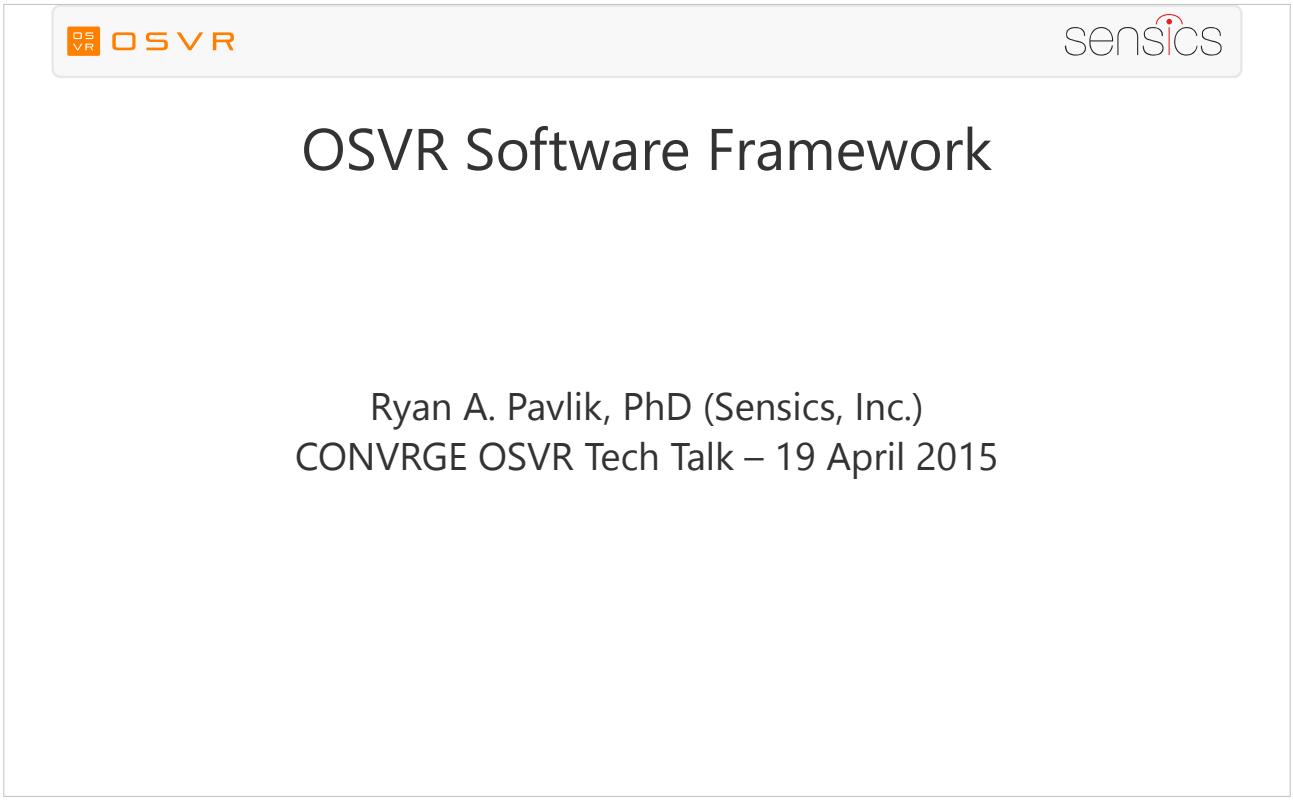
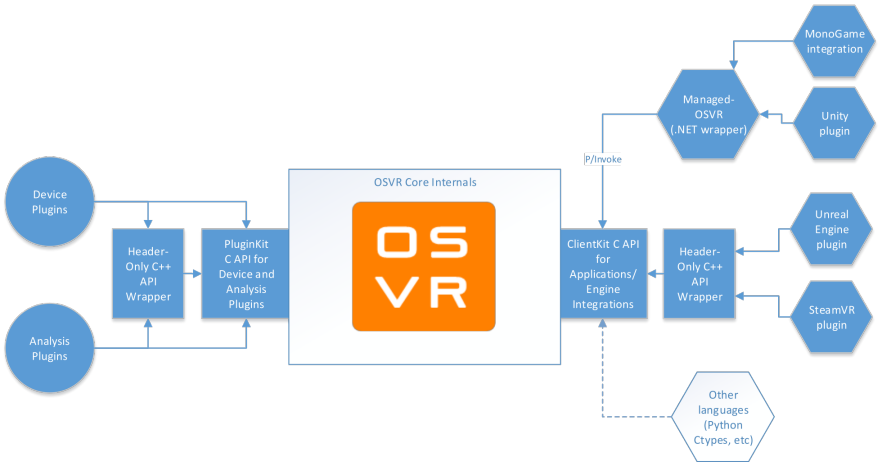


I'm going to present some insight into the core of the OSVR software framework, from my perspective as a senior software engineer for Sensics on the project.

