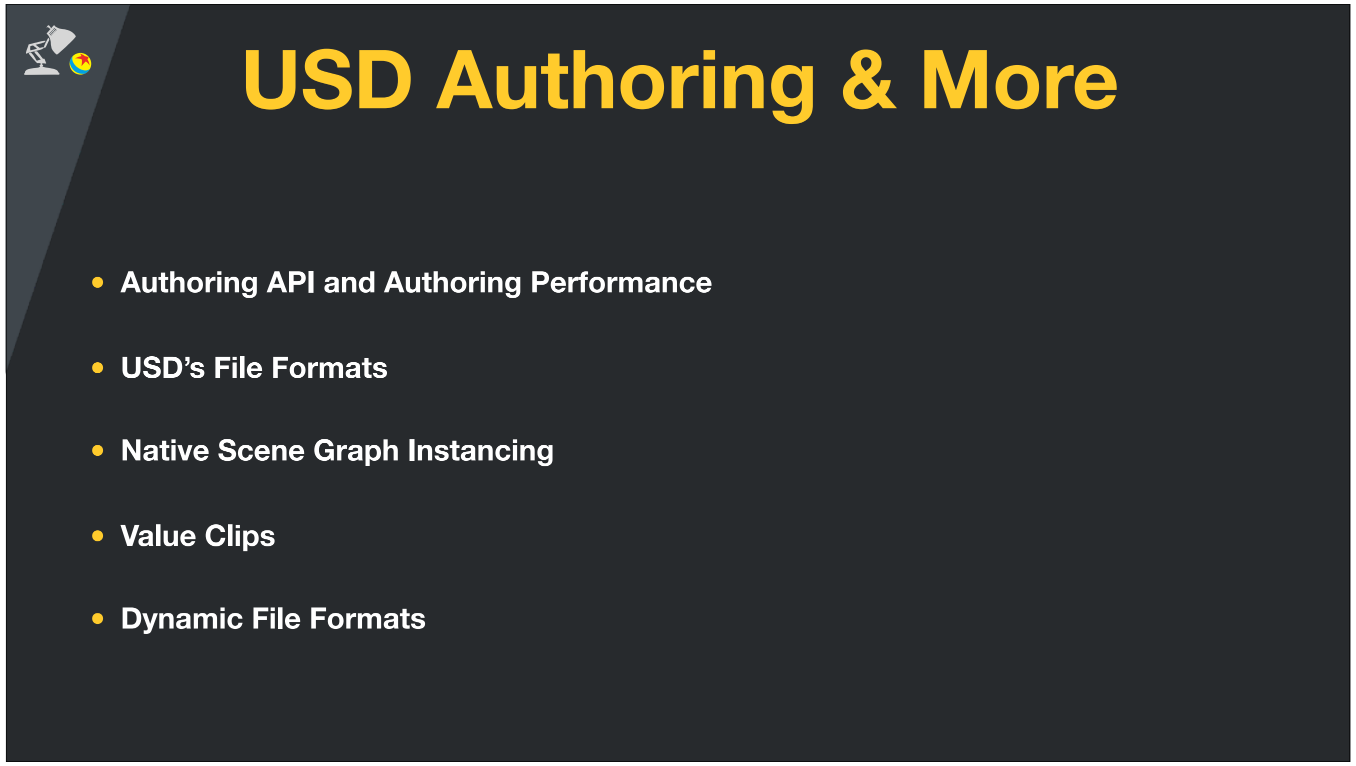
PIXAR
SOFTWARE R&D

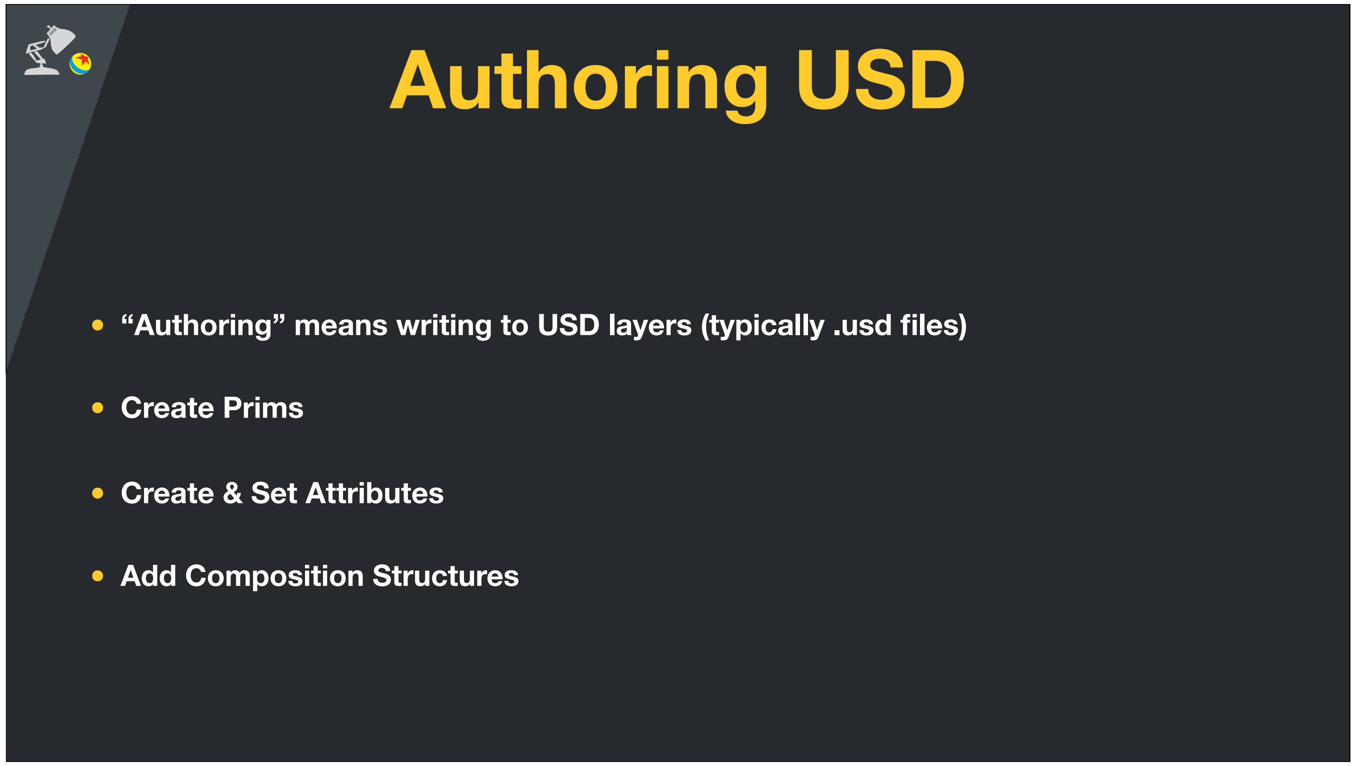
USD Authoring and Advanced Features

Alex Mohr



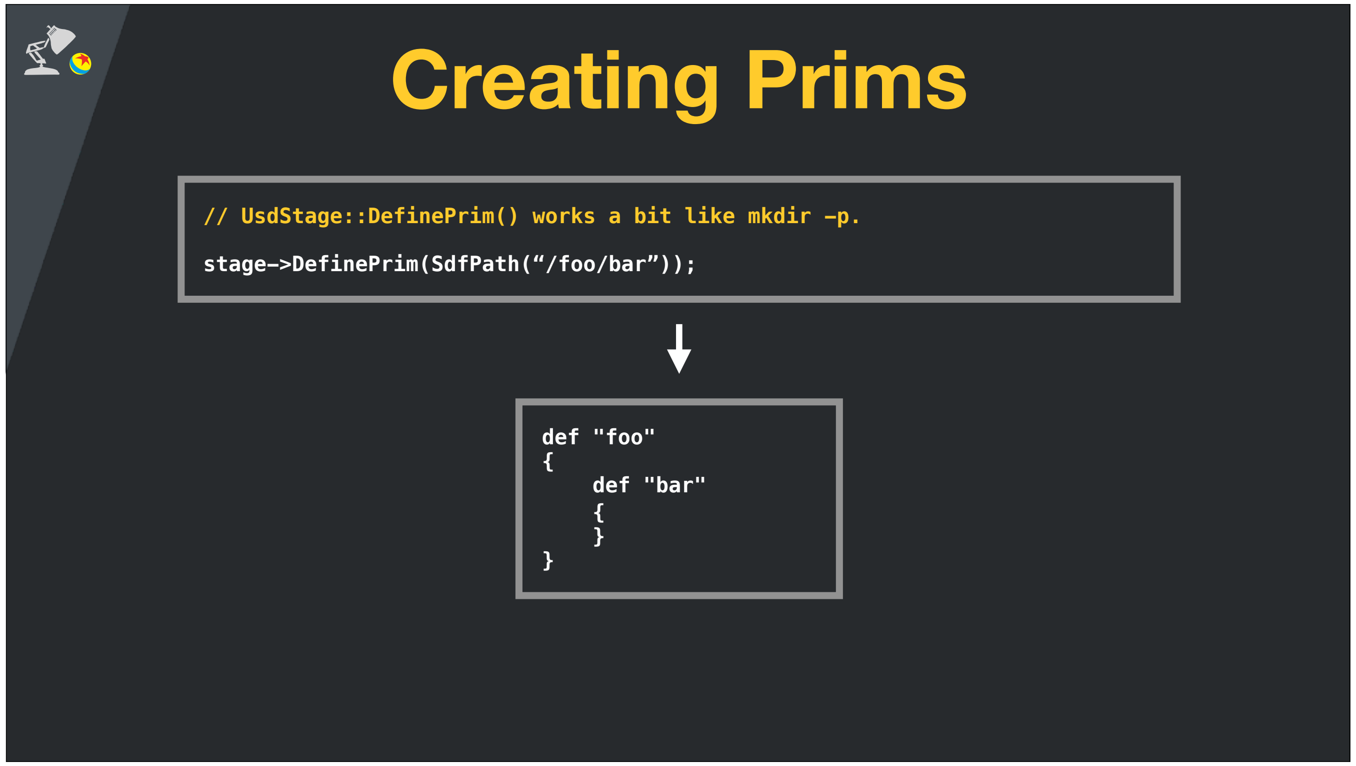
USD Authoring & More

- Authoring API and Authoring Performance
- USD’s File Formats
- Native Scene Graph Instancing
- Value Clips
- Dynamic File Formats



Authoring USD

- “Authoring” means writing to USD layers (typically .usd files)
- Create Prims
