PIXAR
SOFTWARE R&D

USD Composition

Sunya Boonyatera

Composition

- Overview
- Composition Behavior
- Pcp
- Inspecting Composition Structure

Hi everyone, my name is Sunya and I’m going to be discussing USD’s composition system.

I’m going to be giving a brief overview of the system, then we’re going to discuss the composition behaviors in more detailed. We’ll follow that up with some code-level details about the composition engine and end with some methods for introspecting composition structure.

Composition

- Combine scene description from multiple sources into a single “composed” scenegraph
- Non-destructive overrides, multi-user workflows

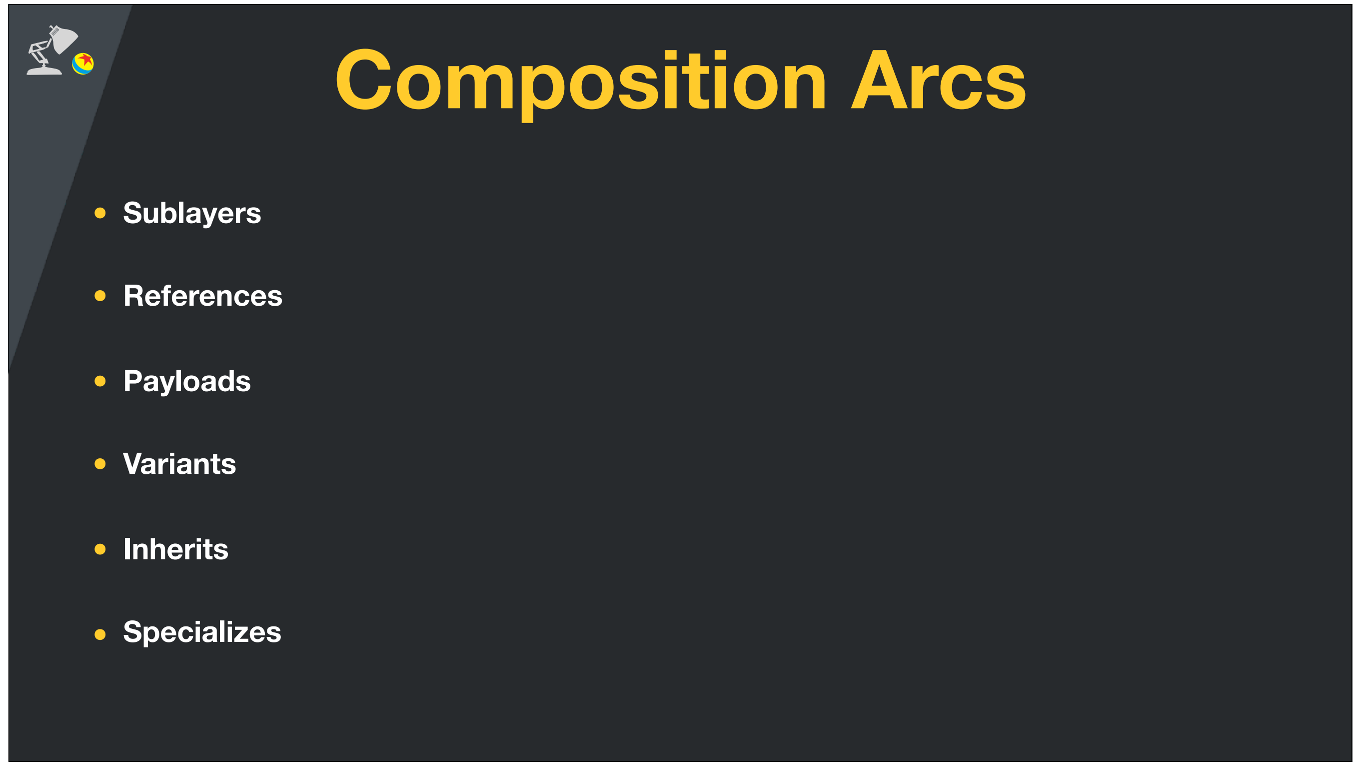
USD’s composition system has a fairly long and rich history. It was originally part of Pixar’s proprietary “Presto” animation software, which was first used on the movie “Brave” released in 2012. It’s been under constant development and improvement and today continues to serve as the backbone of both Presto and USD.

Composition lets users combine scene description from multiple sources (layers) into a single “composed” view. This is extremely useful for content creation pipelines for a number of reasons. Maybe the most important reason is that it allows multiple artists to be working on an asset at the same time.

Composition lets you easily build up large assets from smaller ones. It also gives you the ability to perform non-destructive overrides, so you can change how an asset might look without actually changing the data in the asset itself.

Composition

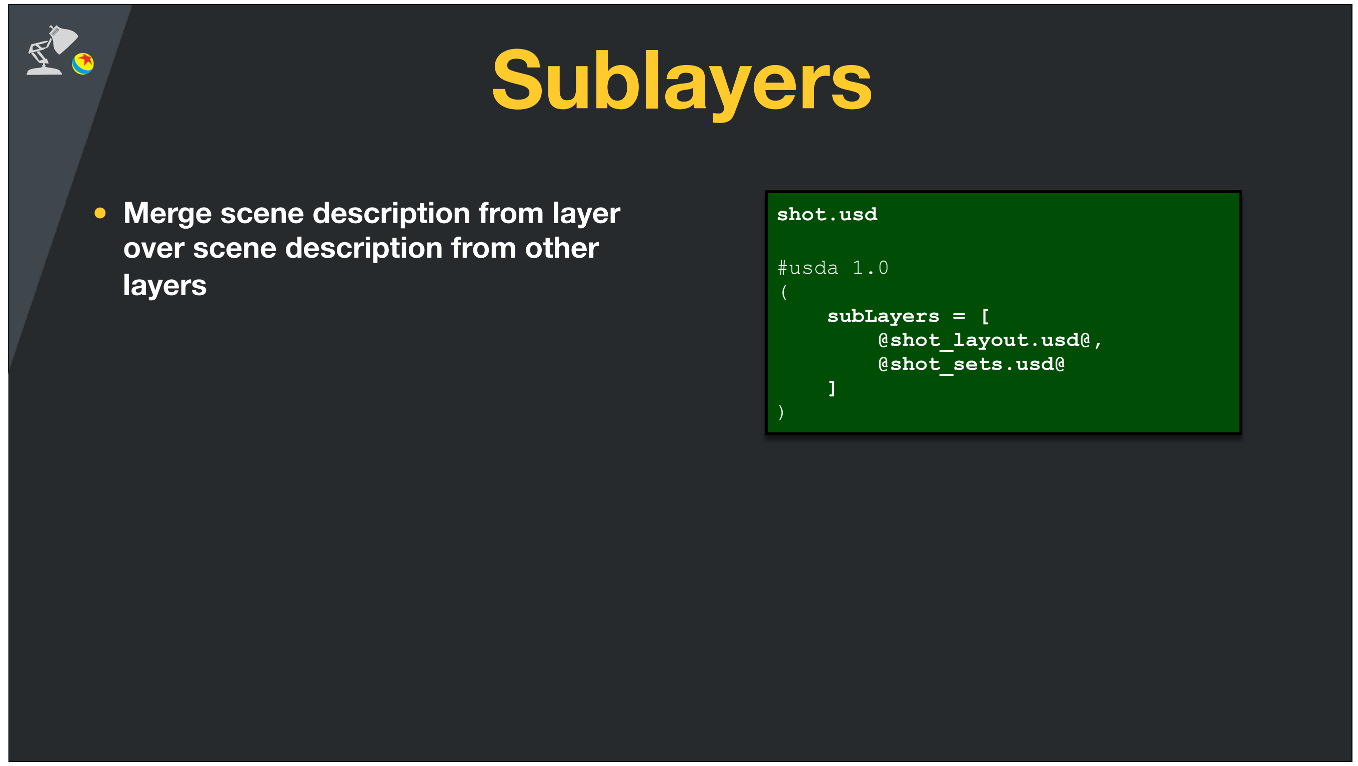
For example, the Antiques Mall set from Toy Story 4 uses the “reference” composition operator (among others) to build up a hugely complex environment from many smaller models.



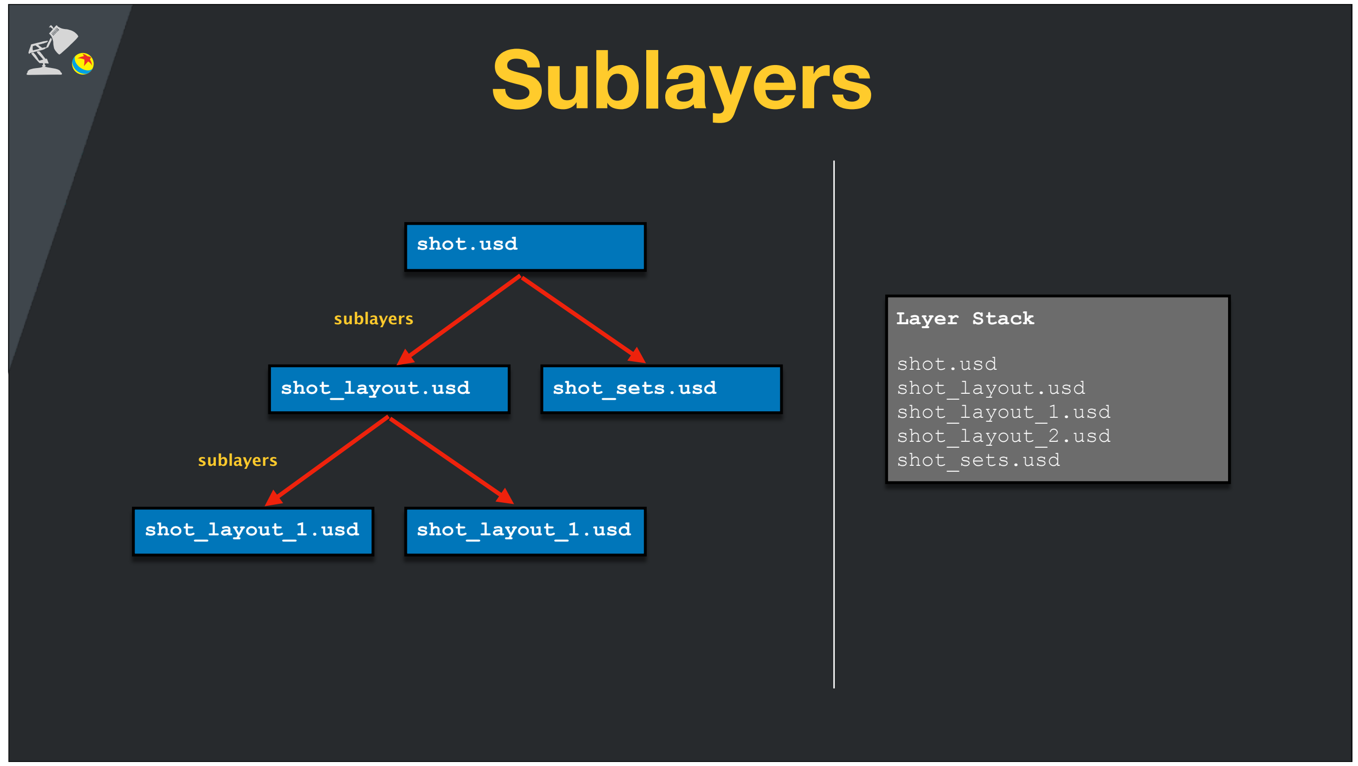
Composition is driven by set of operators (or “arcs”) that are written into scene description. At runtime, the composition engine evaluates these operators and presents the final composed scenegraph to the user.

This is a list of the available composition arcs. I’m going to give an overview of each one and show examples of how they might be used to construct a (very, very) simplified shot featuring DukeCaboom and the Antiques Mall set from Toy Story 4.

These are going to be fairly high level so we can focus on understanding the behavior of each arc. I highly encourage you to see the documentation on our website for more detailed explanations and examples.

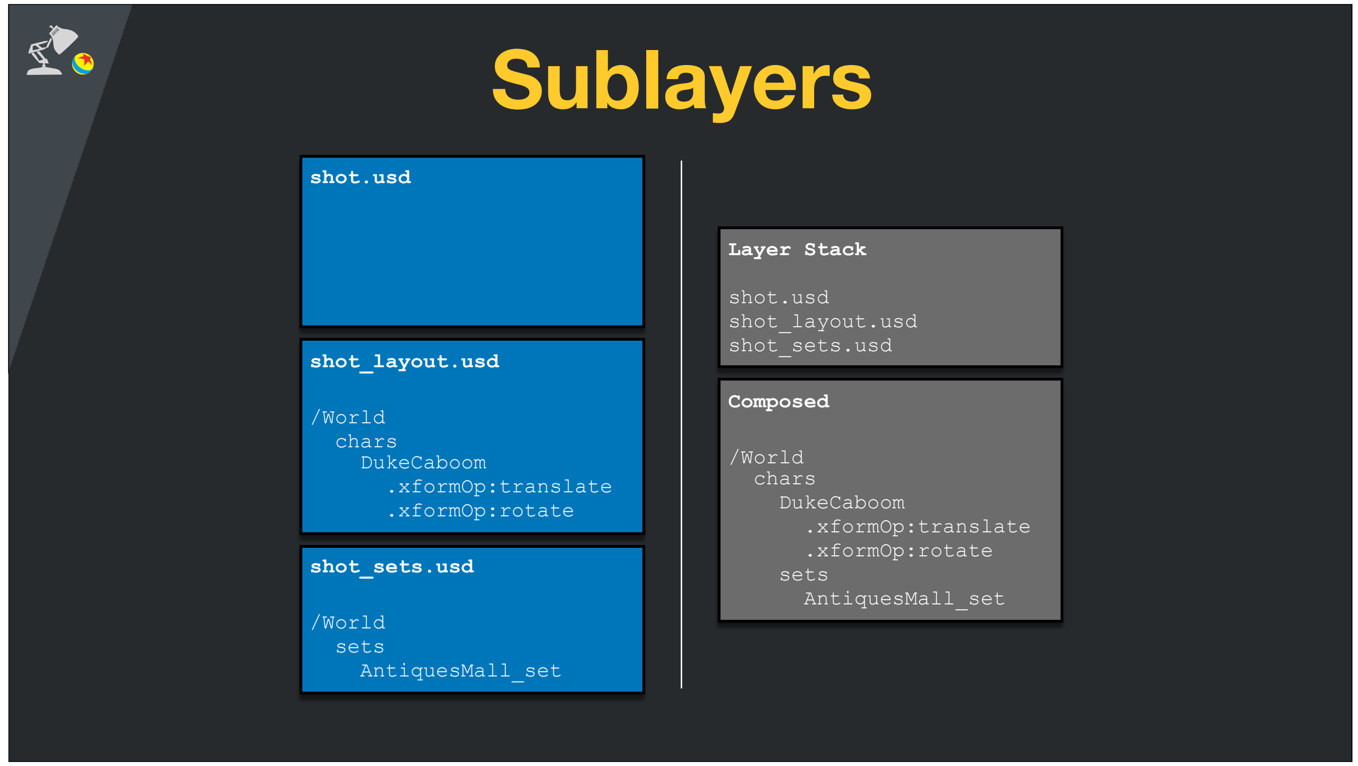


Sublayers are the simplest composition arc. They simply allow you to merge scene description in a layer over other layers. On the right, you can see an example of what this would look like in scene description — we have a root layer called “shot.usd” that specifies an ordered list of sublayers “shot_layout.usd” and “shot_sets.usd”.