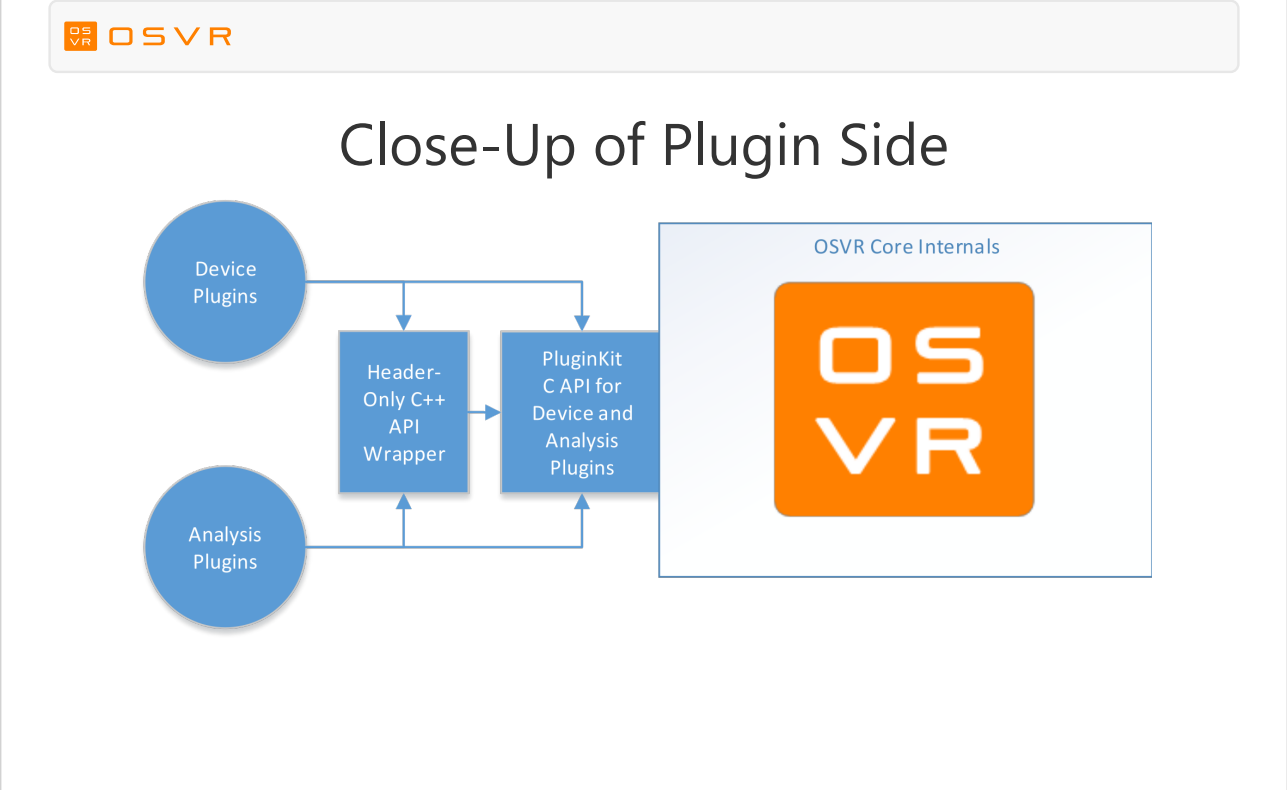
OSVR

A View of the System



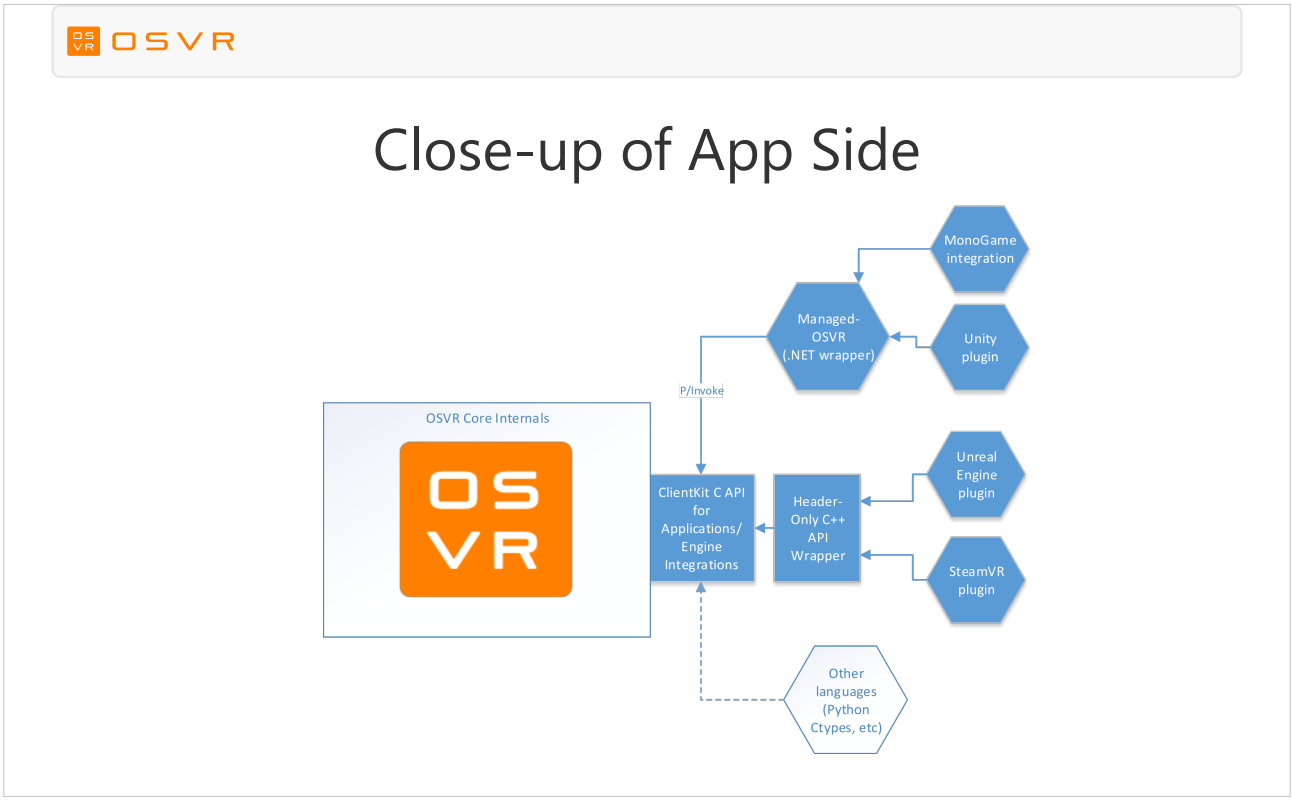
In my mind, when I think of the OSVR Core, it has basically two sides – two basic public APIs – and it serves as a universal interface between them.

Now, if you're developing experiences using one of the game engine integrations available (the far right side of the diagram, we'll zoom in shortly), some of this presentation is probably lower-level than you need to know, and by design, you won't have to think about it. Nevertheless, I think there's value in getting a better idea of how OSVR facilitates interoperability between application software and devices.



So this is what I call the “left side of the stack” - the plugin, server, or device side. Specifically, this is the “PluginKit” API: the API that allows you to write software that interacts directly with hardware to expose generic interfaces (device plugins) or software modules that expose generic interfaces based on input from other interfaces (analysis plugins).

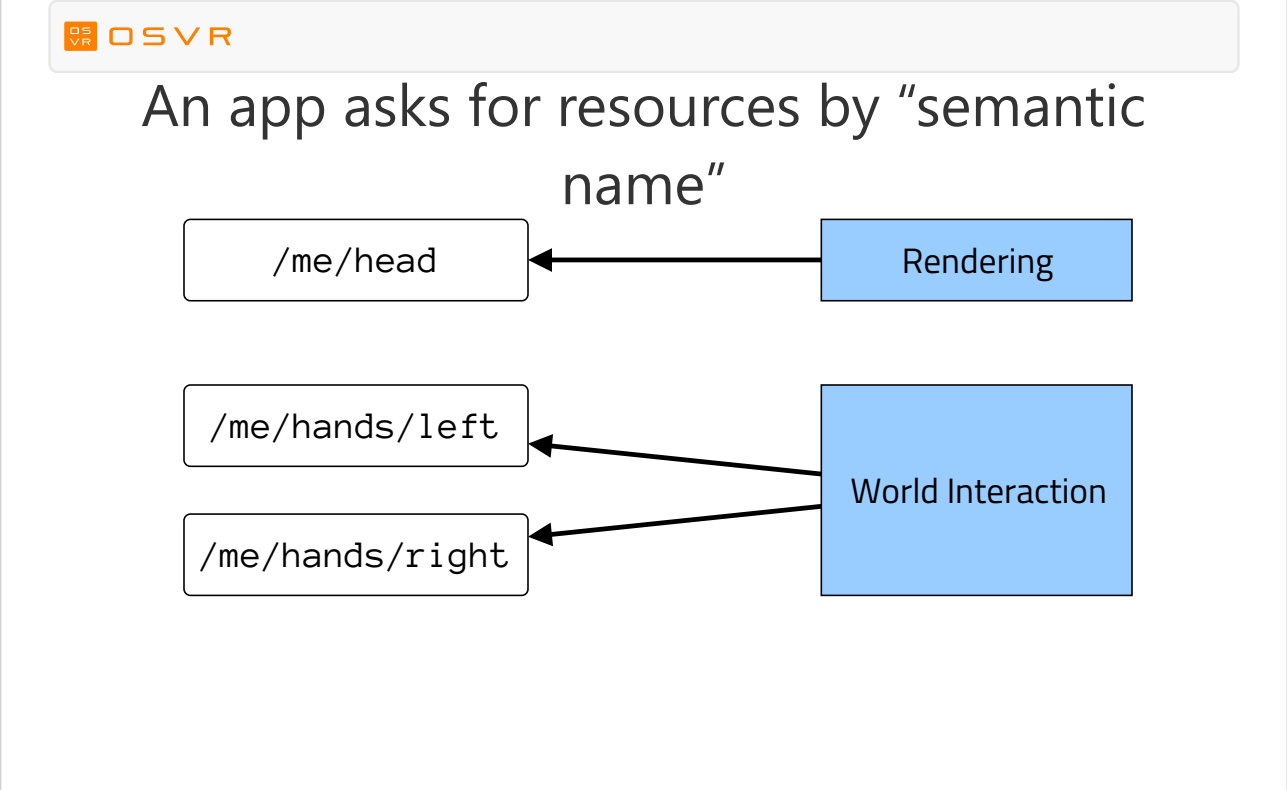
For cross-compiler compatibility, ABI stability, and easier foreign function interfaces (language bindings), the basic API is actually expressed as a C API. There's a header-only C++ wrapper included for safer, more concise and expressive use of the API: it gives you many of the advantages of a C++ API without having to use the same version of Visual Studio (or even the same compiler family) as the core.



Now, on the “right side of the stack,” we get the client, game, or application API, formally known as ClientKit. This is how a VR experience interacts with the functionality provided by plugins (the left side) – getting input data, setting up rendering, etc.

As with PluginKit, ClientKit is a C API (for the same reasons, with FFI made more important because of game engine bindings) with a similar C++ header-only API wrapper.

