Both of these have a max age. When that expires, the effect is removed.