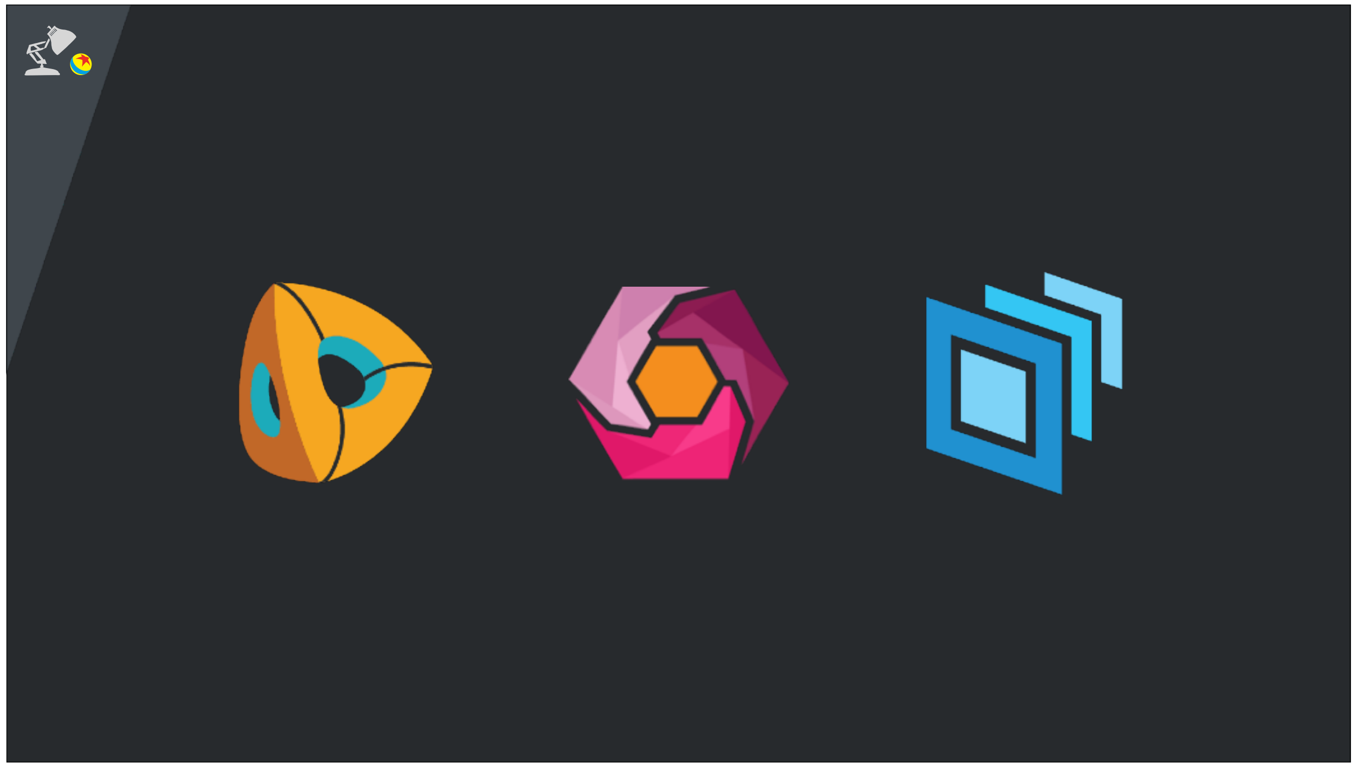
PIXAR
SOFTWARE R&D

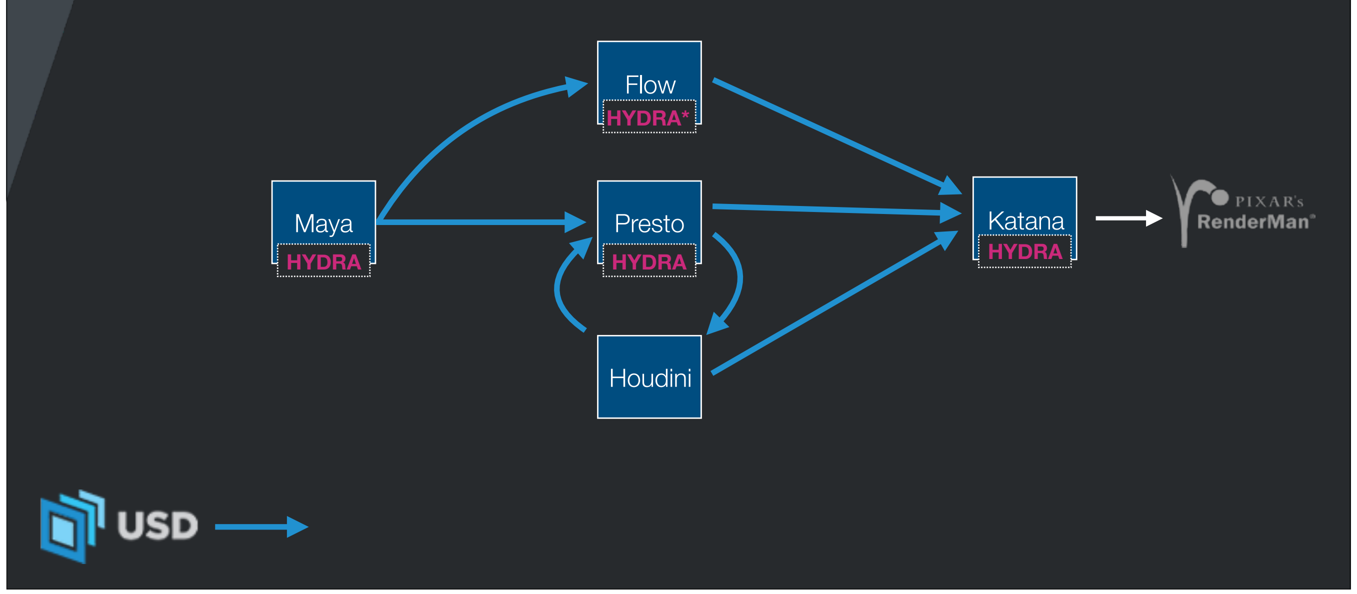
USD Introduction and Overview

George ElKoura

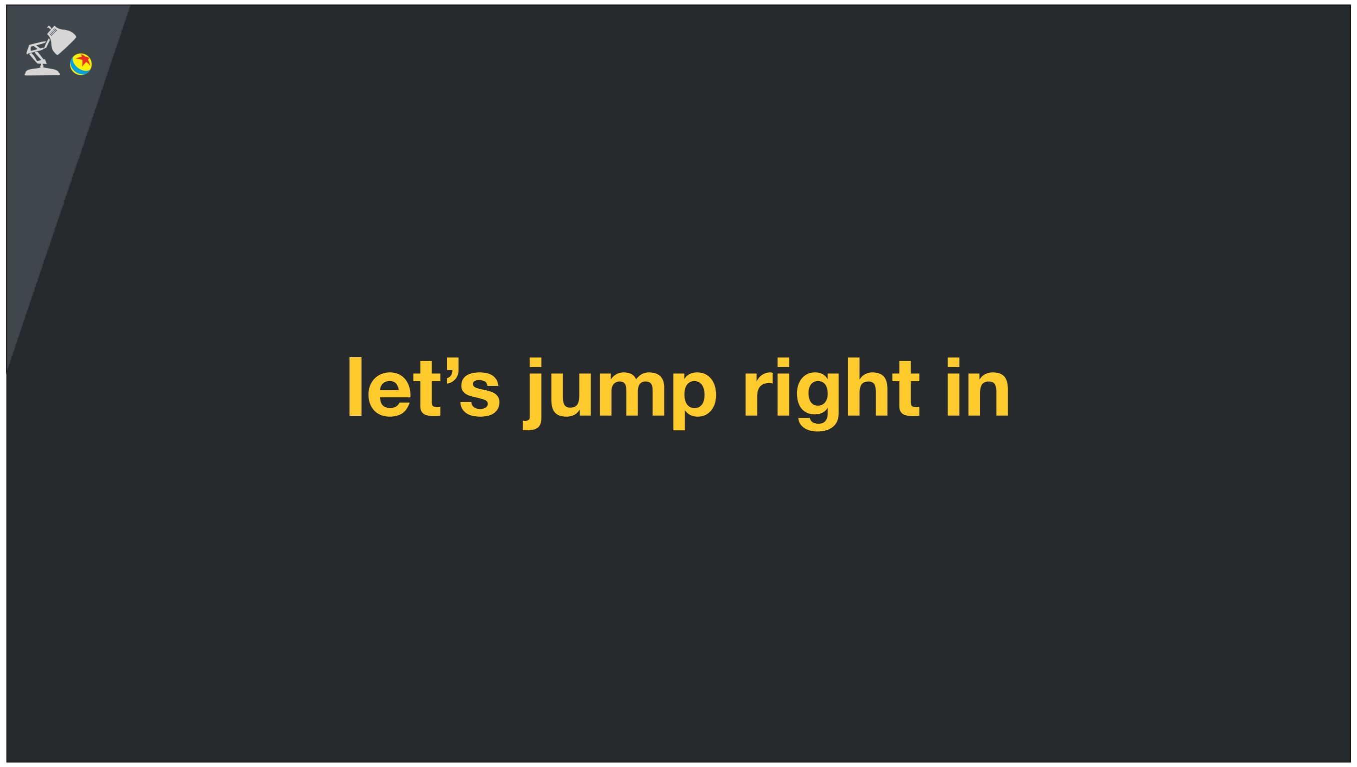


-OpenSubdiv,
-Hydra and
-USD make up the core of our 3D open source offerings at Pixar. We're going to spend the next 3 hours together talking about Hydra and USD

Software in Our Pipeline

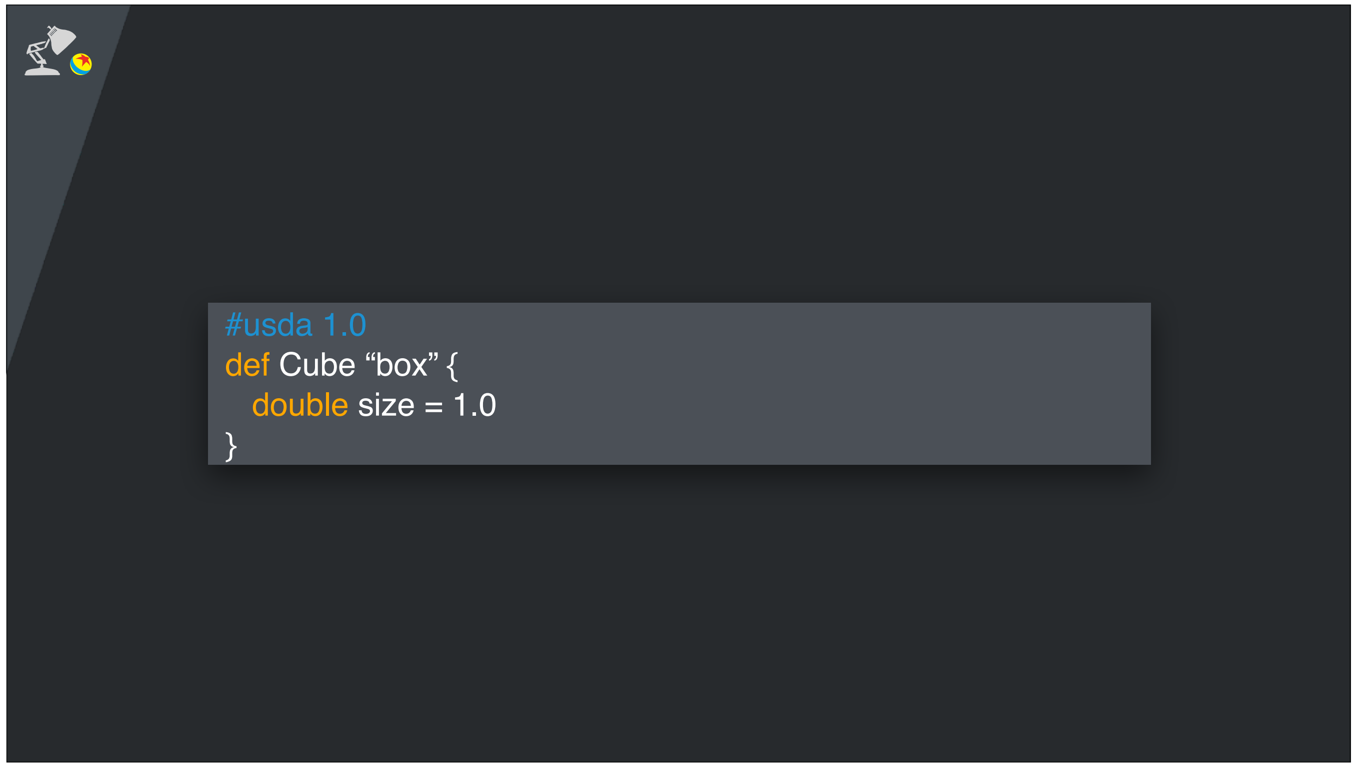


Here's a very rough and simplified overview of the software that we use in our pipeline, just to give you an idea of where these technologies fit in. Anywhere you see Hydra, you can also assume that OpenSubdiv is used at least there as well. And of course all the blue arrows show how USD is used as the main conduit in our pipeline.



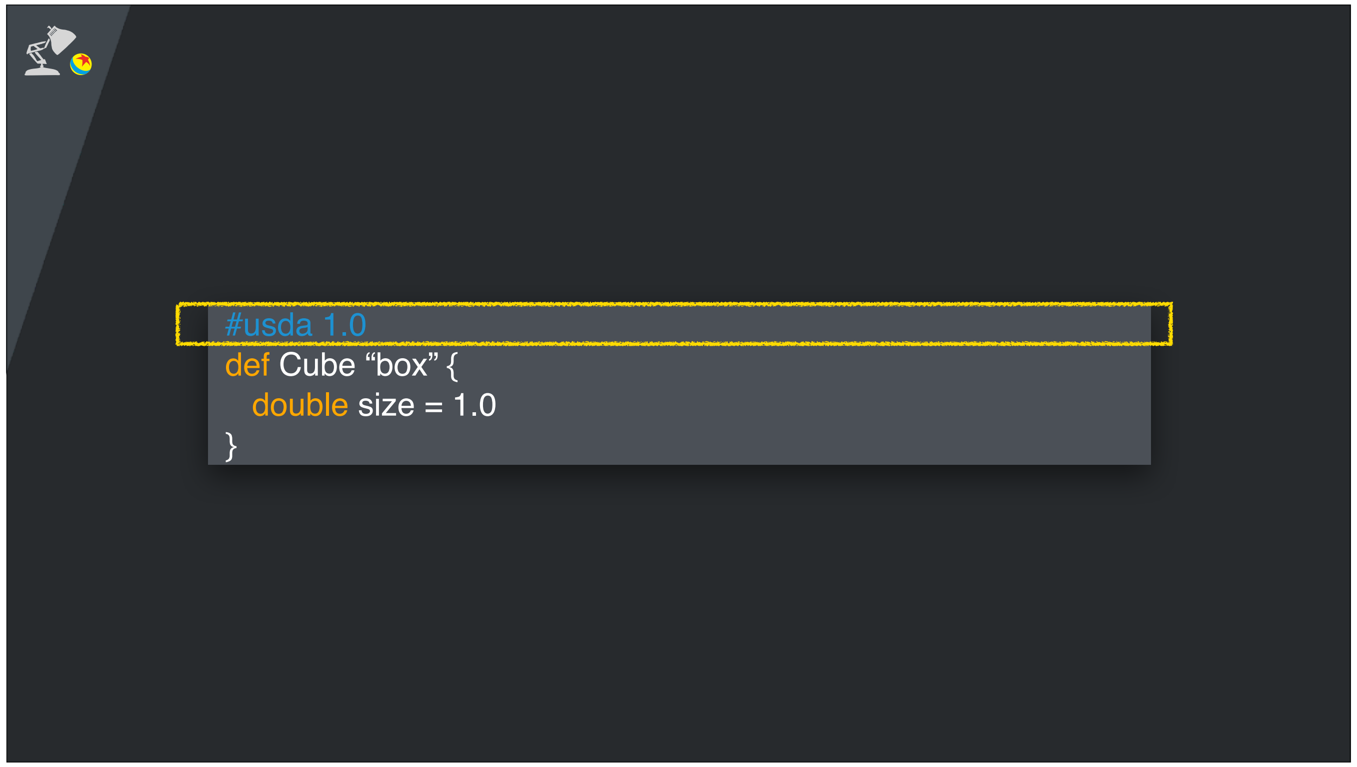
let’s jump right in

Okay now, let’s jump right in, we’ll start by looking at what it takes to represent some very simple geometry is USD.