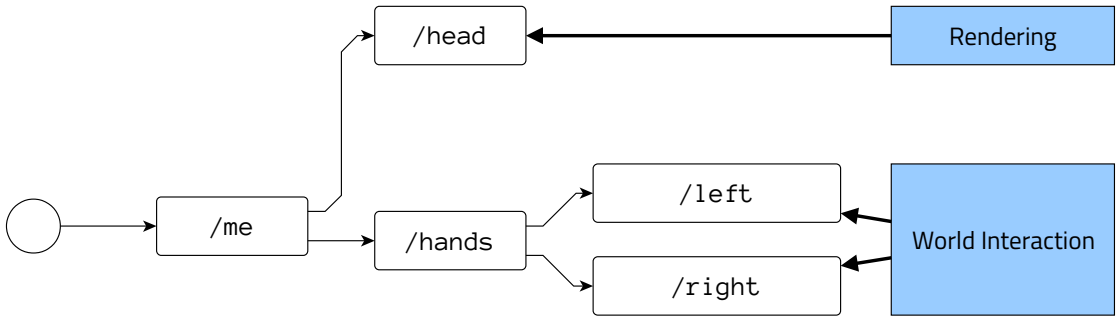
However, there can be a few more layers here, because while you can write an application directly using ClientKit (C or C++), you don't have to. If you're using a game engine, an integration can be written for that engine, then you can create your content in the environment you're familiar with. For instance, there is an integration with Unreal Engine, that takes the OSVR information and concepts and maps them into that engine's design. Our Unity plugin is actually two layers away: there's a generic .NET/Mono binding to OSVR called Managed-OSVR (the reusable stuff that doesn't depend on Unity), that gets used by the Unity plugin making those experiences drag-and-drop simple to create.



So starting over on the right side, let's consider a hypothetical VR application that uses head tracking for rendering and your left and right hands to do some interaction in the game world. In OSVR, the way an application does this is by requesting those resources with a “semantic name” - “/me/head”, “/me/hands/left”, etc. Here, “semantic” indicates a name with application-linked meaning.

OSVR

Actually, a “semantic path” (like a good URL)



And, if those names looked like pathnames, or directory names, you're exactly right: these semantic names actually form part of what we call the “Path Tree” in OSVR. Just like a filesystem path, or closer still a good URL or web app API, resources are identified by paths with meaning.

OSVR

So what?

- This is a higher-level API: the game asks for what it wants by its *meaning*, not by its data source
  - Left hand position, not tracker data from Hydra sensor 0
- So we can make sure the game gets the data with the suitable meaning, with a variety of hardware
- To find the actual data source, we must look at another part of the path tree...

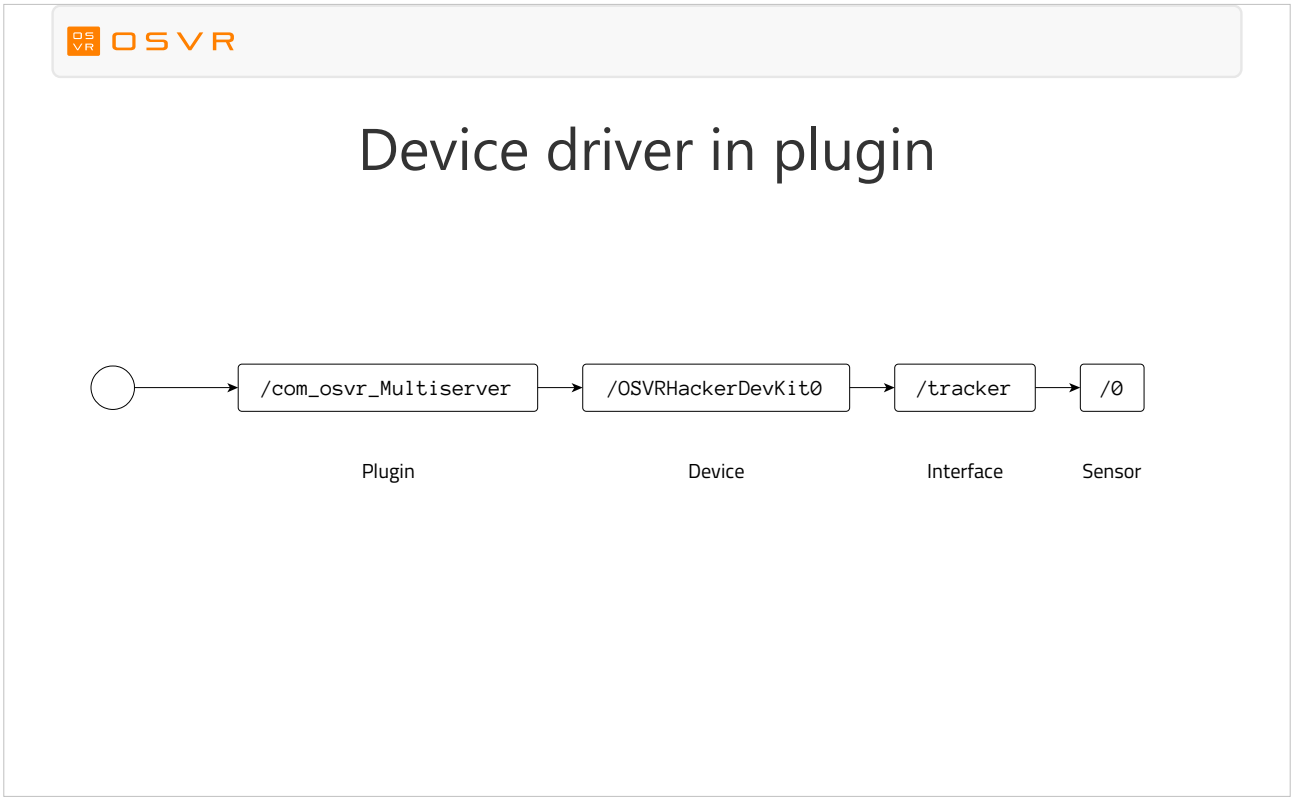
What is the point of having the path tree? Why not just say “sensor 0, sensor 1,” etc.?

The simplest reason is hardware independence: The world doesn't get all its VR-related hardware and software from one vendor; you want to be able to get the particular solutions that best meet your needs – the best HMD, the best input devices, the best trackers, etc.

But further, think of what information the game is providing to OSVR. When it asks for the position of /me/hands/left, we know it wants your left hand position, instead of just saying “Hydra sensor 0”. With this high-level information on meaning, OSVR can work behind the scenes to make the desired data available to the application, regardless of what is tracking your left hand (a Hydra, a STEM, a Leap Motion, some vision-based tracker...)

This is all the application needs to know, and that keeps the application compatible with lots of current devices, as well as future ones.

So now, let's look inside of OSVR and see what it's doing to get that data. For that, we have to go to the left side of the diagram.

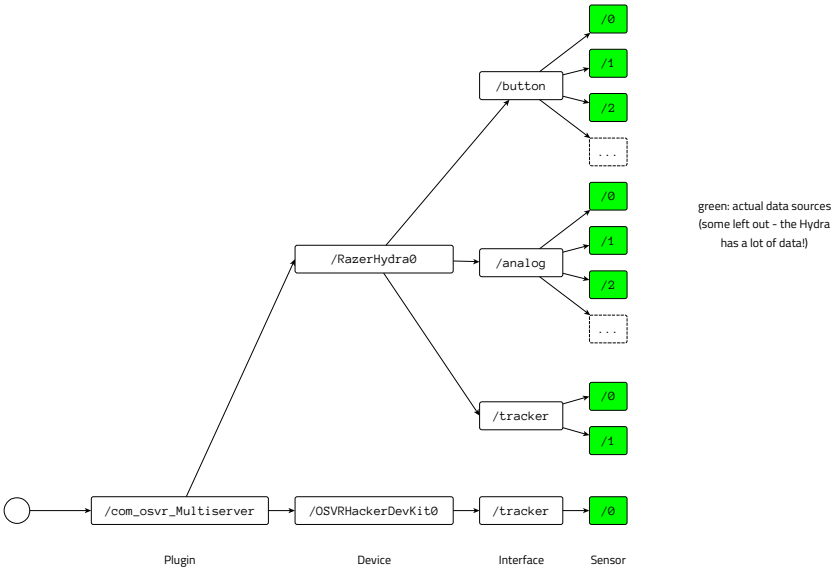


Data from a device driver in a plugin (here, the HDK tracker in one of the core bundled plugins) is made available at a path that generally has four parts: first the plugin, then the device name, then the interface class name, then the sensor ID.

This is a different part of the same path tree the semantic names were in.

