Modular Scene Delegation and Manipulation in Hydra

Problem

Current Hydra architecture employs a scene delegation mechanism to fill a data structure called a “render index” with concrete prim types that represent a flattened version of the source scene. This has worked really well for the purposes of fast preview renderers, but has fallen short when we tried to push it to full final frame renderers. Primarily the issue is that the flattened representation of the scene is not as rich as one might want to present to a full-featured renderer backend. The prim hierarchy is flattened and the data we can pass through is constrained by the scene delegate API, which makes the current system hard to extend. It also places undue burden on general purpose scene delegates to be aware of the unique needs of specific renderers and production pipelines.

Goal

In addressing the limitations of Hydra’s current architecture, our main goal is to allow renderers access to as much of the scene data as possible in as high a fidelity as possible. By allowing more access, we address the current limitations imposed on what can be accessed from the scene (for example, a renderer can’t currently ask for scene data that is not already represented through the scene delegate’s object model). By allowing high fidelity data, we mean that renderers can get at the data from the scene without loss of information (e.g., through the prim hierarchy flattening process that Hydra currently performs).

We’d also like a place in the new system to perform scene transformations currently performed in the scene delegate, like transform concatenation and provide a more structured way in which such transformations can be modularized and configured per renderer.

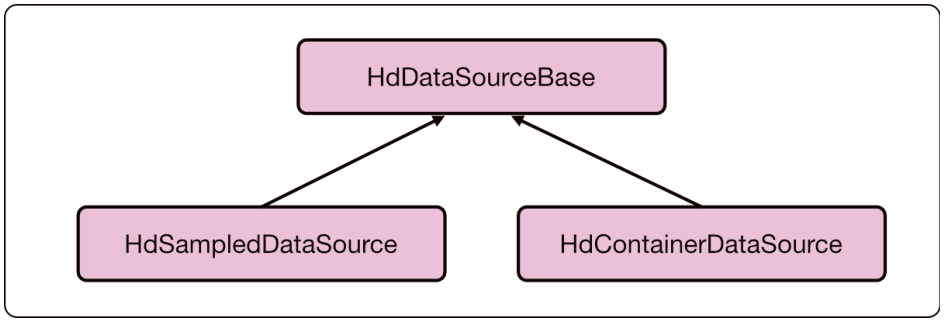
With this system it’s also important to preserve the features of the scene delegate API that made it fast. In particular, the scene delegate API lets scenegraphs decide how to implement **`Get()`** functions for best performance, which enables things like on-demand loading of USD data. We want to keep using zero-copy value types to transport data. We also need to preserve the ability to narrowly invalidate attributes in the renderer, which is currently done via dirty bits in the change tracker.

Proposal

We propose a new mechanism that represents the scene to the renderer backend as a full scene hierarchy, each node of which is capable of indirecting its requests to the delegated source scene. Our proposed system is also flexible and it allows custom data to be transported easily.

We introduce two new concepts: data sources and scene indices. In the following sections we discuss in detail these concepts, but from a high level a data source provides the data indirection, and a scene index provides the ability to manipulate scene data. Finally, we discuss how schemas can be used to give more meaning to data sources, how dependencies can be used across data sources, and how to locate data sources.

HdDataSource Class Hierarchy



The root of this hierarchy is a class called **`HdDataSourceBase`**, an empty base class. Then we have some more interesting classes:

- **`HdContainerDataSource`**: This is the base class representing nodes that contain other data sources. Examples of containers are “prims” and “namespaces” (e.g., groupings of attributes that are somehow related). It is a pure virtual base class with the following API:
    - **`bool Has(const TfToken &name)`**: returns whether or not something called “name” is contained in it.
    - **`TfTokenVector GetNames()`**: returns all the names that are contained in it.
    - **`HdDataSourceBase Get(const TfToken &name)`**: returns a data source for “name”.
- **`HdSampledDataSource`**: This is the base class for sampled values. Where **`HdContainerDataSource`** might represent prims, this class would represent attribute values, with the ability to supply multi-sampled data. It is a pure virtual base class with the following API:
    - **`VtValue GetValue(Time shutterOffset)`**: returns the value of this source at a given time offset relative to the current global time.