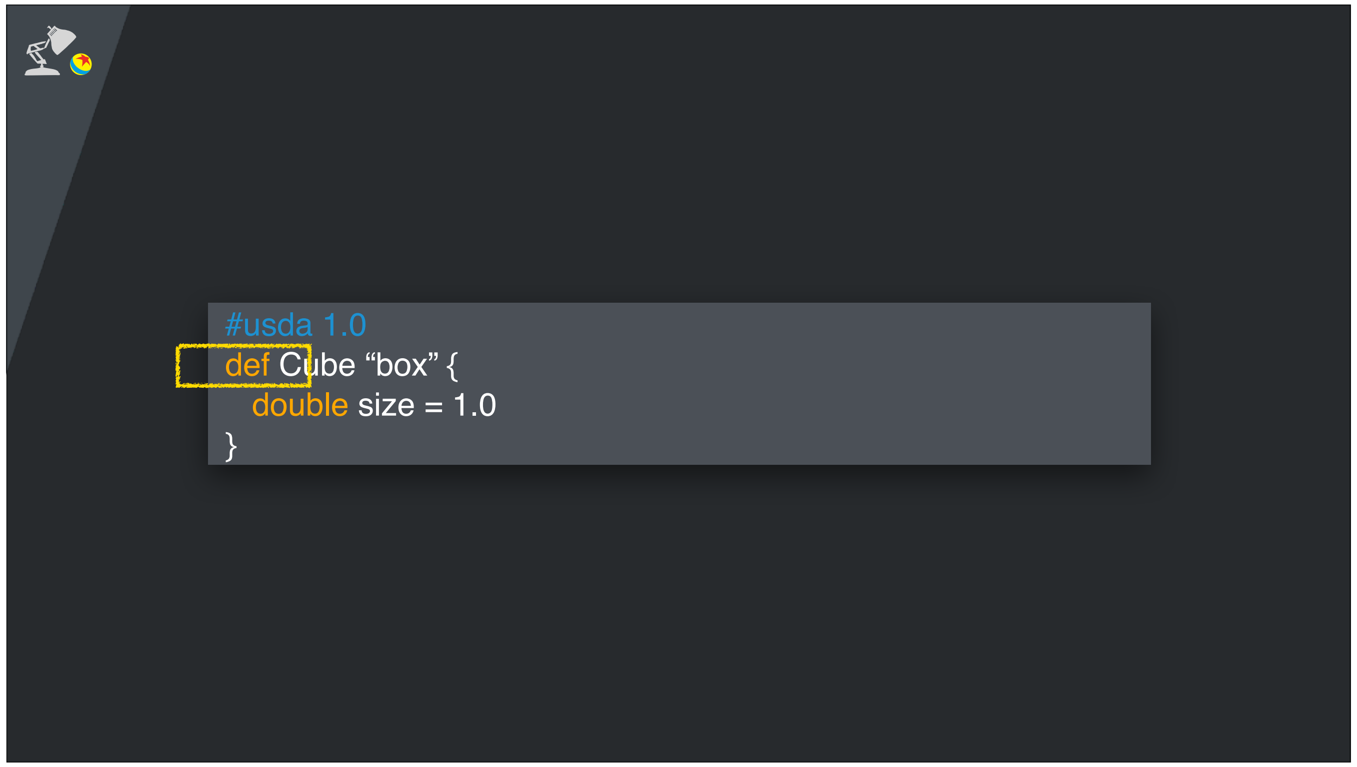
```
#usda 1.0
def Cube “box” {
   double size = 1.0
}
```

This is a unit cube — super simple, but already illustrates some concepts.



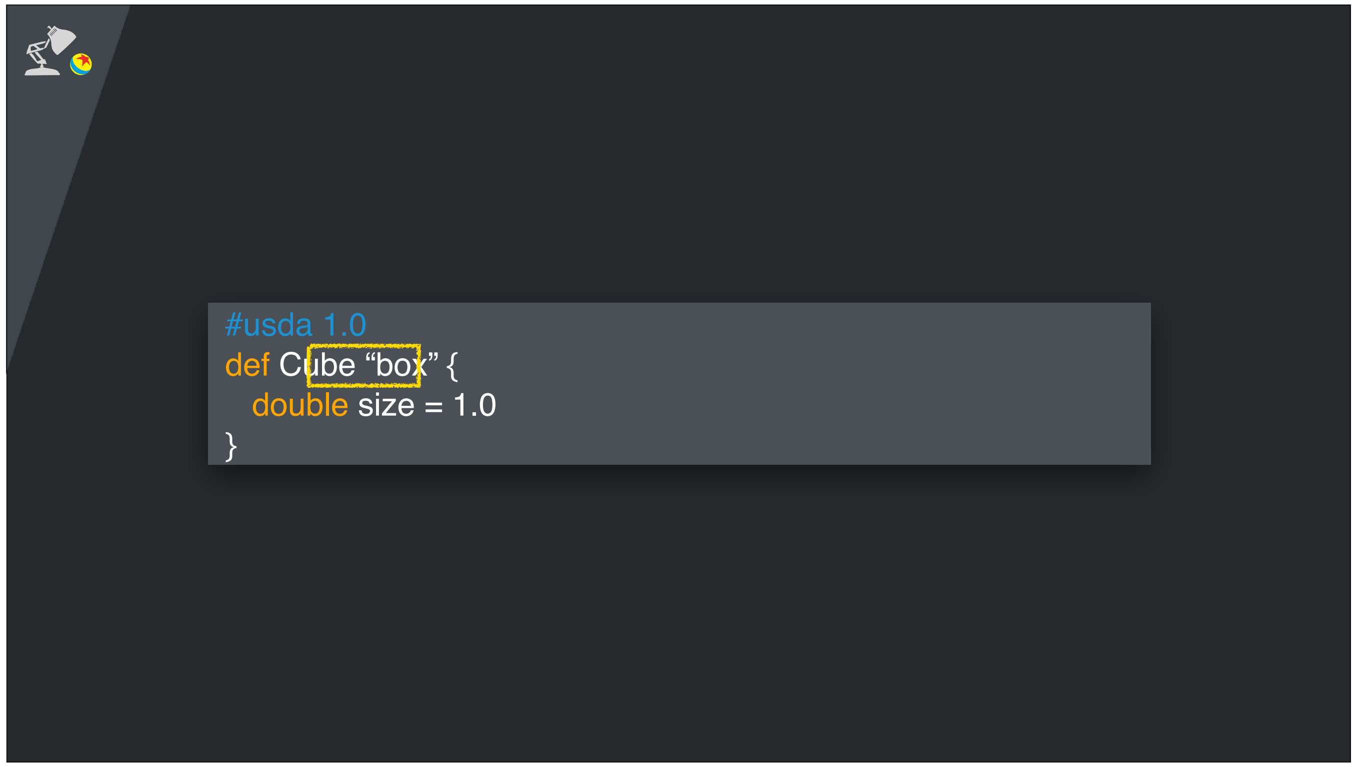
```
#usda 1.0
def Cube “box” {
   double size = 1.0
}
```

This is a file version for usda files. This is a good time to say that what we’re seeing here is a file format that ships with USD that’s super useful for representing scenes in human-readable, declarative, ASCII. Note that USD itself is not a file format, and we’ll talk more about that later.



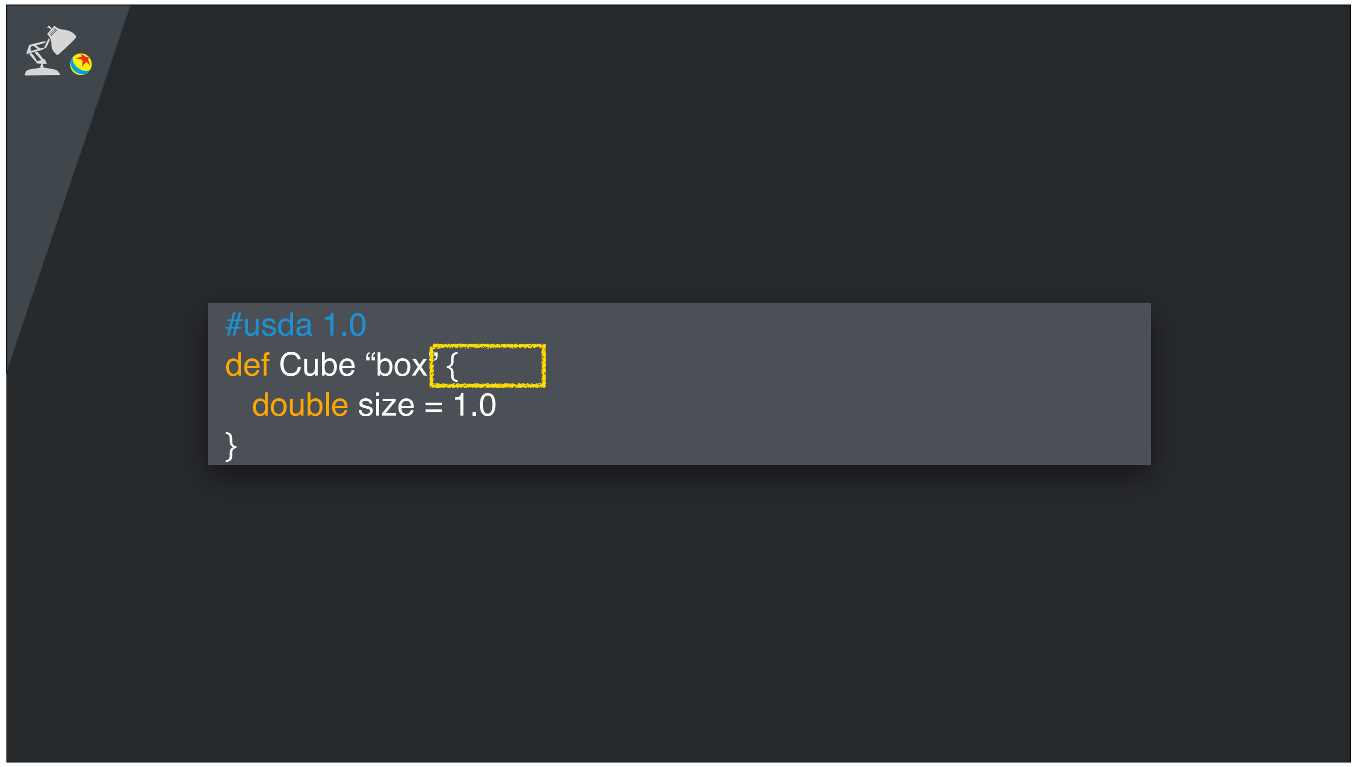
```
#usda 1.0
def Cube “box” {
   double size = 1.0
}
```

Okay the “def” here is a specifier that tells USD that we’re defining a new prim. We’ll see other specifiers too like “over” and “class” which have specific meanings and capabilities.



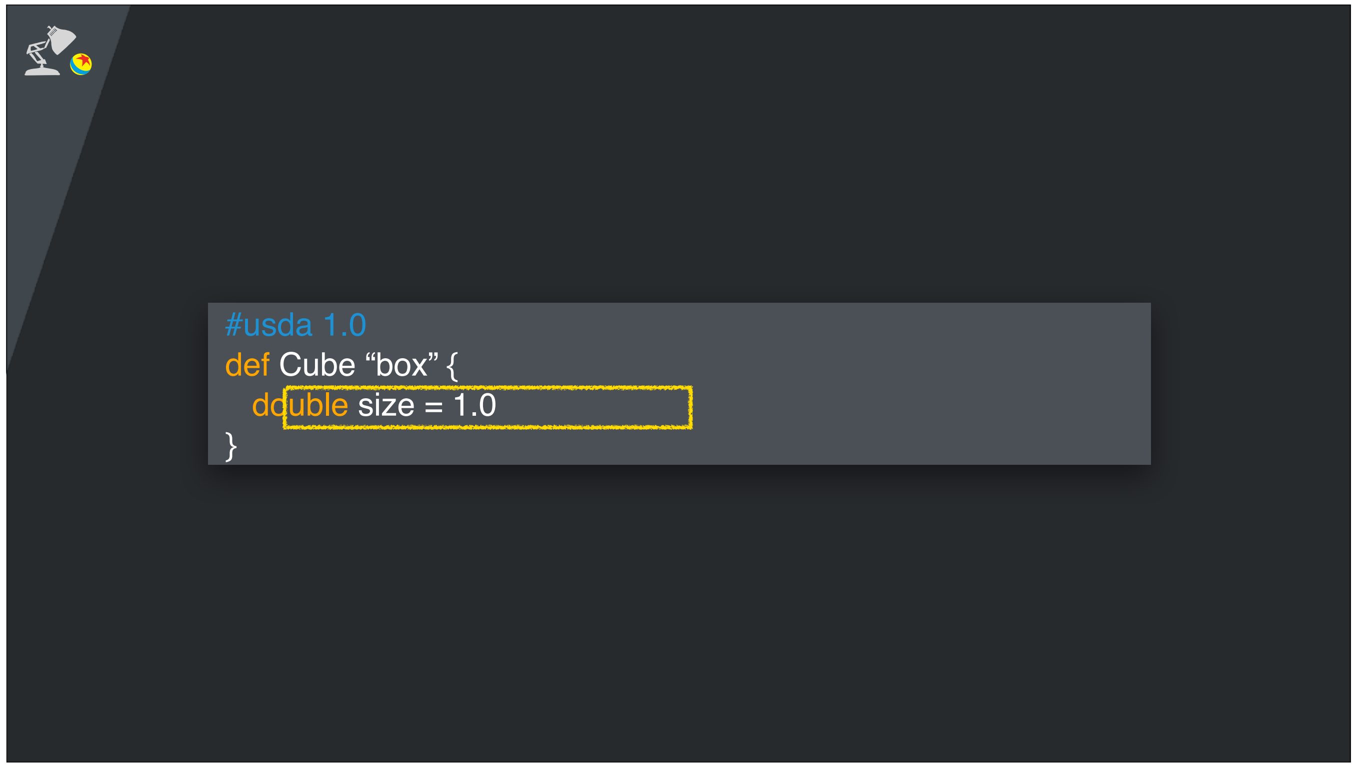
```
#usda 1.0
def Cube “box” {
   double size = 1.0
}
```

The word “Cube” here specifies the type of prim that we want. This type is described by a schema, and we’ll learn more about schemas a bit later.



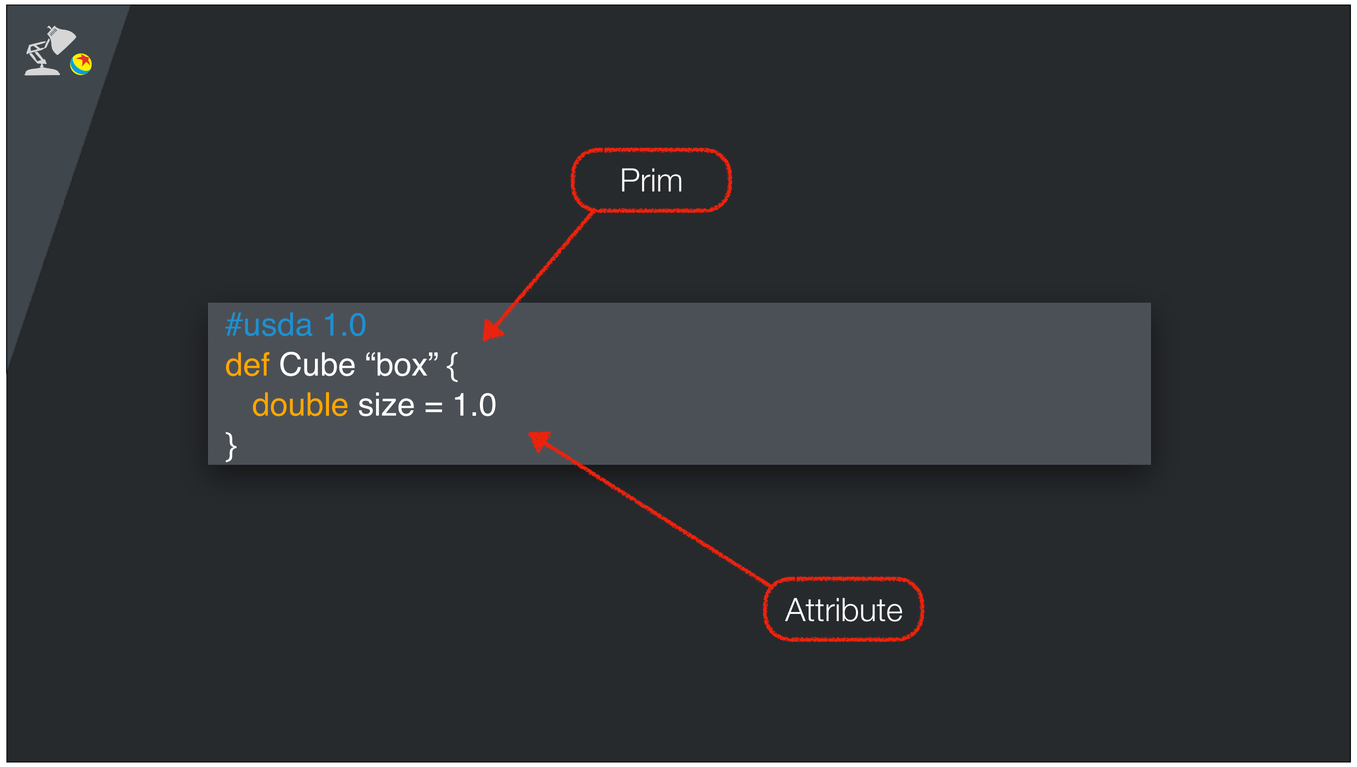
```
#usda 1.0
def Cube “box” {
   double size = 1.0
}
```

“box” is just want I want to call this cube.