OSVR

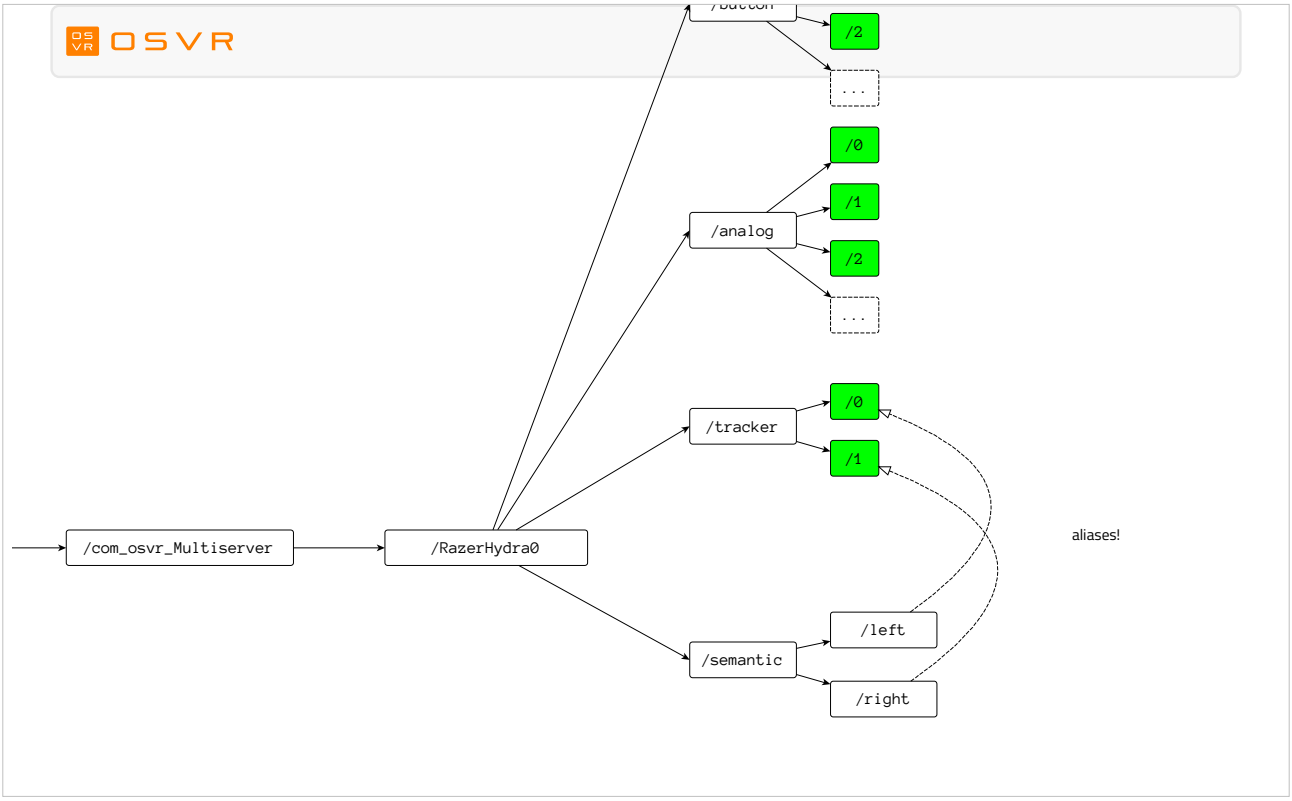
Add another input device



Just like the part of the tree we've already seen, we can have lots of resources. Here, for instance, is an abbreviated version of the tree for an HDK and a Razer Hydra. Notice that the Hydra exposes button, analog (joystick and analog trigger), and tracker interfaces. I've left off some of the sensor nodes: the Hydra has lots more than 3 analog and button sensors.

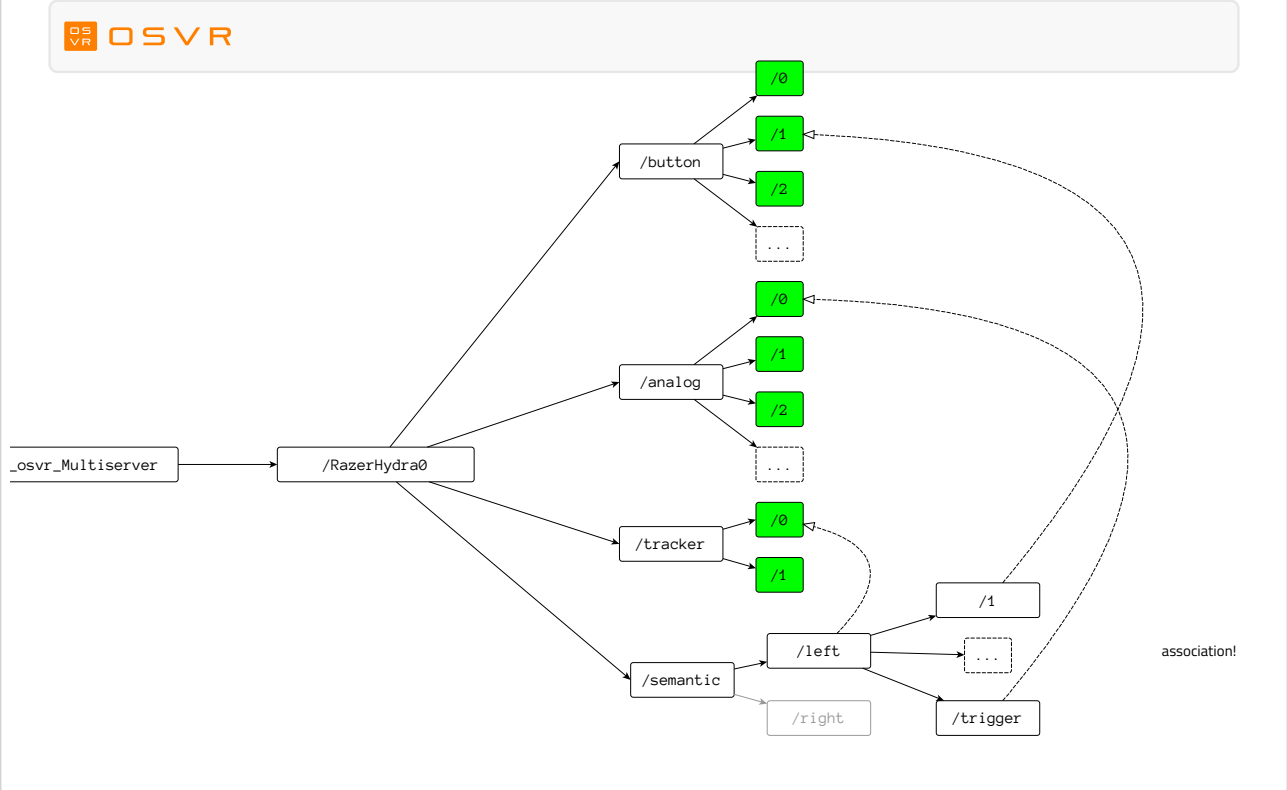
I've highlighted the “sensor” nodes in green – these are where the data is effectively emitted.

So this is a nice tree, but these aren't particularly hardware-independent semantic paths: in fact, they're hardware dependent, and they're just numbered sensors, not very semantic.



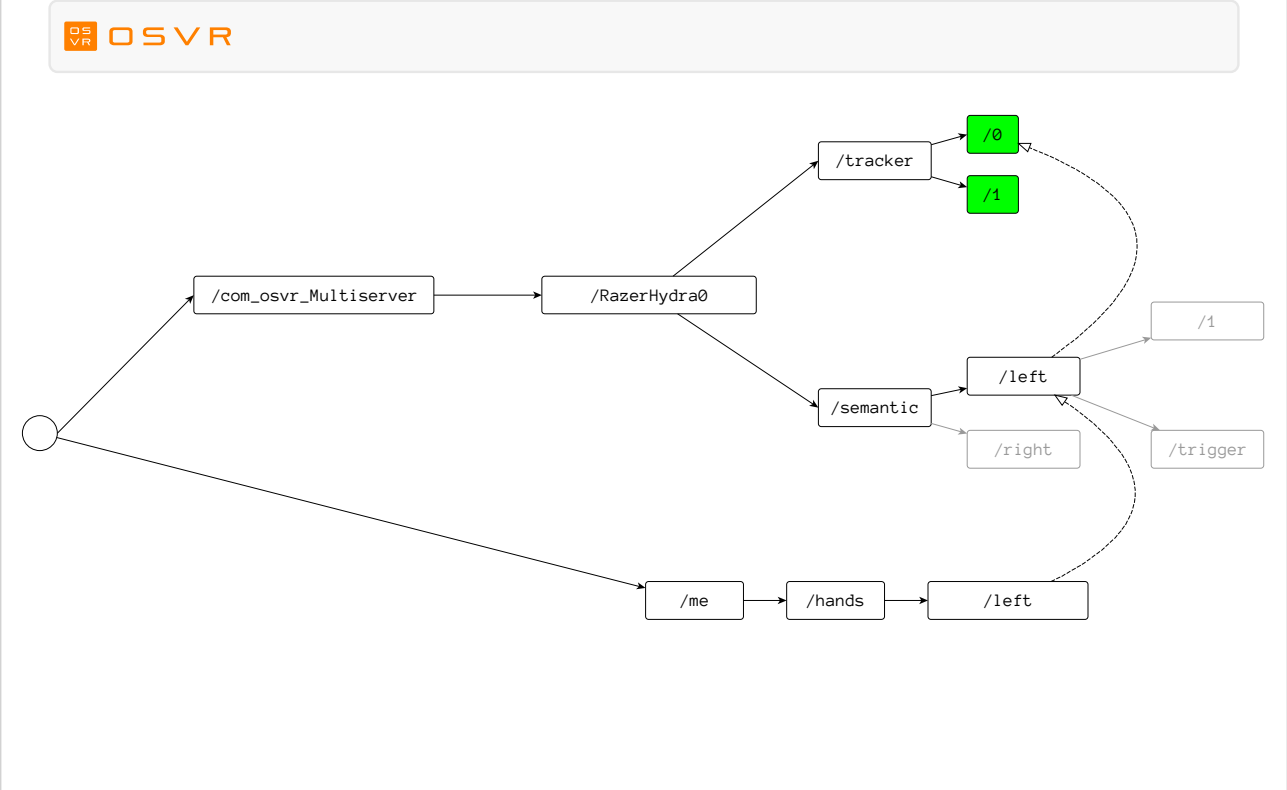
However, the self-description metadata in an OSVR driver is responsible for not only indicating the interface classes provided, but also describing the various sensors and their relationships in a semantic way. This builds a semantic tree of aliases at the device level: hardware-specific semantic names.