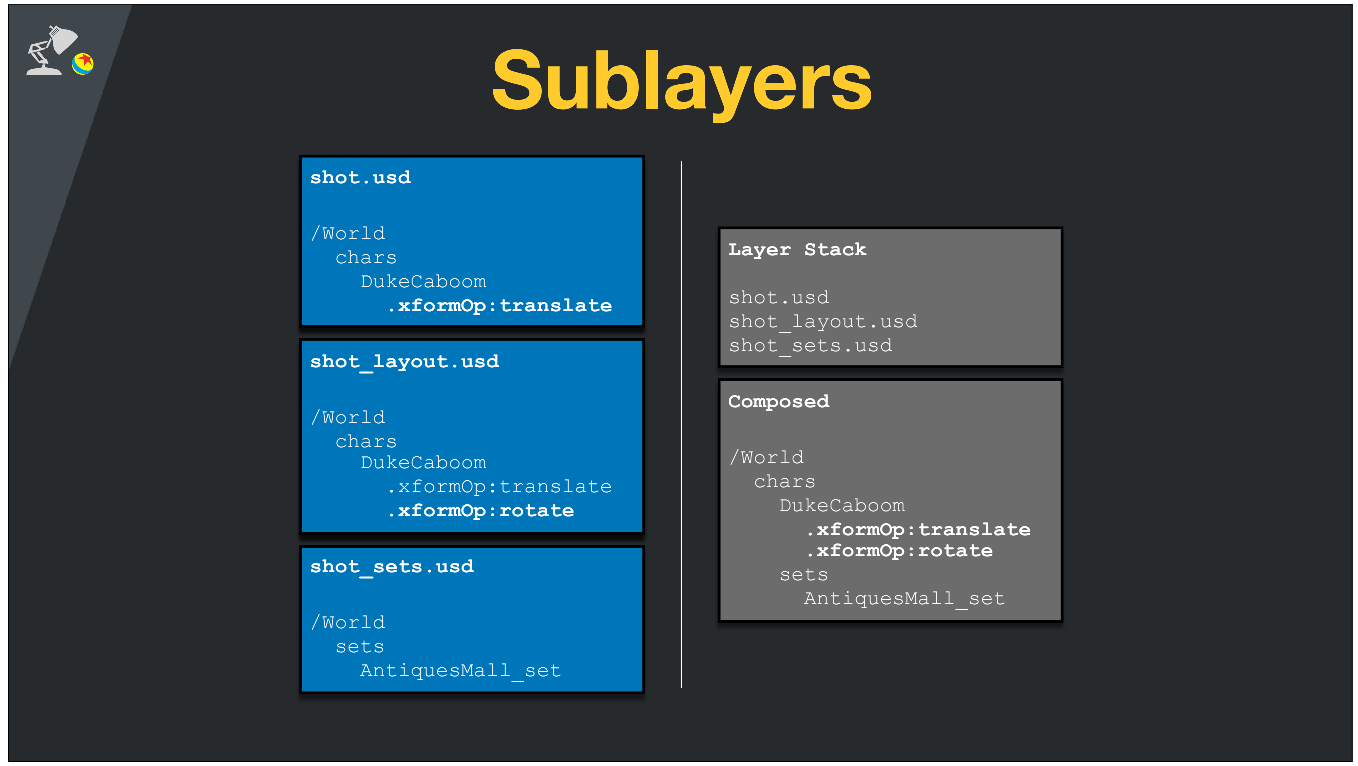
The root layer, its ordered list of sublayers, etc. form an ordered tree of layers. Composition does a depth-first traversal of this tree to produce an ordered list of layers called a “layer stack”, with the root layer at the top of the stack.

Scene description “opinions” in higher layers are considered stronger than those in lower layers in the composed scenegraph.



For our example shot, we're going to build up our scenegraph structure using sublayers. On the left you see the scenegraph in the various sublayers and on the right the resulting composed scenegraph. You can see in the composed result that the scenegraphs from the sublayers are just merged together.

Setting up our shot this way enables multiple users to be working on the shot at the same time. For example, you could imagine a set dresser and a layout artist working on this shot at the same time, but each making changes to a different sublayer.



In this example DukeCaboom has been positioned a certain way in the layout layer using the "translate" and "rotate" attributes. Now another artist might come in and start moving DukeCaboom around the shot by authoring the "translate" attribute in the shot.usd layer.
<click>
We now have two values (or "opinions") for the "translate" attribute in this layer stack, but the opinion from shot.usd wins because shot.usd is the strongest layer in the layer stack. This is an example of a non-destructive override: we've changed DukeCaboom's position in the shot without directly modifying the old value, just overriding it.

References

- **Bring in composed scenegraph hierarchy for a prim in another layer stack**
- **Referenced scenegraph encapsulated under “referencing” prim**

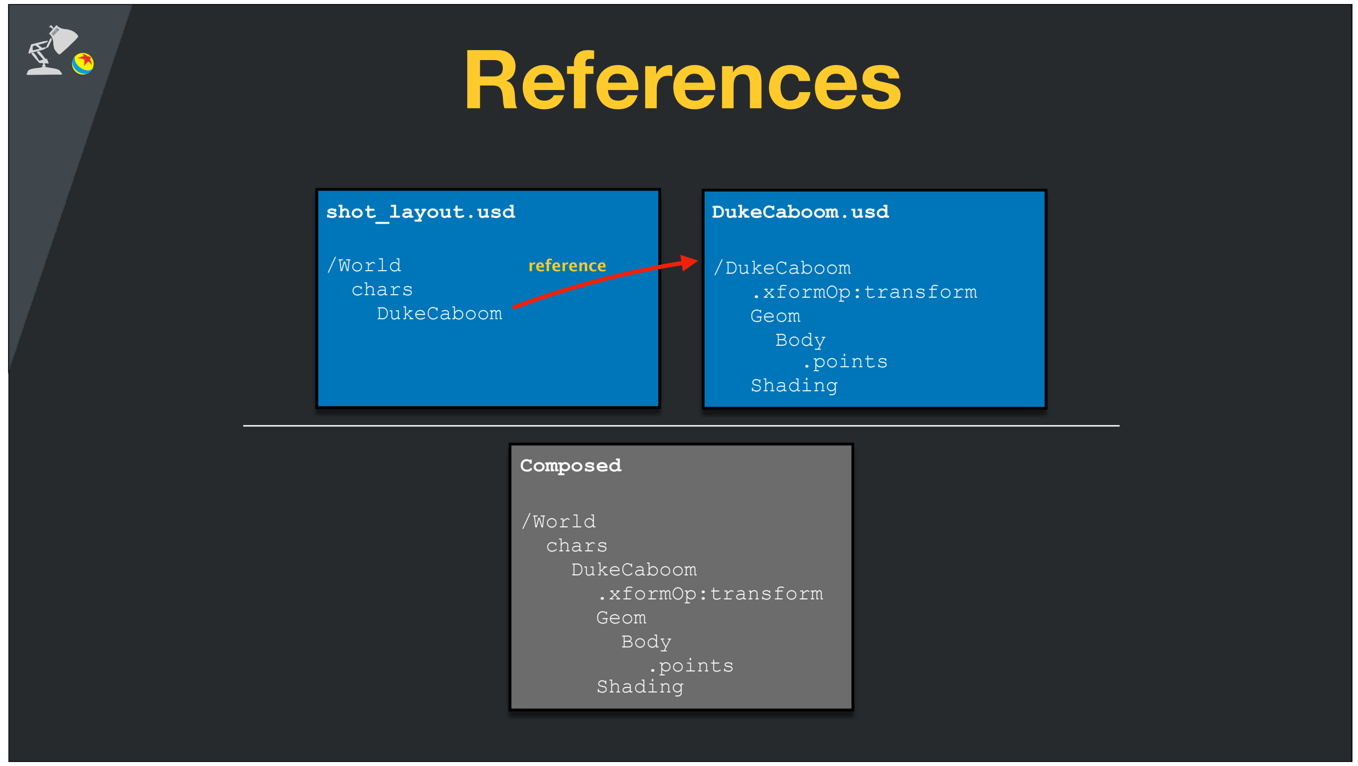
```
shot_layout.usd

def “World”
{
  def “chars”
  {
    def “DukeCaboom” (
      references = [
        @DukeCaboom.usd@</DukeCaboom>
      ]
    )
    {
    }
  }
}
```

Now that we’ve established a very basic shot structure using sublayers, we can start to populate it. To do this we can use the “reference” composition arc to bring in the composed scenegraph for a prim in another layer stack. This is how we can build up large assets from smaller assets.

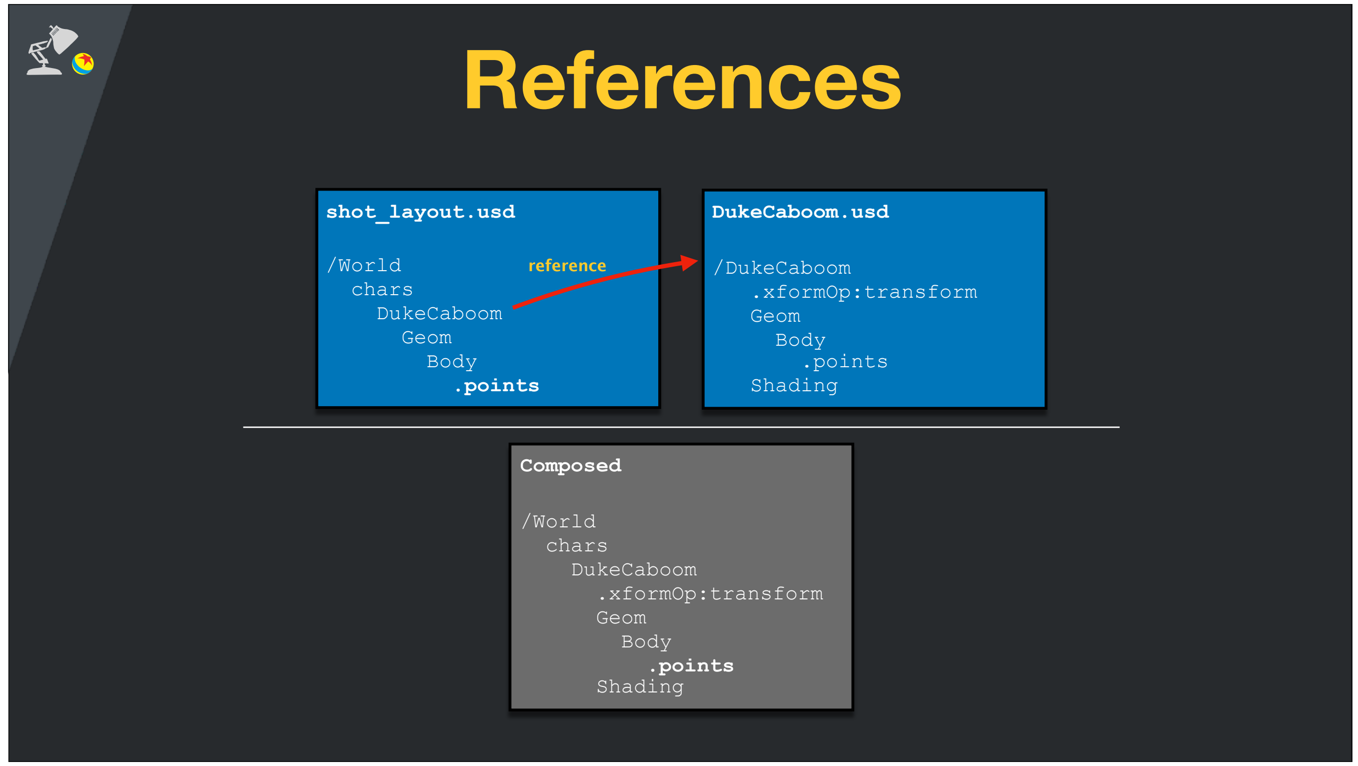
References

```
DukeCaboom.usd

/DukeCaboom
   .xformOp:transform
   Geom
     Body
       .points
   Shading
```

Let’s say that the DukeCaboom model has been created in a separate DukeCaboom.usd layer. This model might itself be built up using sublayers and other composition arcs, but for simplicity I’m just using one layer.



In the layout layer, we can author a reference to the DukeCaboom model. At the bottom, you can see that composition brings in the contents of the DukeCaboom model into the composed scenegraph where the reference was authored.



Just like with sublayers, we can do non-destructive overrides for values from across the reference arc. For example, if we author a different set of points in the shot_layout.usd layer, those values will be used in the composed scenegraph because they're "stronger" than the opinions in the referenced asset. We'll talk more about this strength ordering a bit later.