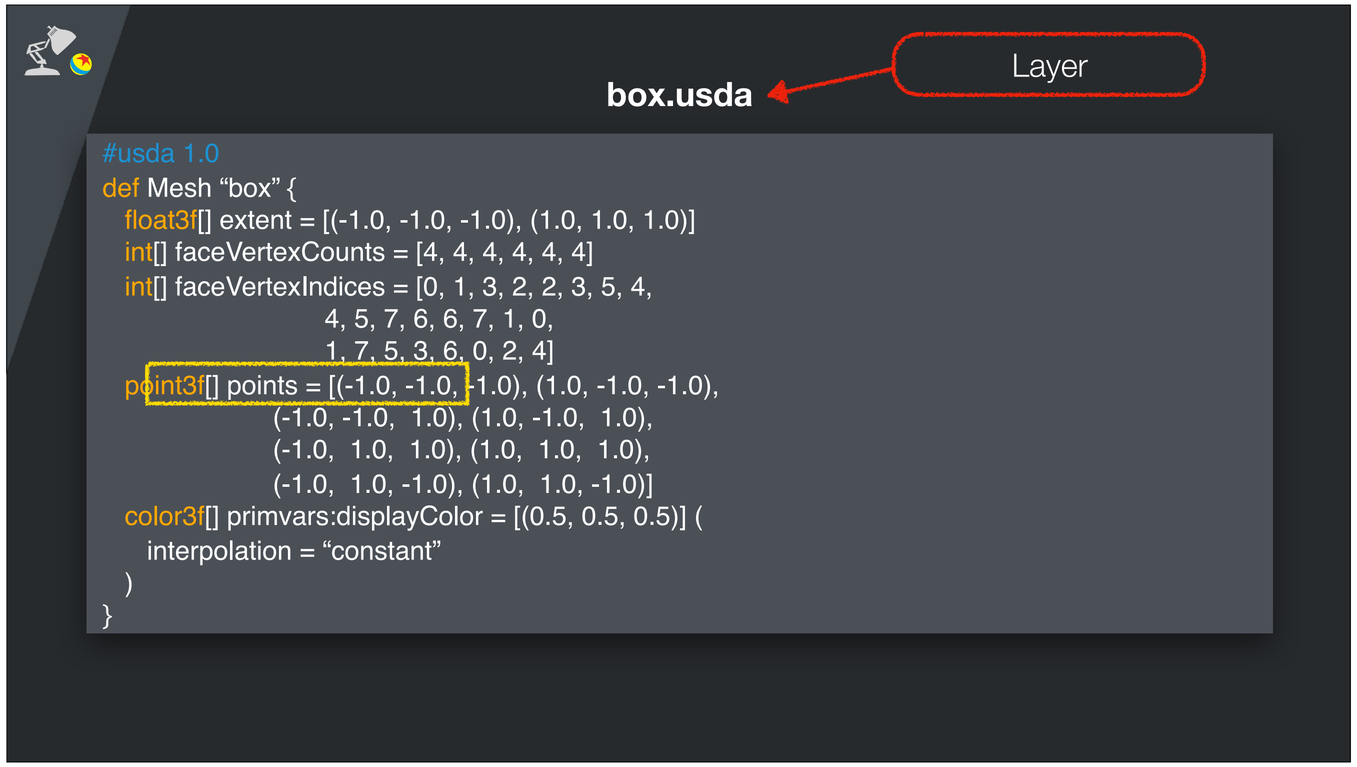
```
#usda 1.0
def Cube “box” {
   double size = 1.0
}
```

On this next line, we specify what we want the size of the cube to be. We do this with an “attribute”, and the schema defines the names, the types and other interesting things about the attributes.



```
#usda 1.0
def Cube “box” {
   double size = 1.0
}
```

To recap, this tiny usd file defines a prim and authors an opinion for the value of an attribute

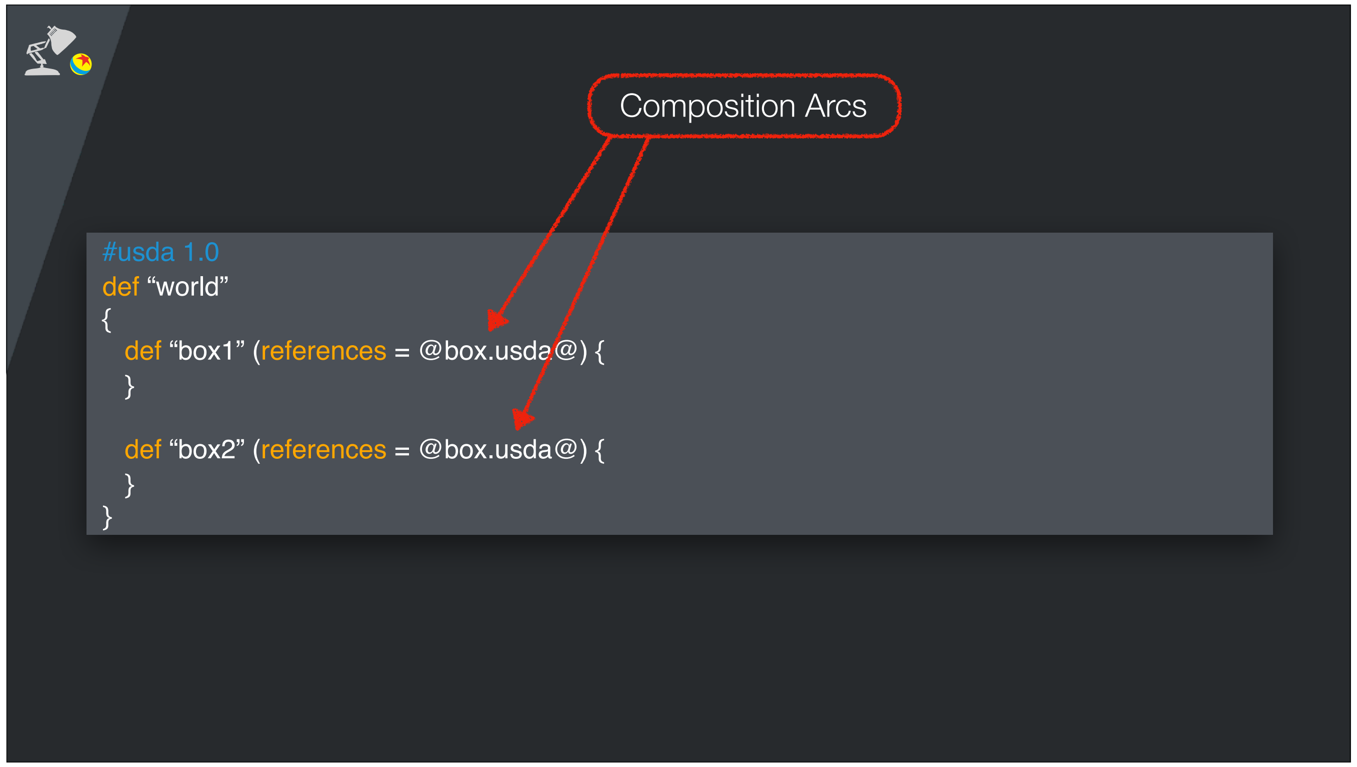


box.usda

```
#usda 1.0
def Mesh “box” {
   float3f[] extent = [(-1.0, -1.0, -1.0), (1.0, 1.0, 1.0)]
   int[] faceVertexCounts = [4, 4, 4, 4, 4, 4]
   int[] faceVertexIndices = [0, 1, 3, 2, 2, 3, 5, 4,
                              4, 5, 7, 6, 6, 7, 1, 0,
                              1, 7, 5, 3, 6, 0, 2, 4]
   point3f[] points = [(-1.0, -1.0, -1.0), (1.0, -1.0, -1.0),
                       (-1.0, -1.0,  1.0), (1.0, -1.0,  1.0),
                       (-1.0,  1.0,  1.0), (1.0,  1.0,  1.0),
                       (-1.0,  1.0, -1.0), (1.0,  1.0, -1.0)]
   color3f[] primvars:displayColor = [(0.5, 0.5, 0.5)] (
      interpolation = “constant”
   )
}
```

Okay a cube is super cool, but doesn’t give us enough to talk about. Let’s instead use a much more general mesh type that lets us represent arbitrary meshes, like we have here. This is a lot more interesting and will help us illustrate some more concepts. Here you see a few more types that are supported including arrays

- of points
- and counts and indices
- But let’s tuck all this away in
- a file we’ll call “box.usda”
- This is also called a layer, like a photoshop layer, for reasons that will become obvious throughout the course



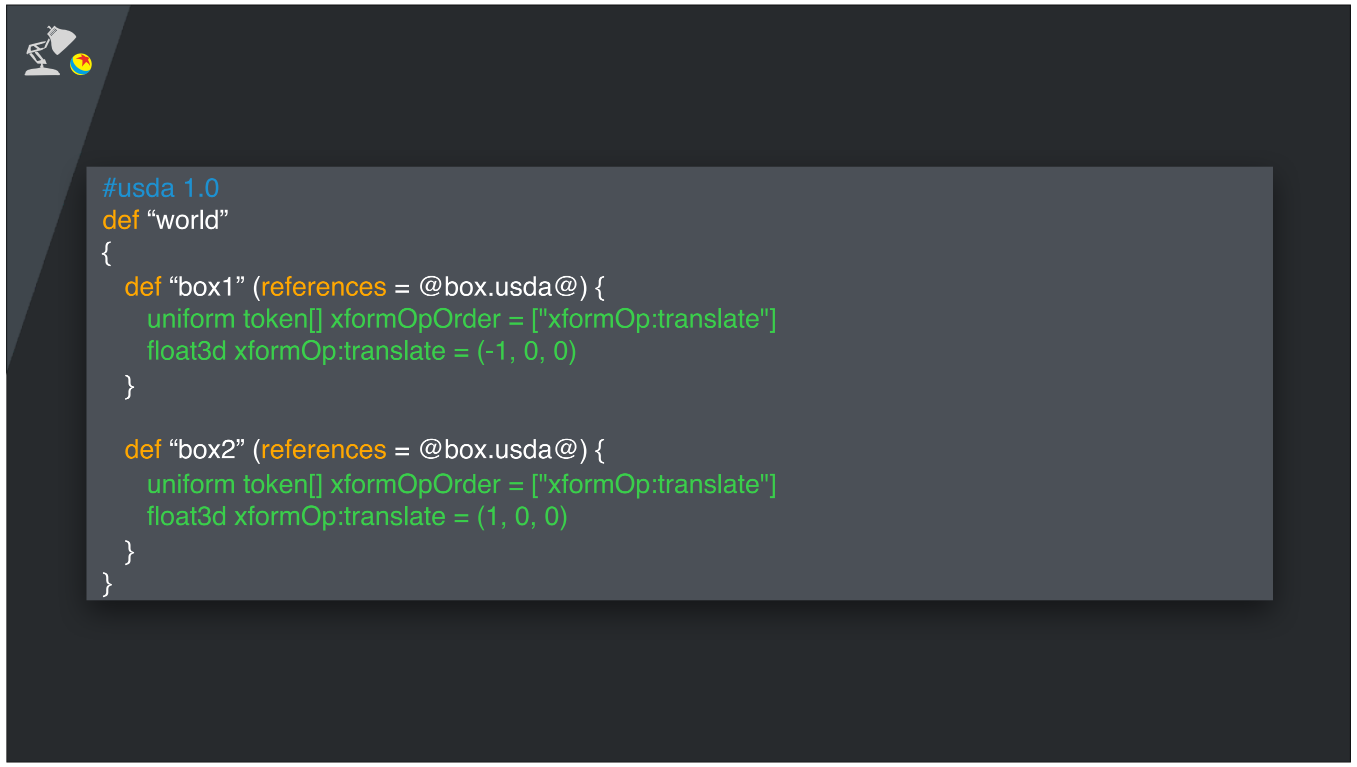
```
#usda 1.0
def “world”
{
   def “box1” (references = @box.usda@) {
   }

   def “box2” (references = @box.usda@) {
   }
}
```

Two cubes are better than one and two cubes that reference the same cube definition are even better

- The two references are an example of composition arcs, or composition operators. USD is all about these, and we’ll learn about more of them later on

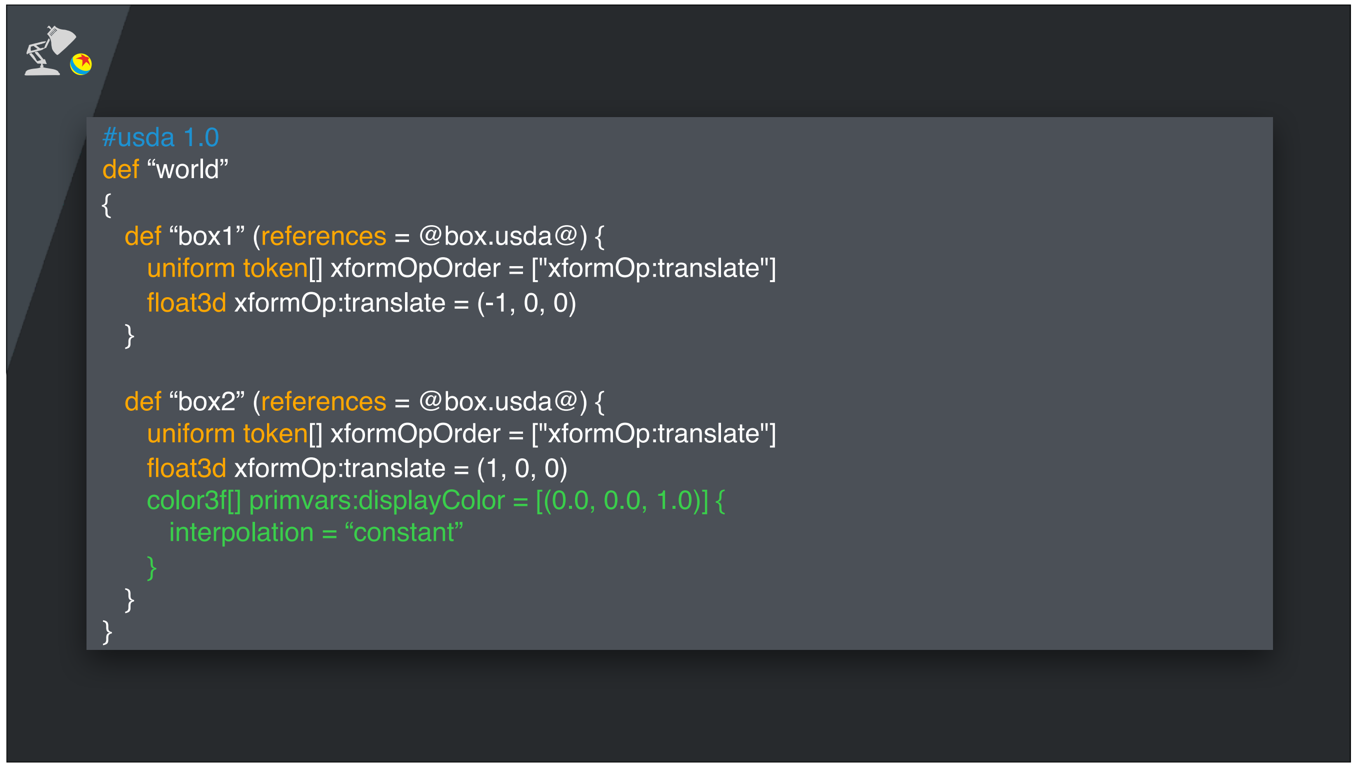
Okay now we have two boxes, but they’re on top of each other… let’s move them over a little



```
#usda 1.0
def “world”
{
   def “box1” (references = @box.usda@) {
      uniform token[] xformOpOrder = ["xformOp:translate"]
      float3d xformOp:translate = (-1, 0, 0)
   }

   def “box2” (references = @box.usda@) {
      uniform token[] xformOpOrder = ["xformOp:translate"]
      float3d xformOp:translate = (1, 0, 0)
   }
}
```

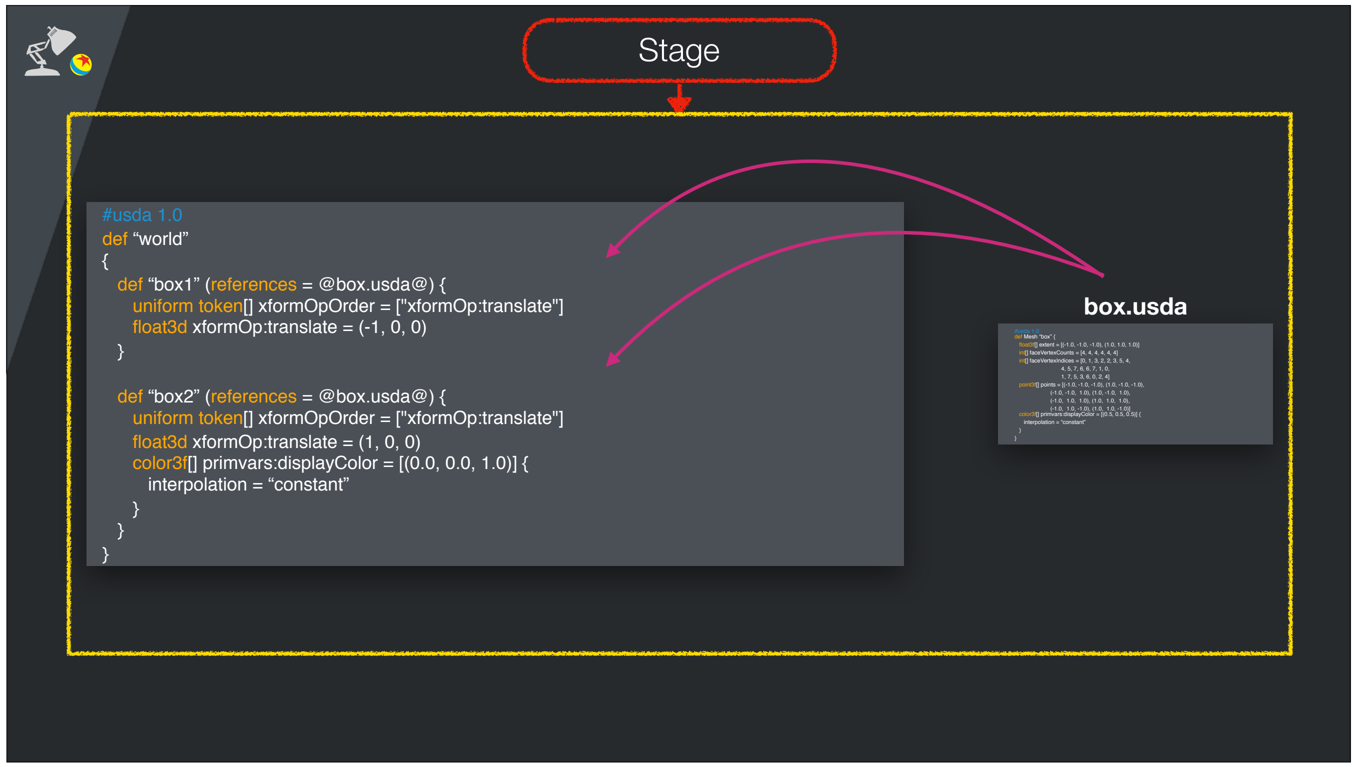
Okay now we have two of them, and they’re not overlapping, the text in green expresses opinions about how much I’d like each of them to be translated…but they’re both gray — I’d like to override the color of one of them to be blue…



```
#usda 1.0
def “world”
{
   def “box1” (references = @box.usda@) {
      uniform token[] xformOpOrder = ["xformOp:translate"]
      float3d xformOp:translate = (-1, 0, 0)
   }

   def “box2” (references = @box.usda@) {
      uniform token[] xformOpOrder = ["xformOp:translate"]
      float3d xformOp:translate = (1, 0, 0)
      color3f[] primvars:displayColor = [(0.0, 0.0, 1.0)] {
         interpolation = “constant”
      }
   }
}
```

Again, the text shown in green expresses opinions that override the attribute values from the referenced in scene.