You can see here that /com_osvr_Multiserver/RazerHydra0/semantic/left is an alias for …./tracker/0: effectively describing the tracker sensor 0 as being “left”. Similarly, tracker sensor 1 is “right”.



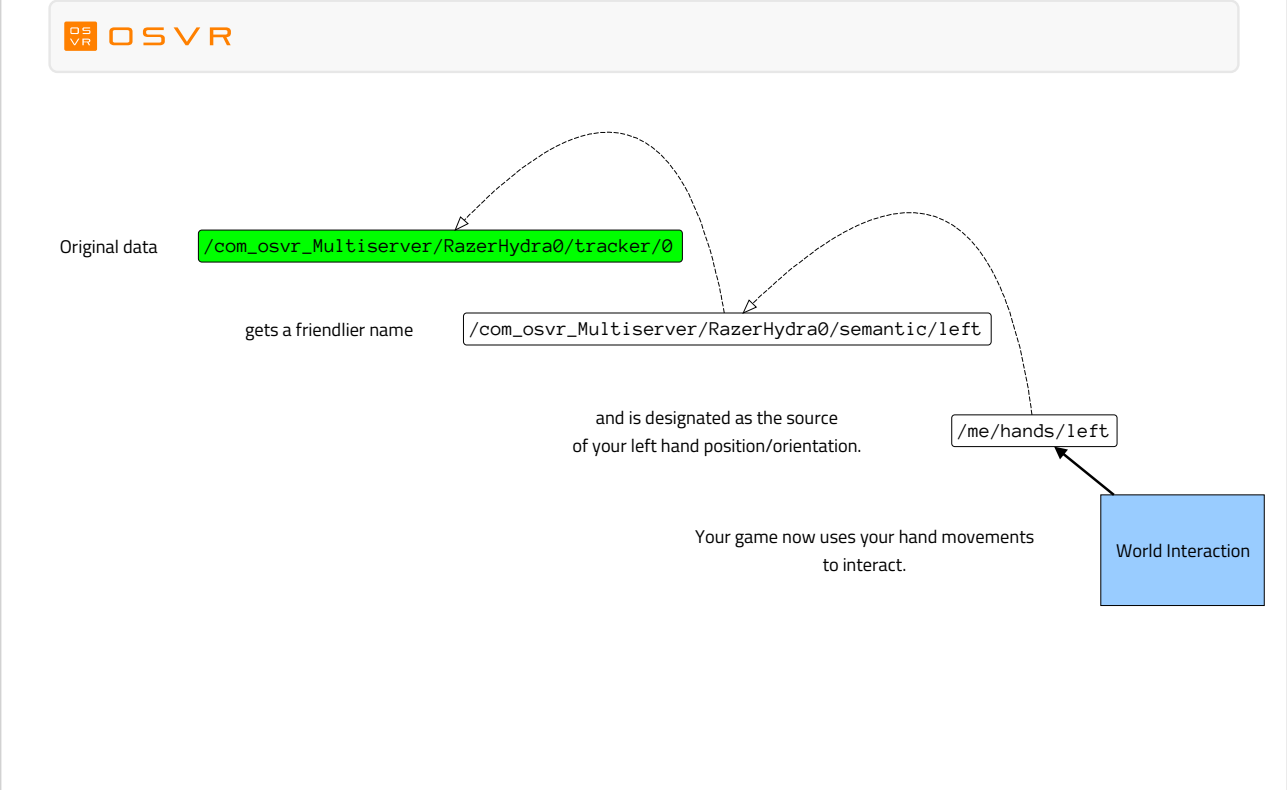
In this case, the left and right Hydra controllers each have a number of buttons and analogs. You now see that how the path tree can describe association: There is a button labeled with the number 1, so ..../semantic/left/1 is that button on the left controller, and ..../semantic/left/trigger is the analog trigger on the left controller.

Of course, many nodes are left out on this diagram for clarity. Those left off include the equivalent aliases that exist under ..../semantic/right: /1 goes to button sensor 8 if I recall correctly, etc.



We started out by talking about an application that wanted to use your left hand to interact with the environment, and at last we're seeing both parts of the tree in the same picture. Here, /me/hands/left is an alias for /com_osvr_Multiserver/RazerHydra0/semantic/left which is an alias for /com_osvr_Multiserver/RazerHydra0/tracker/0. We've now traced back from the semantic name to the underlying hardware, behind the scenes.

It's nice to see the tree here, and see how that semantic name resolves, but we're essentially mixing two data structures in this picture: the tree structure of the names, with the graph structure of the aliases. Let's look at just the aliases for a moment.

