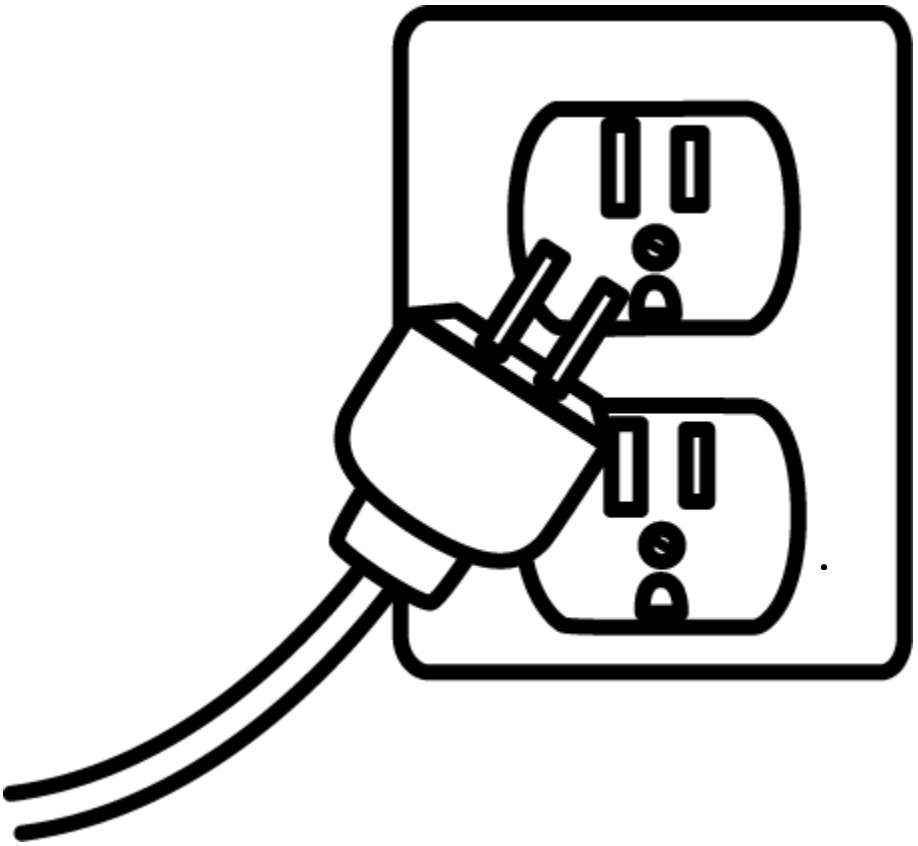
PLUG



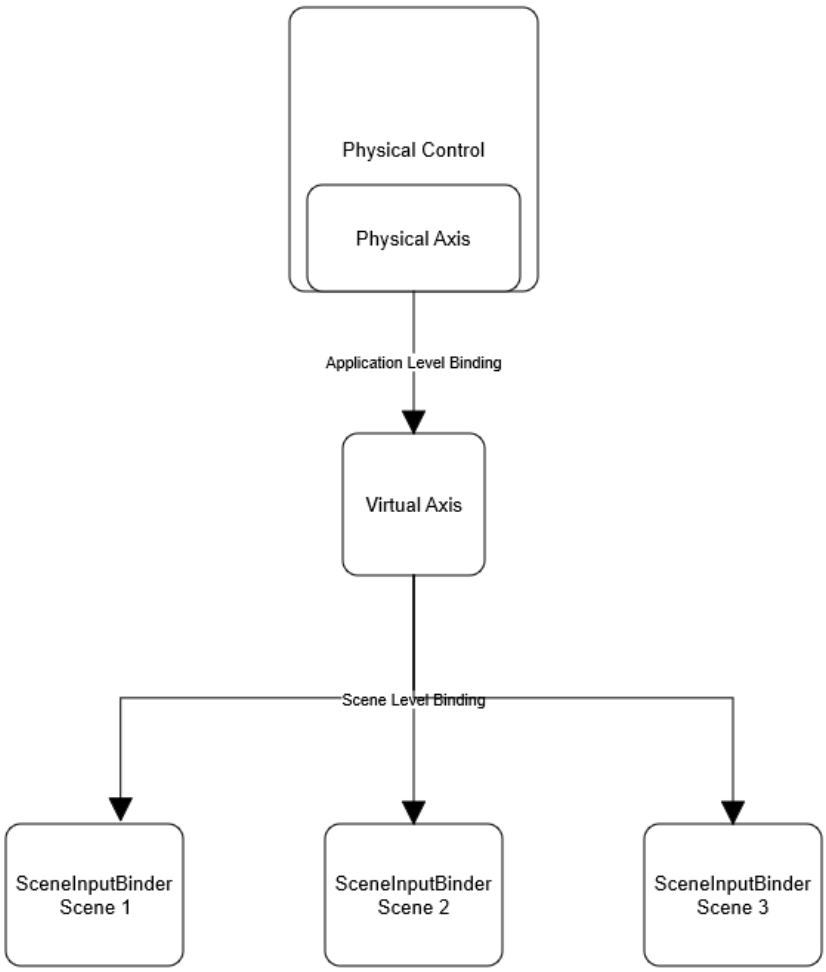
THE UNIFIED EXTENSIBLE INPUT MANAGER FOR UNITY