Stage

```
#usda 1.0
def "world"
{
  def "box1" (references = @box.usda@) {
    uniform token[] xformOpOrder = ["xformOp:translate"]
    float3d xformOp:translate = (-1, 0, 0)
  }

  def "box2" (references = @box.usda@) {
    uniform token[] xformOpOrder = ["xformOp:translate"]
    float3d xformOp:translate = (1, 0, 0)
    color3f[] primvars:displayColor = [(0.0, 0.0, 1.0)] {
      interpolation = "constant"
    }
  }
}
```

box.usda

```
#usda 1.0
def Mesh "box" {
  float3f[] extent = [(-1.0, -1.0, -1.0), (1.0, 1.0, 1.0)]
  int[] faceVertexCounts = [4, 4, 4, 4, 4, 4]
  int[] faceVertexIndices = [0, 1, 3, 2, 2, 3, 5, 4,
                             4, 5, 7, 6, 6, 7, 1, 0,
                             1, 7, 5, 3, 6, 0, 2, 4]
  point3f[] points = [(-1.0, -1.0, -1.0), (1.0, -1.0, -1.0),
                      (-1.0, -1.0,  1.0), (1.0, -1.0,  1.0),
                      (-1.0,  1.0,  1.0), (1.0,  1.0,  1.0),
                      (-1.0,  1.0, -1.0), (1.0,  1.0, -1.0)]
  color3f[] primvars:displayColor = [(0.5, 0.5, 0.5)] {
    interpolation = "constant"
  }
}
```

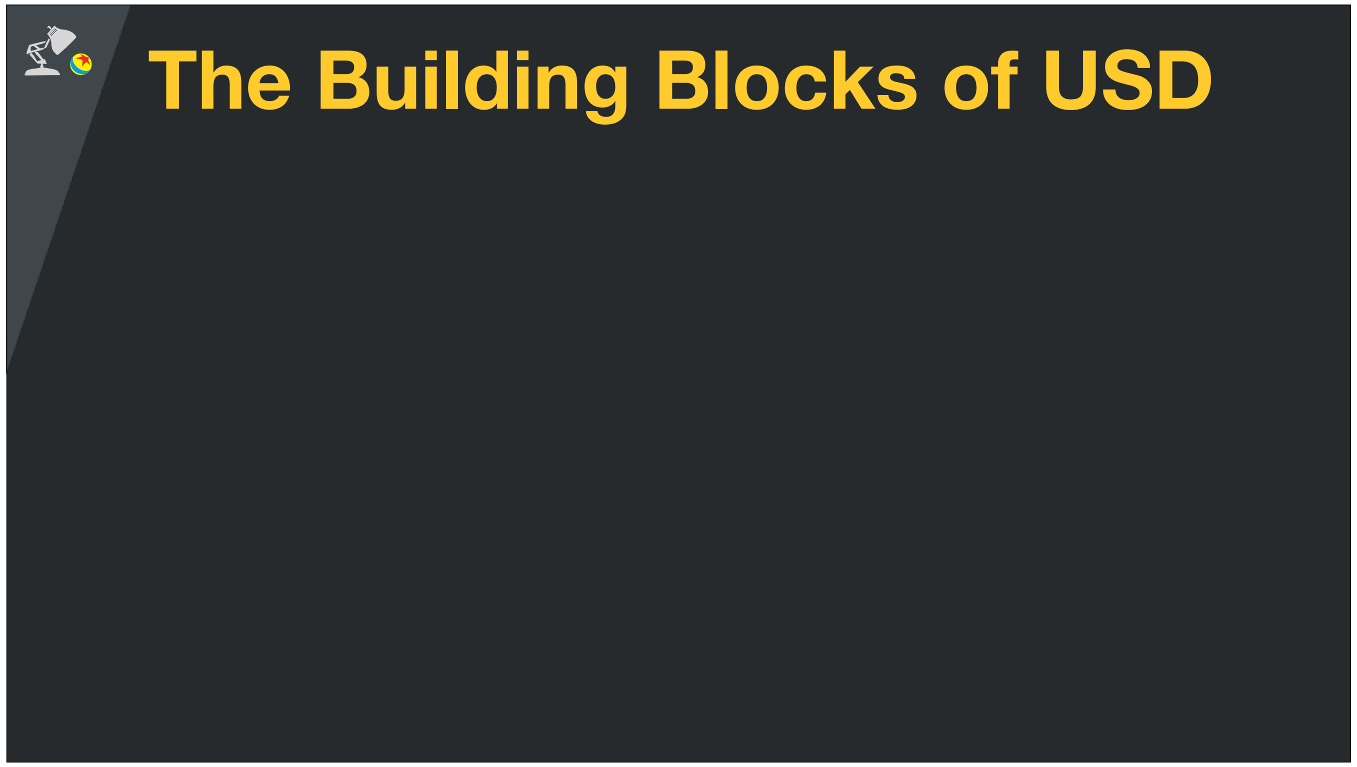
- When we resolve all the references and all the composition arcs, and the override values, we present *all* of
- those through the common main interface to USD,
- and we call it the stage.

What is USD?

Okay now we know enough to answer the question: what is USD?
- USD is the runtime engine that composes scenes and resolves scene values.
- It's used for interchange and data transfer between many applications, and it is also used as the 3D scene data format for some applications.
- Its features allow for building of full complexity scenes and allow for collaboration across many artists and studios even.

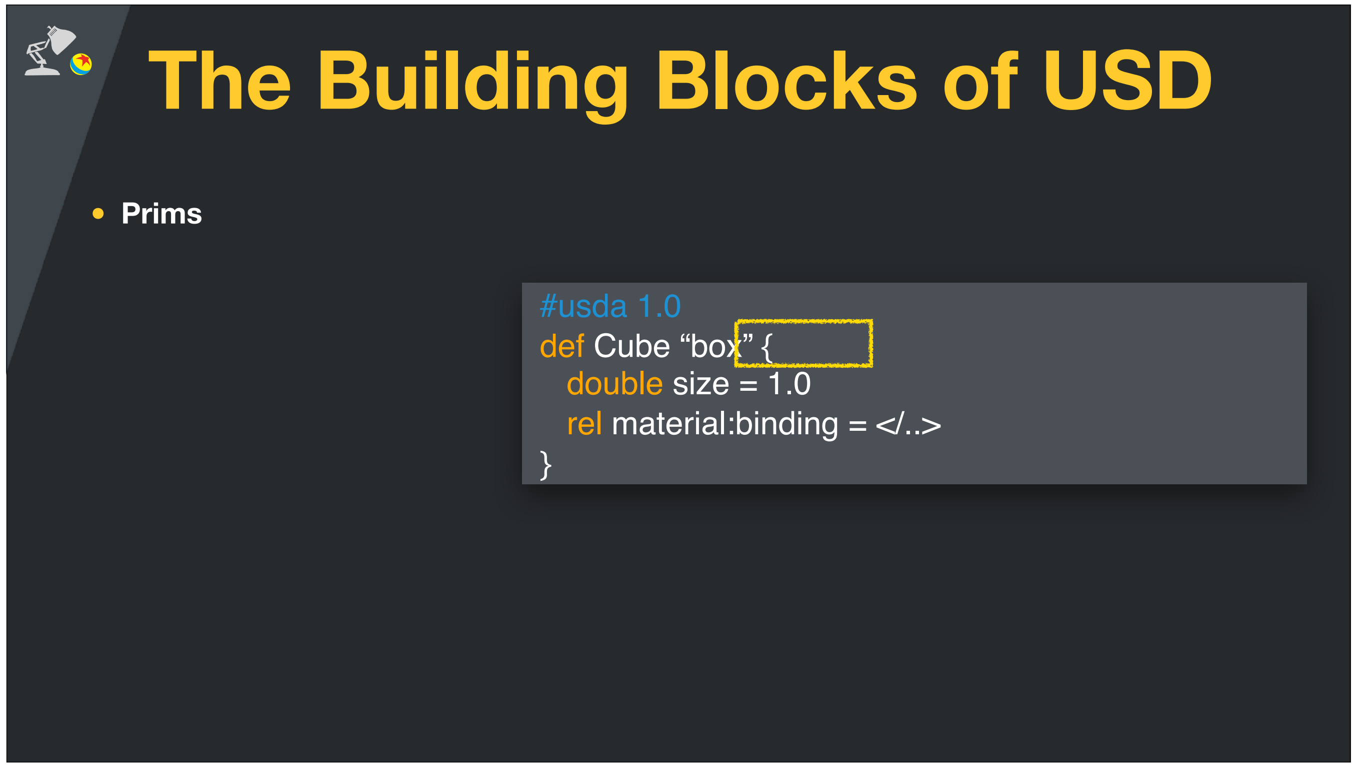
Why USD?

- USD is the result of many years of iteration at Pixar, so you can take advantage of and learn from our experience and mistakes.
- USD's direct heritage is the composition engine in Presto, our in house animation, rigging and simulation system, and marionette before that — so USD builds on the experience of many talented filmmakers over decades.
- It's also why some concepts appear complicated — there's usually a good reason, and hopefully by the time this course is over, you'll get a good sense of why things are the way they are.
- USD was primarily designed for making feature-film quality content, with all the scale that that implies.



The Building Blocks of USD

- Okay now let’s talk about the variety of objects that make up USD

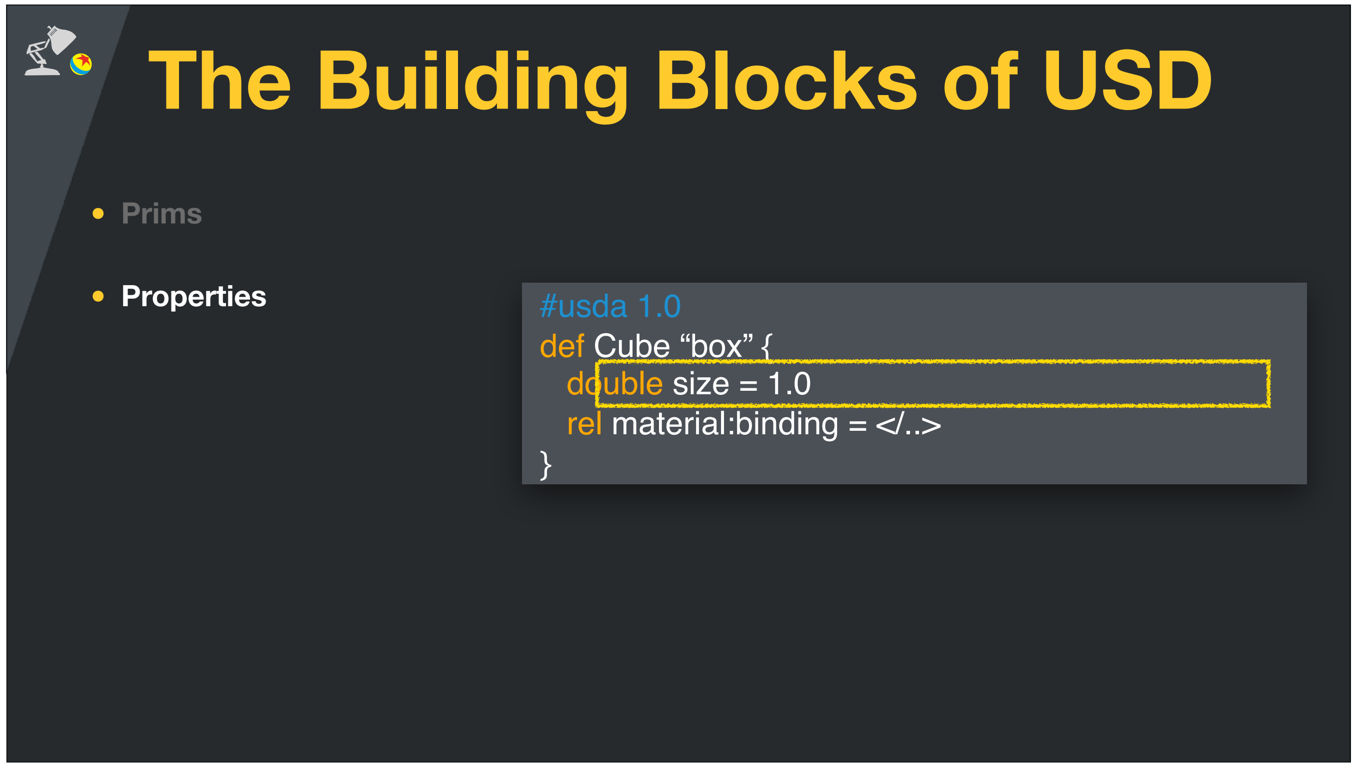


The Building Blocks of USD

- **Prims**

```
#usda 1.0
def Cube “box” {
   double size = 1.0
   rel material:binding = </..>
}
```

First we have prims, like this box, and typically prims have a type,
- in this case Cube

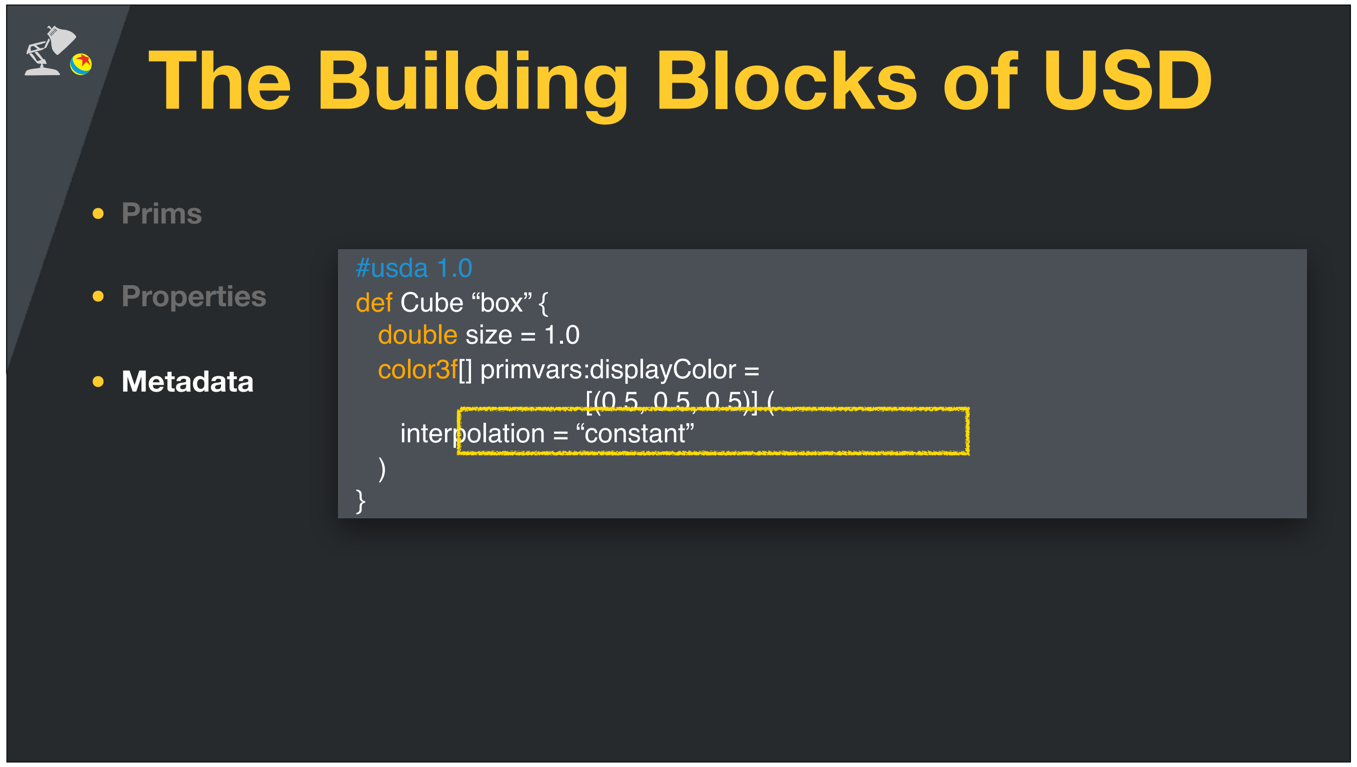


The Building Blocks of USD

- Prims
- **Properties**

```
#usda 1.0
def Cube “box” {
   double size = 1.0
   rel material:binding = </..>
}
```