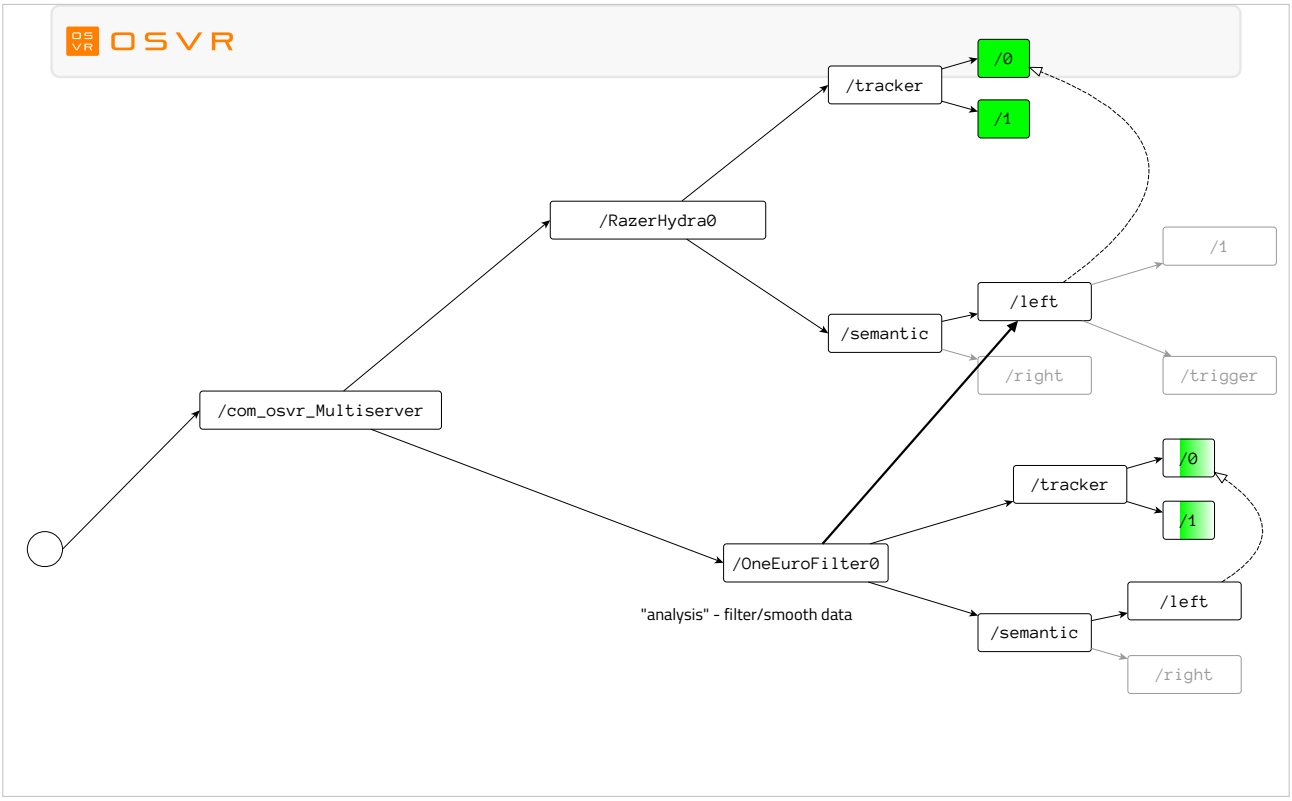


So we start at the source of the data: the hardware sensor and interface, which then gets a semantic name within that device, which is then assigned as the source of the general/generic semantic name for your left hand. The app just asks for the left hand.

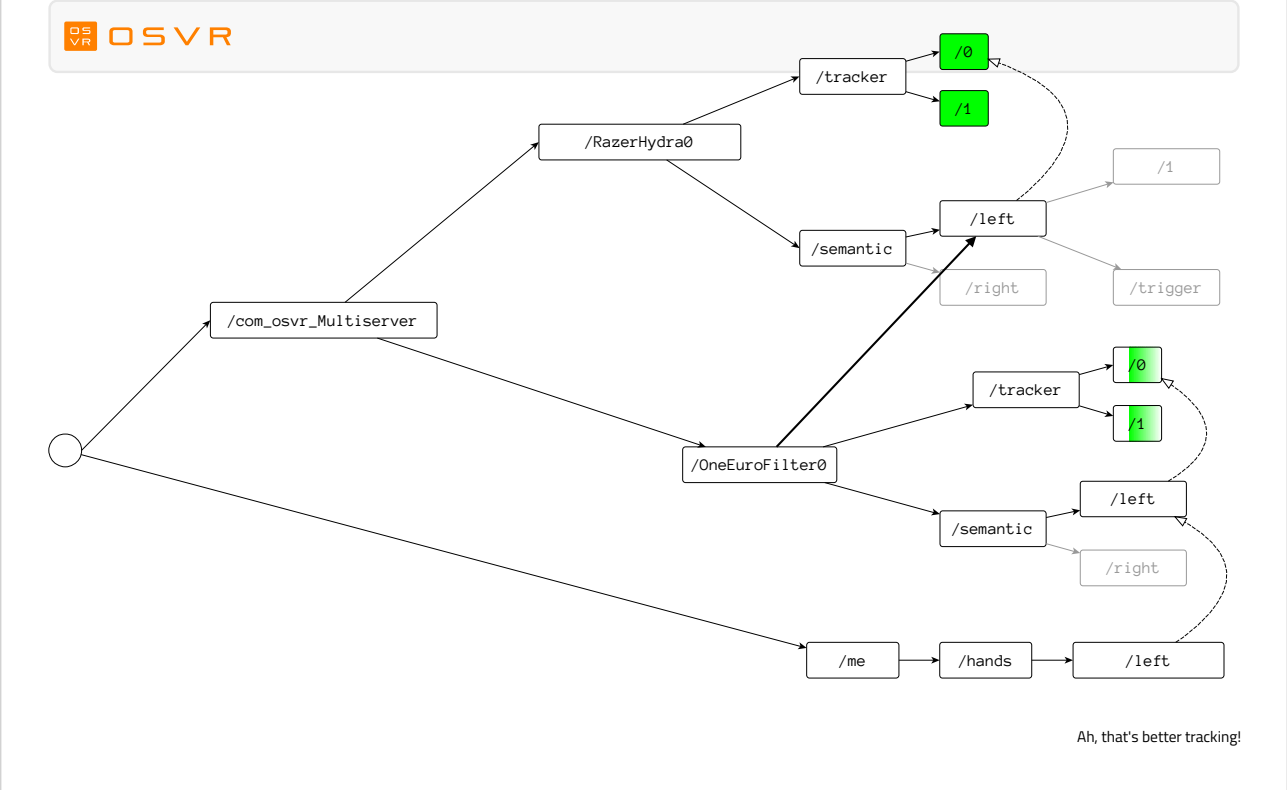
Furthermore, this configuration can actually automatically configure itself: the descriptor for the Hydra has a section that basically says, “if there isn't already an alias for /me/hands/left, put one there that points to the left Hydra controller” - but these aliases could of course be manually configured for more complex VR systems.

Note that the arrows could go either direction: I don't mean to indicate a pull or push data flow, just had to pick a consistent direction for those alias arrows. Application software can choose to receive callbacks or to query state, and both techniques work independent of how the underlying hardware is accessed – the library makes that transparent.

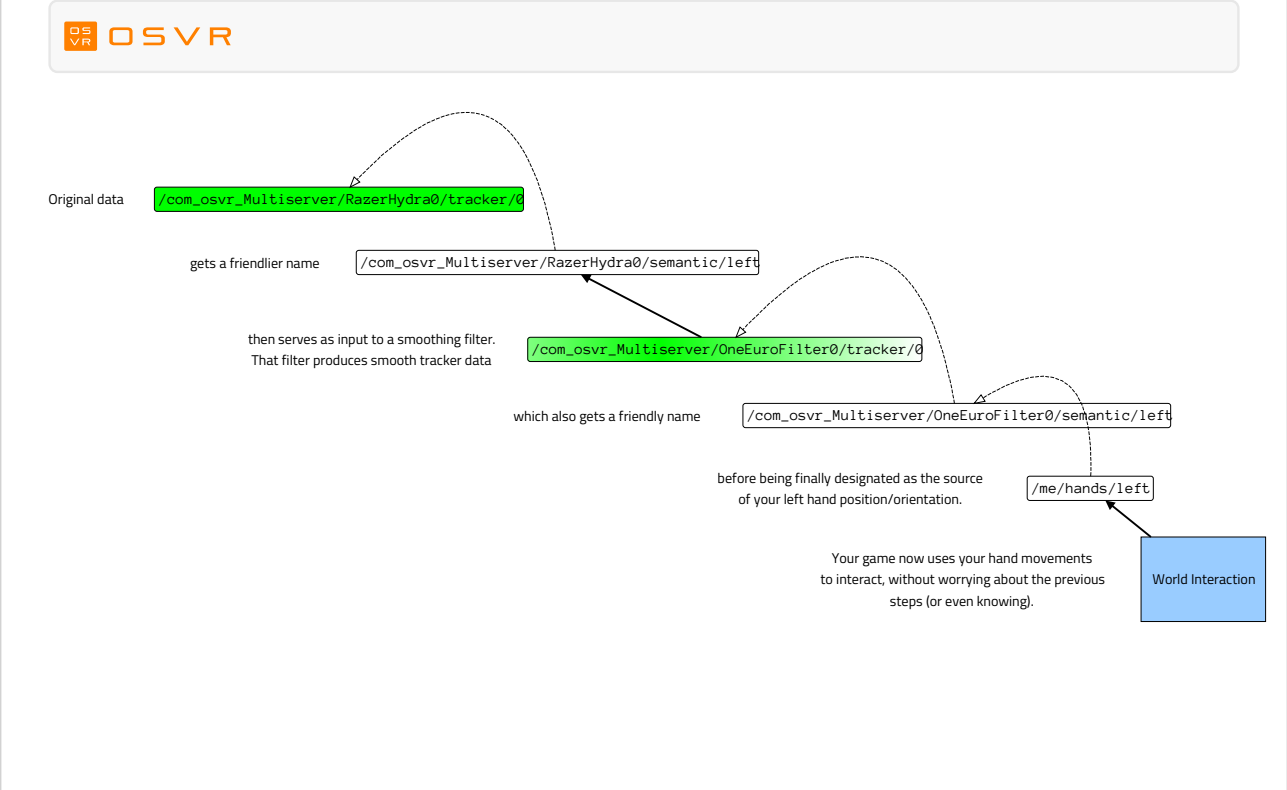
By the way, you can have arbitrarily many aliases between the name the application uses and the hardware, with no performance degradation: at runtime those steps are collapsed down, the data isn't actually taking a “hop” for each alias.