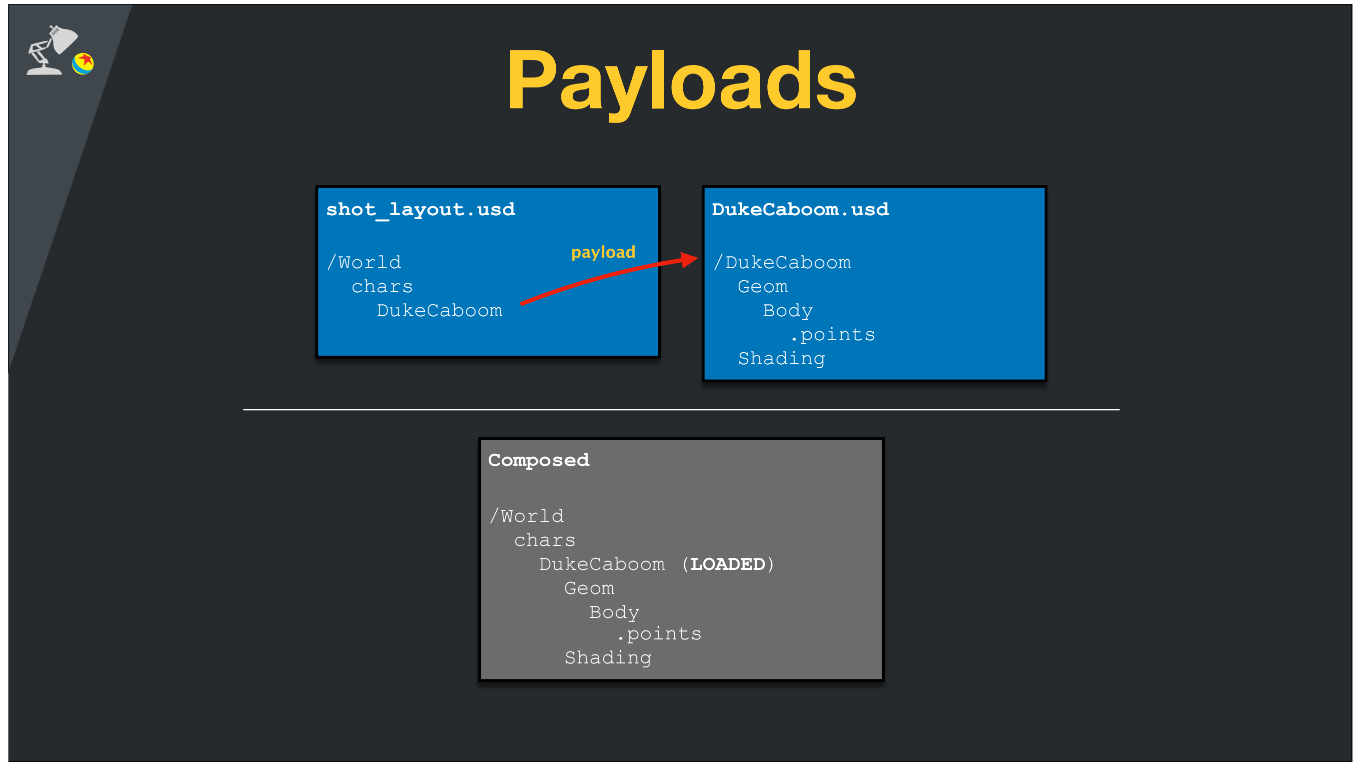
I also want to note that you can reference the same asset into a shot as many times as you want. For example, when building the AntiquesMall set, we might have a single prop asset that gets referenced multiple times so we have copies of the same asset distributed throughout the set.

Payloads

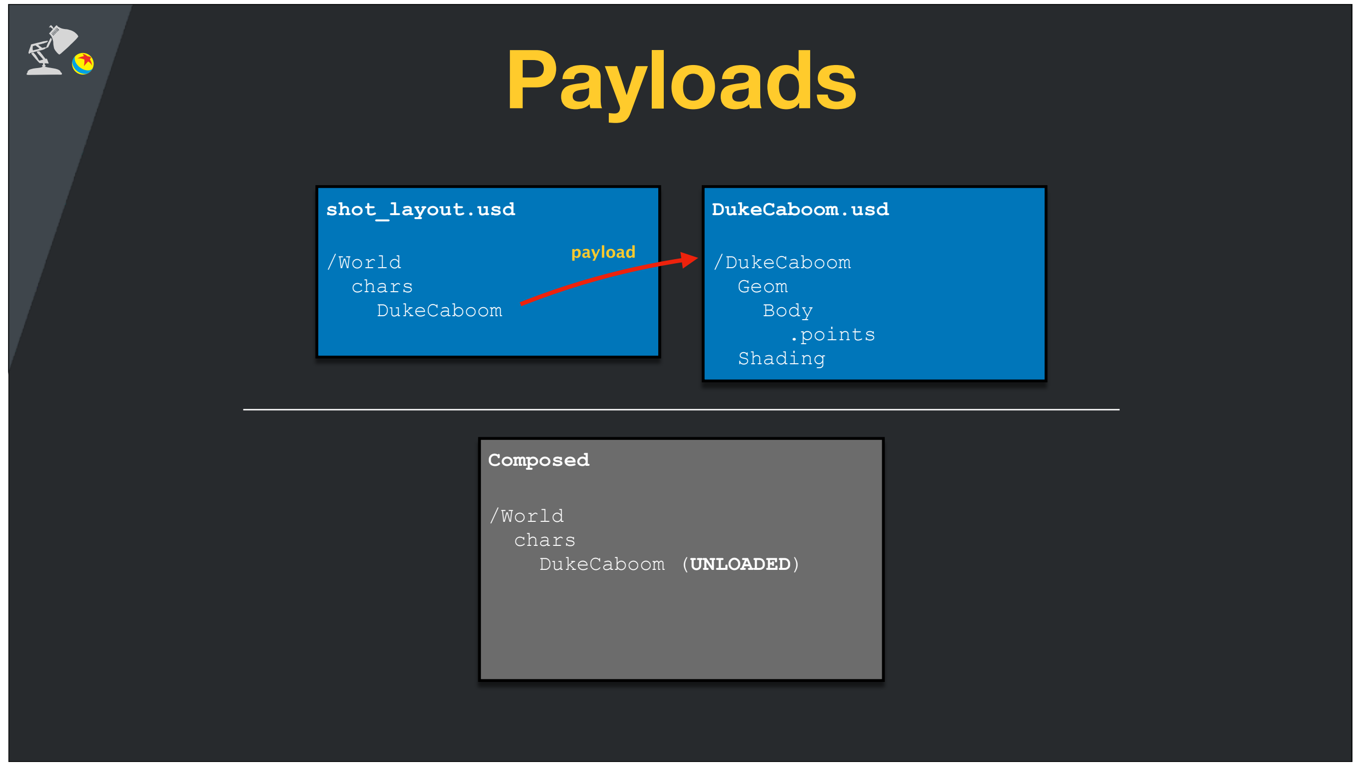
- **Optionally loadable references**
- **Allows clients to decide at runtime to load expensive scene description**
- **See UsdStage::Load, UsdStage::Unload**

```
shot_layout.usd

def “World”
{
  def “chars”
  {
    def “DukeCaboom” (
      payloads = [
        @DukeCaboom.usd@</DukeCaboom>
      ]
    )
    {
    }
  }
}
```

References let us build up large aggregates assets from smaller pieces, but in some cases you may want to give consumers the ability to manage what gets composed at runtime. The “payload” composition arc provides this functionality. Payloads are references that may be optionally loaded or unloaded at runtime via various API calls.

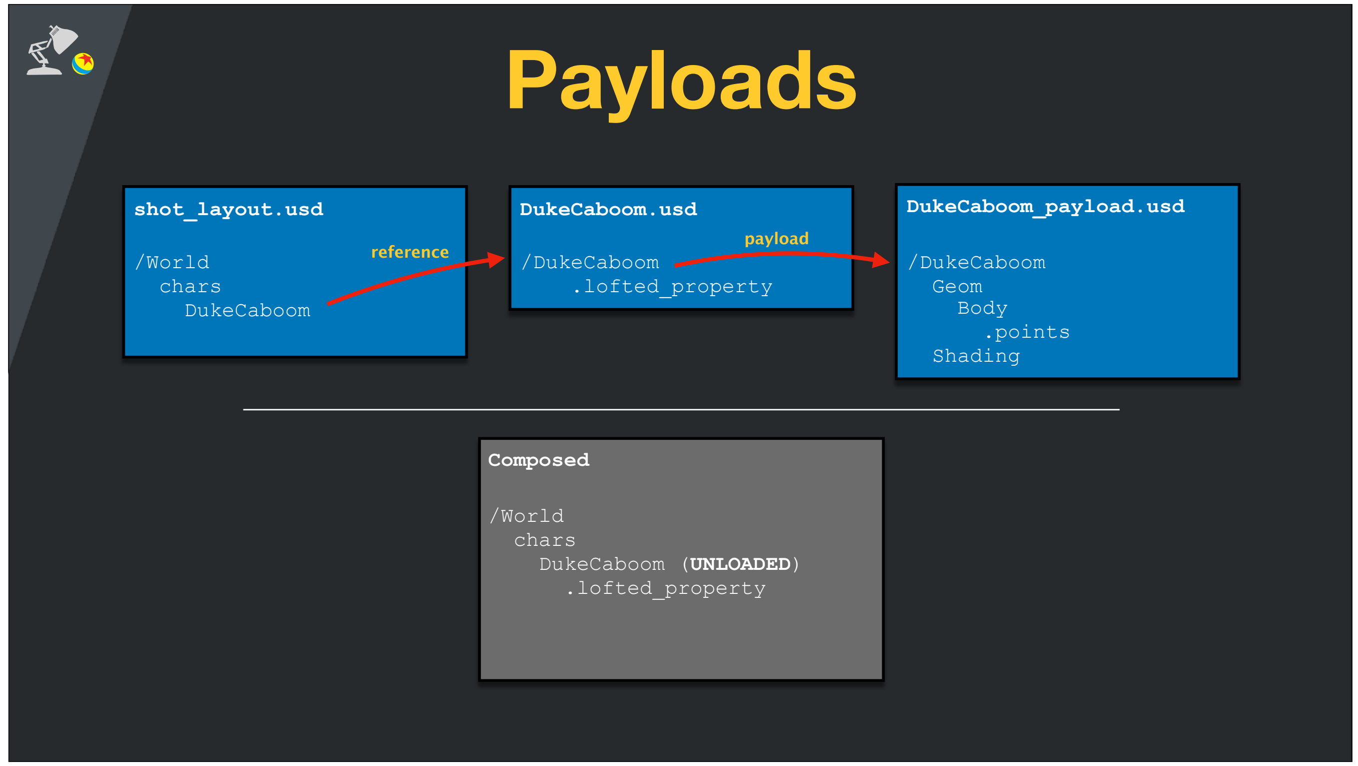


Let's say we use a payload arc to bring DukeCaboom into our shot instead of a reference like before. When the payload is loaded, the composed result looks just like it did when we used a reference.

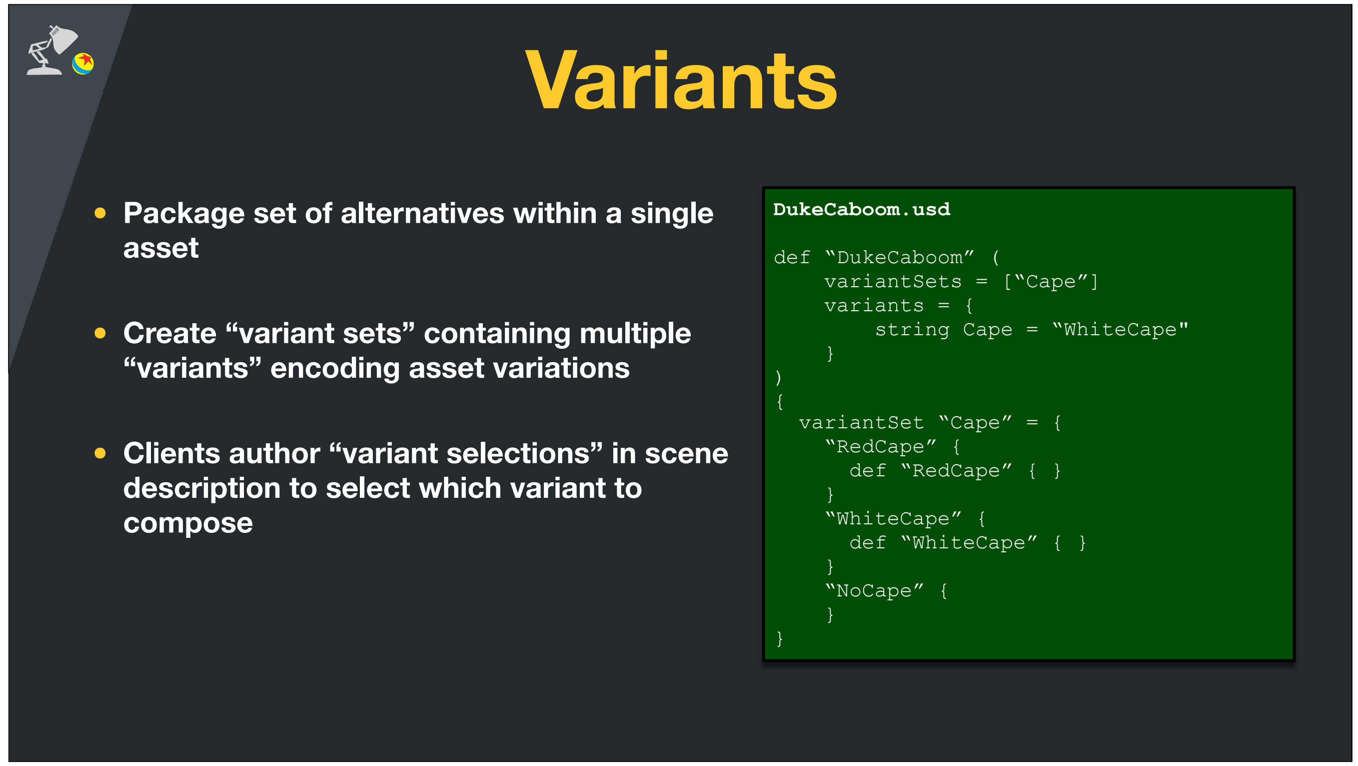


But let’s say there’s a set dresser working on the AntiquesMall set and they don’t need to see Duke. At runtime, they can choose to not load DukeCaboom’s payload, in which case the contents of the payload will be ignored during composition, saving memory and runtime cost.