Scene Data Sources

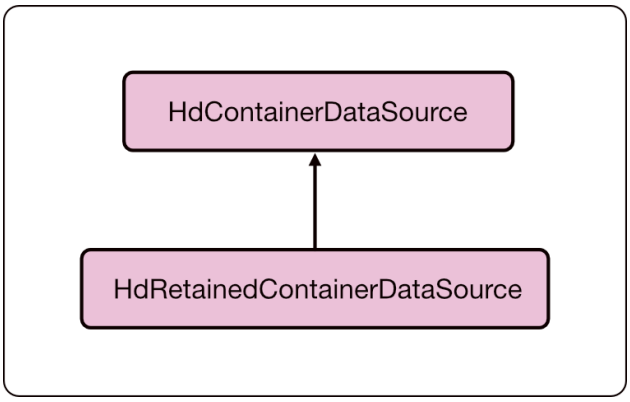
With these two classes in place, you can now imagine adapting arbitrary scenes. You can derive the data sources for your scene and start fulfilling the required APIs by delegating to your scene.

A concrete example might be a USD-backed data source. In that case we could, for every prim, create an **`HdContainerDataSource`**-derived object that indirects the question of what "names" it contains to the underlying **`UsdPrim`** fully lazily. For an attribute, like color, we could create an **`HdSampledDataSource`**-derived object that lazily indirects the **`GetValue`** call to the underlying **`UsdAttribute`**. For a **`UsdRelationship`** to a prim, we could use an **`SdfPath`**-valued data source or a data source holding an array of **`SdfPath`**s. For more discussion about addressing objects in Hydra, see the Locating Data Sources section later in this document

Retained Data Sources

There is another approach that this system is designed for, which we call "retained" data sources. With retained data sources, the answers can be hard-coded into the objects and returned immediately, without any required scene backing. This would allow you to build or extend scenes procedurally. Data sources from a variety of scenes could be spliced together, with all of it abstracted from the core of Hydra.

For example, a **`HdRetainedContainerDataSource`** derives from **`HdContainerDataSource`**:



Ownership Model

Currently we're experimenting with a shared ownership model of data source objects for expediency and convenience. We're evaluating the performance costs of such a model. The specific form it will take in the end is undecided.

Hydra Schemas

The **`HdDataSource`** API is very flexible, but unstructured by design. Reading the scene may include lots of code that looks like this:

```
dataSource->Get("someToken")
dataSource->Get("thatToken")
```

To add some structure back, we introduce classes called Hydra Schemas that formalize what fields and types we expect on certain classes of objects. Unlike USD schemas, these are not code generated, they are simply hand-written data structures that are there for convenience and safety.

For example you could construct a schema like this:

```
HdContainerDataSource topoDataSource = primDataSource->Get(“meshTopology”);
HdMeshTopologySchema topoSchema(topoDataSource);
```

Afterwards, instead of:

```
topoDataSource->Get("faceVertexCounts")
```

You can use an `HdMeshTopologySchema` to do something like:

```
topoSchema->GetFaceVertexCounts()
```

The `HdMeshTopologySchema` class would be defined like so:

```
class HdMeshTopologySchema : public HdSchema
{
  public:
     HdMeshTopologySchema(HdContainerDataSource container);

     HdIntArrayDataSource GetFaceVertexCounts();
     HdIntArrayDataSource GetFaceVertexIndices();
     HdIntArrayDataSource GetHoleIndices();
     ...
};
```

A nice distinction is that the schema could do the appropriate type-checking for you and other common validation before returning you a data source that contains the value you're looking for (so you still need to call GetValue() on the result in both cases).