Now let's consider another similar scenario. Back to the path tree, you can see we've added a new node called “OneEuroFilter0”. It's exposing two sensors on a tracker interface, which also have local semantic names, but the data isn't coming directly from a piece of hardware. This is an analysis plugin that contains a smoothing filter for the (noisy) raw tracking data that comes from the Hydra. The filled arrow shows that the One Euro filter is actually acting as a client in a sense, taking in data from the Razer Hydra's trackers, and generating smoothed tracker data from it.

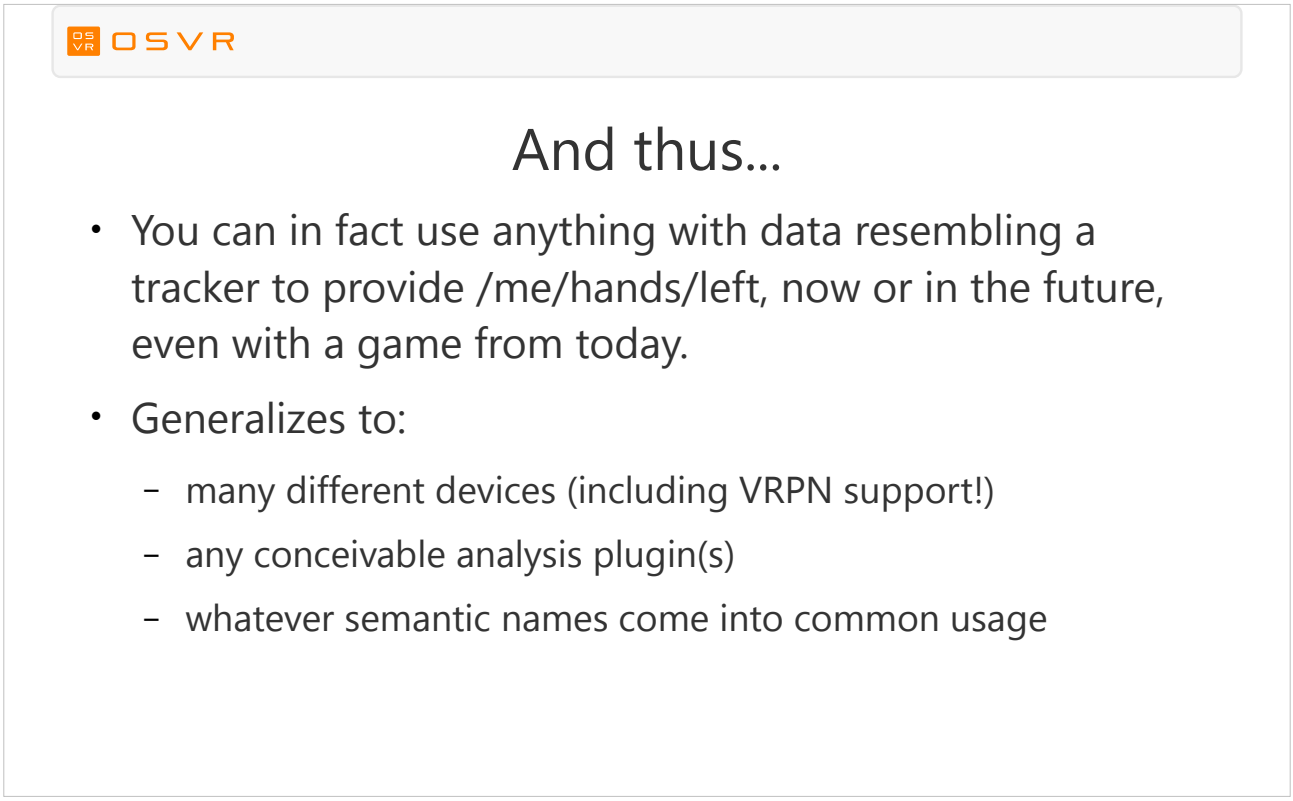


So, if we want smoothed tracking for our left hand, we can now point the alias /me/hands/left to the semantic/left under the OneEuroFilter instead of directly from the Hydra. No changes to the application are needed, despite the fact we just switched out the source of the data from underneath it: as far as the application is concerned, it knows where /me/hands/left is and that's as far as it has to think about.



If we again take away the tree and just look at the aliases and data usage, you can see the extra two nodes between the semantic name and the hardware that provide us with improved tracking data.

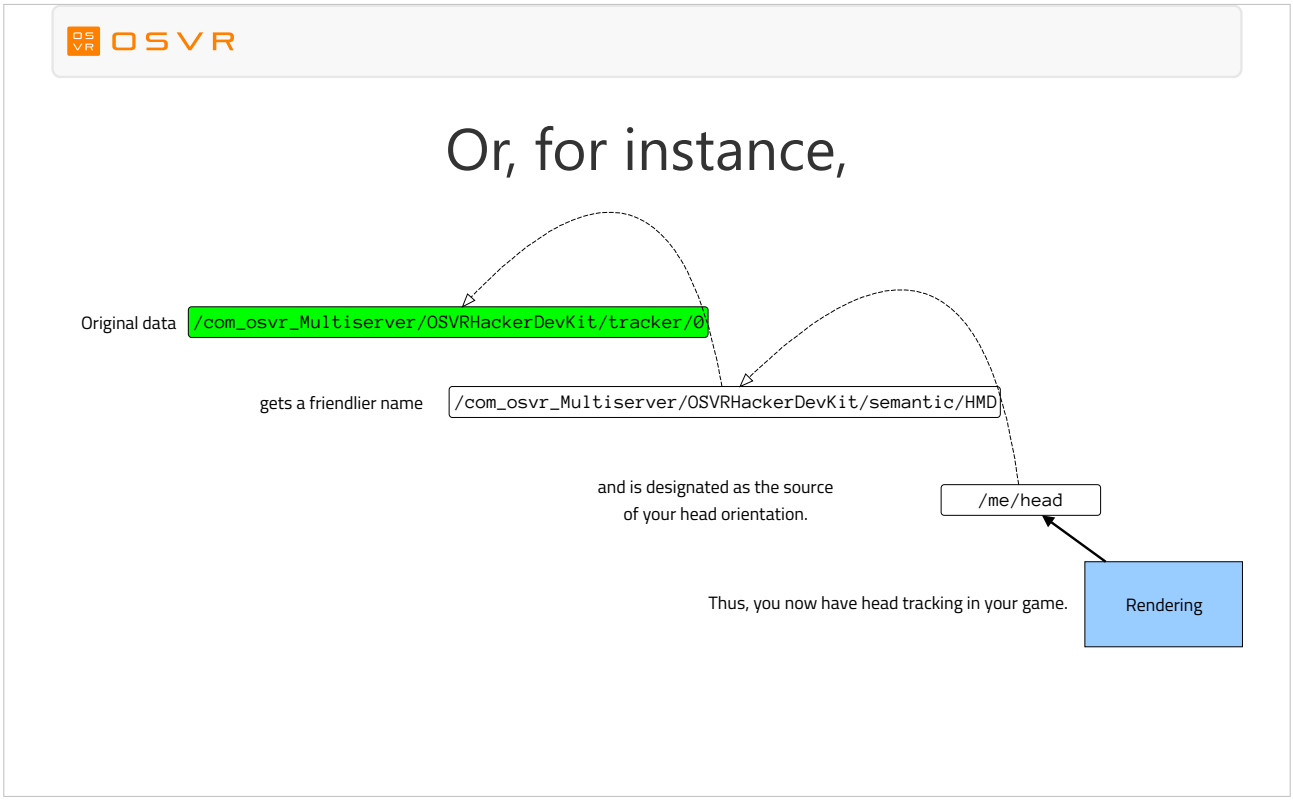
For the sake of completeness, I should point out that because the raw unfiltered tracking data from the Hydra is noisy (which is typical of most tracking technology), we actually set up this configuration, with the filter already inserted, by default. If you wanted to, you could manually add an alias from /me/hands/left directly to the Hydra tracking – that would override the automatic aliases – or point an application directly at the hardware-specific name to get the raw data, but that's not recommended, especially the “hardware-specific name in application” part.