Introduction

PLUG grew out of my need to integrate various Virtual Reality tracking devices and custom game controllers in a unified and extensible manner. As I was attacking this anyway, I also created a virtual-input layer to allow players to remap controllers at play-time as this is a feature of most modern games.

Control Flow

Input control flow proceeds in an event driven fashion through three layers:



The physical layer is read by am extensible set of API specific modules called input providers that implement the **PlugInputProvider** interface. It is not necessary for a client app to register or describe the input providers, they are found automatically on game start by the Plug system.

Input providers have two functions: Discovery and Input. Discovery means that they supply to the system a description of every button, axis or other input control attached to the system. These controls are grouped by the device they are part of, and the input provider that provided them such that the X axis of the mouse detected by the Unity input system might be named *UNITY.Mouse.X*. A joystick X axis discovered by DirectX might be described as *DX.Logitech Wingman.X* and so forth. The exact names of providers, devices and controls are all up to the input provider. The only limit is that two different provider should never have the same provider name.

Controls come in 4 varieties or types:

- Digital Controls
  This are buttons that can be either down or up and send an event when their state changes
- Analog Controls
  These are controls that have a state represented by a floating point number. They send an event whenever that number changes.
- Normalized controls
  These are analog controls who value is bounded between 0.0 and 1.0 inclusive.
- Ascii Controls
  These are controls that return an integer representing a key press. They send sn event every time a new key is pressed.

Input is reported as device and control callback events. There are C# events for the adding and removing of devices as well as state changes of any of the controls. The callbacks are all routed through a singleton class called the **PlugInputProviderManager.** If all an application wants is low level access to physical controls, it can register itself with the provider manager to get all of these callbacks.

Virtual Axes

In order to allow for user reconfigurable controls, a second layer is provided calls the virtual axis manager. This is also implemented through a singleton class called **PlugVirtualAxes.** (Axes is the plural of axis and will be a term that is used many times in this document.) A virtual axis is like an alias for a control. It has a type that matches one of the four control types and can be bound to any physical control of the same type.

This binding is stored in an XML file and is soft. It may be set either at game development time using the Virtual Axes editor window, or at run-time from inside the game.

The game developer can define as many axes as he or she wishes in the Virtual Axes editor window and assign their default mappings to physical controls. Any number of virtual axes can map to the same control, but any given virtual axis can only be mapped to a single physical control. (Which is to say that the virtual axis to physical axis relationship is n:1 .)