- Create & Set Attributes
- Add Composition Structures



Creating Prims

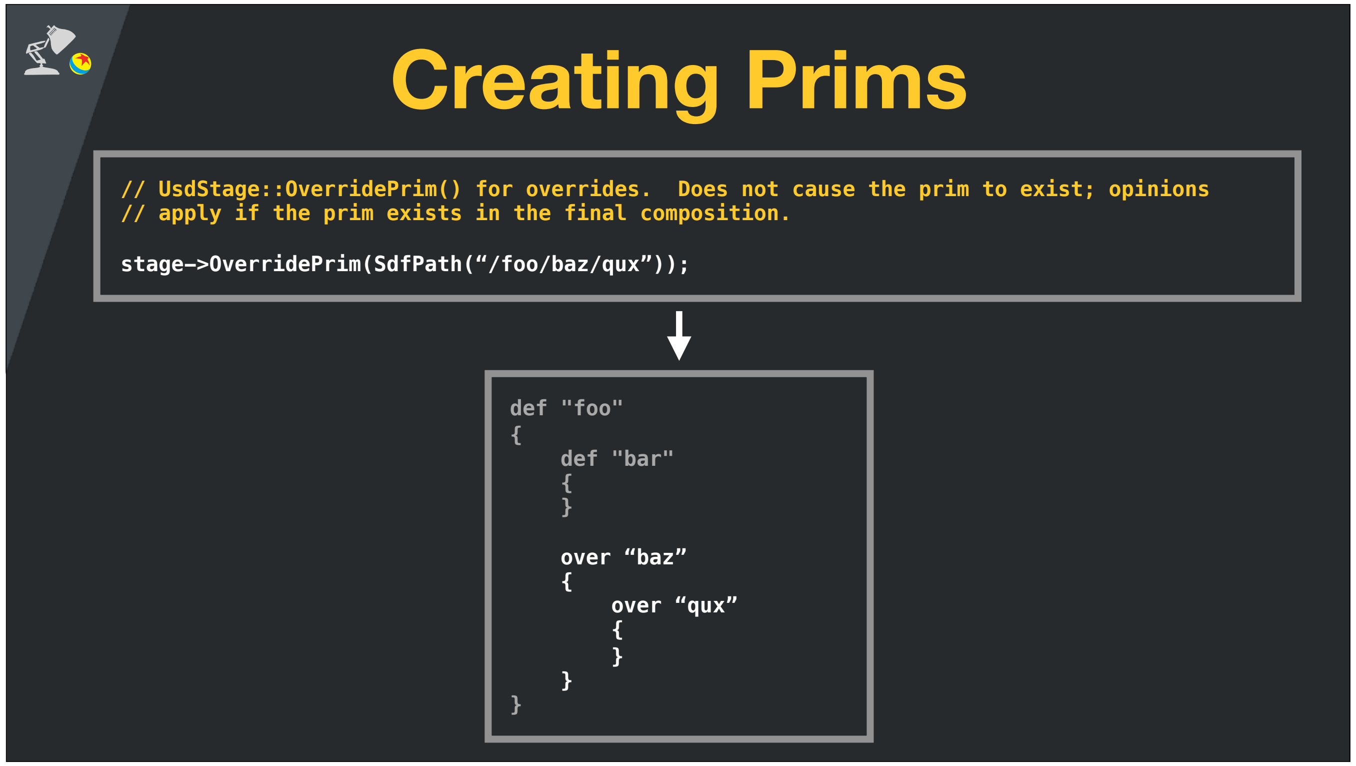
```
// UsdStage::DefinePrim() works a bit like mkdir -p.

stage->DefinePrim(SdfPath(“/foo/bar”));
```

↓

```
def "foo"
{
    def "bar"
    {
    }
}
```

First: Code is C++, but everything is available in Python too.

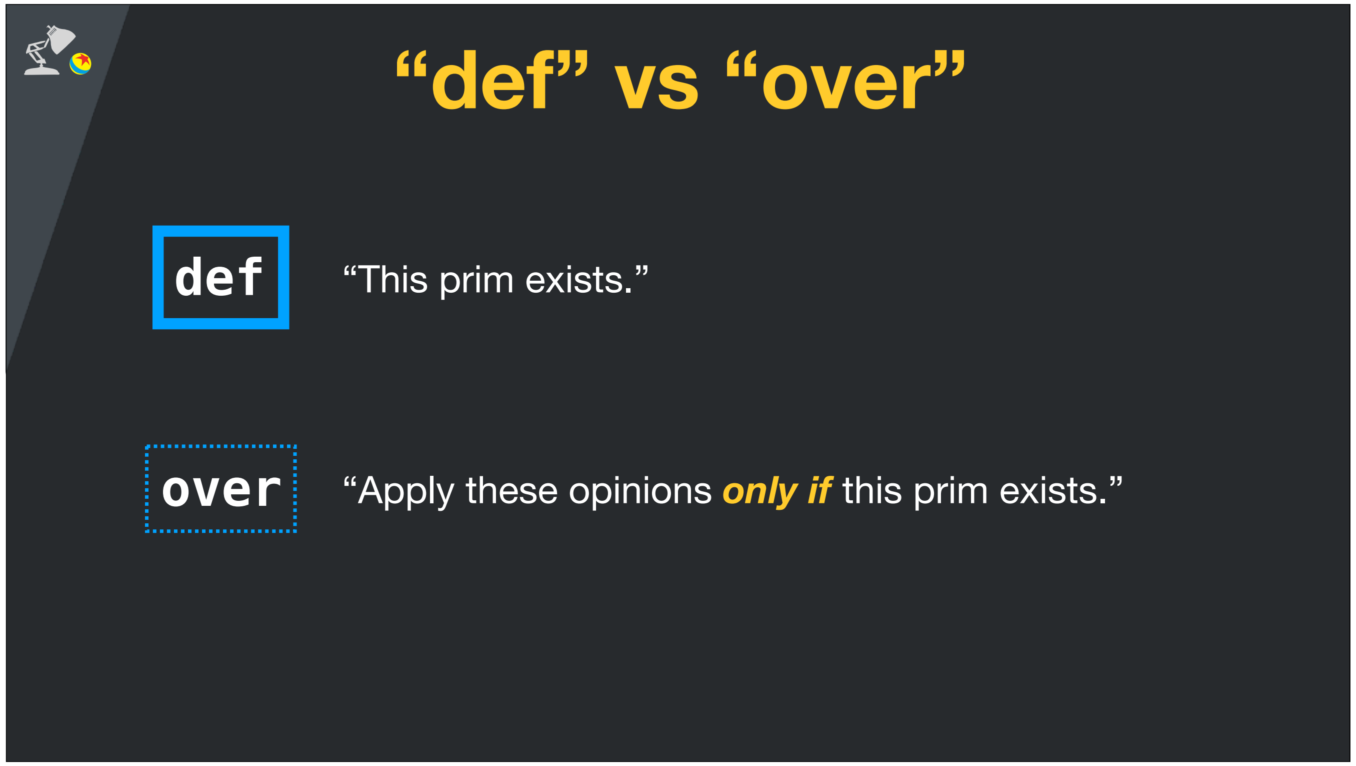
This is a nice way to programmatically build up prim hierarchies.