Note that payload loading/unloading is a runtime flag, so loading/unloading a payload doesn’t author any changes to layers.

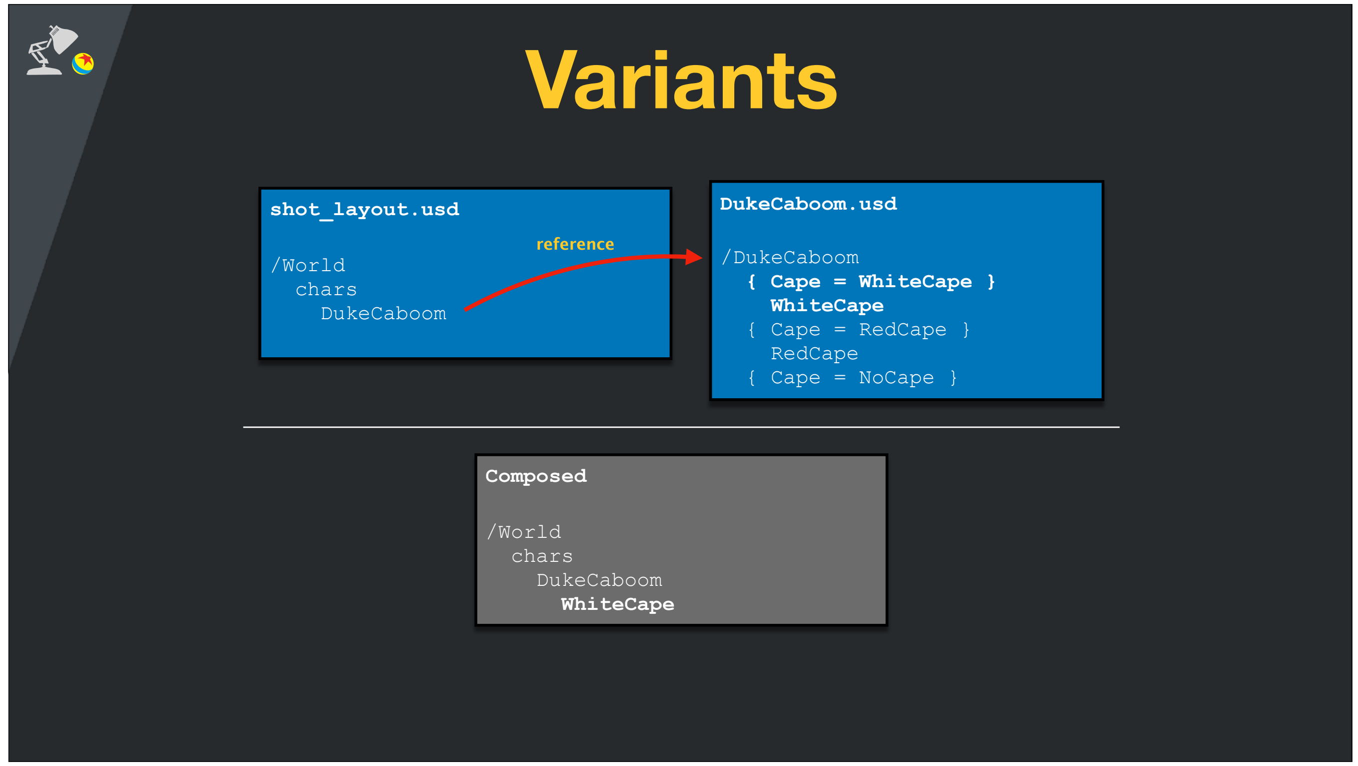


In the Pixar pipeline, we use payloads heavily to allow users to pay for only what they need. But we've found that there are some properties or metadata in assets that we want to always have available, even if the payload is unloaded. To support this, we typically introduce an intermediate layer in our asset structure that contains the stuff we always we want to see in the scenegraph. This layer then uses a payload arc to make all of the expensive data optionally loadable. Internally we refer to this as "lofting" opinions outside the payload.

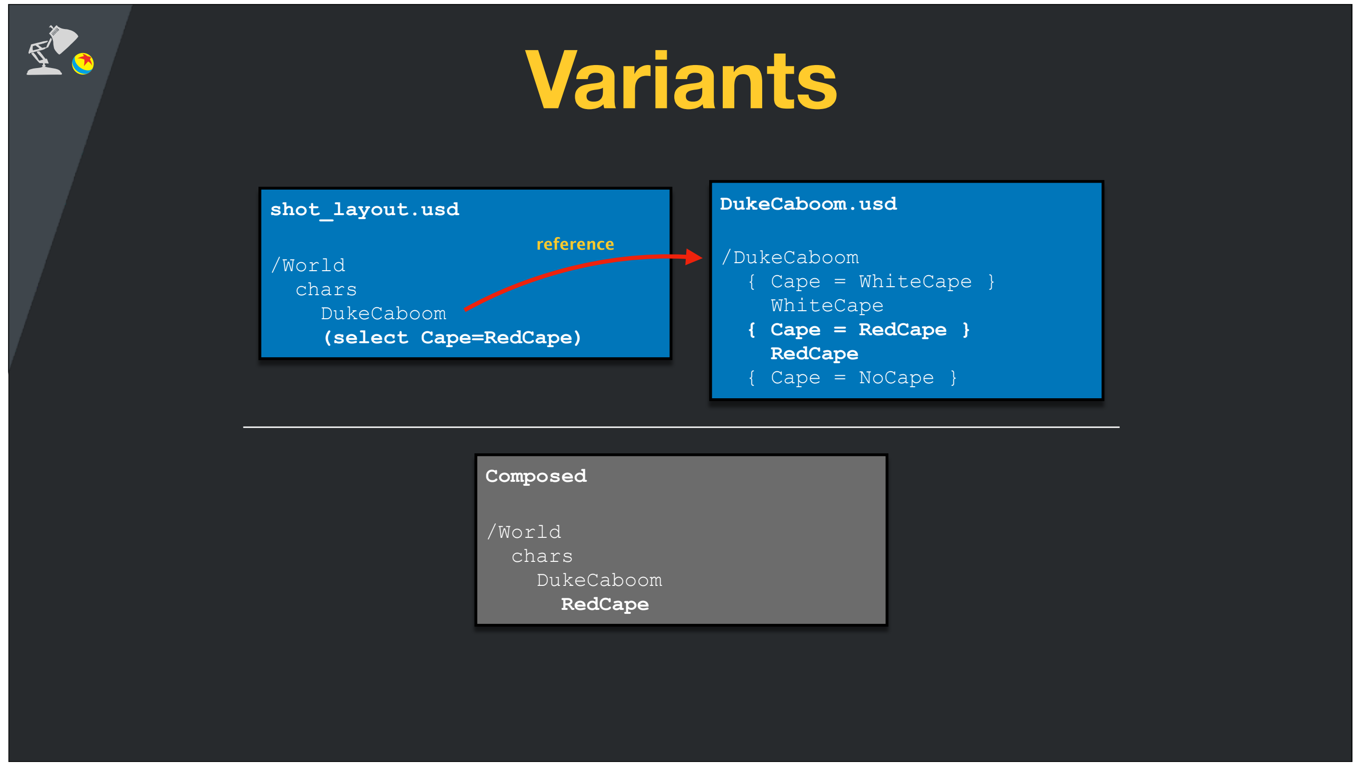


The next arc I want to talk about is the “variant” arc. Variants allow you to package up a set of alternatives within a single asset. Users create “variant sets” that contain “variants” in an asset, then author “variant selections” to select which variants should be considered during composition.

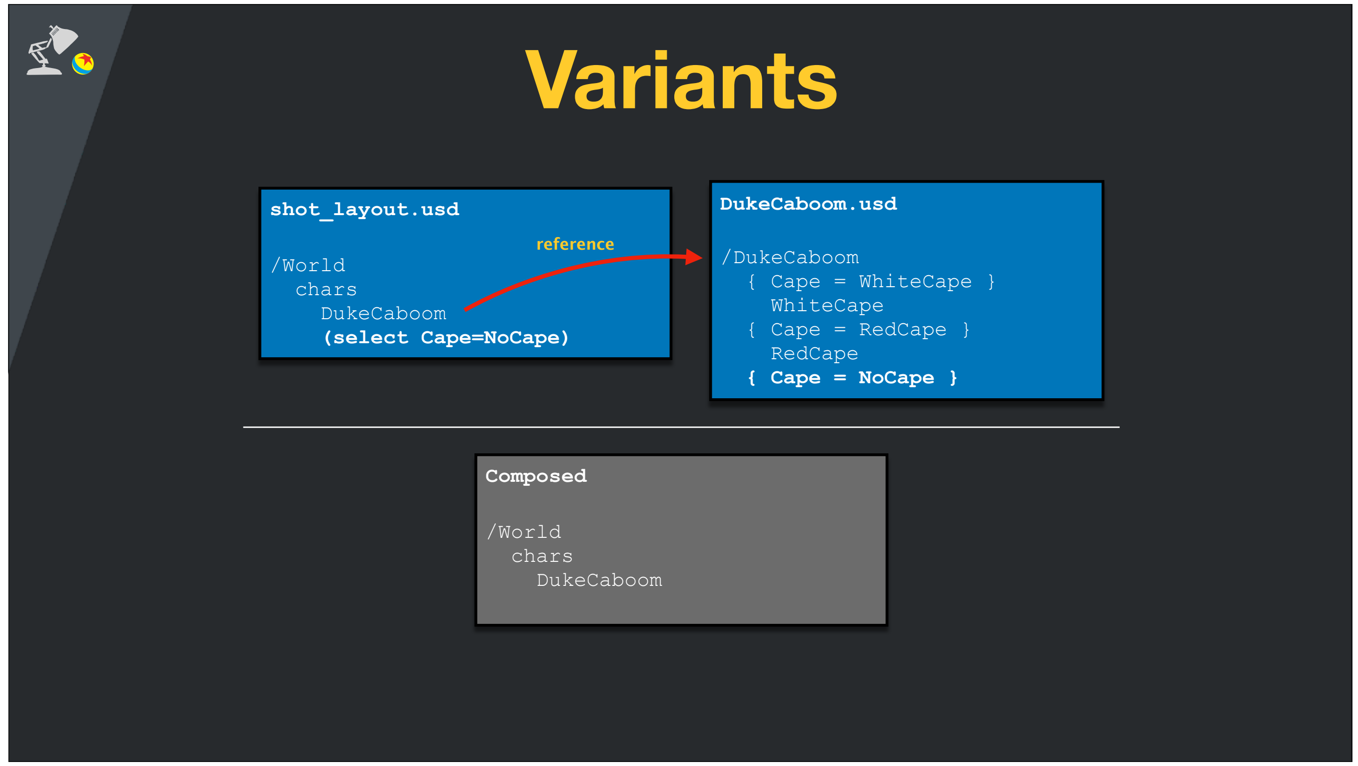
The example on the right shows our DukeCaboom asset with a variant set allowing users to choose between different cape variants and a variant selection that chooses the WhiteCape variant.



When we reference this DukeCaboom asset into our shot, composition will bring everything under the "WhiteCape" variant into the composed scenegraph because that's the authored variant selection in the asset.



However, we can author a different variant selection in our shot layer, like Cape=RedCape. This selection will override the selection authored in the asset, so our composed scenegraph winds up with the RedCape instead.



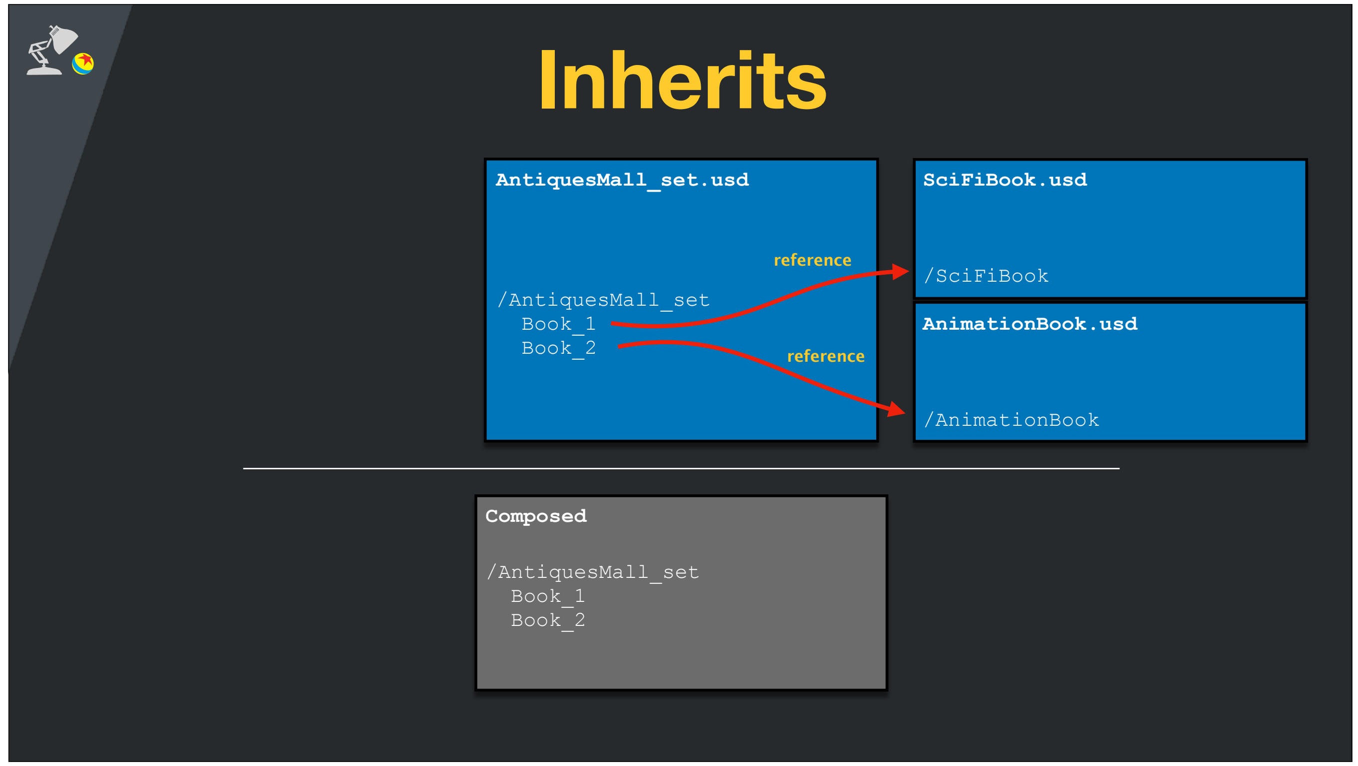
And similarly for the NoCape variant. One important thing to note is that a prim can have multiple variant sets. For example, instead of having the a single “Cape” variant set that encodes both the presence of the Cape and its color, we could have used two variant sets. This can give users finer-grained control over the variation.