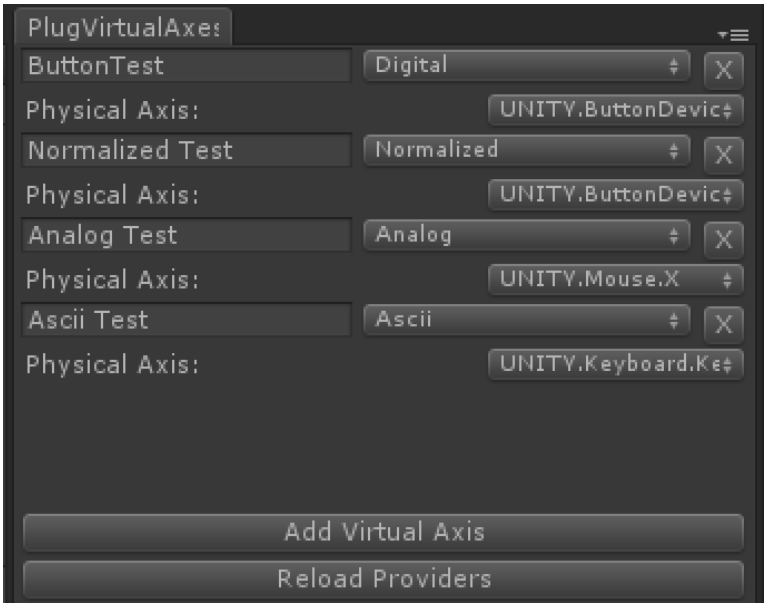
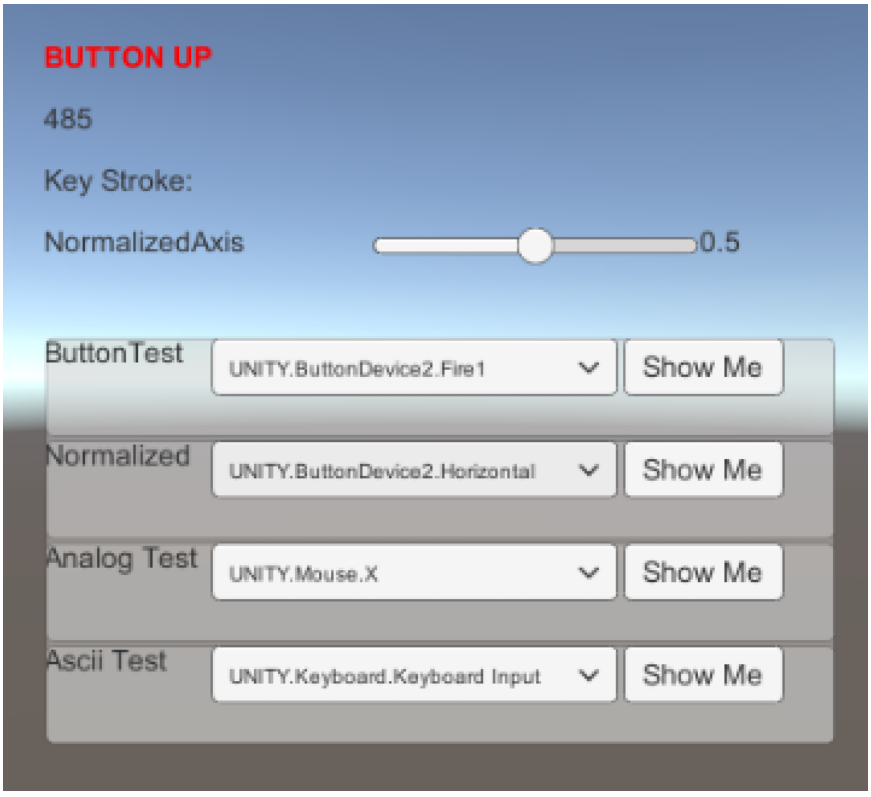


Figure 1The Virtual Axes Editor Window

The number of virtual axes is fixed at run-time, but API calls are provided to read and set the virtual axis to physical axis binding. Below is a screenshot from the examples that shows how to create an input remapping GUI inside of your UNITY game.