Creating Prims

```
// UsdStage::OverridePrim() for overrides.  Does not cause the prim to exist; opinions
// apply if the prim exists in the final composition.

stage->OverridePrim(SdfPath(“/foo/baz/qux”));
```

```
def "foo"
{
    def "bar"
    {
    }

    over “baz”
    {
        over “qux”
        {
        }
    }
}
```



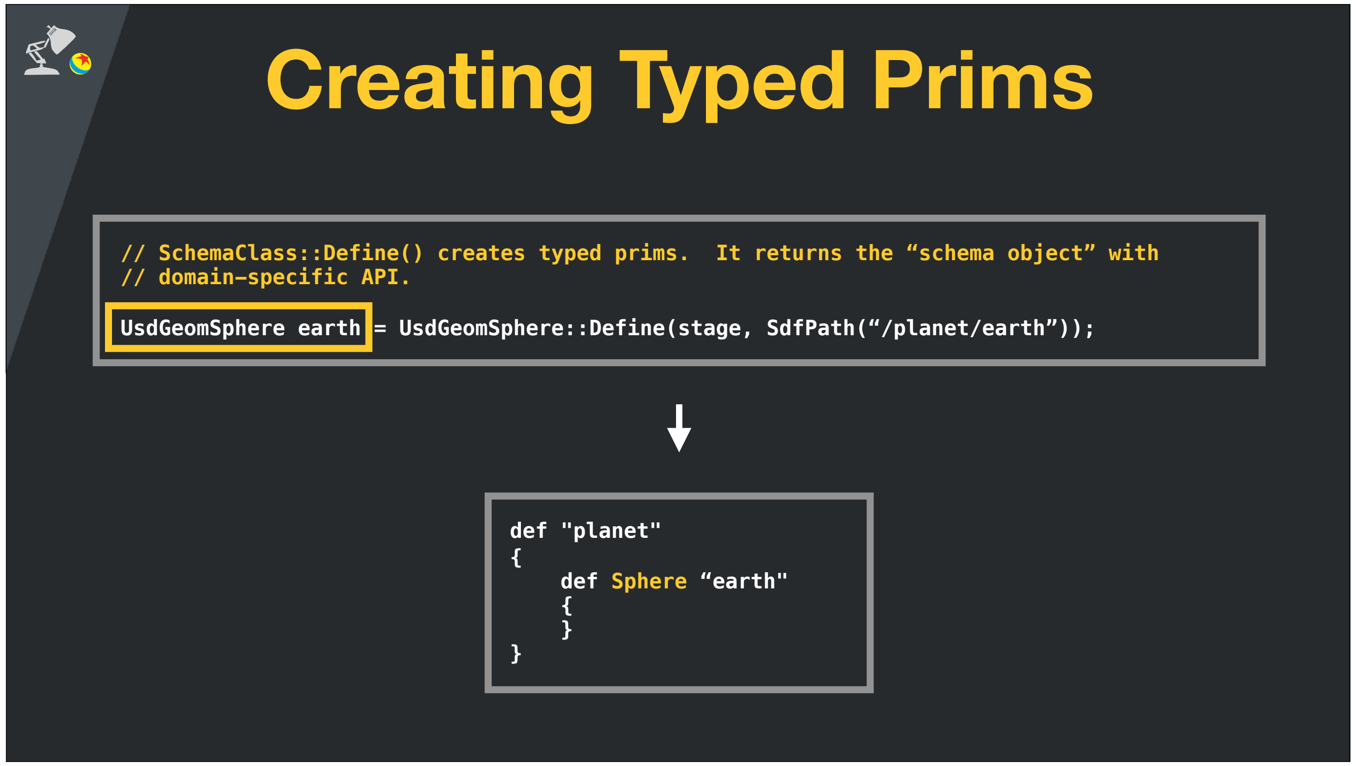
“def” vs “over”

`def` “This prim exists.”

`over` “Apply these opinions ***only if*** this prim exists.”

def: short for ‘define’ — intent is to create a prim. Default traversals will visit these prims.

over: short for ‘override’ — intent is to apply opinions if another layer ‘def’s in the composition.

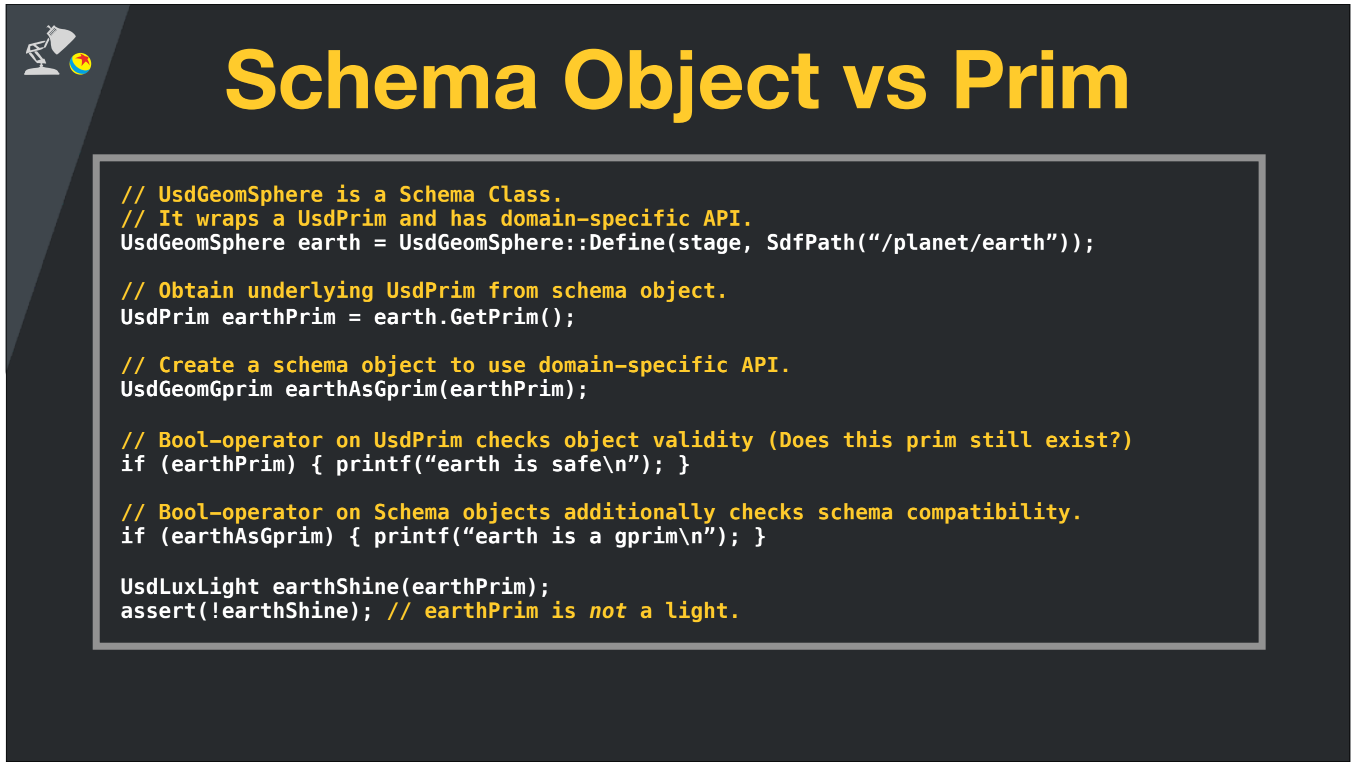


Creating Typed Prims

```
// SchemaClass::Define() creates typed prims.  It returns the “schema object” with
// domain-specific API.

UsdGeomSphere earth = UsdGeomSphere::Define(stage, SdfPath(“/planet/earth”));
```