Inherits

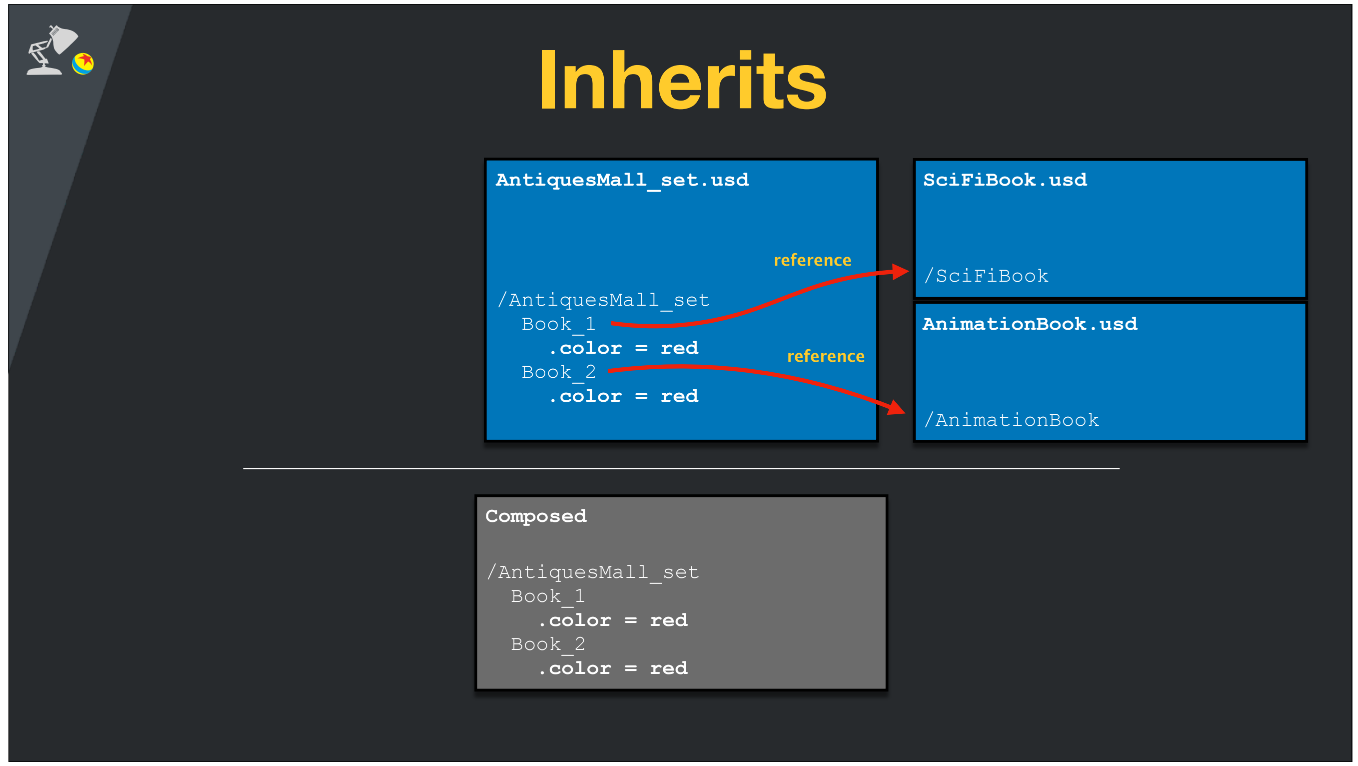
- **Allow mass edits and overrides on all prims of a given classification**

```
SciFiBook.usd

class "class_Book"
{
}

def "SciFiBook" (
    inherits = [
        </class_Book>
    ]
)
{
}
```

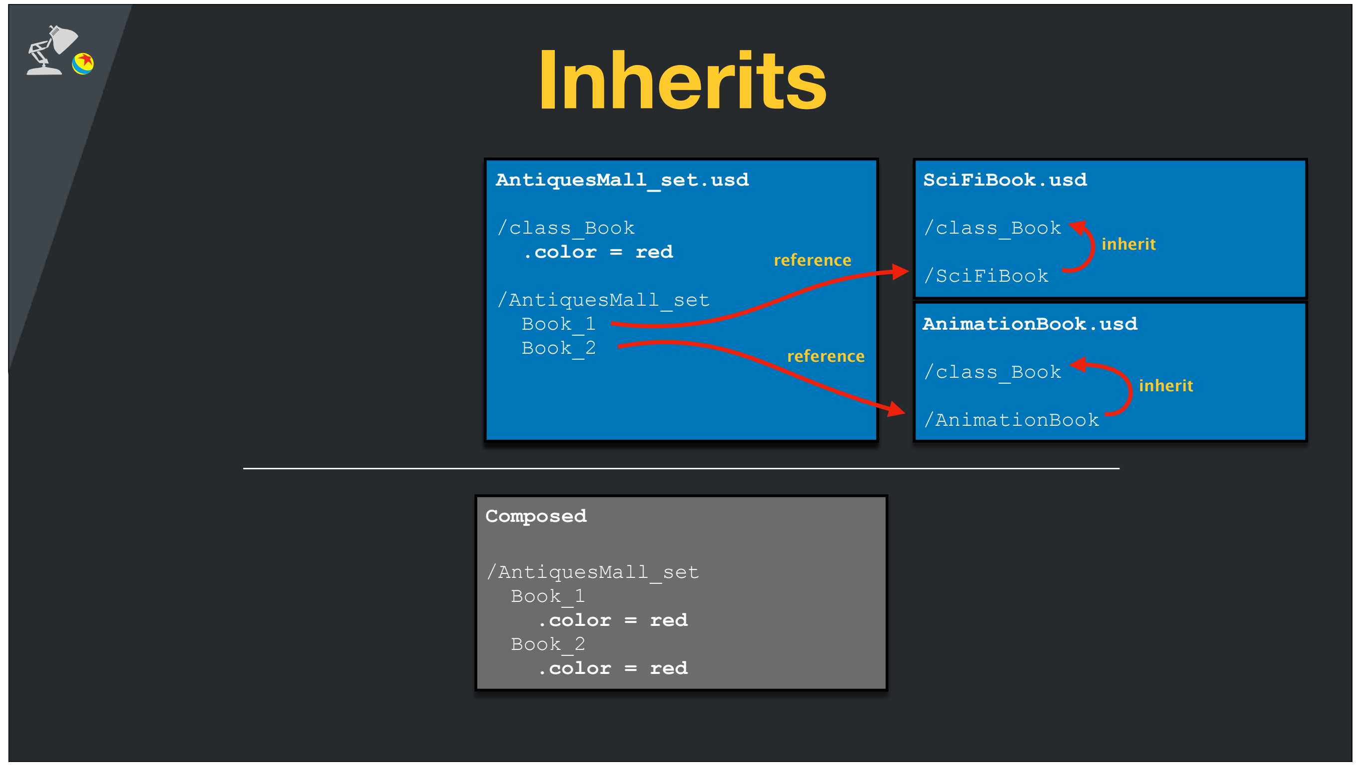
The next composition arc to discuss is inherits. Inherits allow you to perform mass overrides on all prims of a given classification in a minimal way.



Let’s say in the AntiquesMall set I want to have a bunch of different books scattered around. We can use references to bring in two different book assets like so. But suppose that in the AntiquesMall set, I want to apply an override to all of the books, for example let’s say I want all the books to be red.

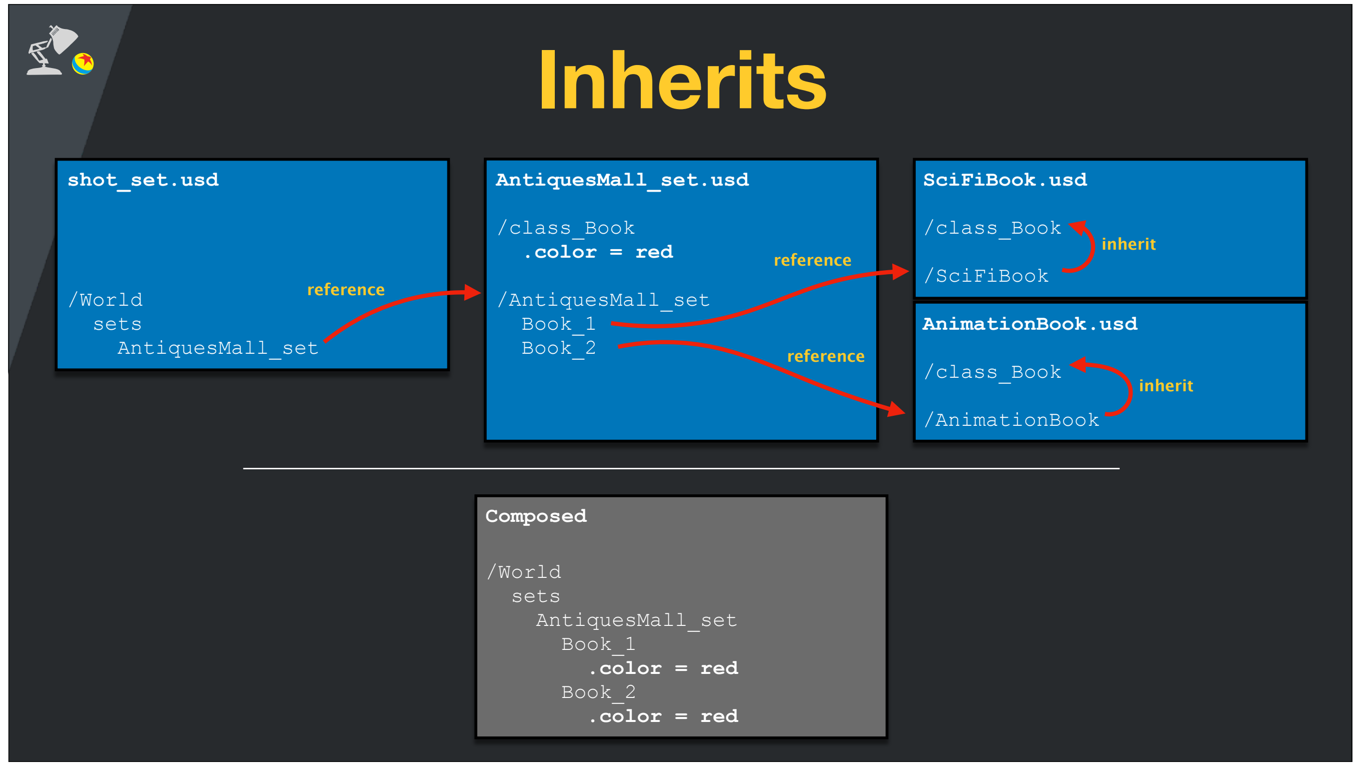


One way we can do this is to author an override on each Book in the AntiquesMall set, as shown here. This gives us the result we want, but is not ideal for a number of reasons. It's not scalable — if you have a huge number of books in the set, you have to find them all and author the same override everywhere. And if you add another book to the set, you have to remember to apply the same color too!

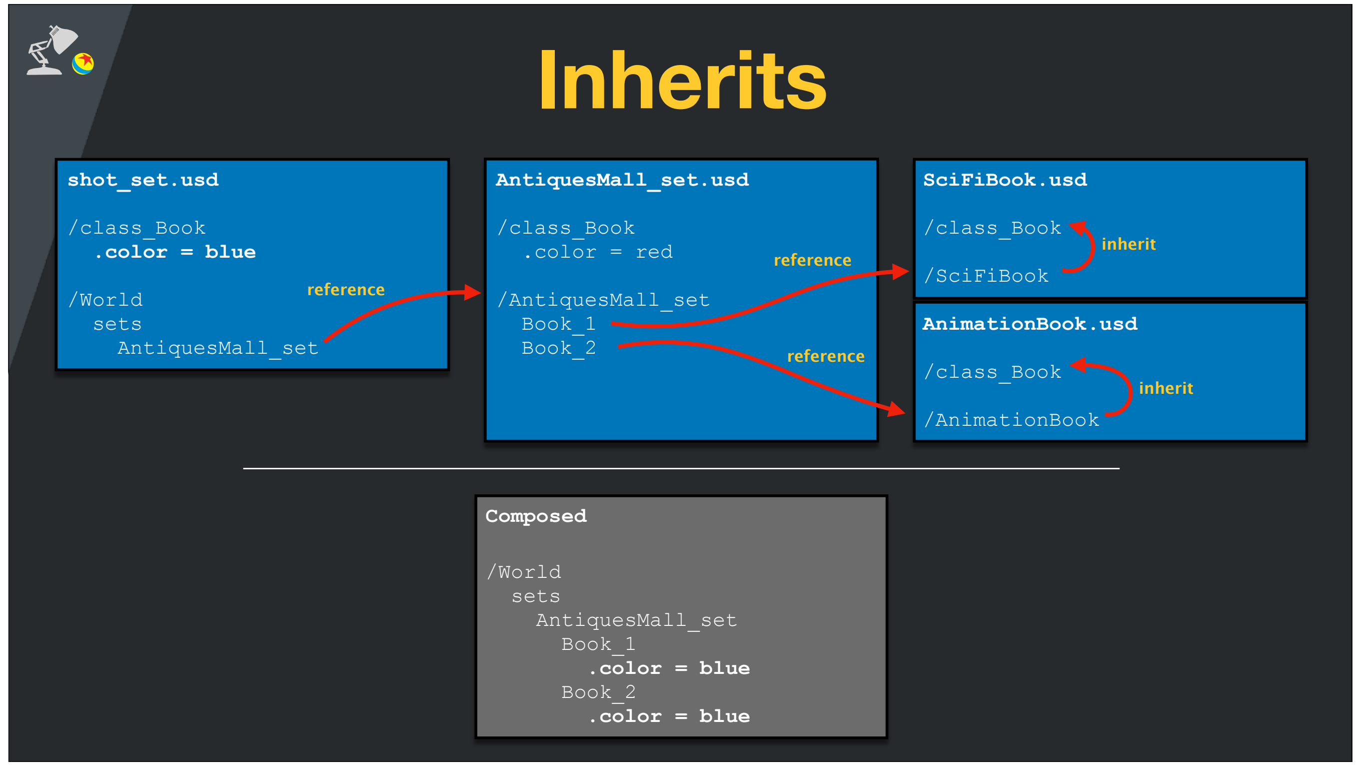


We can use inherit arcs to solve this problem. In our two assets, we author inherit arcs to a prim called "class_Book". The name could be whatever we'd like, but the important thing is that they're the same. Then in the AntiquesMall set, we can simply author the color=red opinion on class_Book, and they'll be inherited down to all of the books.