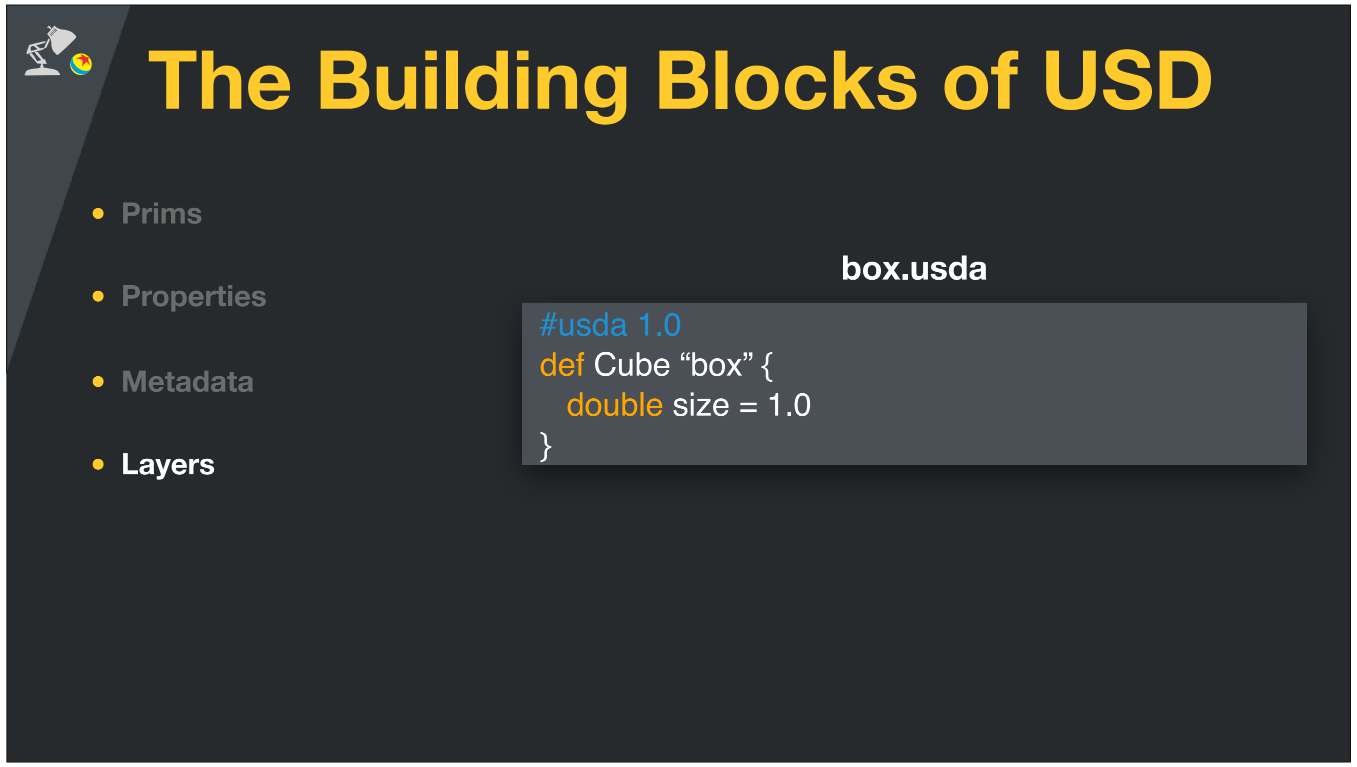
Next we have properties, we’ve looked at them before
- These include attributes like the size of a box
- And relationships, which allow for associations between prims and properties


The Building Blocks of USD

- Prims
- Properties
- **Metadata**

```
#usda 1.0
def Cube "box" {
  double size = 1.0
  color3f[] primvars:displayColor =
                          [(0.5, 0.5, 0.5)] (
    interpolation = "constant"
  )
}
```

- Next we have metadata, which are extra pieces of information that we can attach to prims and properties, and layers even. In this case we're showing the interpolation metadata on a primvar.



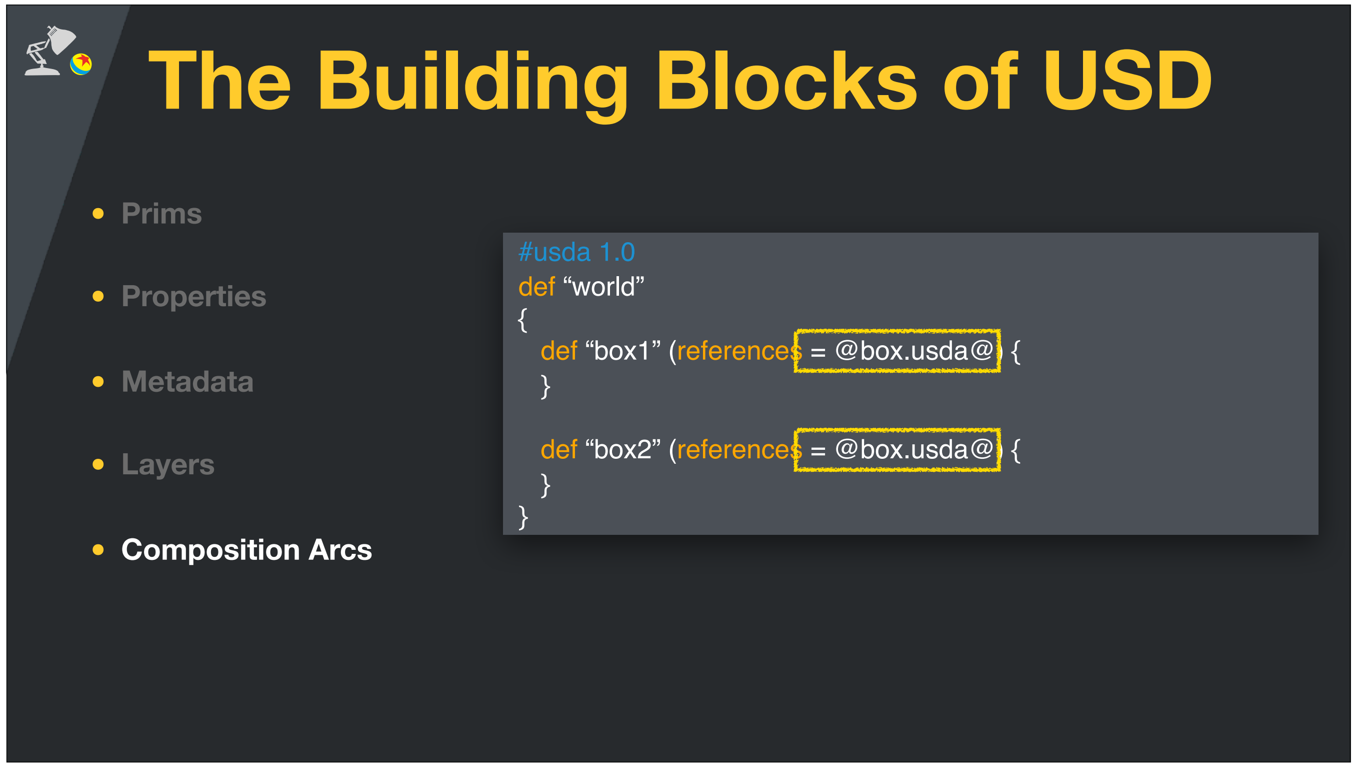
The Building Blocks of USD

- Prims
- Properties
- Metadata
- Layers

box.usda

```
#usda 1.0
def Cube “box” {
   double size = 1.0
}
```

Layers are a container for prims and properties organized hierarchically, and they are the objects that are composed and referenced by the USD composition engine.



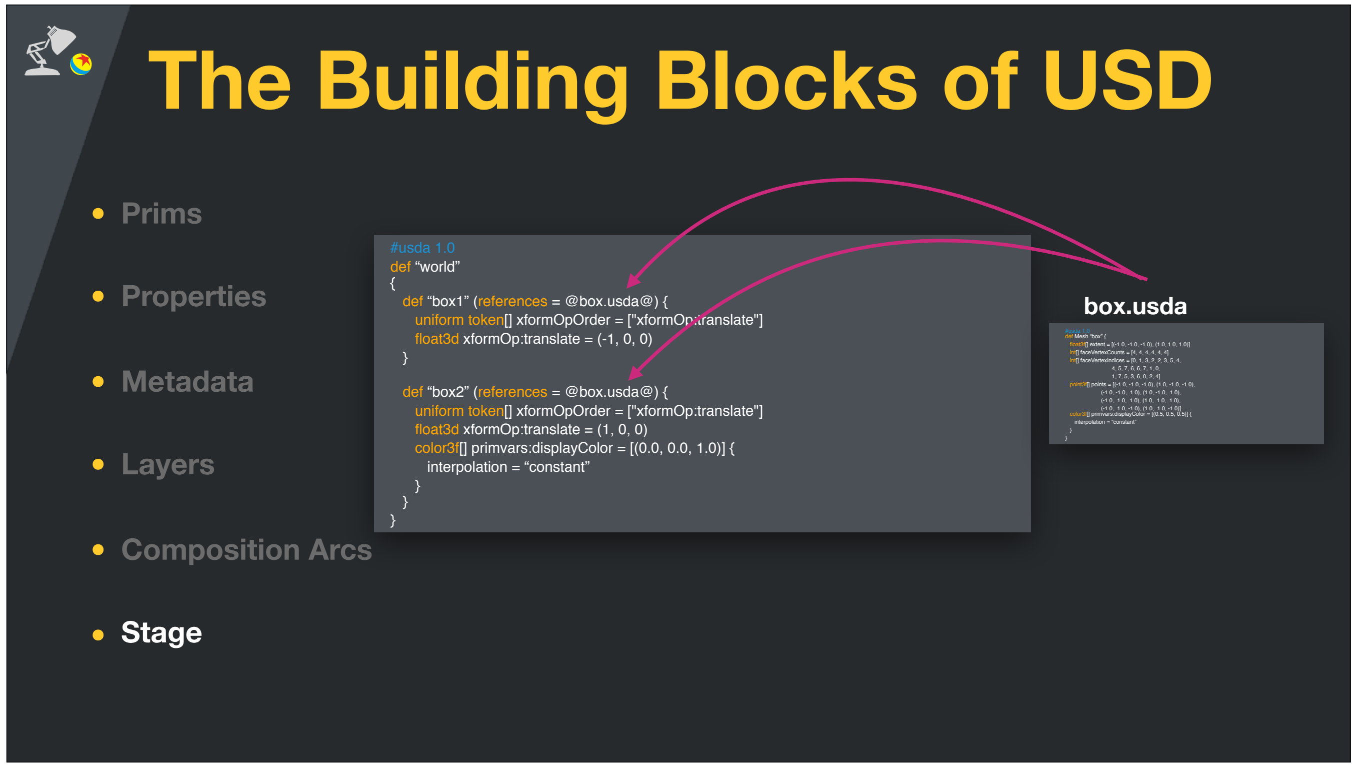
The Building Blocks of USD

- Prims
- Properties
- Metadata
- Layers
- **Composition Arcs**

```
#usda 1.0
def “world”
{
   def “box1” (references = @box.usda@) {
   }

   def “box2” (references = @box.usda@) {
   }
}
```

- Composition arcs are the operators that USD provides in order to facilitate assembly of a scene, and they are quite powerful and when used together can unlock cool workflows in your pipeline, we’ll hear much more about them from Sunya in the next section of the course.



The Building Blocks of USD

- Prims
- Properties
- Metadata
- Layers
- Composition Arcs
- **Stage**

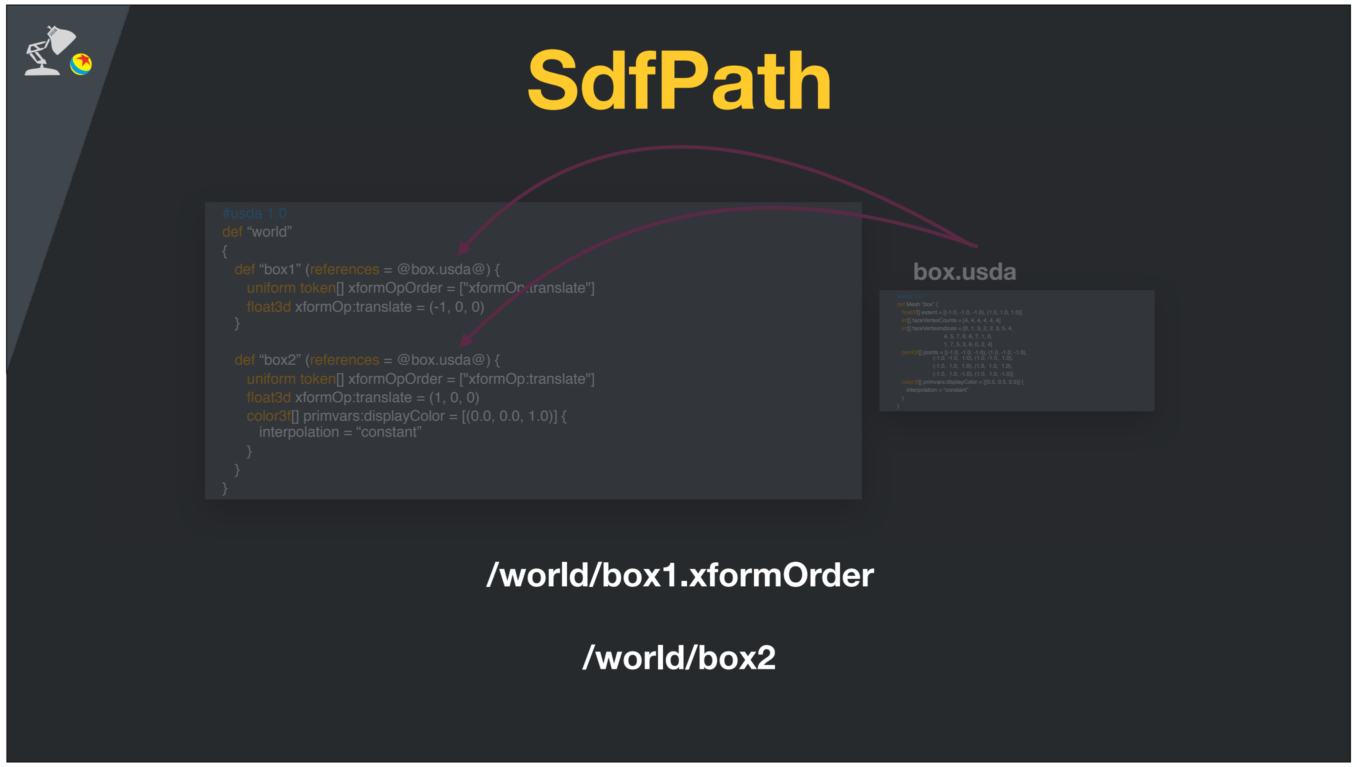
```
#usda 1.0
def "world"
{
  def "box1" (references = @box.usda@) {
    uniform token[] xformOpOrder = ["xformOp:translate"]
    float3d xformOp:translate = (-1, 0, 0)
  }

  def "box2" (references = @box.usda@) {
    uniform token[] xformOpOrder = ["xformOp:translate"]
    float3d xformOp:translate = (1, 0, 0)
    color3f[] primvars:displayColor = [(0.0, 0.0, 1.0)] {
      interpolation = "constant"
    }
  }
}
```

box.usda

```
#usda 1.0
def Mesh "box" {
  float3f[] extent = [(-1.0, -1.0, -1.0), (1.0, 1.0, 1.0)]
  int[] faceVertexCounts = [4, 4, 4, 4, 4, 4]
  int[] faceVertexIndices = [0, 1, 3, 2, 2, 3, 5, 4,
                             4, 5, 7, 6, 6, 7, 1, 0,
                             1, 7, 5, 3, 6, 0, 2, 4]
  point3f[] points = [(-1.0, -1.0, -1.0), (1.0, -1.0, -1.0),
                      (-1.0, -1.0,  1.0), (1.0, -1.0,  1.0),
                      (-1.0,  1.0,  1.0), (1.0,  1.0,  1.0),
                      (-1.0,  1.0, -1.0), (1.0,  1.0, -1.0)]
  color3f[] primvars:displayColor = [(0.5, 0.5, 0.5)] {
    interpolation = "constant"
  }
}
```

Finally, a stage is the result of composing all the layers starting at a root layer. The stage is typically the object that is queried for scene information and traversal. It gives you composed answers for prims, their position in namespace and final values for attributes and so on.