Representing Dependencies

One of the fundamental requirements in a scene is the ability to express relationships. In the renderer, for example, these turn into invalidation dependencies. In today's Hydra parlance, a dependency would help answer the question: when this prim's attribute x changes, what other Hydra prims and dirty bits need to be set? Current Hydra architecture doesn't have a great way of representing these dependencies.

Here we show how such a dependency system can be built with this new architecture, as an example of the system's expressiveness. Rather than implementing support for a fixed set of dependencies known to the system, we can declare and track dependencies in an extensible manner via data sources within the scene itself. We propose expressing such dependencies as **`HdDataSource`**s with the following named-fields:

```
{
  dependedOnPrimPath: SdfPath
  dependedOnDataSourceName: HdDataSourceLocator
  affectedDataSourceName: HdDataSourceLocator
}
```

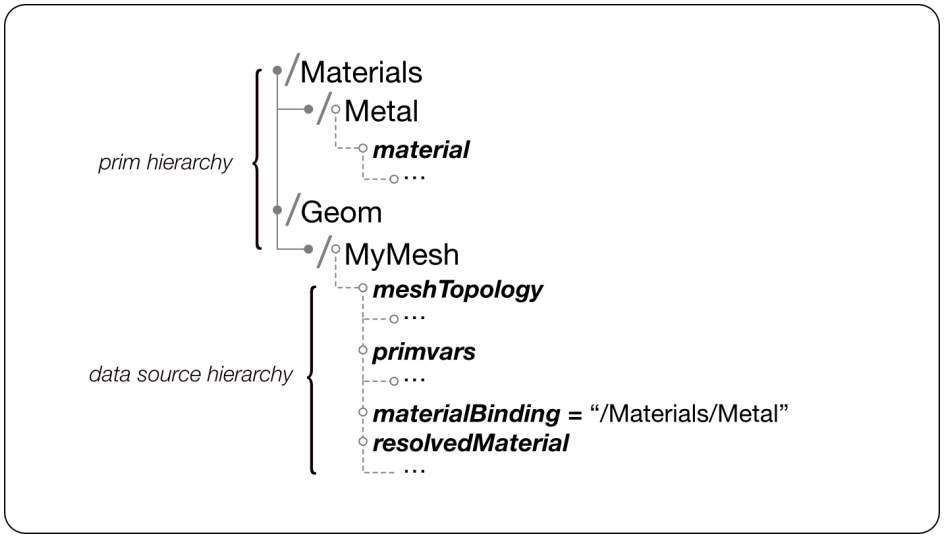
This data source can be placed in an `HdContainerDataSource`, which could live at the prim level as `'__dependencies'`. Let's call the prim owning `'__dependencies'` *this*. With that in mind, let's define the above fields:

- **`dependedOnPrimPath`**: The path to the prim on which *this* depends.
- **`dependedOnDataSourceName`**: The data source name on `dependedOnPath` on which *this* depends.
- **`affectedDataSourceName`**: The data source on *this* that depends on `dependedOnDataSourceName` and `dependedOnPrimPath`.

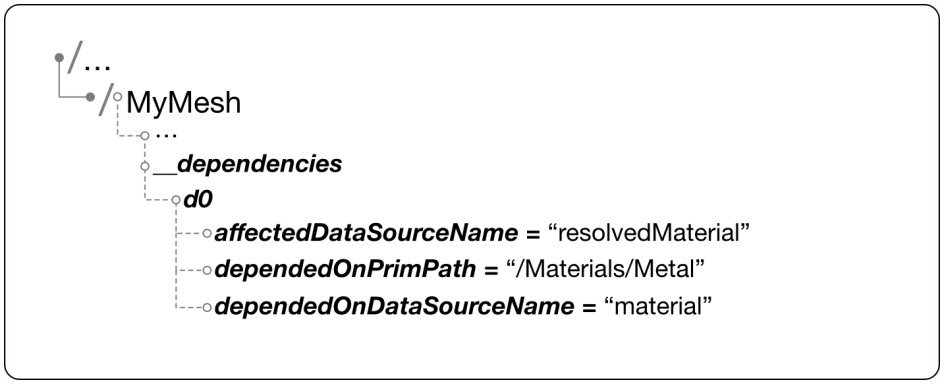
Dependency Example: Material Bindings

Let's examine the common (but non-trivial) case of tracking changes relating to material bindings.