This address the problems with the previous approach. Also notice that we didn't have to author the inherit composition arcs in the AntiquesMall set, they were part of the Book assets themselves.



Taking this a step further, when we reference the AntiquesMall set into our shot we'll continue to see the red color on all books.



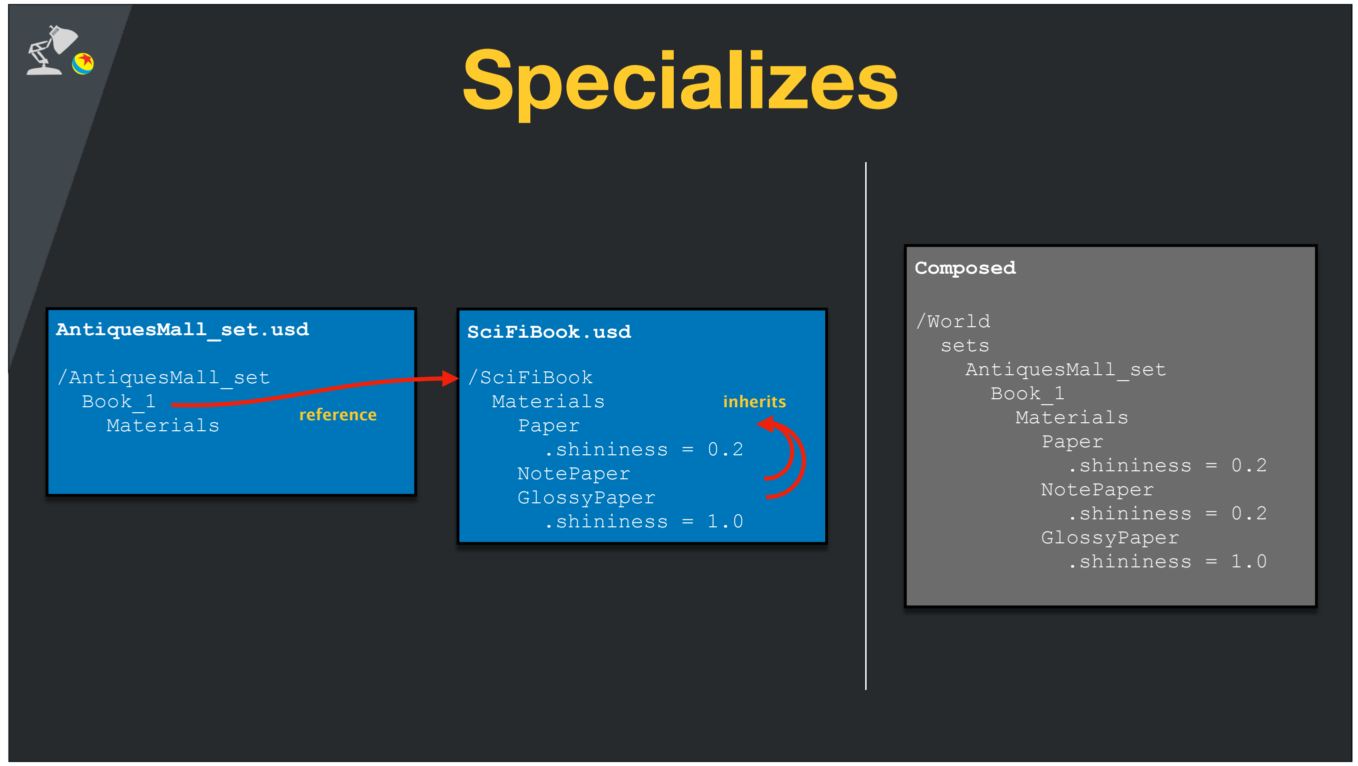
But we can override the class opinion in the shot and make all of the books blue instead! So we can see that inherits are a pretty powerful mechanism for broadcasting edits to many prims.

Specializes

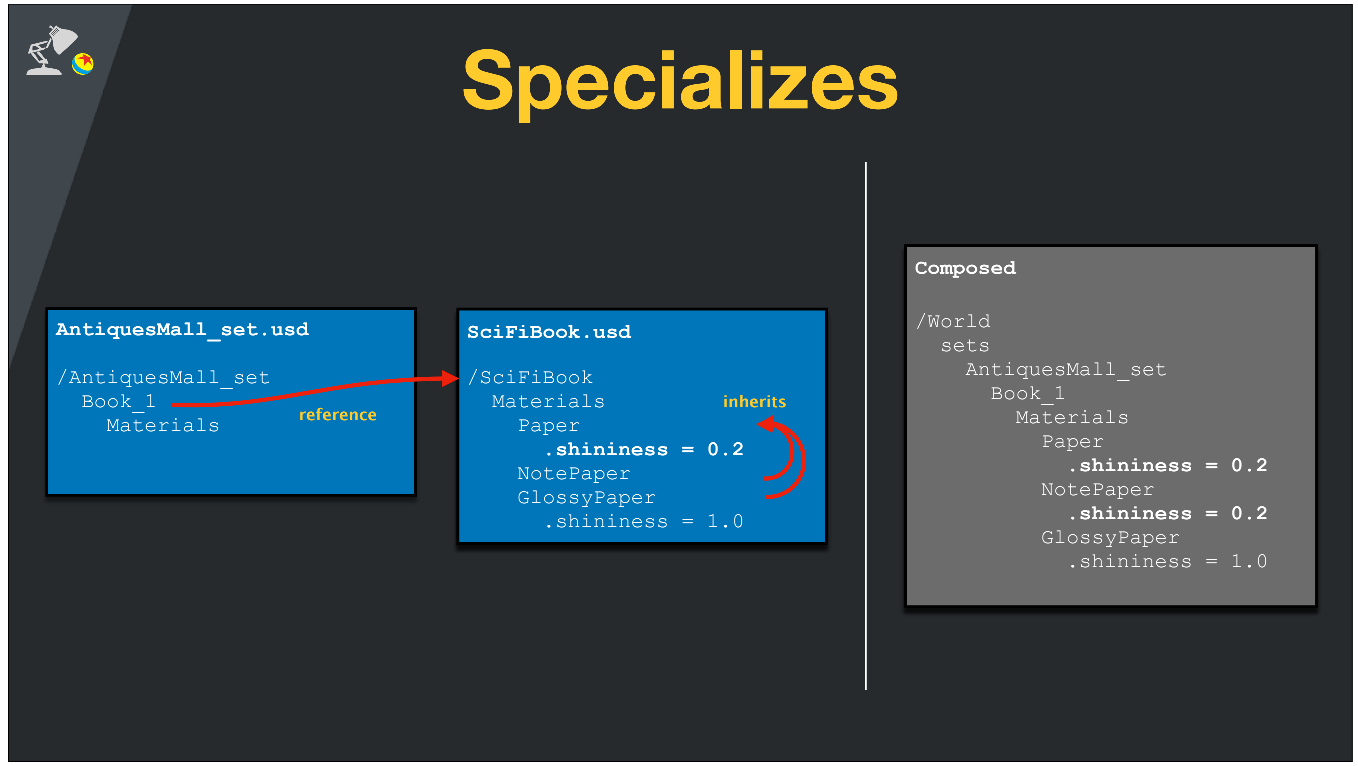
- **Enables refinement of specialized prim from base prim**
- **Developed to help with defining shader and material libraries in USD**
- **Similar to inherits, but opinions from base prim are always weakest**

```
SciFiBook.usd

def “SciFiBook”
{
  def “Materials”
  {
    def “Paper”
    { }

    def “NotePaper” (
      specializes = [
        </SciFiBook/Materials/Paper>
      ]
    ) { }

    def “GlossyPaper” (
      specializes = [
        </SciFiBook/Materials/Paper>
      ]
    ) { }
}
```

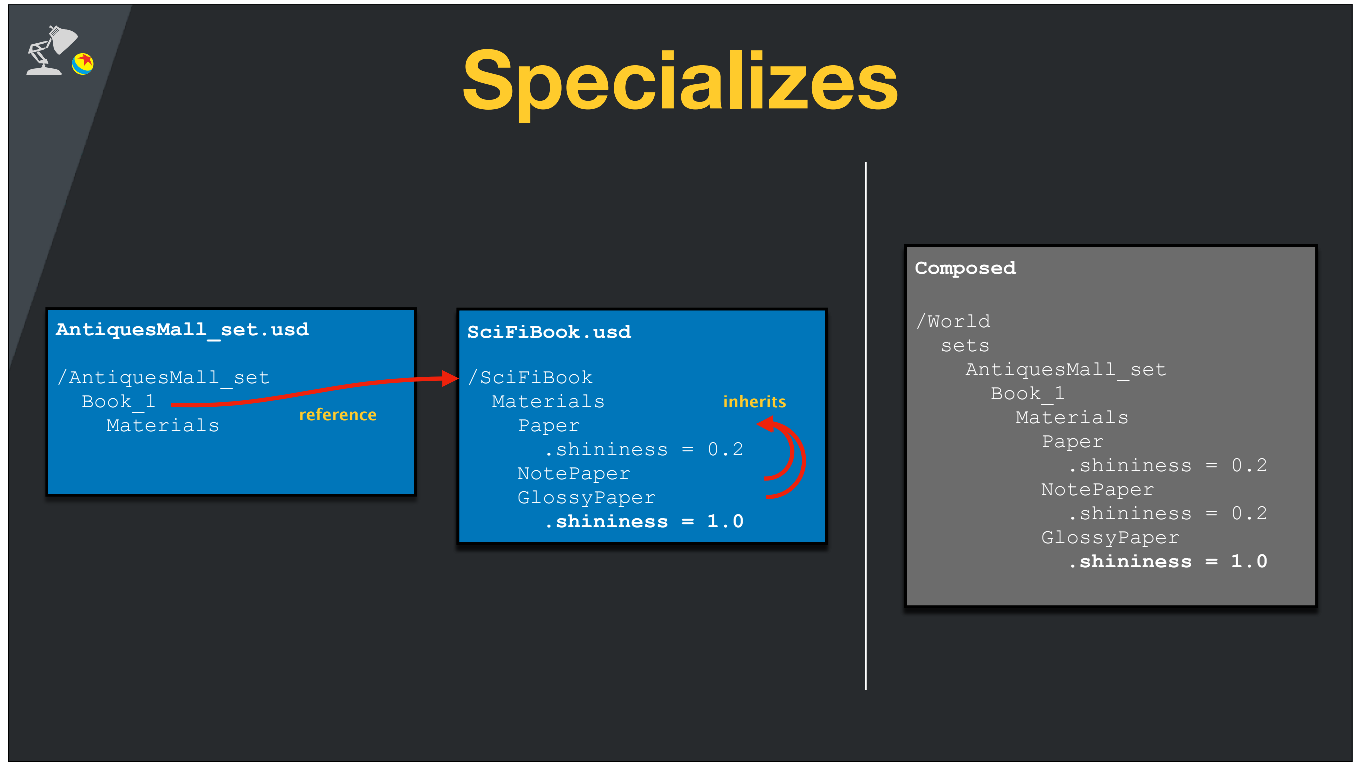
The last composition arc to talk about is the “specialize” arc. This is the most recent addition to the composition engine and was developed to help define material libraries in USD. This arc behaves like inherits, but has different strength ordering rules — opinions in the specialized prim are always weaker than opinions in the specializing prim. That’s kind of complicated, the easiest way to understand this is through an example.



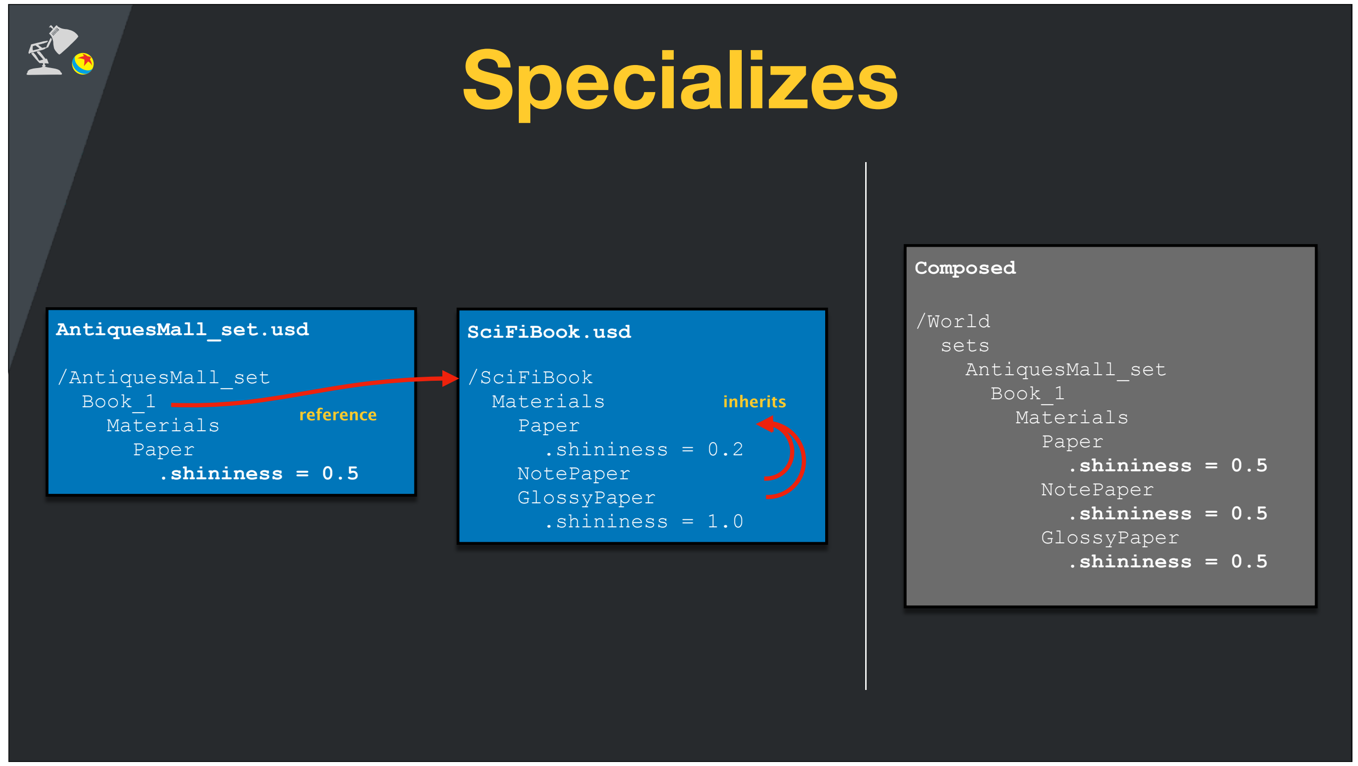
In our SciFiBook asset, we want to define a set of materials for the pages in the book. We have a base “Paper” materials that defines some common properties, and two other types of paper that both inherit from that prim. Glossy paper is always shinier than regular paper, so for the “GlossyPaper” material we want to specify a higher shininess value than basic Paper.
<click>
When we reference this into the AntiquesMall set, the composed result looks like this.