↓

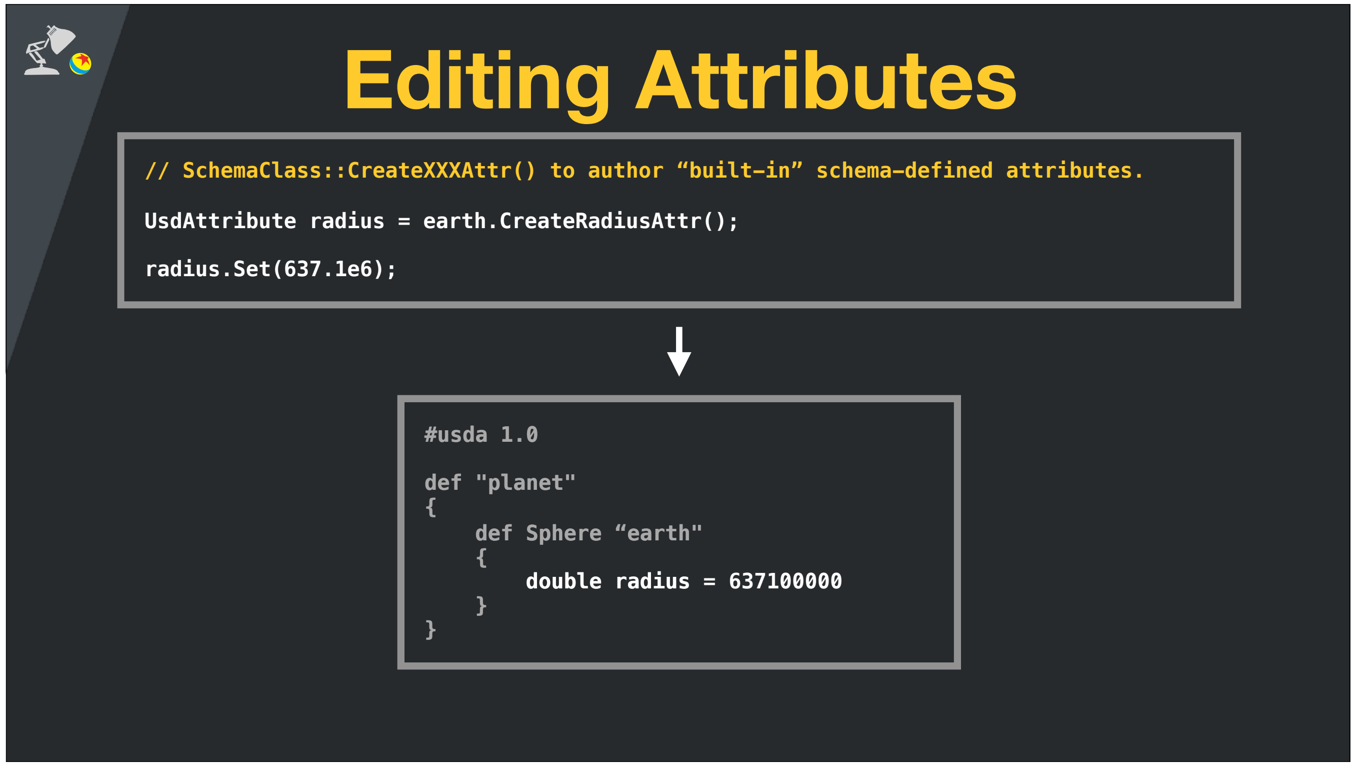
```
def "planet"
{
    def Sphere “earth"
    {
    }
}
```



Schema Object vs Prim

```
// UsdGeomSphere is a Schema Class.
// It wraps a UsdPrim and has domain-specific API.
UsdGeomSphere earth = UsdGeomSphere::Define(stage, SdfPath(“/planet/earth”));

// Obtain underlying UsdPrim from schema object.
UsdPrim earthPrim = earth.GetPrim();

// Create a schema object to use domain-specific API.
UsdGeomGprim earthAsGprim(earthPrim);

// Bool-operator on UsdPrim checks object validity (Does this prim still exist?)
if (earthPrim) { printf(“earth is safe\n”); }

// Bool-operator on Schema objects additionally checks schema compatibility.
if (earthAsGprim) { printf(“earth is a gprim\n”); }

UsdLuxLight earthShine(earthPrim);
assert(!earthShine); // earthPrim is not a light.
```

<at end> Schema classes do not check validity on use, so you can use them to author schema-specific data on prims that do not yet adhere to the schema.

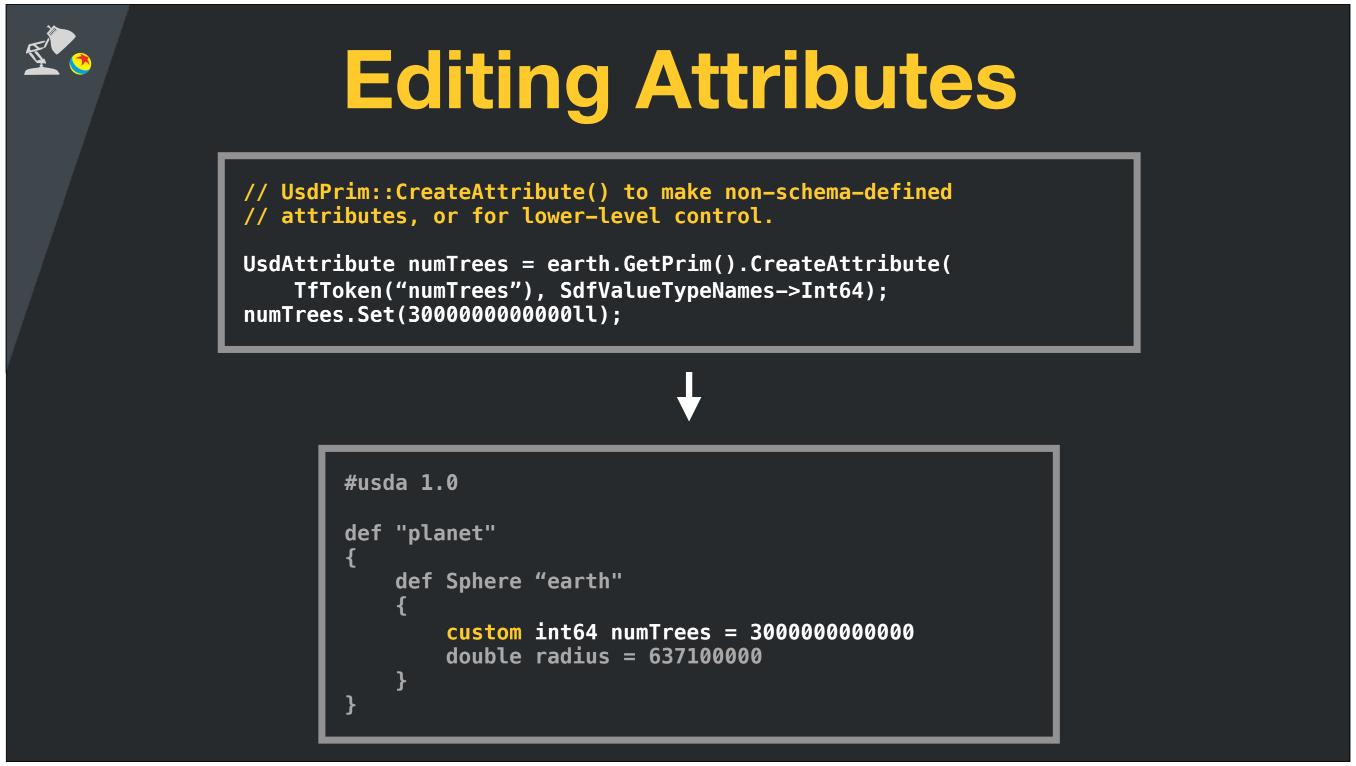


Editing Attributes

```
// SchemaClass::CreateXXXAttr() to author “built-in” schema-defined attributes.

UsdAttribute radius = earth.CreateRadiusAttr();

radius.Set(637.1e6);
```

```
#usda 1.0

def "planet"
{
    def Sphere “earth"
    {
        double radius = 637100000
    }
}
```

<end> By default USD’s linear units are centimeters, but layer metadata ‘metersPerUnit’ can indicate different units.