Consider a render delegate that consumes the material on a gprim using a **“`resolvedMaterial`”** data source. Such a data source could depend on the **“`material`”** data source of the bound material, represented by the **“`materialBinding`”** data source.

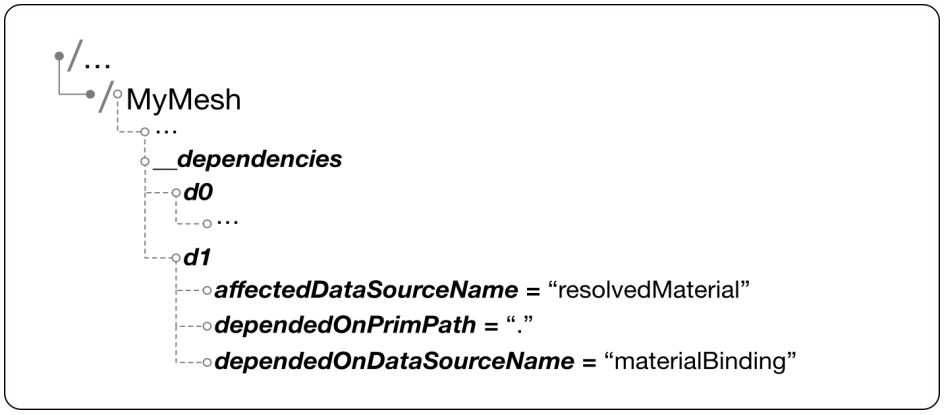


In the above scene, the dependency between the **“`resolvedMaterial`”** data source and the material the geometry is bound to would be expressed on **`/Geom/MyMesh`** in this form (as defined and interpreted by the dependency schema):



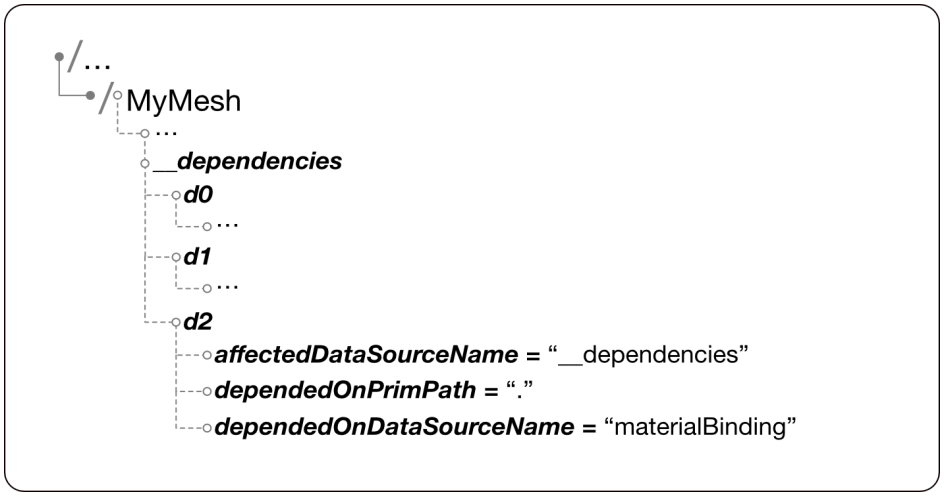
So with this alone, dirtying of **`/Materials/Metal "material"`** data source will result in **`/Geom/MyMesh's "resolvedMaterial"`** being dirtied. But changes to the mesh's **`"materialBinding"`** data source can also affect the value of the mesh's **`"resolvedMaterial"`**.

So we can declare a second dependency to track that:



So now when MyMesh's own **`"materialBinding"`** is dirtied, its **`"resolvedMaterial"`** data source value is also flagged as dirty.