SdfPath

```
#usda 1.0
def "world"
{
  def "box1" (references = @box.usda@) {
    uniform token[] xformOpOrder = ["xformOp:translate"]
    float3d xformOp:translate = (-1, 0, 0)
  }

  def "box2" (references = @box.usda@) {
    uniform token[] xformOpOrder = ["xformOp:translate"]
    float3d xformOp:translate = (1, 0, 0)
    color3f[] primvars:displayColor = [(0.0, 0.0, 1.0)] {
      interpolation = "constant"
    }
  }
}
```

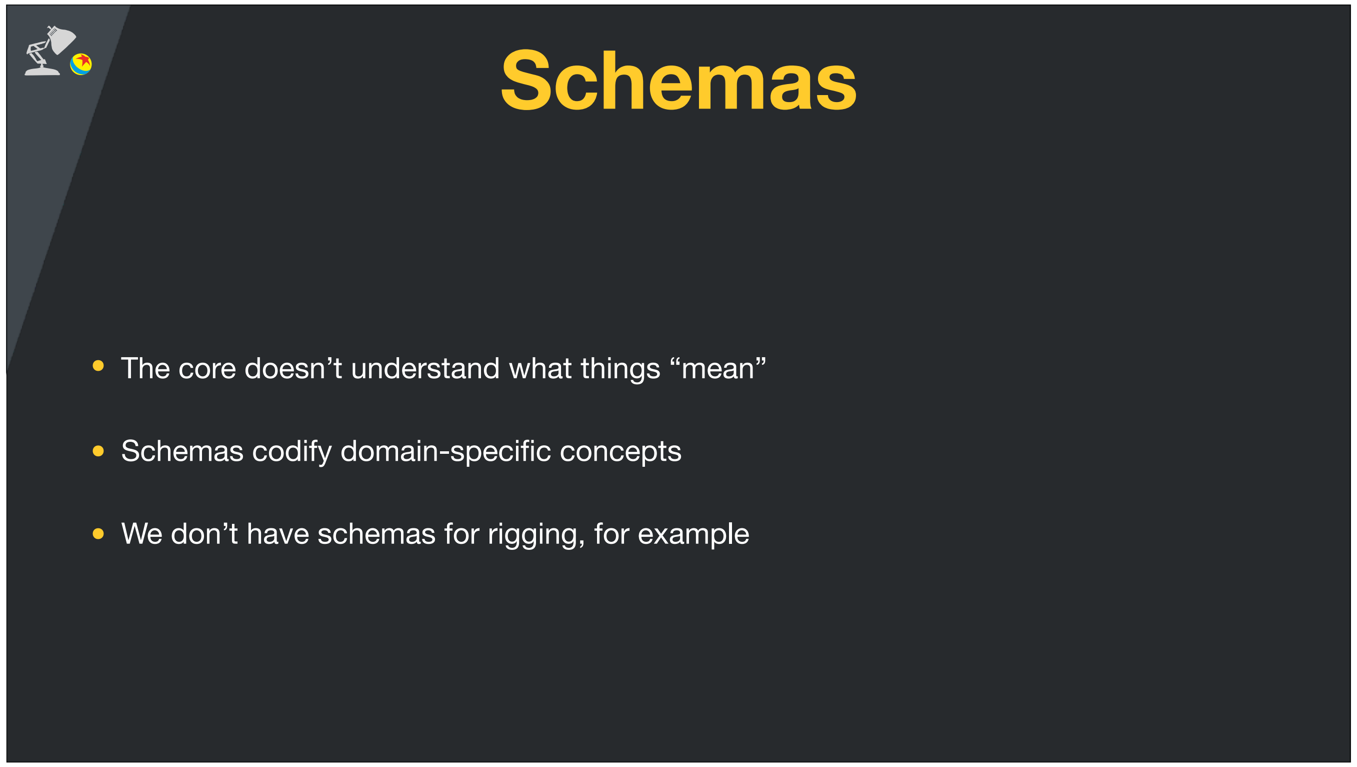
box.usda

```
#usda 1.0
def Mesh "box" {
  float3f[] extent = [(-1.0, -1.0, -1.0), (1.0, 1.0, 1.0)]
  int[] faceVertexCounts = [4, 4, 4, 4, 4, 4]
  int[] faceVertexIndices = [0, 1, 3, 2, 2, 3, 5, 4,
                             4, 5, 7, 6, 6, 7, 1, 0,
                             1, 7, 5, 3, 6, 0, 2, 4]
  point3f[] points = [(-1.0, -1.0, -1.0), (1.0, -1.0, -1.0),
                      (-1.0, -1.0,  1.0), (1.0, -1.0,  1.0),
                      (-1.0,  1.0,  1.0), (1.0,  1.0,  1.0),
                      (-1.0,  1.0, -1.0), (1.0,  1.0, -1.0)]
  color3f[] primvars:displayColor = [(0.5, 0.5, 0.5)] {
    interpolation = "constant"
  }
}
```

/world/box1.xformOrder

/world/box2

One thing worth mentioning is that prims and properties are namespace objects
- And in USD, they are addressed through an "SdfPath". Here are two examples, the first shows the path to the xformOrder attribute and the second shows a path to the box2 prim. SdfPaths are a main component of the USD and Hydra APIs



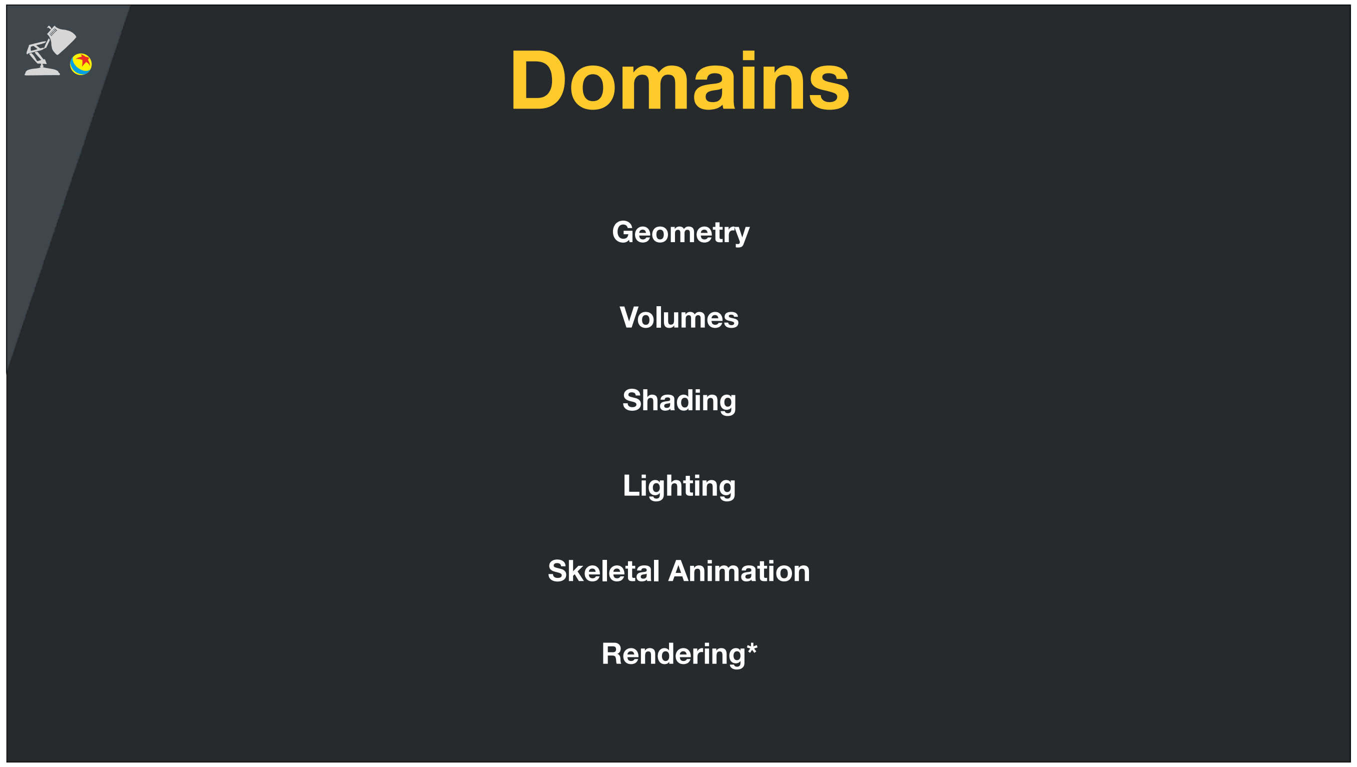
Schemas

- The core doesn't understand what things "mean"
- Schemas codify domain-specific concepts
- We don't have schemas for rigging, for example

Another thing I barely touched on in the earlier slides, but is important to understand is Schemas, you'll hear us talk about schemas a lot. Basically a schema is a collection of properties that give a prim meaning.

- The core itself doesn't understand what any of the properties and prims mean. Schemas give them meaning, for example if a prim represents a mesh, or a cube as we saw earlier, there are schemas that back them up to define exactly which properties are required.
- We have lots of examples of schemas as part of USD, and I'll cover the areas in the next slide
- And there are areas where we don't have schemas, for example rigging. This is because we initially set out to solve the interchange problem, and rigging is a bit daunting for interchange (and well, baby steps), but there's no technical reason why USD couldn't encode rigging schemas for example.

Okay what do we support?



Domains

Geometry

Volumes

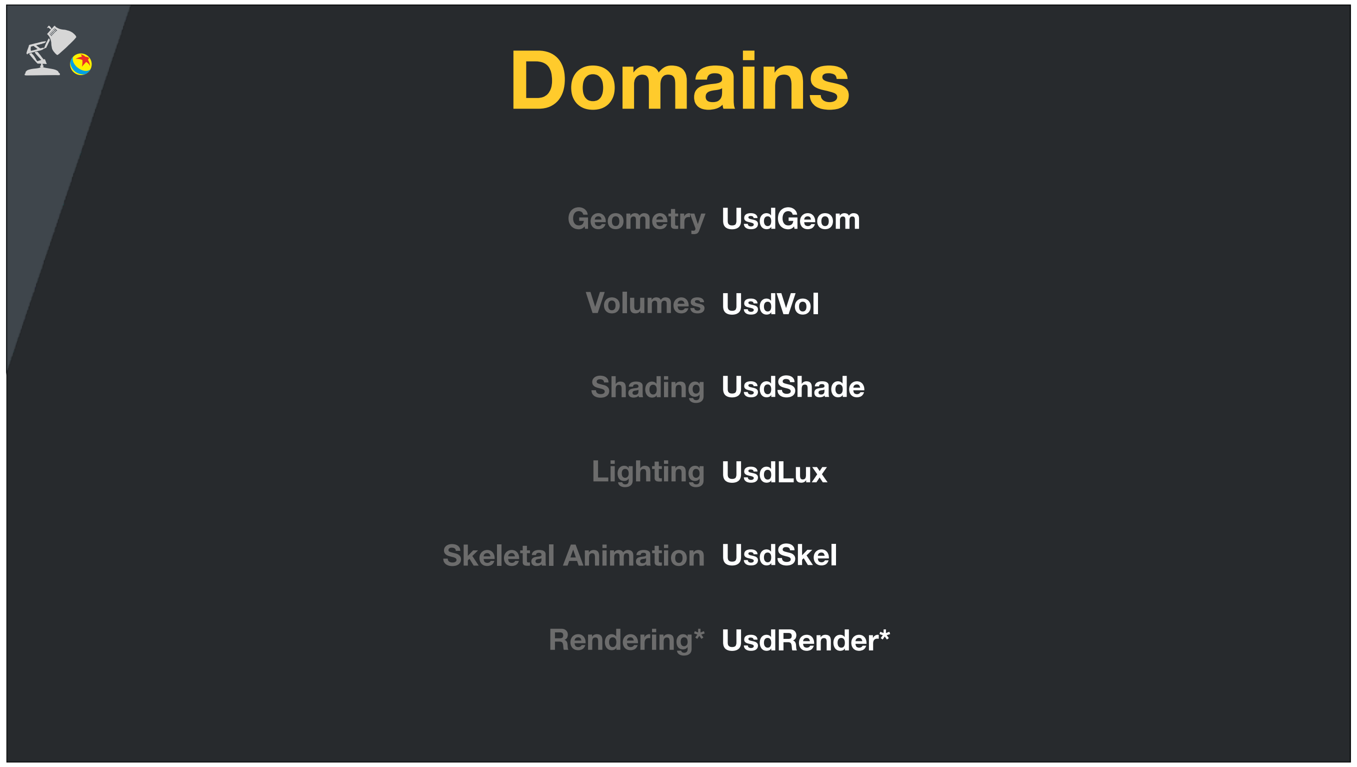
Shading

Lighting

Skeletal Animation

Rendering*

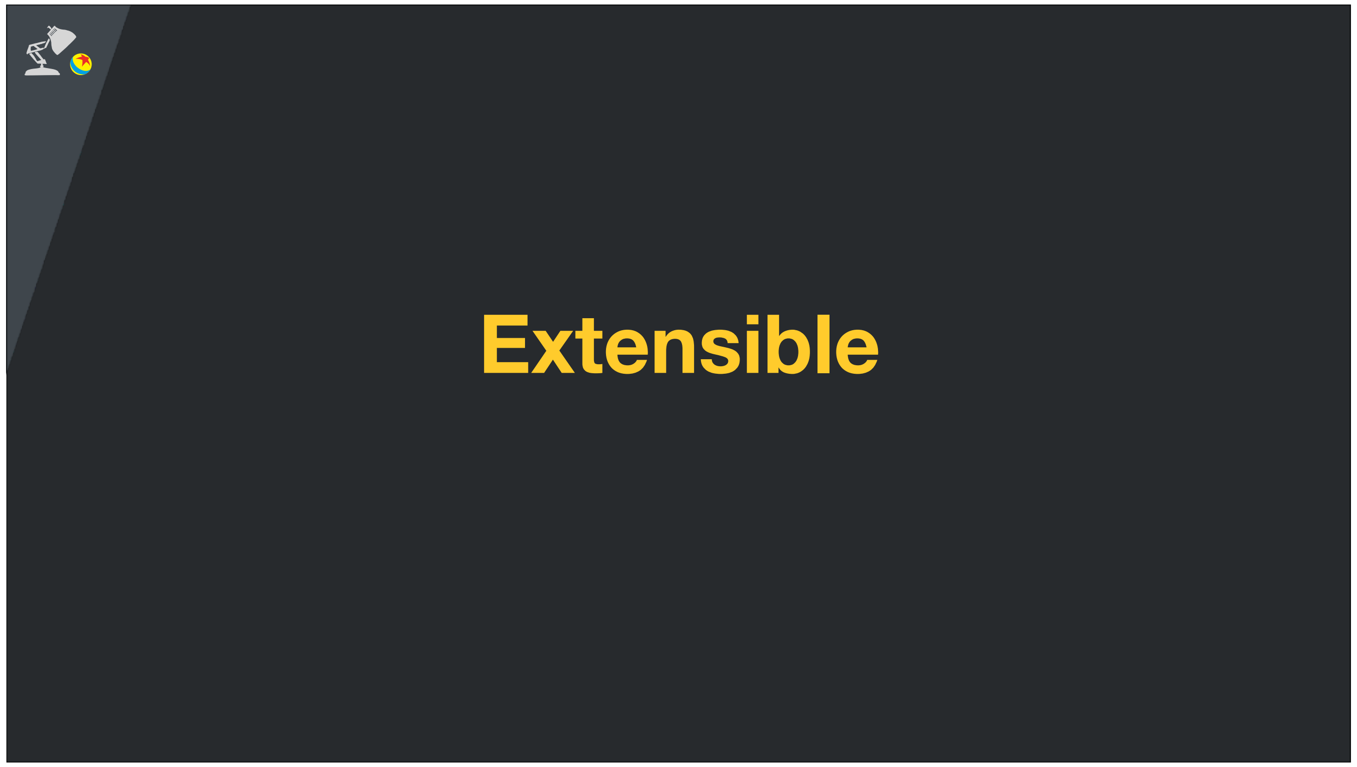
Here are the domains that we currently support and we occasionally add more domains (like Rendering is one that we're actively working on) and we also extend existing domains, for example, we recently added blend shapes to our skeletal animation schemas.



Domains

Geometry	UsdGeom
Volumes	UsdVol
Shading	UsdShade
Lighting	UsdLux
Skeletal Animation	UsdSkel
Rendering*	UsdRender*

And here are the libraries they correspond to, we'll often reference these domains by their library names

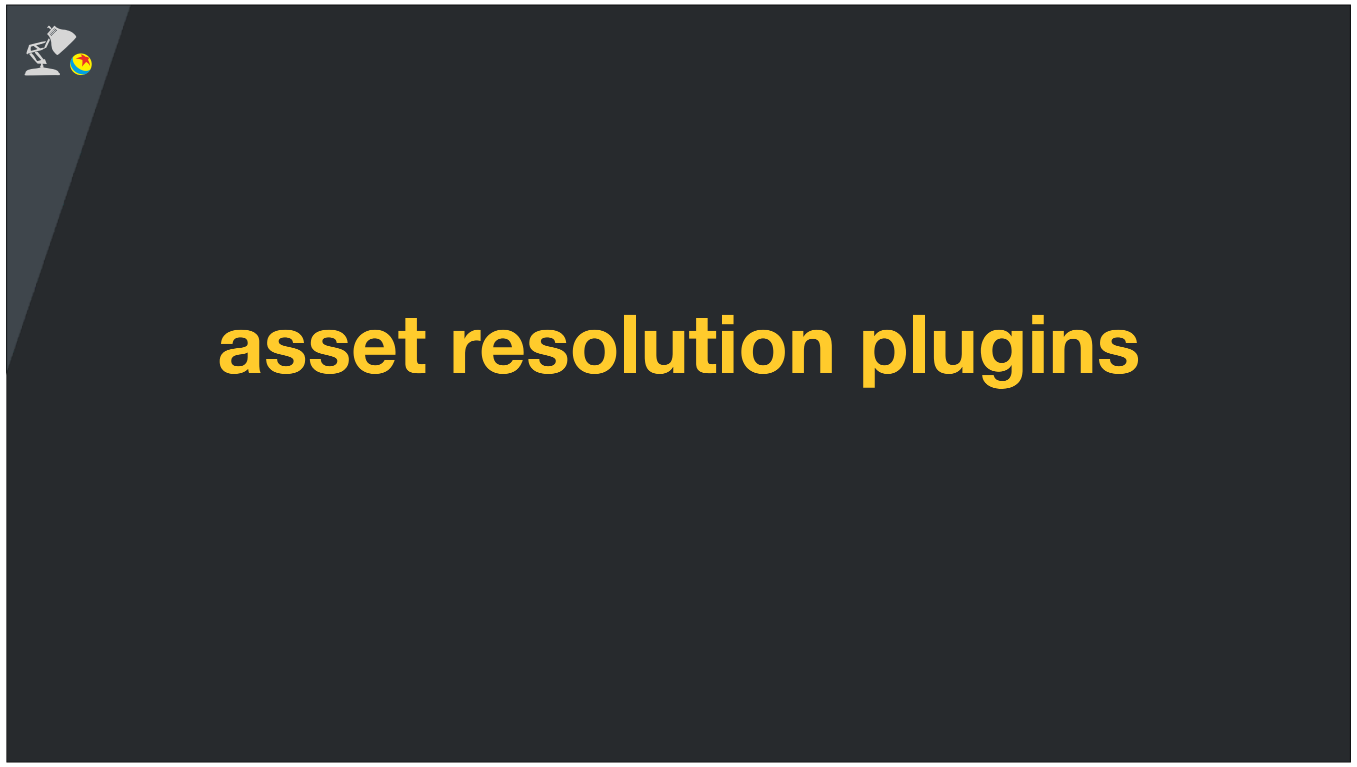


Extensible

The USD Ecosystem is extensible. We have plugin opportunities along many points of our library stacks.