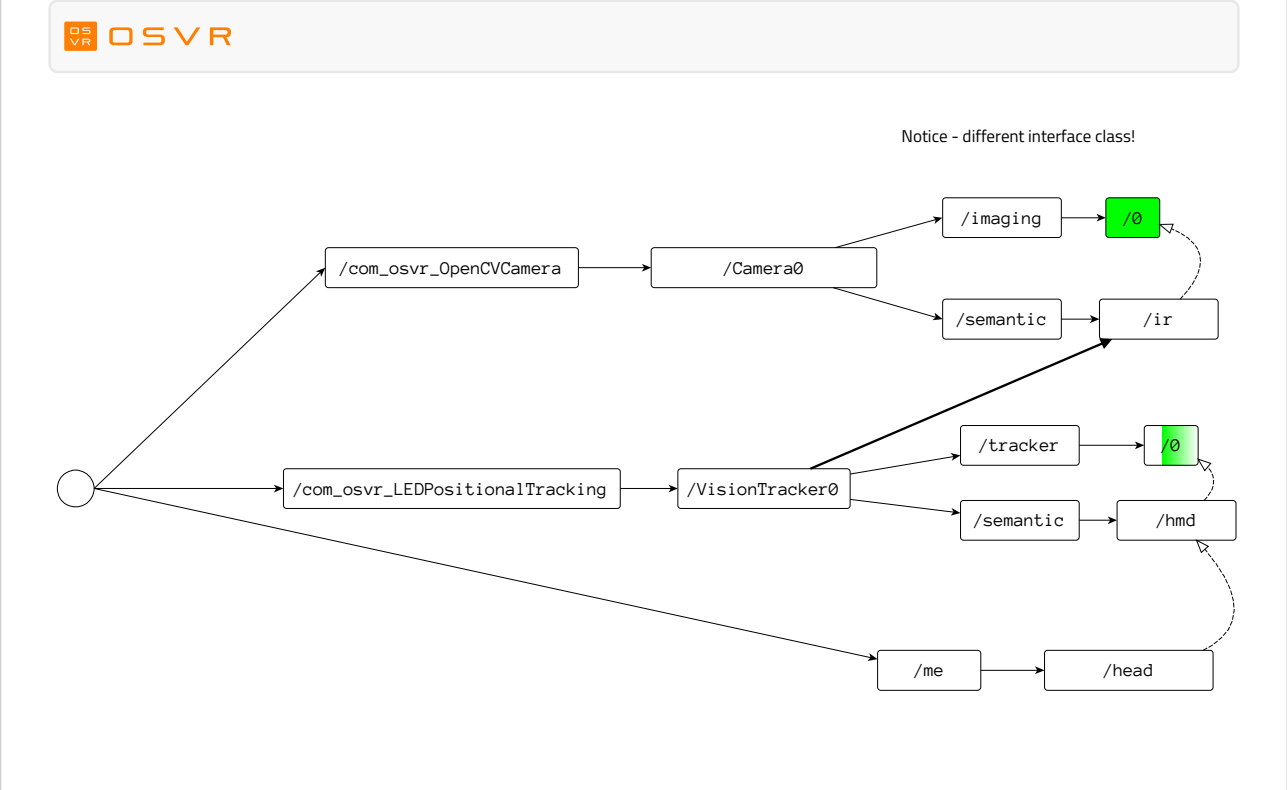
With the basics of the path tree as we've discussed, you can see how an application written today that wants /me/hands/left will work today and into the future, no matter what the source of the data that "resembles tracker data". This is the power of the combination of generic interfaces and semantic names with arbitrary levels of aliases.

While I picked a convenient example that I happen to use a lot, this works with many different devices. There are a growing number with drivers written against the OSVR PluginKit API. The hundreds of devices supported by the existing de-facto standard in commercial and academic VR, VRPN, are also supported: you just need to provide the descriptor that would have come with an OSVR-native plugin. This also generalizes to any analysis plugin you can think of: not limited to a single input to a single output, or presenting the same output interface as it takes in.

And of course, there's more than just /me/hands/left – you can use any arbitrary semantic name, and we encourage you to use very meaningful names to increase the flexibility of what OSVR can do to get you that data. We're working on a suggested set of common semantic names, and you're welcome to help.



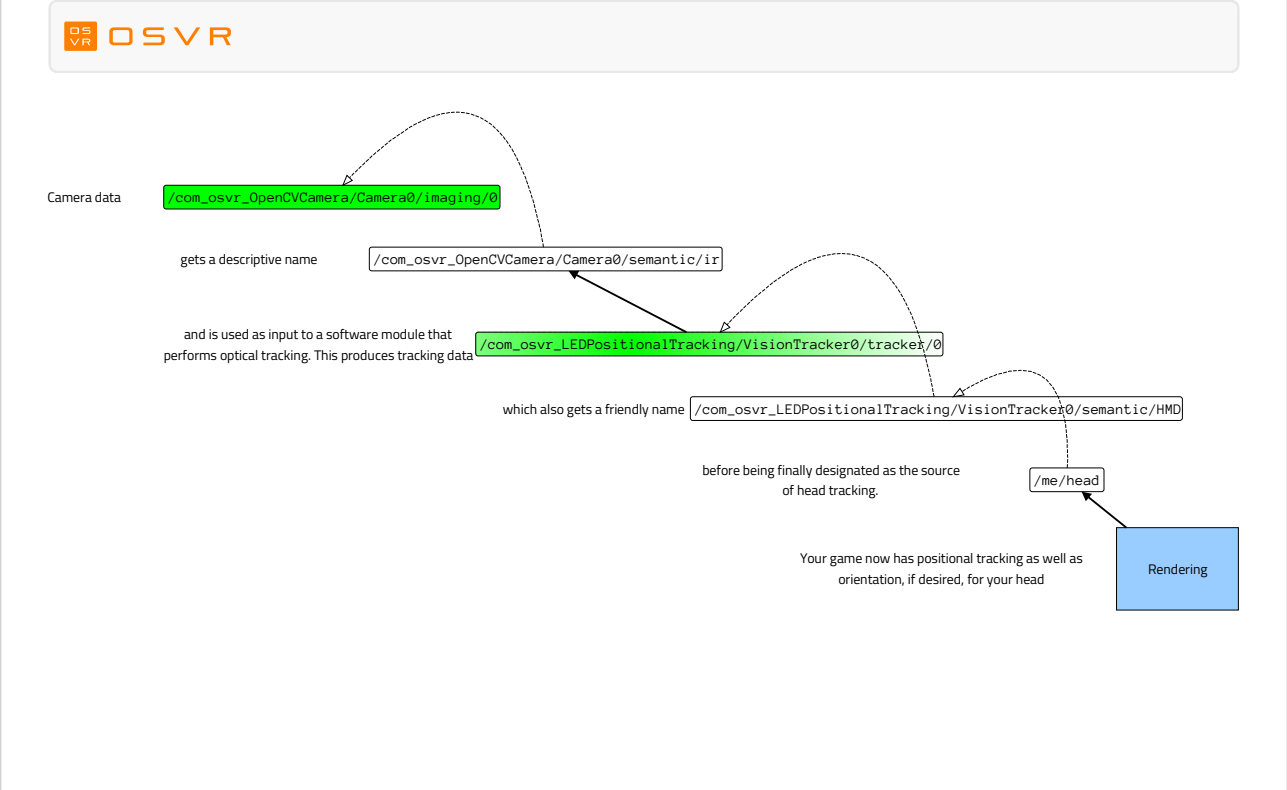
As a final example and a bit of a tease, here's the diagram showing how the rendering engine in your game gets head tracker data from the OSVR HDK's integrated orientation tracker.



But now, let's say there's an IR camera in the mix, and some IR LEDs on the HMD front panel. You can take the IR imaging data from the camera and feed it into an “LEDPositionalTracking” analysis plugin, which exposes a single sensor on a tracker interface.

Notice that here, unlike the One Euro filter, the input and output data of the analysis are different interfaces: we've turned imaging into tracking.

You can then point /me/head at the output of the analysis plugin.



When we look at just the route to the rendering engine, the camera plugin gets data, and gives it a nice name. An algorithm in an analysis plugin uses that imaging data to run a marker-based vision tracking algorithm, and the results are exposed as a sensor on a tracker interface.

The sensor gets a local friendly name, is set as the source of /me/head, and now, the application gets positional tracking as well if desired (if it asks for full pose data or registers a pose callback), instead of just orientation as before. The application doesn't need to know how the position was computed, or who wrote the algorithm – so algorithms can compete and improve – or even that a vision-based tracker was used: the data still has the same semantic name.

OSVR sensics

For more information:

- OSVR developer portal
  - osvr.github.io
- Sensics – Founding contributor to OSVR, experts working in VR/AR for over a decade
  - www.sensics.com