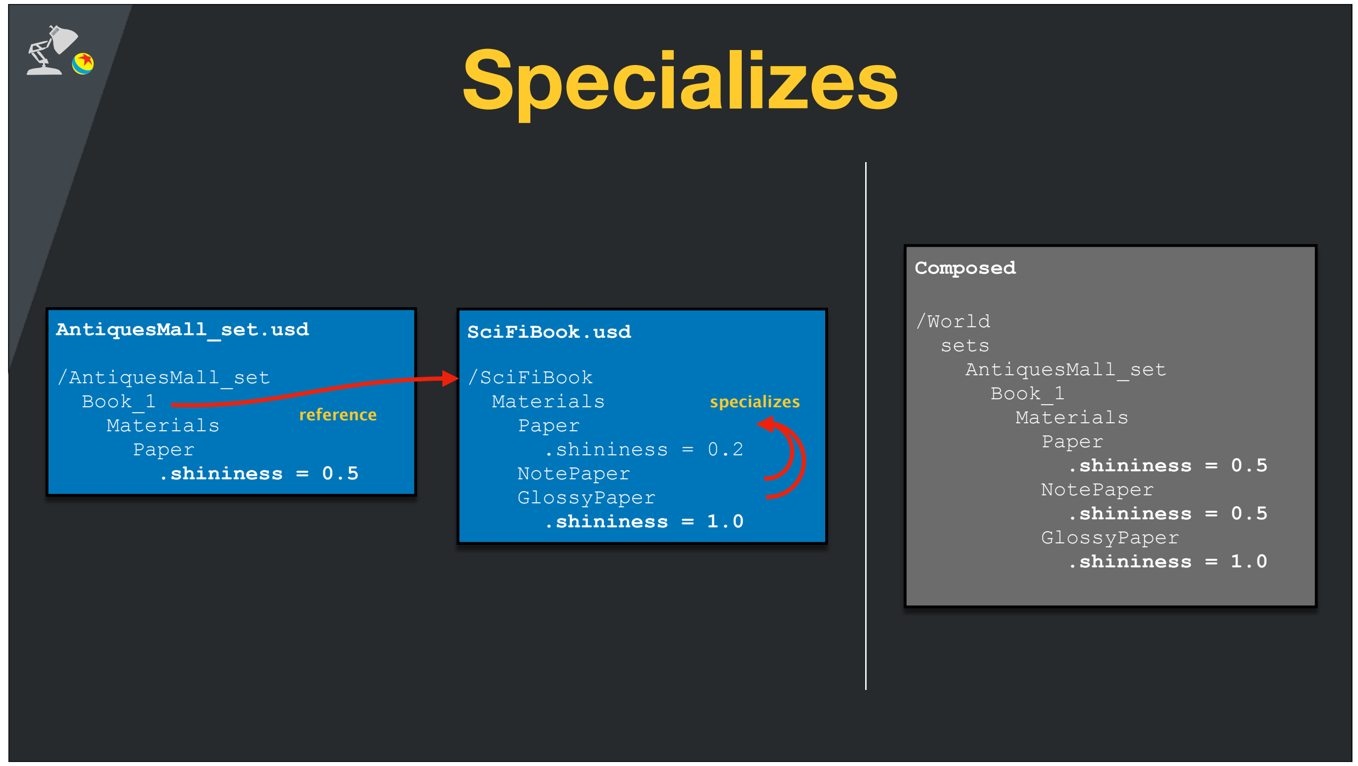
The shininess value for the Paper and NotePaper material are 0.2



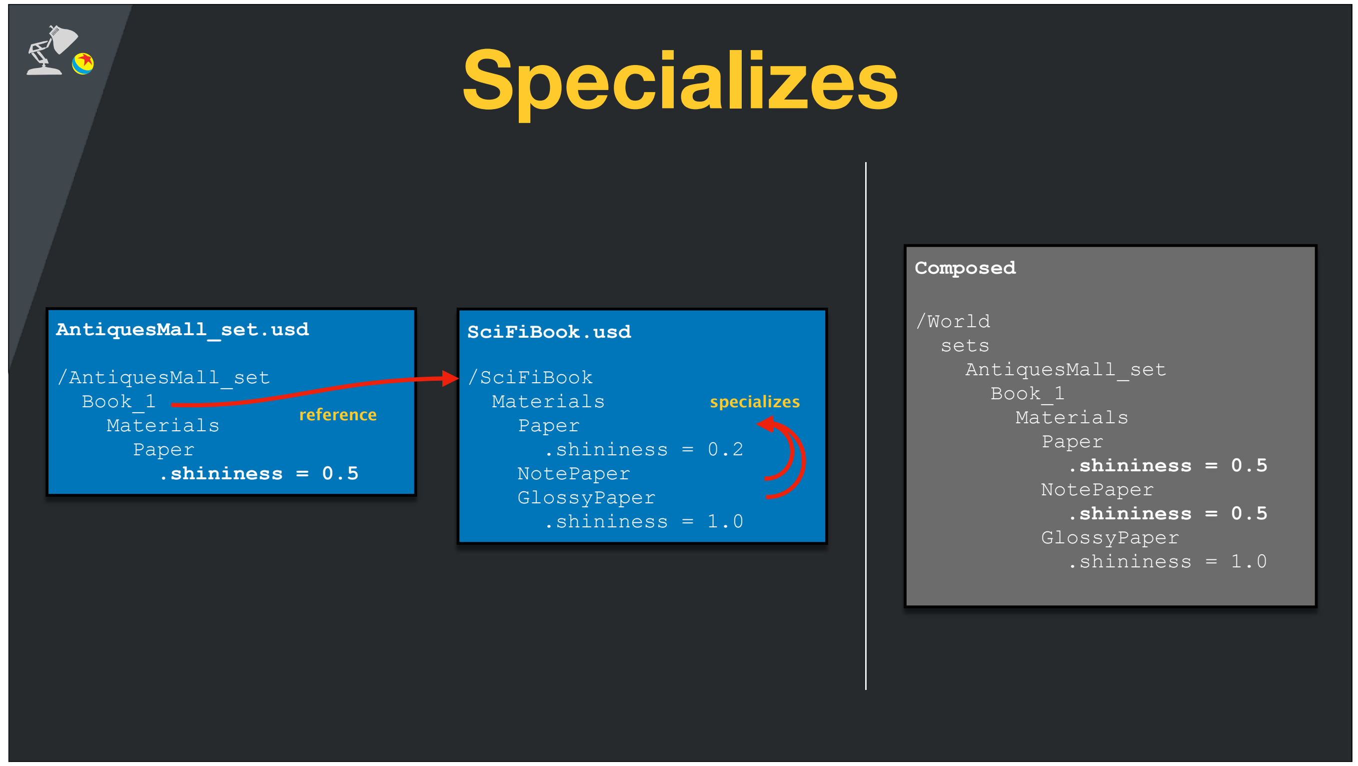
And the shininess value for the GlossyPaper material is 1.0



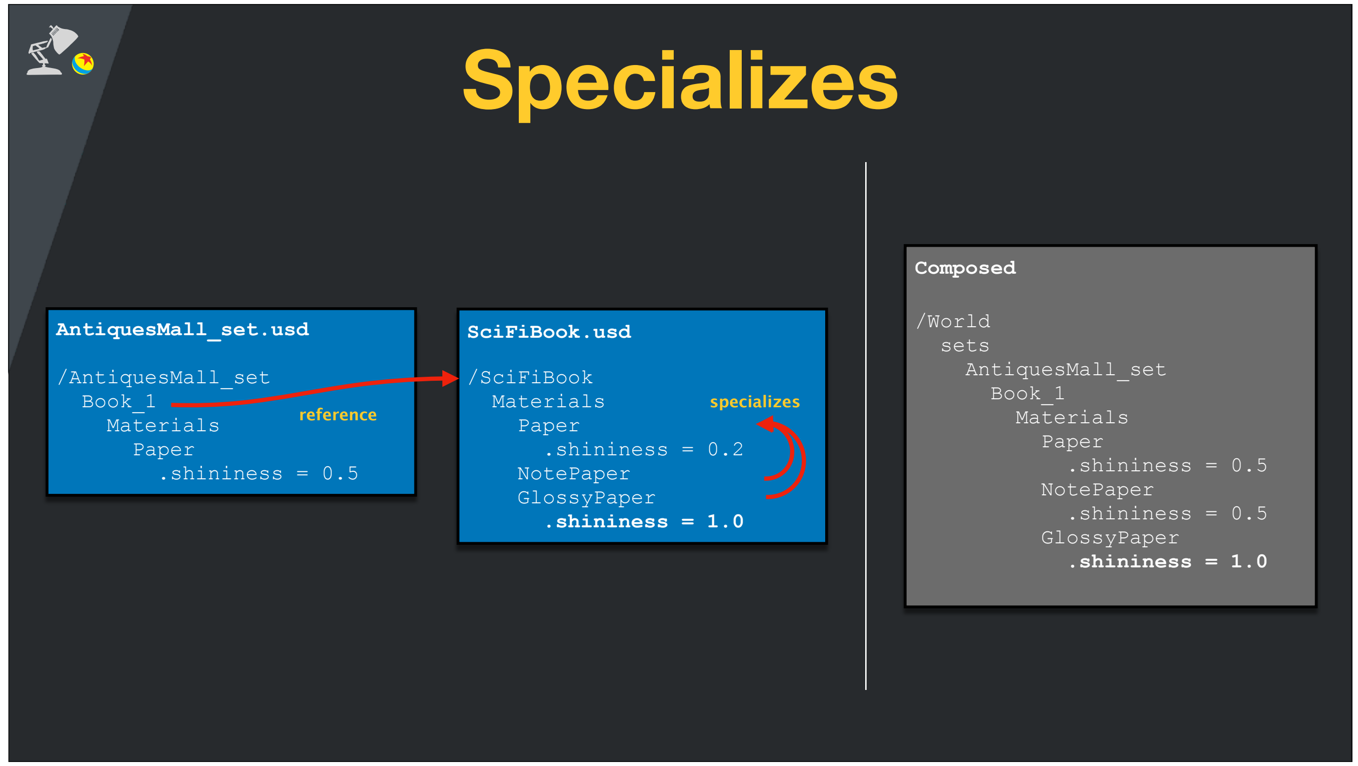
Now suppose in the AntiquesMall set we decide we want the regular pages in this book to be just a bit shinier, so we override the value in the AntiquesMall layer. Thanks to the inherits behavior, this value overrides all of the previous shininess values. Conceptually, this makes sense for NotePaper, since we hadn't specified a value there — we wanted it to be as shiny as the basic Paper material. But it's not what we want for GlossyPaper! We had already refined the GlossyPaper material by specifying its shininess in the book, we don't want any changes to the basic Paper's shininess to override this.



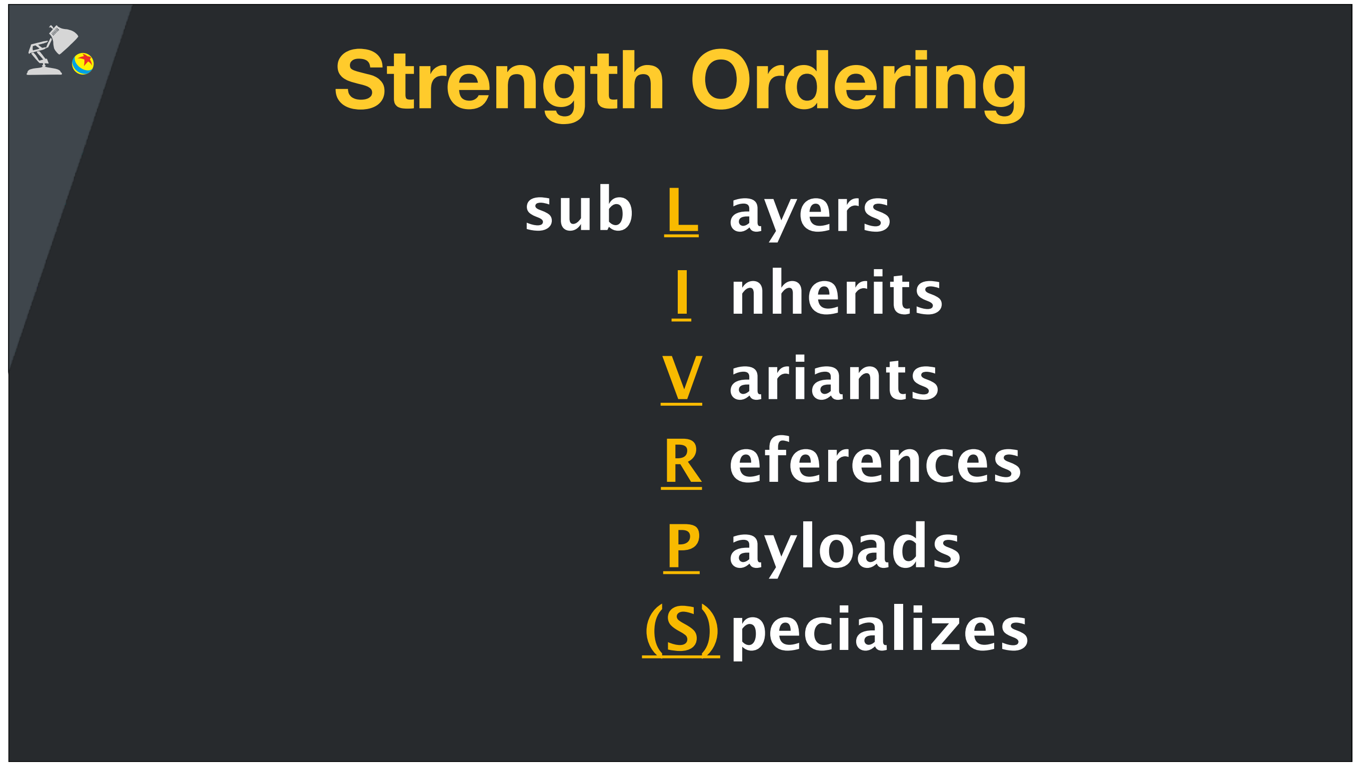
If we just replace the inherits arc with specializes, we get exactly the behavior we want.