Editing Attributes

```
// UsdPrim::CreateAttribute() to make non-schema-defined
// attributes, or for lower-level control.

UsdAttribute numTrees = earth.GetPrim().CreateAttribute(
    TfToken(“numTrees”), SdfValueTypeNames->Int64);
numTrees.Set(3000000000000ll);
```

```
#usda 1.0

def "planet"
{
    def Sphere “earth"
    {
        custom int64 numTrees = 3000000000000
        double radius = 637100000
    }
}
```

Editing Composition

- API Objects for Each Composition Operator

`UsdInherits`

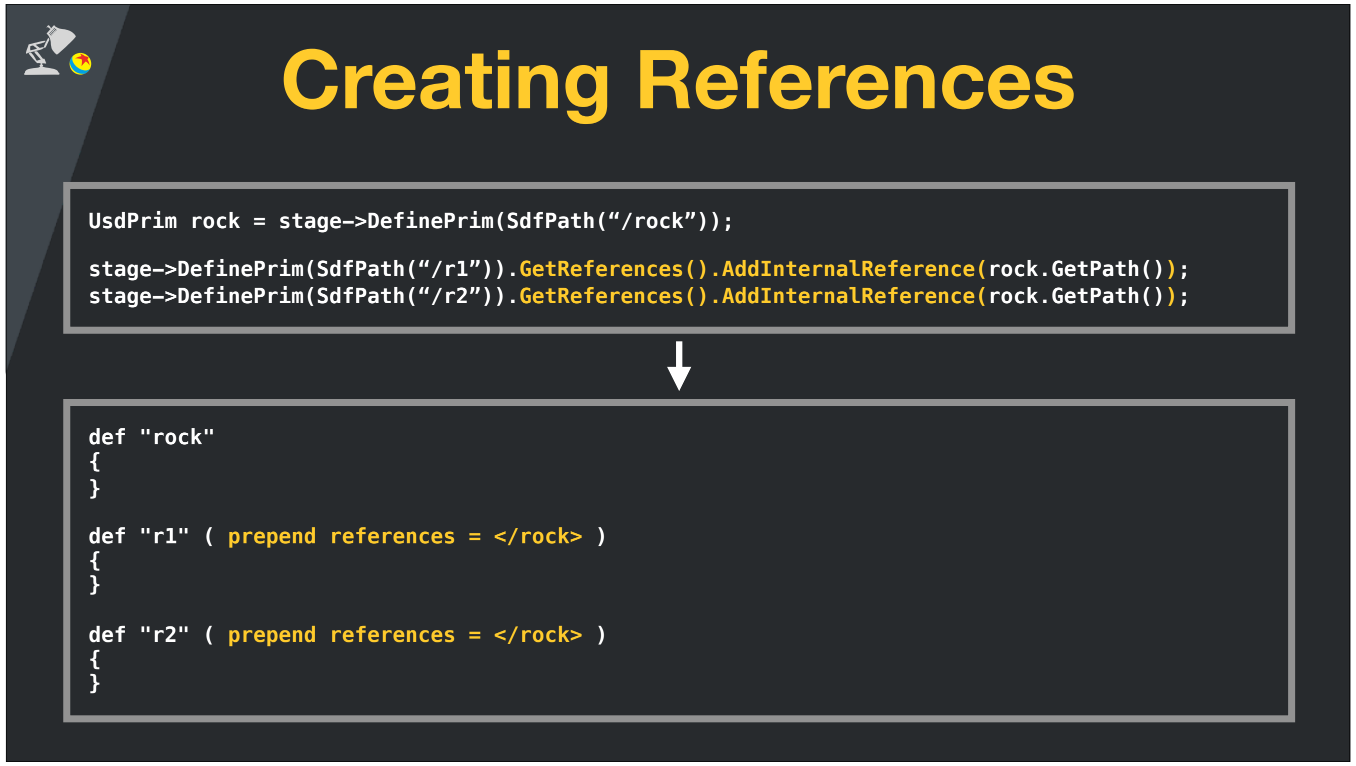
`UsdVariantSets` `UsdVariants`

`UsdReferences`

`UsdPayloads`

`UsdSpecializes`

- Each Provides List Editing Operations (Add / Remove / Set / Clear)



Creating References

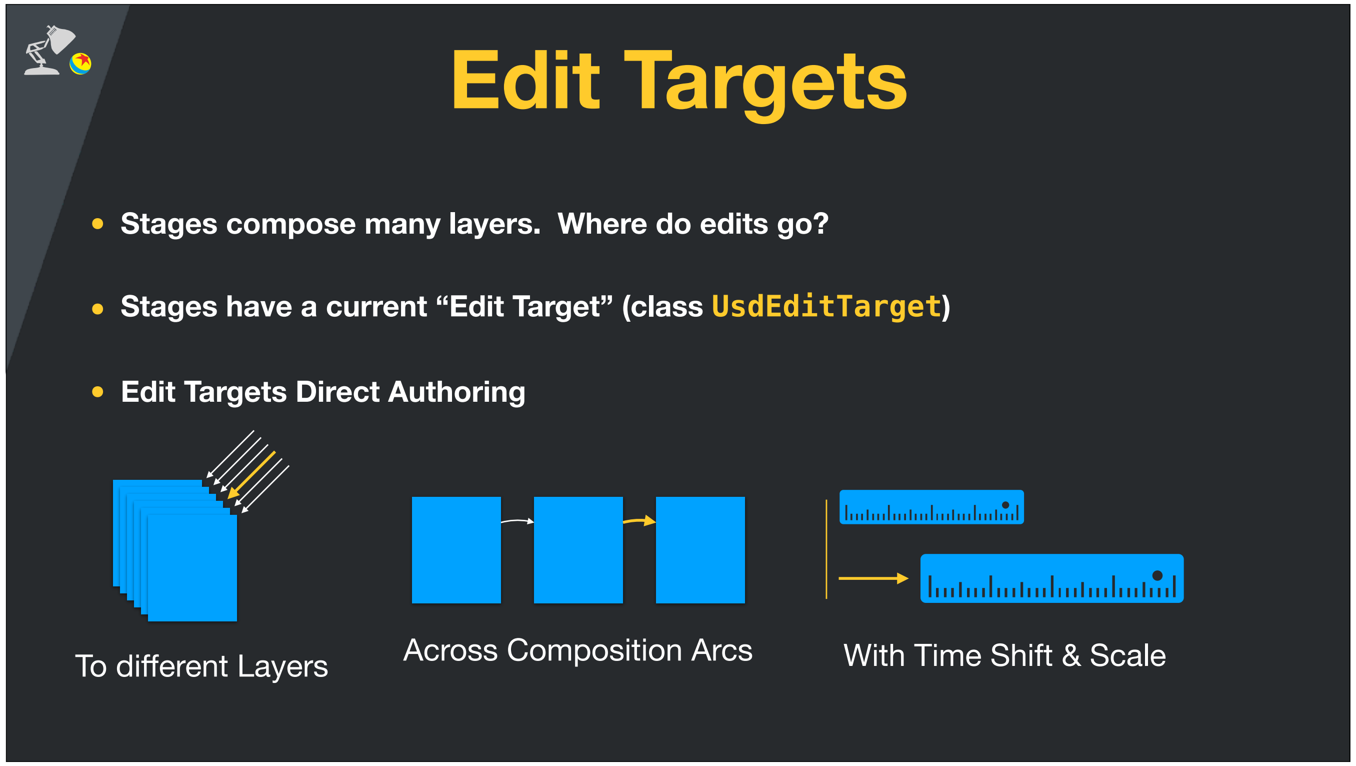
```
UsdPrim rock = stage->DefinePrim(SdfPath(“/rock”));

stage->DefinePrim(SdfPath(“/r1”)).GetReferences().AddInternalReference(rock.GetPath());
stage->DefinePrim(SdfPath(“/r2”)).GetReferences().AddInternalReference(rock.GetPath());
```

```
def "rock"
{
}

def "r1" ( prepend references = </rock> )
{
}

def "r2" ( prepend references = </rock> )
{
}
```

Creating Variants

```
UsdPrim kid = stage->DefinePrim(SdfPath(“/TwoYearOld”));
UsdVariantSet mood = kid.GetVariantSets().AddVariantSet(“mood”);

mood.AddVariant(“elation”);
mood.AddVariant(“anguish”);
```

```
def "TwoYearOld" (
    prepend variantSets = "mood"
)
{
    variantSet "mood" = {
        “anguish" {
        }
        "elation" {
        }
    }
}
```