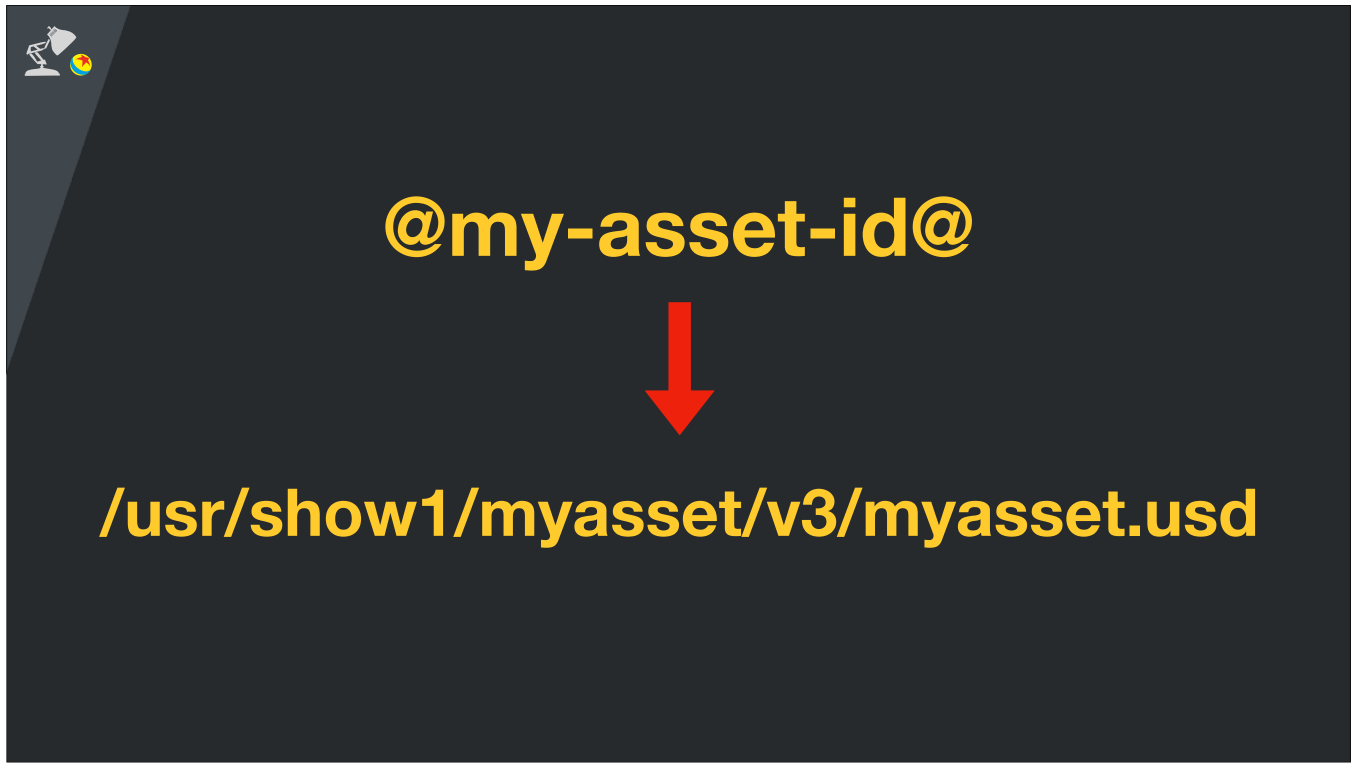
file format plugins

A file format plugin is where you can teach USD how to read various file formats
Also the mechanism where we introduce dynamic file formats which you'll hear more about later in the course



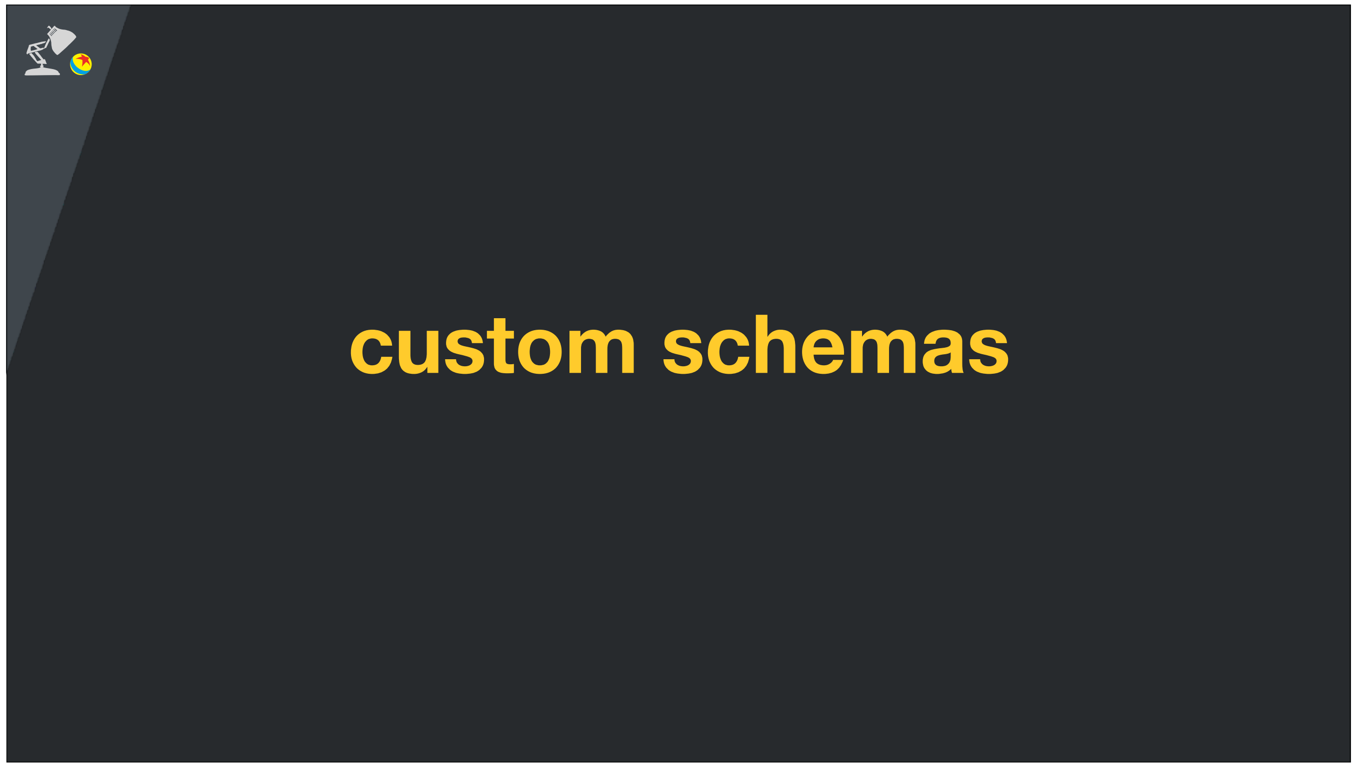
asset resolution plugins

Another way in which USD is extensible is that it gives you an opportunity to intercept asset paths and translate them


@my-asset-id@

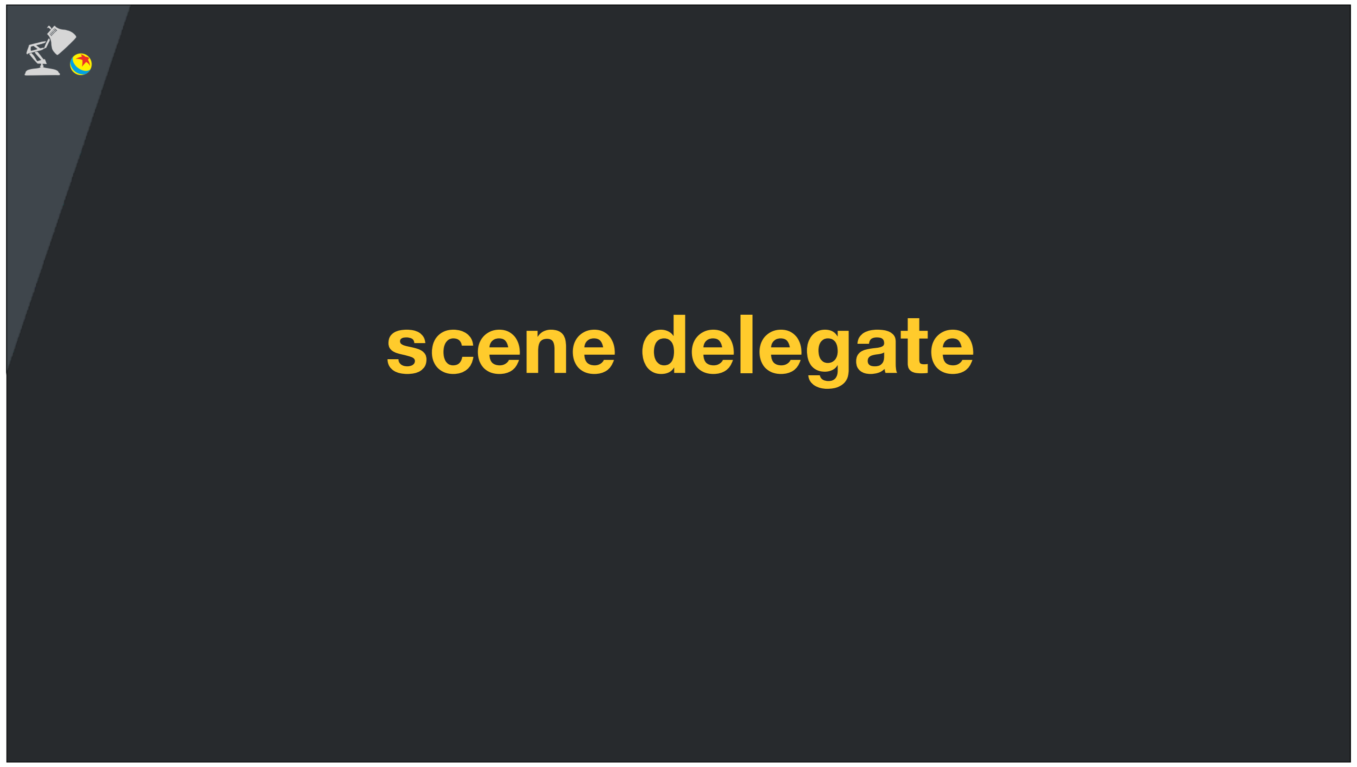
/usr/show1/myasset/v3/myasset.usd

Its main job is to take a reference to an asset encoded for your pipeline and
- turn it into a path that your asset system can open and understand.
This is an area of lots of fun and fruitful discussion also, which you should feel free to come up to us and ask about.



custom schemas

We talked about schemas and domains that we support, well you can write your own!
For example your studio might want a Hair schema, like you'll hear Matt talk about more in the pipeline portion of this course.

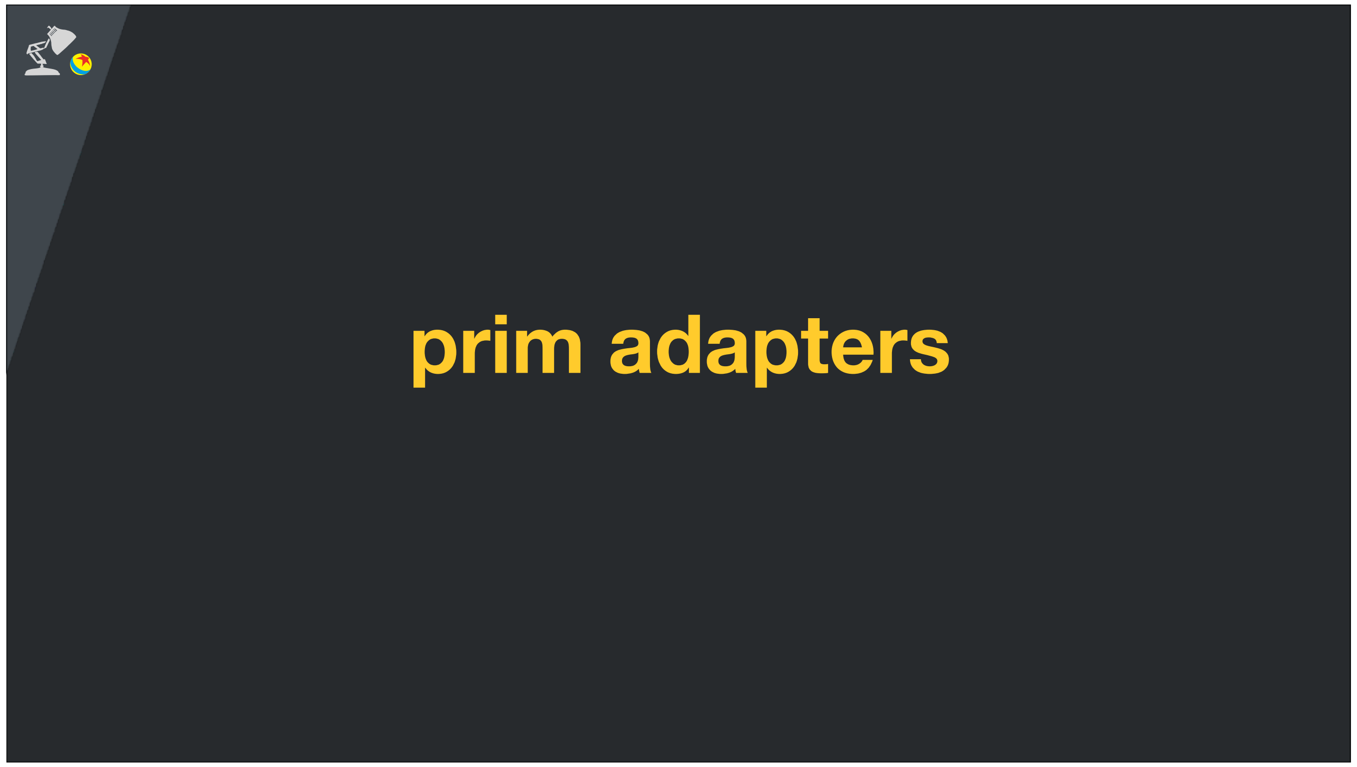


scene delegate

Jumping over to Hydra now, Pol will go into more detail a bit later, but Hydra’s architecture lets you write a plugin to translate your native scenes