The shininess value for the base Paper material and the NotePaper is now 0.5



And the shininess value is the value we refined in the book asset itself, 1.0.



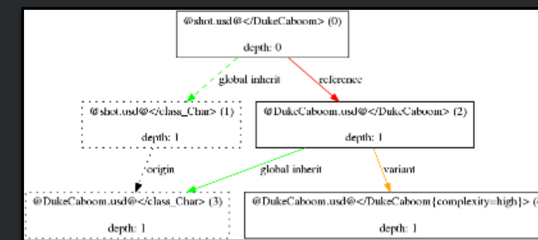
The last aspect of composition behavior I want to talk about is “strength ordering”. The composition engine defines a strength order for the arcs so that it can determine which opinions win when you have a combination of composition arcs on a prim.

This order is given by the acronym LIVRPS, pronounced liver-peas. Opinions from sublayers are stronger than those that come from across an inherit arc, and so on. This ordering is recursive, since each composition arc may introduce other composition arcs. So if a referenced prim introduces other composition arcs, opinions from those arcs are also ordered according to the LIVRPS rule.

Specializes are a tricky case — opinions from specializes arcs are globally weaker than all other opinions; this is how we get the specialization behavior we just saw in the previous example.

Pcp - Prim Cache Population

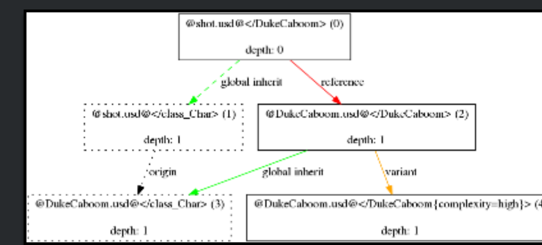
- **Library responsible for evaluating composition operators**
- **Computes “prim index” : graph of locations of scene description opinions and their relative strengths**



OK, so that’s it for the high-level composition behaviors! I want to dive a bit into some of the code-level details. The composition engine is implemented in a library called Pcp, which stands for “Prim Cache Population”. It’s responsible for evaluating all of the composition arcs and generating a “prim index” that shows where the possible opinions are for a given prim and its properties.

Pcp - Prim Index

- Nodes are (layer stack, path) and represent possible sources of opinions
- Edges are composition operators (hence “composition arc”)
- See UsdPrim::GetPrimIndex, PcpPrimIndex



A prim index is represented by an ordered graph, where the nodes represent a layer stack and a prim path that may be sources of opinions for the prim. The edges between the nodes are the composition operators that introduced the node. This is why we call them composition arcs. In code, the prim index is represented by the PcpPrimIndex class, and there’s various API to get access to them.

Inspecting the Prim Index

- Determine referenced assets for isolation or other purposes
- Show UI for possible locations to author opinions for a prim
- Diagnostics — where are opinions coming from and why?

99% of the time, you won't need to deal with Pcp or prim indexes directly. You'll just be using the USD API, which presents a view of the composed scenegraph. But in some cases, you may want to inspect the prim index directly. For example, you might need to figure out what assets are being referenced by a prim so you can isolate them. Or maybe you want to show a UI for authoring opinions that would affect a prim. Or maybe you just need to do some debugging and figure out where opinions are coming from. I'm going to show you a few different ways you can do this just so you have an idea of what's possible.

Inspecting the Prim Index

- “Composition” tab in usdview

The simplest way to inspect a prim index is to use the “Composition” tab in usdview. This will show you a hierarchical view of the prim index.

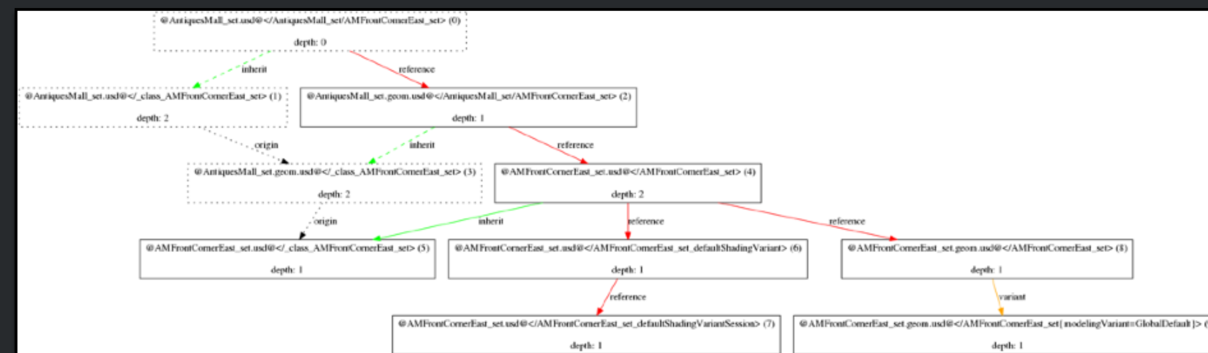
Inspecting the Prim Index

- **API on PcpPrimIndex for programmatically walking graph structure.**

```
def PrintPrimIndex(primIndex):
    def _PrintNode(node):
        print(node.arcType)
        print(node.site)
        for child in node.children:
            _PrintNode(child)

    _PrintNode(primIndex.rootNode)

PrintPrimIndex(usdviewApi.prim.GetPrimIndex())
```

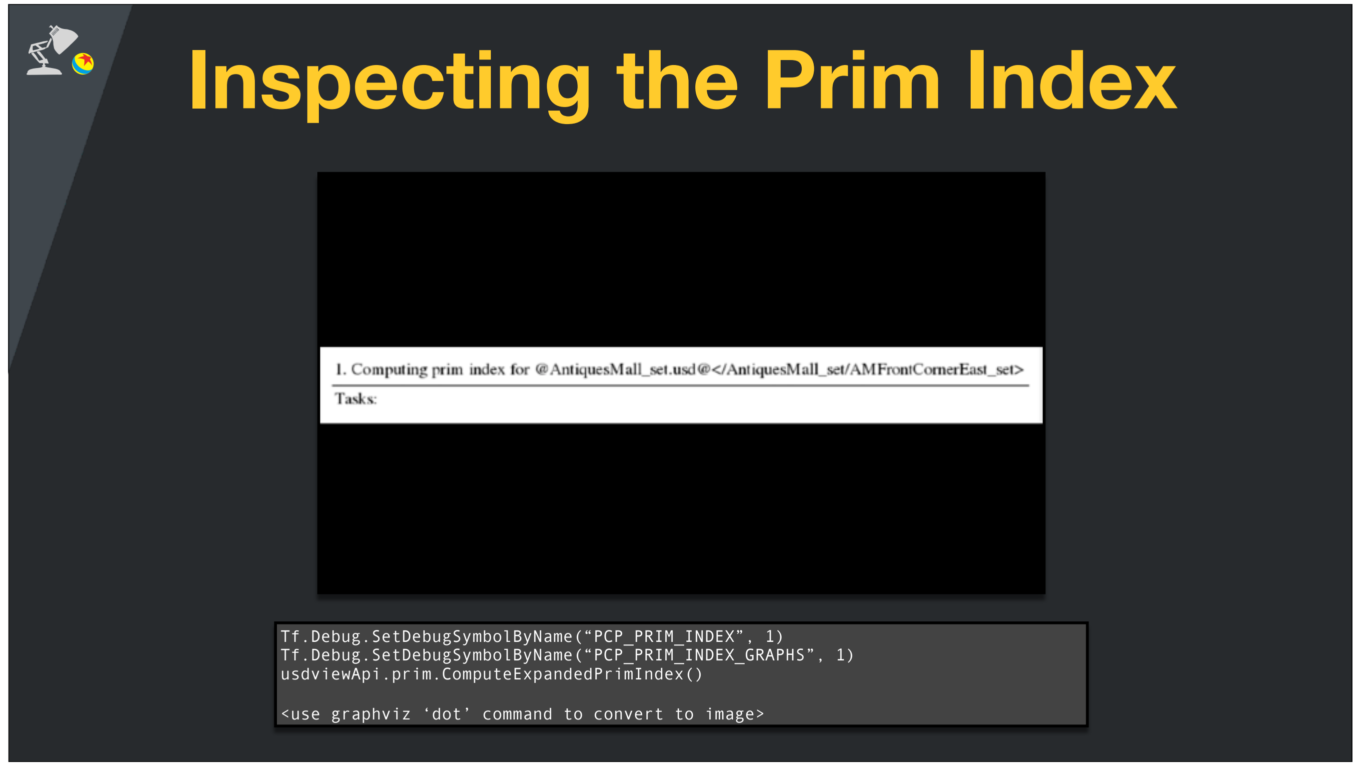
Another thing you can do is to use the Usd and PcpPrimIndex APIs to programmatically walk the graph structure and dump out information.

Inspecting the Prim Index

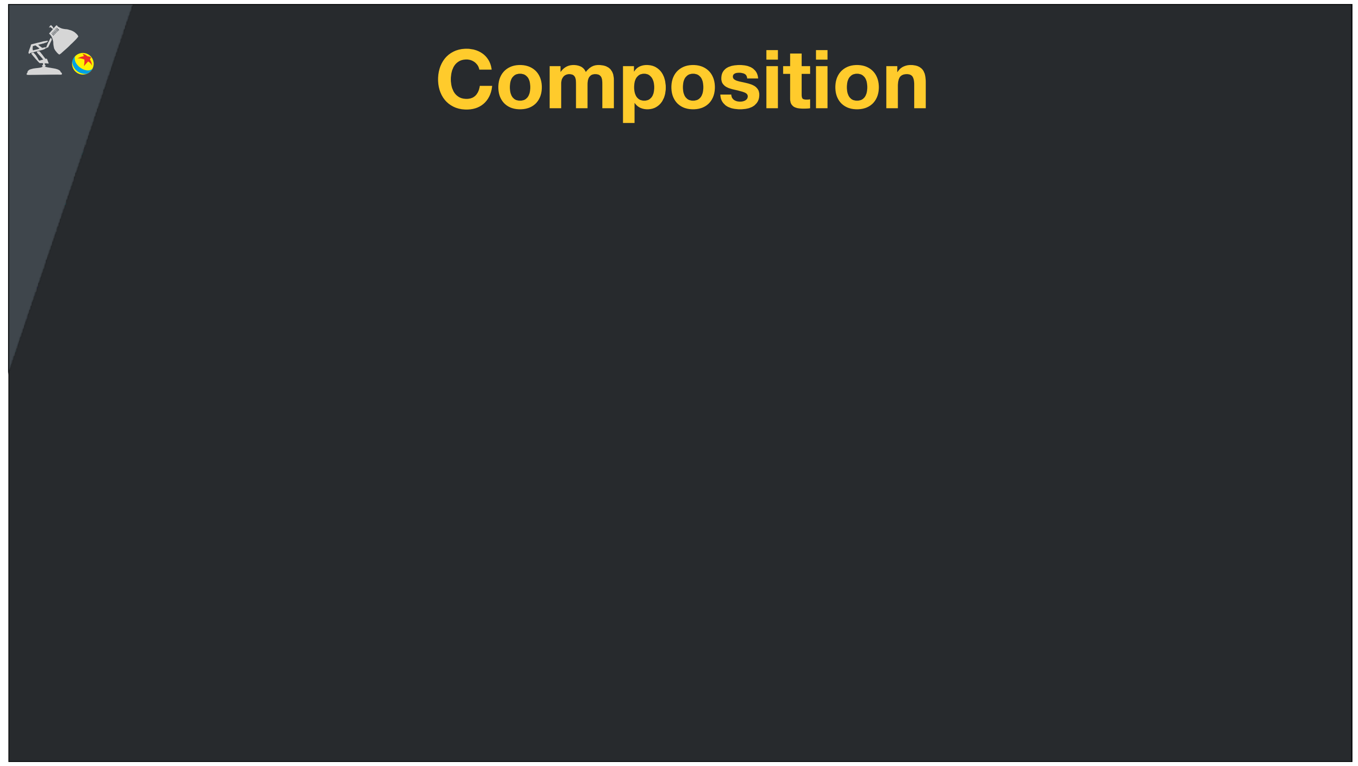



```
primIndex = usdviewApi.prim.GetPrimIndex()
primIndex.DumpToDotGraph('prim_index.dot')

<use graphviz 'dot' command to convert to image>
```

For a graphical view of the prim index, you can use API on PcpPrimIndex to dump a graphviz .dot file, which you can turn into an image like above.



And finally, for the most hard-core debugging information, you can turn on TF_DEBUG flags that will generate a .dot file for each step of composition. You can turn this into a nice image sequence for printf-style debugging of composition.



So that’s it for the composition section of this course. I’m going to turn it over to Alex now to talk about authoring and advanced features in USD.