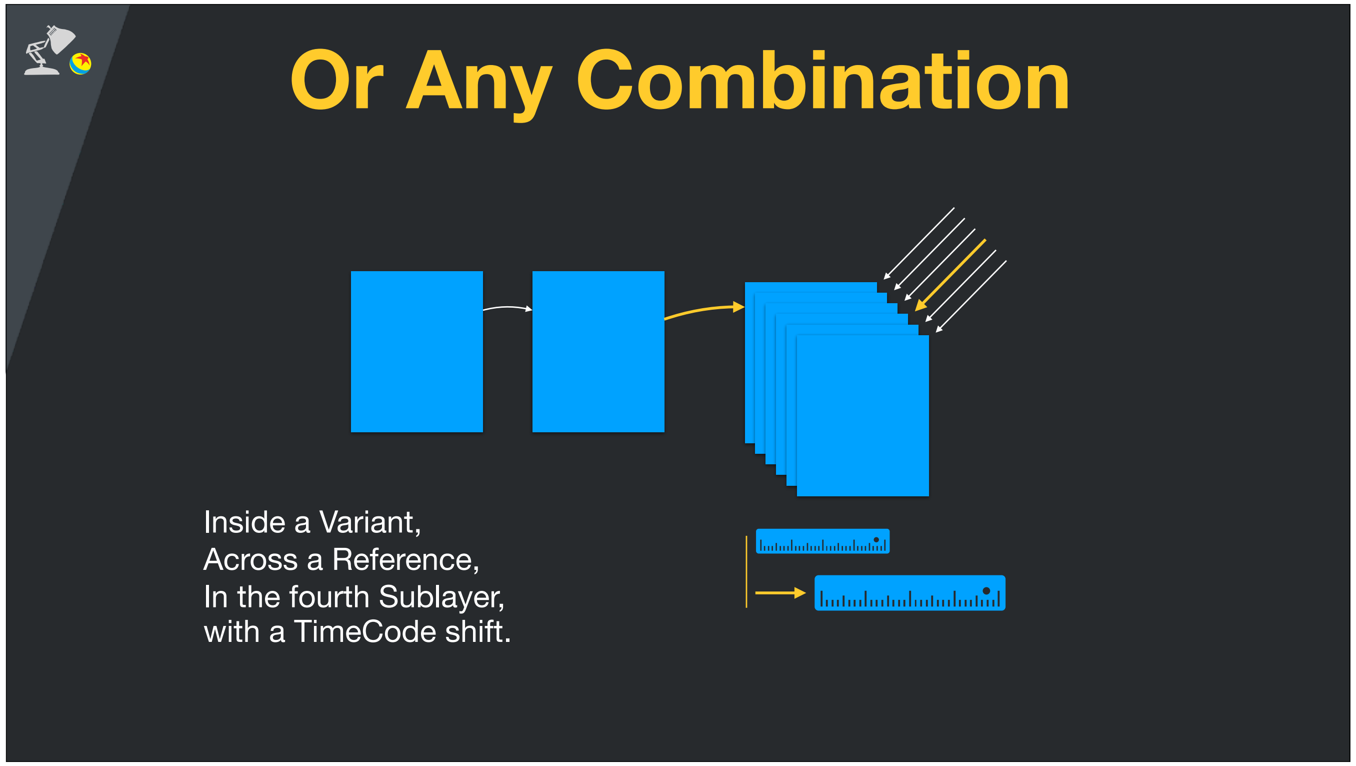
Edit Targets

- Stages compose many layers. Where do edits go?
- Stages have a current “Edit Target” (class `UsdEditTarget`)
- Edit Targets Direct Authoring

To different Layers

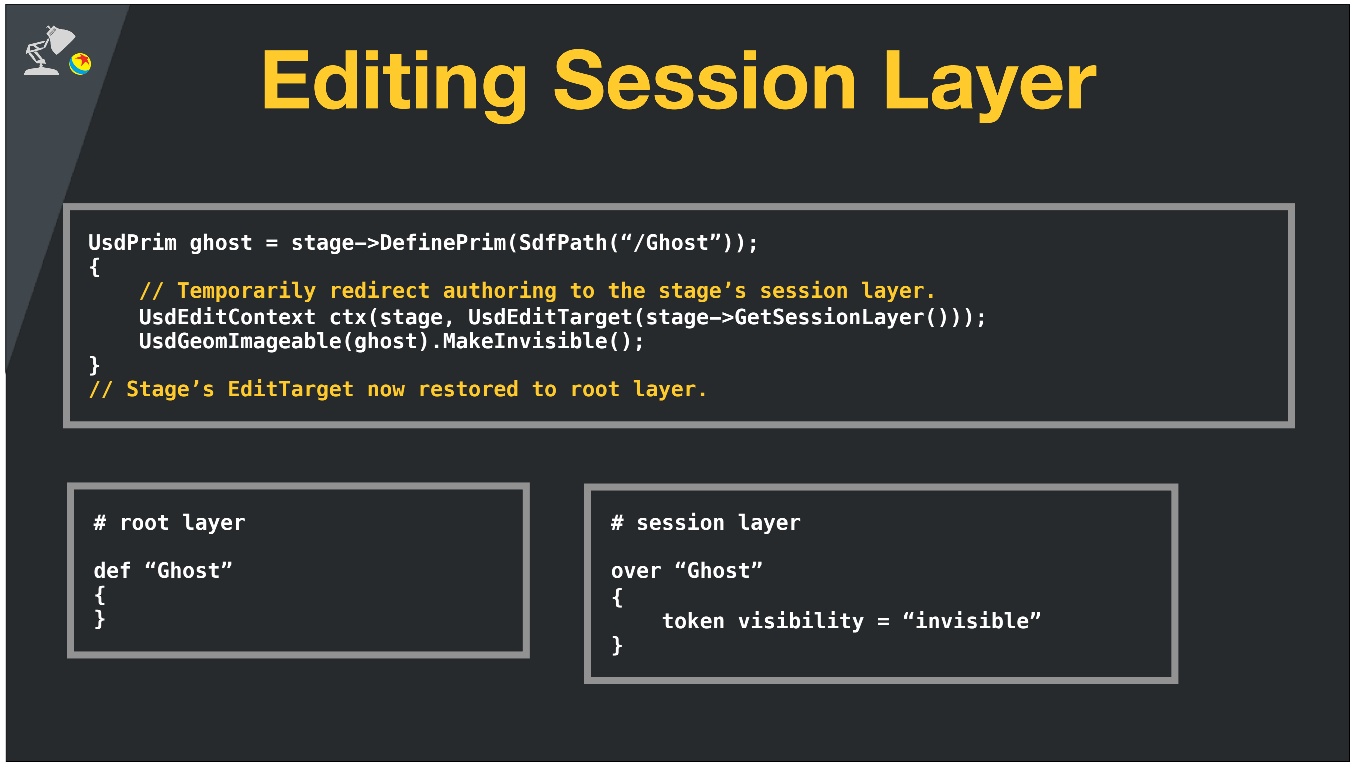
Across Composition Arcs

With Time Shift & Scale



Or Any Combination

Inside a Variant,
Across a Reference,
In the fourth Sublayer,
with a TimeCode shift.



Editing Session Layer

```
UsdPrim ghost = stage->DefinePrim(SdfPath(“/Ghost”));
{
    // Temporarily redirect authoring to the stage’s session layer.
    UsdEditContext ctx(stage, UsdEditTarget(stage->GetSessionLayer()));
    UsdGeomImageable(ghost).MakeInvisible();
}
// Stage’s EditTarget now restored to root layer.
```

```
# root layer

def “Ghost”
{
}
```

```
# session layer

over “Ghost”
{
    token visibility = “invisible”
}
```

First what is a Session Layer? By default UsdStages have a special in-memory layer that’s stronger than all other layers, intended for temporary overrides that are usually not saved. Like a scratch space. When you toggle prim visibility in usdview, those edits go to the session layer.

Editing Variants

```
UsdVariantSet mood = kid.GetVariantSets().GetVariantSet(“mood”);
mood.SetVariantSelection(“elation”);
{
    UsdEditContext ctx(mood.GetVariantEditContext());
    kid.SetDocumentation(“just given a cookie”);
}
mood.SetVariantSelection(“anguish”);
{
    UsdEditContext ctx(mood.GetVariantEditContext());
    kid.SetDocumentation(“finished eating cookie”);
}
```

```
def "TwoYearOld" (
    prepend variantSets = "mood"
)
{
    variantSet "mood" = {
        "elation" ( doc = “just given a cookie” ) {
        }
        “anguish" ( doc = “finished eating cookie” ) {
        }
    }
}
```

Edit Targets

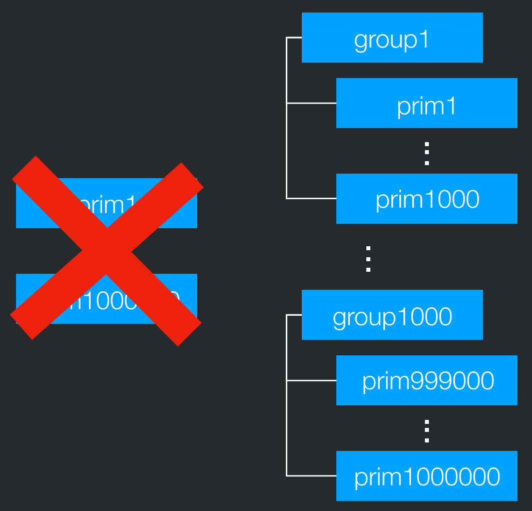
- Edit Targets are powerful and general
- Get a Prim’s `PcpPrimIndex` and walk composition structure
- Construct Edit Targets from `PcpNodeRefs` to edit anywhere
- Check out new higher-level `UsdPrimCompositionQuery` coming soon!
- Project Idea: build a GUI to select EditTargets

Authoring Performance

- USD’s authoring performance is an area for improvement.
- Optimized for Reading. We read a *lot* more than we write.
- Presto heritage; originally a rigging and animation tool.
  - Emphasized interactive response to single control changes.
- USD responds to changes live as they are made.