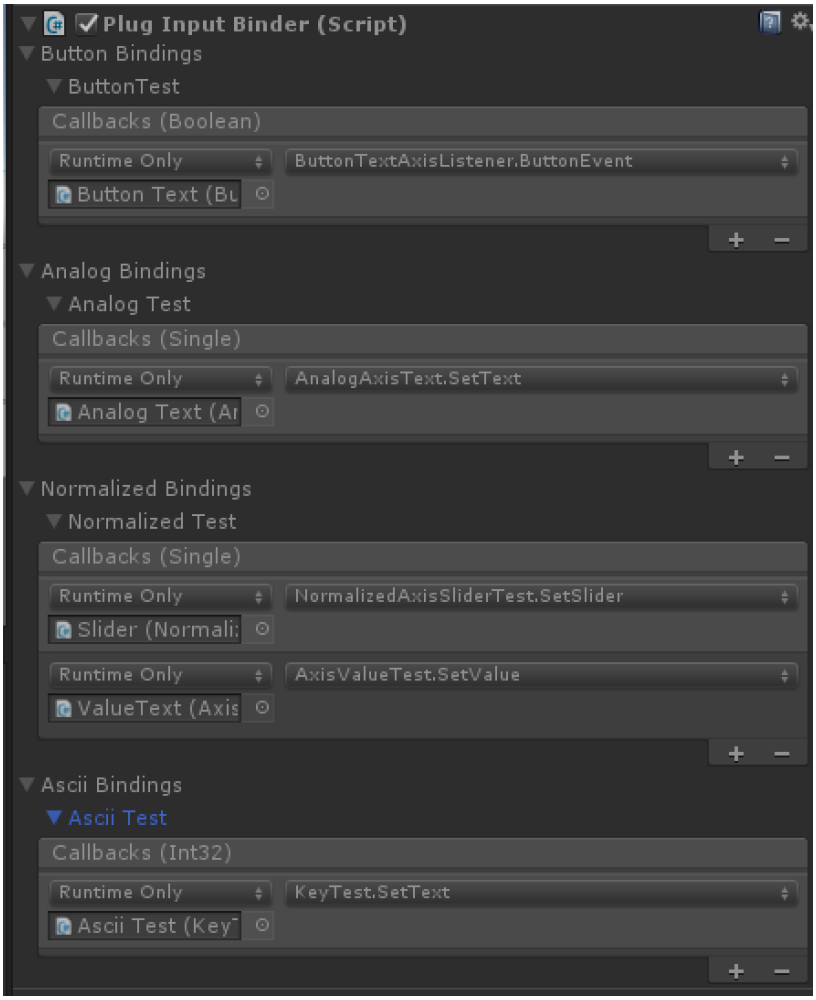
Input Binding Layer

The physical axis layer and the virtual axis layer of the code are both built on top of application-wide data. It doesn't matter which Unity scene you are in, they all share the same physical controls and virtual axes. The input binding layer is different.

The input binding layer is implemented by a Unity prefab called the **PlugInputBInder.** The **InputBinder** contains a single special monobehavior also named Plug Input Binder. This monobehavior is something we call a "Managed Singleton" or a "Scene Singleton."

Scene singletons are unity behaviors that are limited to one per scene and accessible on the scene code level through a public **Instance** variable. They store their state with the scene and recover it when returning to that scene.

The PlugInputBinder makes all of the virtual axis callbacks available as **UnityEvents** so you can use the Unity event system to wire up your application. Below is the inspector for the above scene showing how the **PlugInputBinder** has been used to connect the virtual axes to the output displays in the GUI.



Extending the Input System

The basic plug system just virtualizes the existing Unity input system. In order to add new input sources, all you need do is sub-class the **PlugInputProvider and create a new sub-type**. The system will automatically pick up and instance all classes in your project that do so. WorldWizards makes packs of input providers available at a very modest cost. (eg The Windows DX/USB pack. The VR devices pack, and so forth.) But you can also write your own for any special needs you have. All source code to the system, including the Unity input provider, is included for you to use as a guide and reference.

When writing input providers, it helps to understand the input provider life-cycle. Said cycle is described below:

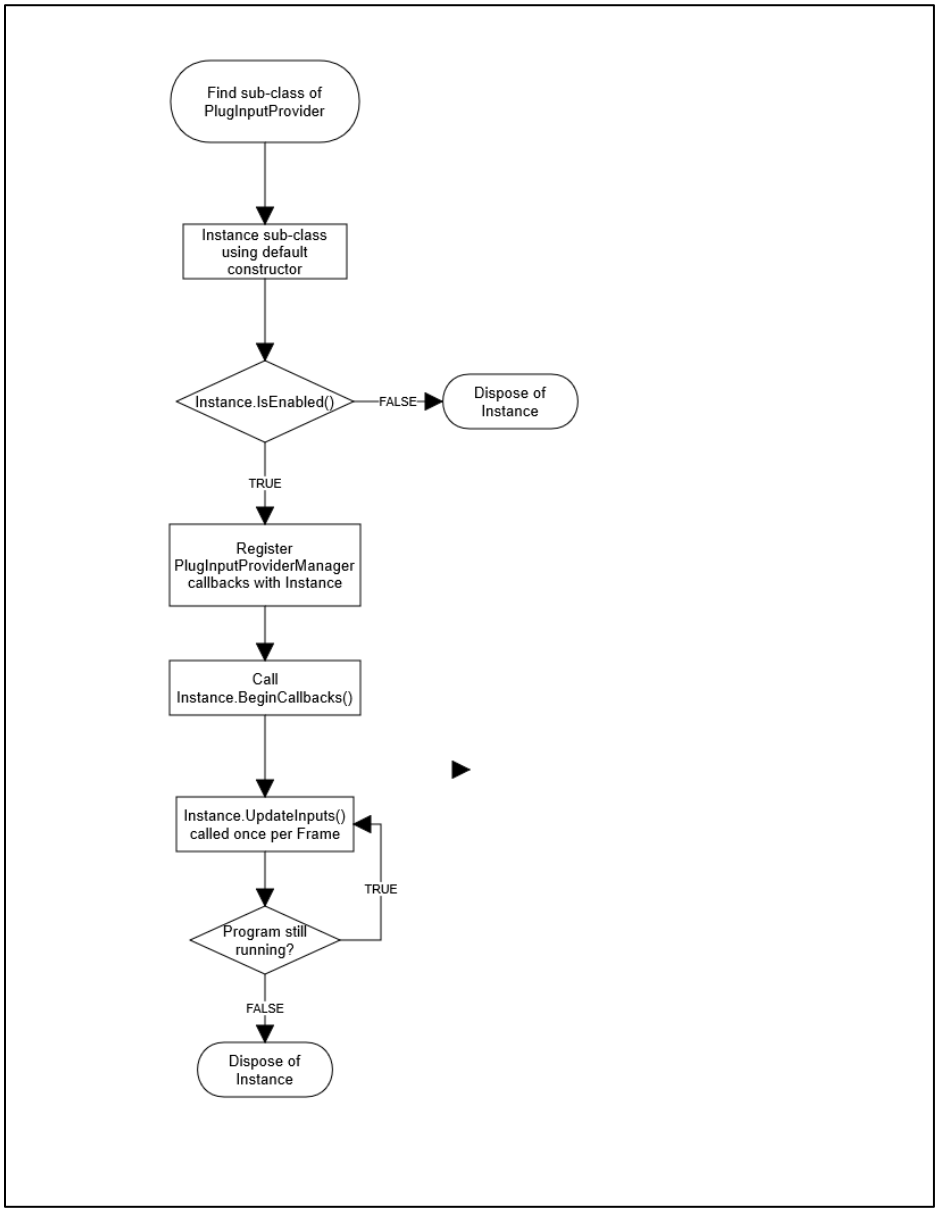


Figure 2: PlugInputProvider lifecycle

The **PlugInputProviderManager** scans and finds the available provider classes in its static initialization phase. (It can be told to repeat this process at any time by calling the *ScanProviders()* API call.)

It then instances each one to make provider object. The *IsEnabled()* method on that object is called. If it returns false then the object is discarded, if it return true then the object is added to the **PlugInputProviderManager**s list of providers and the **PlugInputProviderManager** is registered as the handler for all the object's input events. Finally the object's *BeginCallback()* method is called to inform it that the **PlugInputProviderManager** is ready to handle its events.

On initialization, the object is expected to resolve all its current devices and their component controls, and to make them available via the *Devices* property. When *BeginCallbacks* is called, the object is expected to reply by issuing a *DeviceAdded* callback event for each available device. This is a special case. **All other callbacks must be issued from inside the *UpdateInput()* call to avoid race conditions.** If a provider's underlying device API is based on asynchronous calls, the results of those calls should be stored to be dispatched during *UpdateInput*.

The provider object is kept alive until the program ends, at which point it gets cleaned up.