Finally, note that when **`"materialBinding"`** changes, we should recompute our dependencies. Since the dependencies are expressed as data sources, we can do that simply by declaring a third dependency as such:

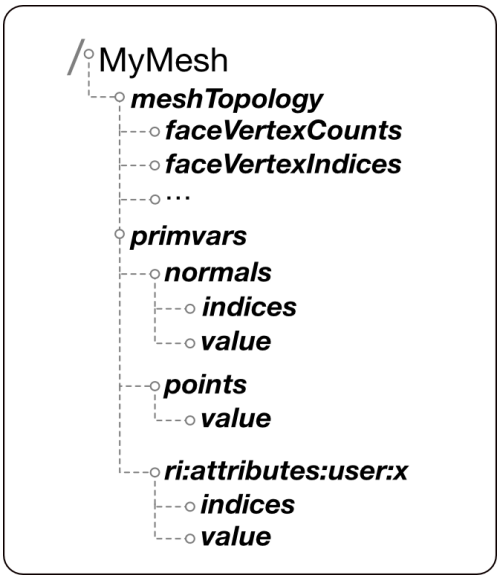


This indicates that the **`"__dependencies"`** data source itself will be dirtied when MyMesh's own **`"materialBinding"`** data source value is changed. The code (or data) backing the **`"__dependencies"`** data source has the specific knowledge to describe the updated dependencies reflecting the **`"materialBinding"`** value.

A general purpose "dependency dirtiness propagating" scene index can ensure that expected notices are sent without specific knowledge of the meaning or usage of the involved data sources. This also means that additional dependencies can be described and tracked without changes to the low-level system.

Locating Data Sources

One of the issues we will have to address is the need to uniquely identify a data source. In USD for example, properties can be uniquely identified with an **`SdfPath`**. Here we explore what the requirements are for these kinds of paths. Consider the following container data source that represents a mesh prim:



Looking specifically at an **`SdfPath`** to the USD scene attribute **`/Geom/MyMesh.primvars:ri:attributes:user:x`** (please ignore whether or not this can actually be an indexed primvar, just imagine that it is) -- we need a way to identify it relative to its prim in a fast way (much like **`SdfPath`** provides for prim-level hierarchy navigation). In other words, we need to represent {**`“primvars”, “ri:attributes:user:x”, “value”`**}.

We’ve considered several possible implementations and we’re currently favoring an implementation where a final leaf “attribute” in Hydra is referenced with both an **`SdfPath`** and an **`HdDataSourceLocator`**. The **`SdfPath`** is only used to uniquely address down to the “prim” level, and the **`HdDataSourceLocator`** is internally represented by a vector of tokens that will take you the rest of the way. In the Dependency Example: Material Bindings example, **`“materialBinding”`** is represented as an **`SdfPath`** data source, since it points to a material prim. In the **`/MyMesh`** example, just above, addressing the indices of the **`normals`** primvar of a specific prim would require adding an **`HdDataSourceLocator`** to the schema along with an **`SdfPath`** to represent the prim.

Tying the Scene Together and Filtering it

We now need to tie all the above concepts together. To do this we propose a new class, **`HdSceneIndexBase`**, which will be the hub for dealing with the scene as a whole, and the concept of an **`HdSceneIndexObserver`**, which is any entity that can monitor changes to the scene index.

HdSceneIndexBase

The fundamental API for a scene index is very simple:

- `HdSceneIndexPrim` **`GetPrim`**`(const SdfPath &primPath)`:
    - This simply returns an HdSceneIndexPrim at the given path. We haven't covered `HdSceneIndexPrim` yet, but basically it is just a container data source that has a type associated with it.
- `TfTokenVector` **`GetChildPrimNames`**`(const SdfPath &primPath)`:
    - Returns all the child names for `primPath`. This can be used to traverse the whole scene.
- `HdDataSource` **`GetDataSource`**`(const SdfPath &primPath,`
  `HdDataSourceLocator &locator):`
    - Retrieves the data source at the given locator for the given prim.
    - This is really just a convenience API and can be implemented in terms of `GetPrim`.