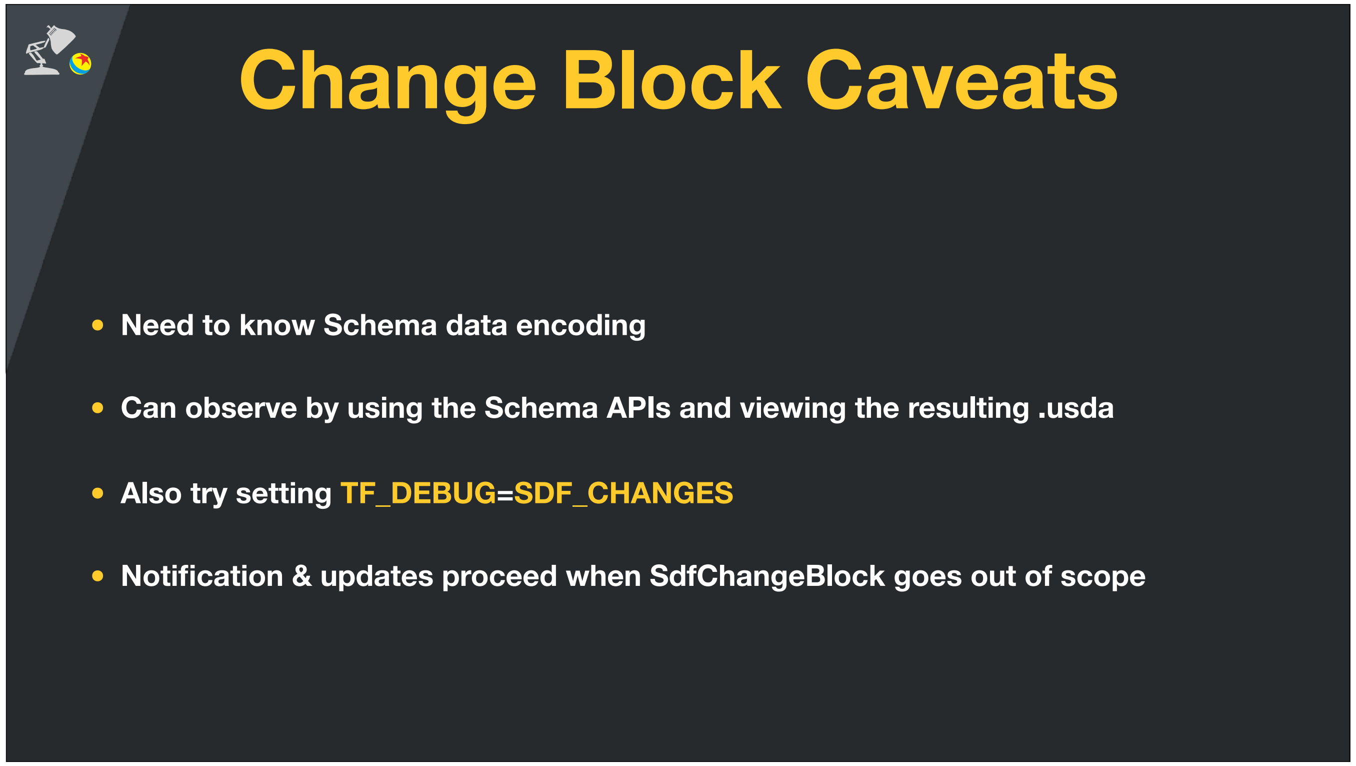
Authoring Perf Tips & Tricks

- Avoid too many sibling prims
  - Lists of names rebuilt per change
  - Adding siblings has $O(n^2)$ behavior
- Use grouping over 10,000s



Change Blocks

- `SdfChangeBlock` defers notification from Sdf (layers) to Usd (stages)
    - Avoids USD's live updates: much faster, but no safety net!
    - Usd doesn't "see" changes, so it can be in an inconsistent state
    - Cannot safely use USD API while `SdfChangeBlocks` are active
- Use Sdf API to write directly to SdfLayers
- Best option for bulk prim creation today



Change Block Caveats

- Need to know Schema data encoding
- Can observe by using the Schema APIs and viewing the resulting .usda
- Also try setting TF_DEBUG=SDF_CHANGES
- Notification & updates proceed when SdfChangeBlock goes out of scope

Authoring Perf Tips & Tricks

Look for better & faster editing options coming in the future

USD’s Native File Formats

- .usda Text File Format
- .usdc “Crate” Binary File Format
- Use the ‘.usd’ extension with either usda or usdc
    - Swap binary for text assets without breaking references
- Both support all USD data (use `usdcat -o` to convert)
- Both are lossless

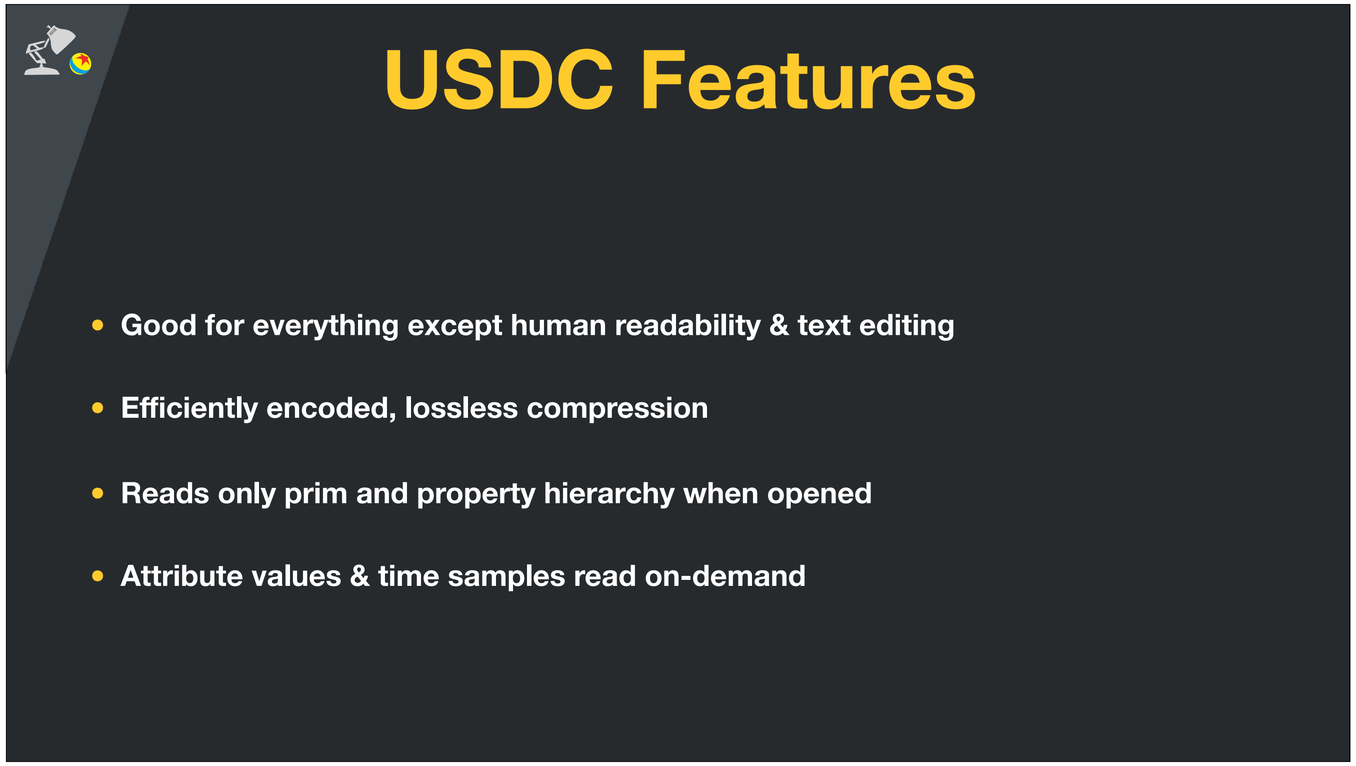
USDZ

- Archive file format co-developed with Apple for network transmission
    - *Is USD running in your pocket?*
- Contains USD files with textures and other assets
- Useful in VFX for packaging and sharing assets across sites