render delegate

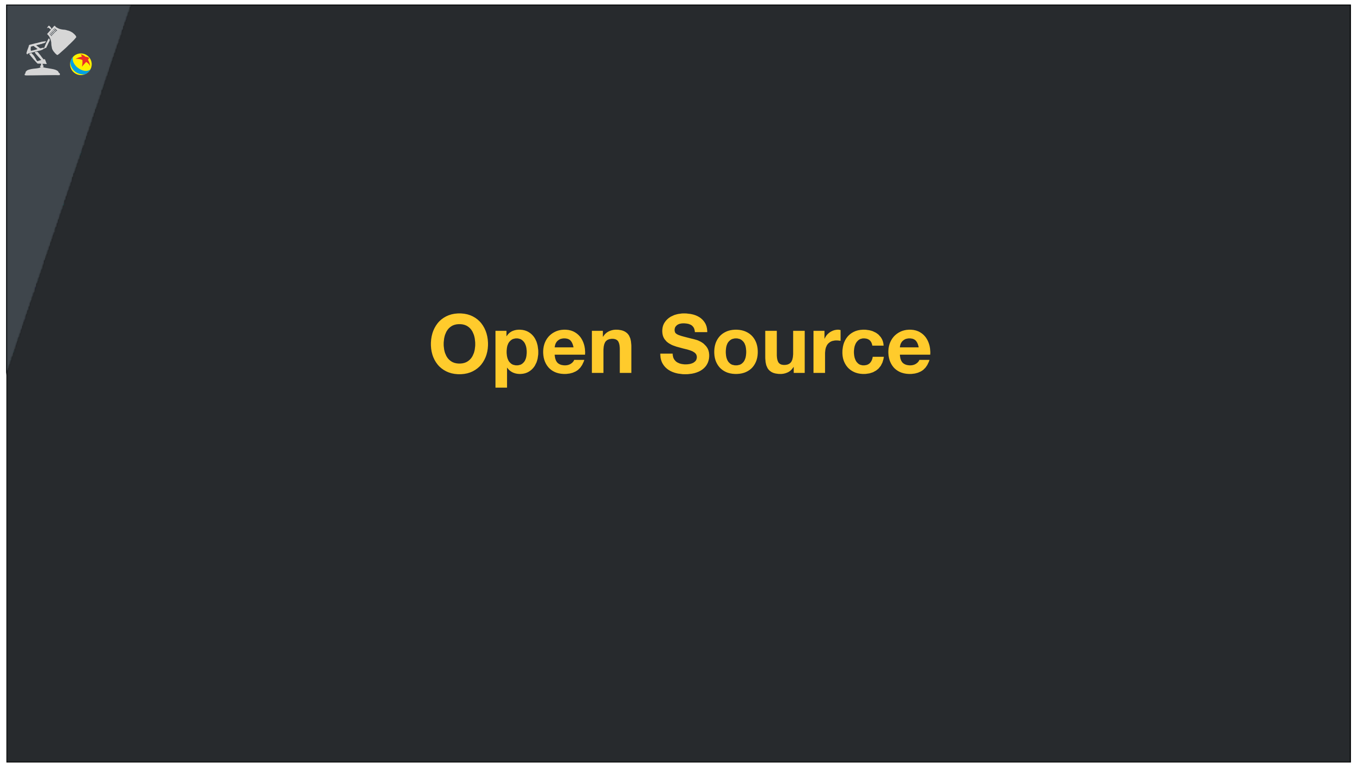
And plugins to use your own renderer



prim adapters

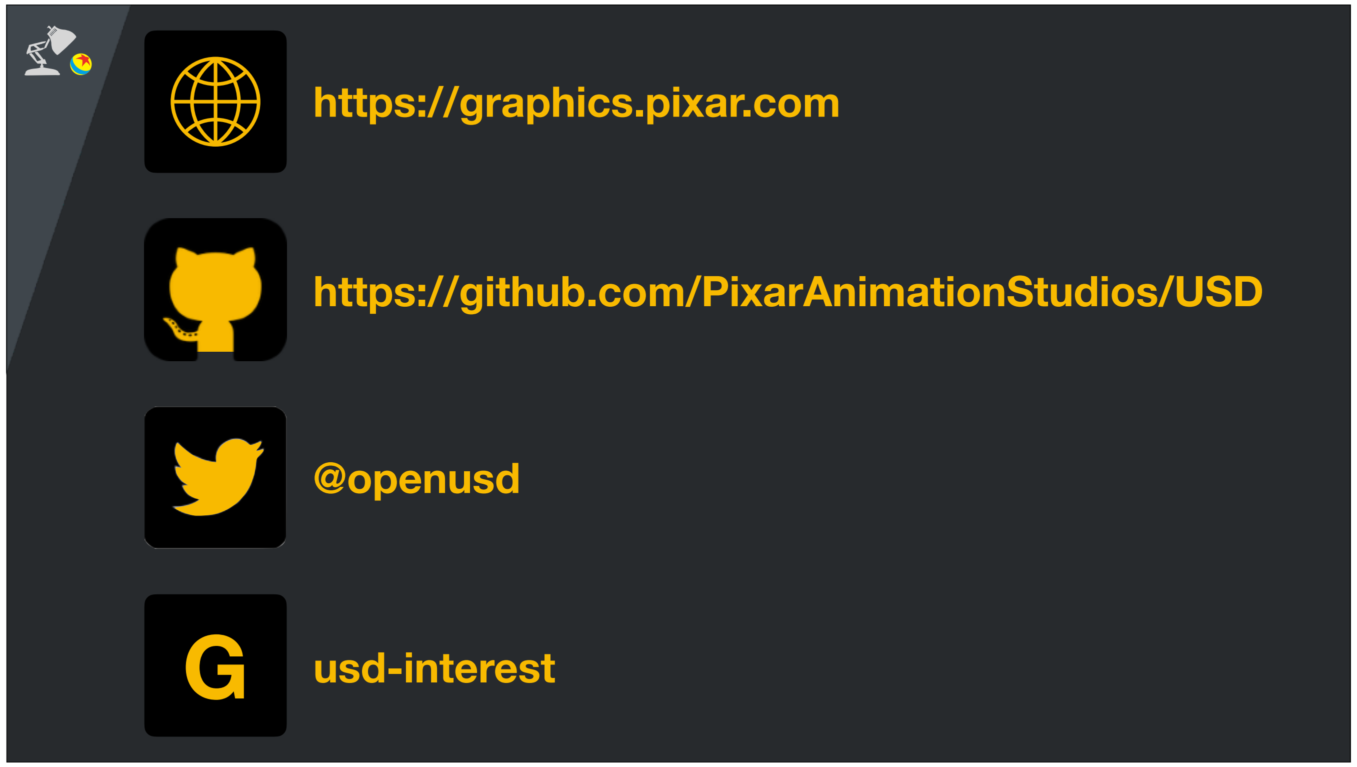
And further and further into the rabbit whole, you can write USD Imaging plugins to visualize the custom schemas you wrote.

You get the idea, USD and Hydra are extremely customizable and extensible



Open Source

Before I hand things over to Sunya, I'll close with just a few notes about our library structures and help that you might need in navigating the open source code base



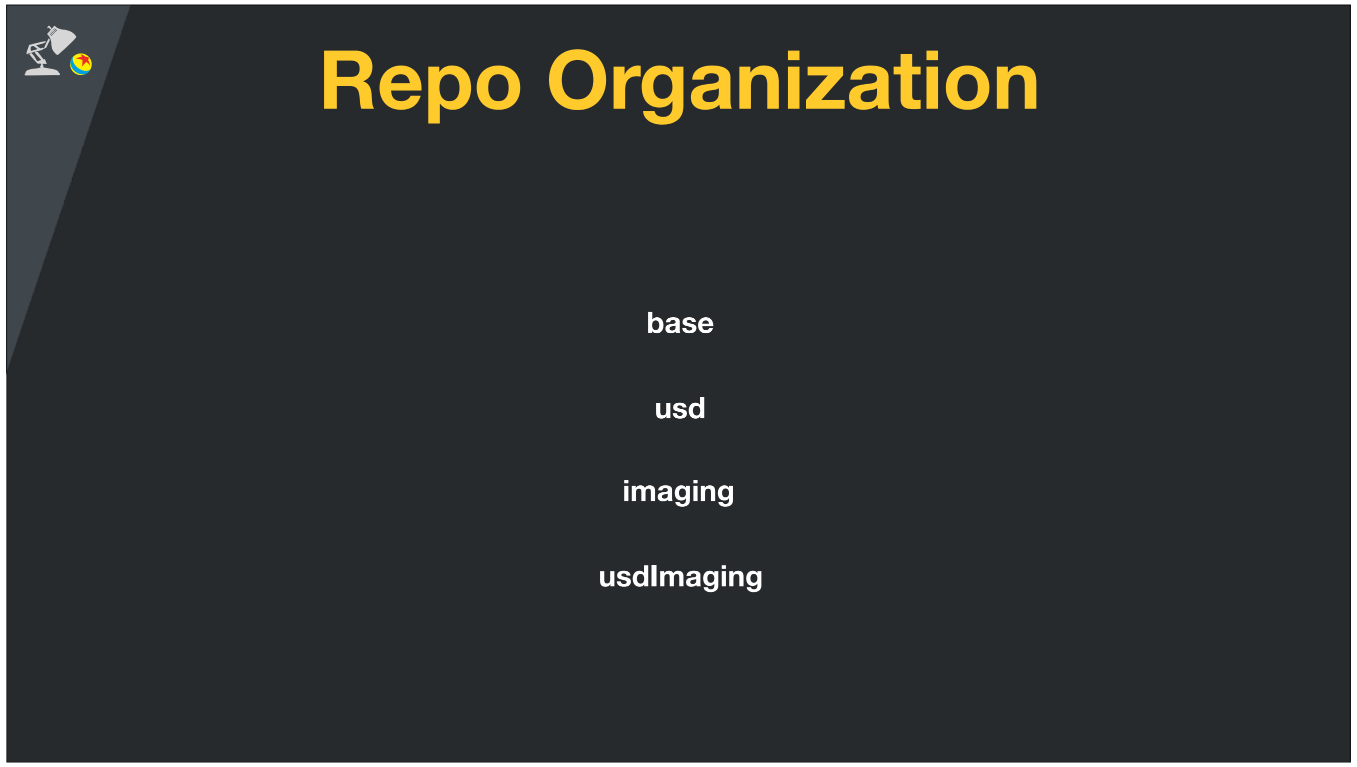
https://graphics.pixar.com

https://github.com/PixarAnimationStudios/USD

@openusd

usd-interest

First of all, there are a handful of ways for you to reach us. On our website, on github, on twitter and on our google interest group



Repo Organization

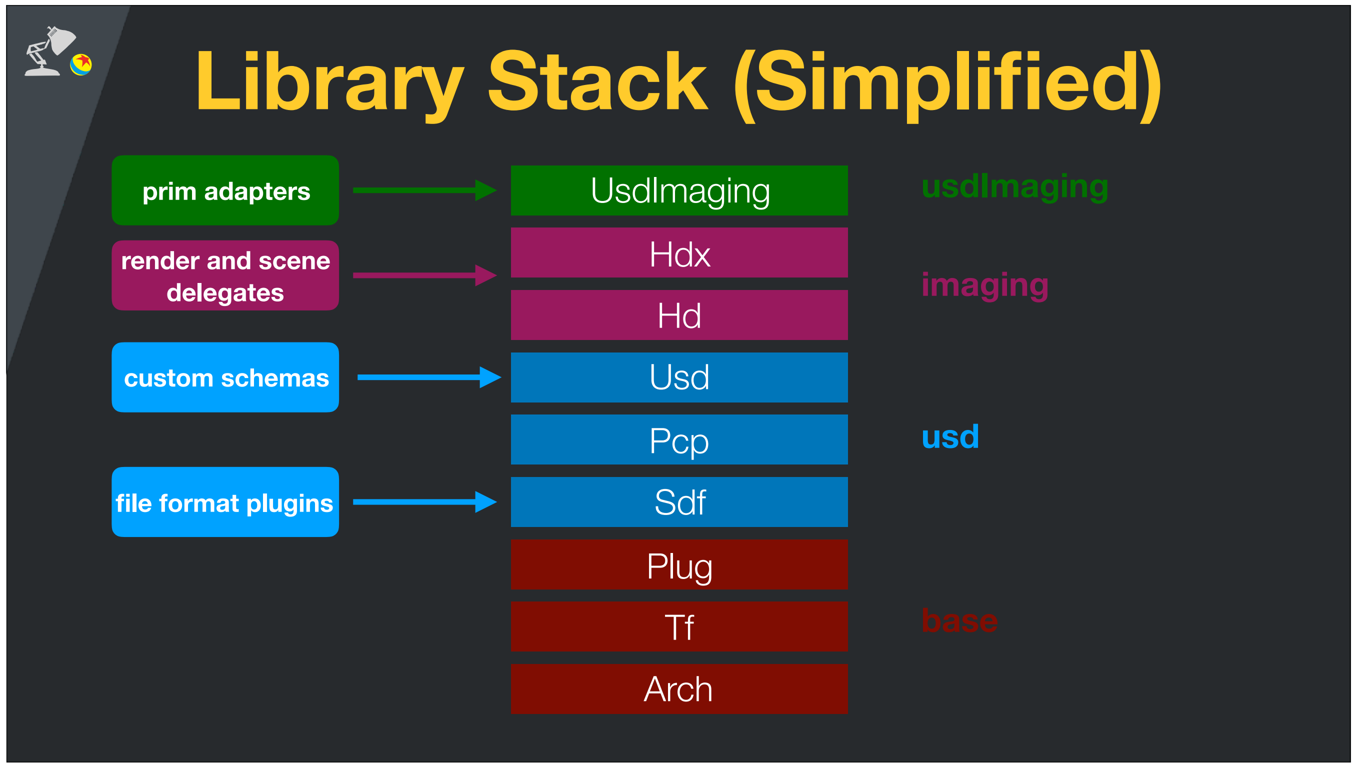
base

usd

imaging

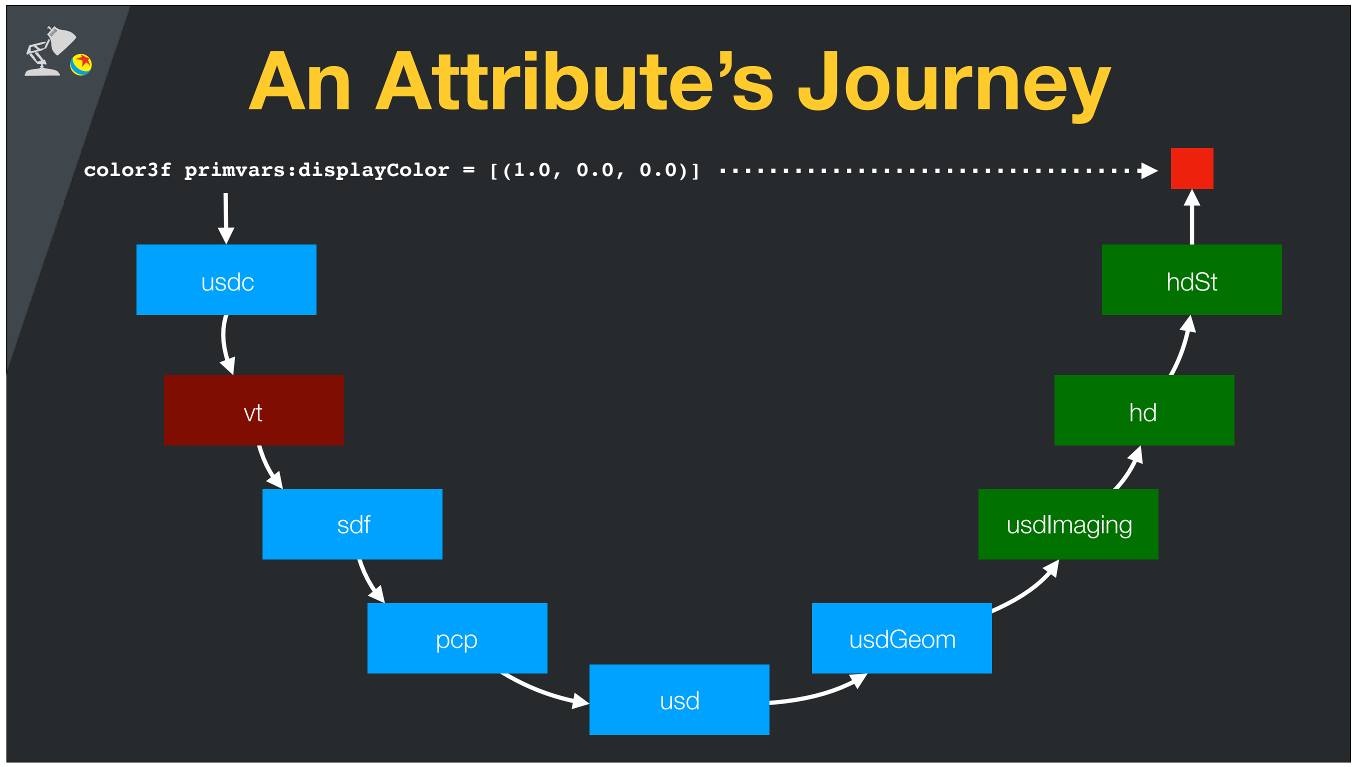
usdImaging

Once you clone the repo, you’ll see these top level folders, let’s see what they mean



Library Stack (Simplified)

- The lowest level of the system is in base. There you have libraries that deal with your foundational utilities, our plugin system, Vt, which implements our basic value types, etc.
- Next, you have a component called usd where most of the usd core lives. This is where you find the code for implementing prims and properties, the composition engine, and most of our core schemas
- This is where file format plugins fit in, for example
- And custom schemas
- Imaging is where the Hydra core lives
- And this is where scene and render delegates come in, this is where our renderers are implemented too (for example, Storm, our fast interactive viewport-style renderer)
- Nearer to the top of the stack is UsdImaging, and that mainly contains the implementation of the USD scene delegate for Hydra, and it's where it all comes together. This is where you'll find usdview for example, our USD viewer application
- The prim adapters I talked about implement APIs defined here.



An Attribute's Journey

```
color3f primvars:displayColor = [(1.0, 0.0, 0.0)]
```

Okay now let's take something concrete like how a displayColor primvar might travel through this library stack…The system is actually pull-based, so we're going to imagine that the imaging system requested this primvar.

- We're assuming this primvar is stored in a USD binary file, so the usdc file format plugin gets asked to fetch the value
- Which it packages in a VtValue — keep in mind usdc is super efficient, as you'll hear Alex talk about later, and we're going to invoke zero copies here — all the arrays that travel through this stack are also copy-on-write
- Next, sdf is the library that knows about layers and prim and property specs
- The layers in sdf are composed by pcp, the composition engine, and where the final resolved values are made available
- usd is the entry-level API where client queries end up, it provides the UsdStge, prim and property APIs
- usdGeom is the schema library that gives these properties meaning
- This is queried by usdImaging, the USD scene delegate for Hydra
- Which itself is queried by Hydra, in hd
- On behalf of the render delegate, in this case our GL backend, implemented in hdSt, which is where the implementation for the Storm renderer mostly lives
- And all that for a red pixel <extra enter>
- While this looks complicated, we care deeply about its performance, besides the GPU memory, there's exactly one copy of this data, and the responsibilities and concerns are nicely decoupled.

… and with that …