This API exists on a base class **`HdSceneIndexBase`**. In order to understand why you might want to derive different kinds of **`HdSceneIndexBase`**s, we need to examine the idea of a scene index observer.

HdSceneIndexObserver

A scene index observer is an entity that is able to monitor changes to the observed scene index. An observer needs to satisfy the following API:

- `void` **`PrimAdded`**`(const HdSceneIndexBase &sender,`
  `const SdfPath &primPath,`
  `const TfToken &primType)`
- `void` **`PrimRemoved`**`(const HdSceneIndexBase &sender,`
  `const SdfPath &primPath)`
- `void` **`PrimDirtied`**`(const HdSceneIndexBase &sender,`
  `const SdfPath &primPath,`
  `const HdDataSourceLocator &sourceLocator)`

(Note that the actual calls in the implementation will all be vectorized versions; these are shown here for simplicity. The final implementation will consider several different possibilities for vectorization, but the message vocabulary will remain.)

`PrimAdded` is called when a prim is either added or needs to be fully invalidated.

`PrimRemoved` is called when a prim is removed, which also includes all descendants.

`PrimDirtied` is called when a particular locator needs to be updated. This allows for very fine granularity invalidation.

As you can see these methods are all about getting notified when changes happen to the scene. In the Representing Dependencies section, we hinted at how a scene index can be used to send these notices to dependents as well; we'll now see how. To start thinking about this let's consider some uses cases:

Render Delegate with Observer

If a renderer wants to subscribe to scene changes, it can use an observer. As the observer sends the render delegate scene updates, the render delegate can update rendering data.

Debugging Tool with Observer

Similarly, an observer and a given scene index can be used by a UI widget to display and introspect the renderable scene any changes to it.

HdSceneIndex with Observer

Here is where the power really starts to shine. Now if you consider adding your own scene index with an observer to another scene index, the observing scene index can now provide filtering capabilities to the original scene index coming from the scene delegate.

A filtering scene index might perform transformation concatenation down to leaf prims in order to allow for render delegates to opt-in to transform-flattening behavior when desired. This could be accomplished by replacing each prim's **"xform"** datasource with one computed from concatenation, and correspondingly forwarding **PrimDirtied** notices down-namespace. Such modularization also removes the burden of implementing that functionality within each individual delegating scene index.

Similarly, another powerful use alluded to earlier is the general purpose dependency forwarding scene index. Such a scene index would use an observer to observe an input scene and then forward dependency-based notices downstream based on the tracking and interpretation of the **"__dependencies"** data sources.

Other potential uses include lightweight merging of multiple scene inputs, hosting of procedural generation systems, and tessellation of analytic geometry primitives to meshes.

Roll-Out and Backwards Compatibility

In order to minimize disruption in rolling out the changes described in this document, we plan on providing two "shim" or "emulation" layers: one on the front end and one on the backend, to allow existing clients using the existing scene delegate and render delegate APIs until an explicit effort is made by the client to move to the new APIs. **The goal here is to allow scene delegates and render delegates to modernize their usage of the APIs independently.**

The frontend and the backend emulation layers will have the same limited expressiveness as current Hydra, but they would allow us to roll out the new technology in a phased way, validating the approach every step of the way and constantly testing for performance.

Frontend Legacy Emulation

This layer allows for client applications to populate prims using a render index and scene delegates without modifying the existing code. Hydra will internally create a scene index for these prims and transform them into a data source representation. This enables both a scene delegate that wants to use the old API and a render delegate that wants to directly consume data sources to coexist. Furthermore, this enables scenes from legacy scene delegates to coexist with scenes that populate scene indices to produce one renderable scene.

Backend Legacy Emulation

This layer allows for client render delegates to keep using the same APIs, which use a scene delegate to read rendering data. Hydra will create a scene delegate that reads from the data source representation, and pass that into the render delegate. This enables a scene coming from data sources and a render delegate using the old API to coexist.