USDA Features

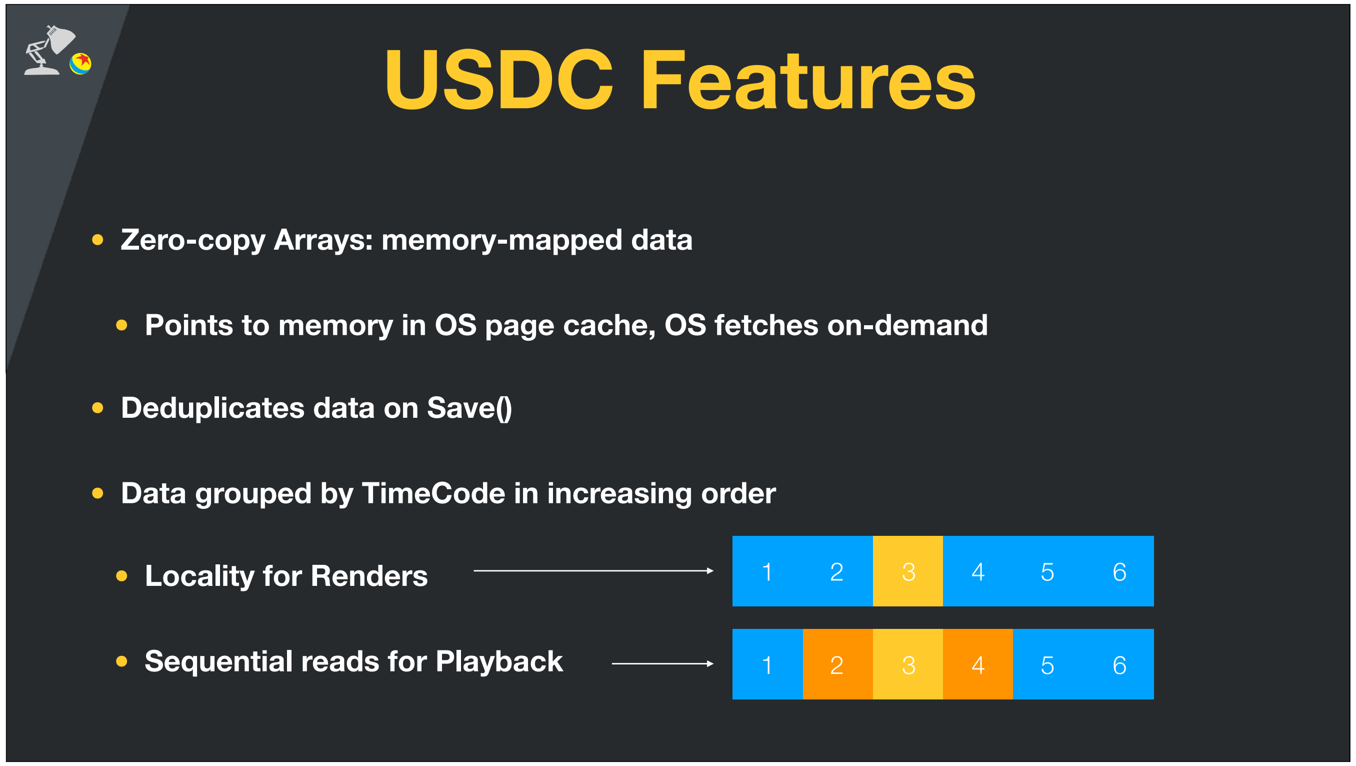
- Great for assembling & positioning assets with References & SubLayers
- Human readable, editable in a text editor
- Fully read into memory when opened

```
#usda 1.0
(
    defaultPrim = "Kitchen_set"
    upAxis = "Z"
)

def Xform "Kitchen_set" (
    kind = "assembly"
)
{
    def Xform "Arch_grp" (
        kind = "group"
    )
    {
        def "Kitchen_1" (
            add references = @./assets/Kitchen/Kitchen.usd@
        )
        {
            double3 xformOp:translate = (71.10783386230469, -43.28064727783203, -1.8192274570465088)
            uniform token[] xformOpOrder = ["xformOp:translate"]
        }
```



USDC Features

- Good for everything except human readability & text editing
- Efficiently encoded, lossless compression
- Reads only prim and property hierarchy when opened
- Attribute values & time samples read on-demand

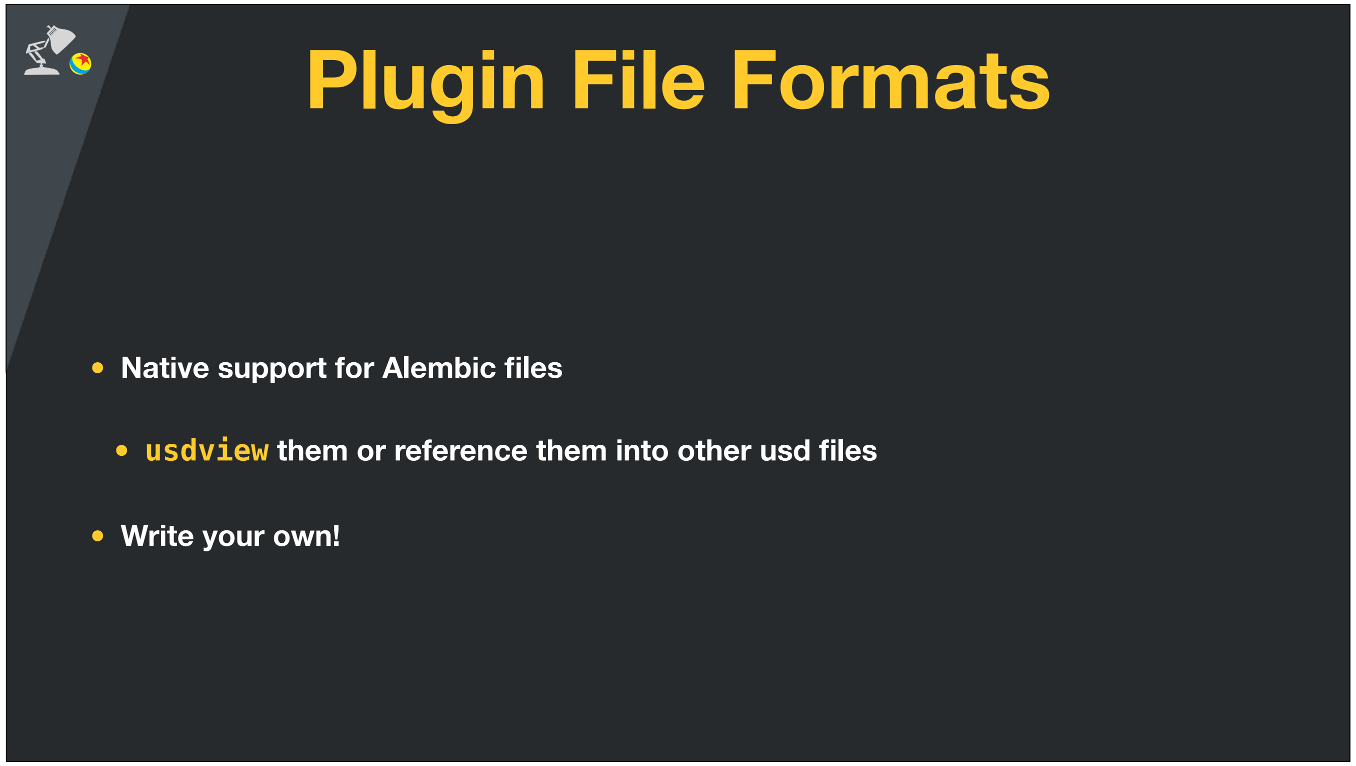


USDC Features

- Zero-copy Arrays: memory-mapped data
  - Points to memory in OS page cache, OS fetches on-demand
- Deduplicates data on Save()
- Data grouped by TimeCode in increasing order
  - Locality for Renders →
  - Sequential reads for Playback →

1	2	3	4	5	6

1	2	3	4	5	6



Plugin File Formats

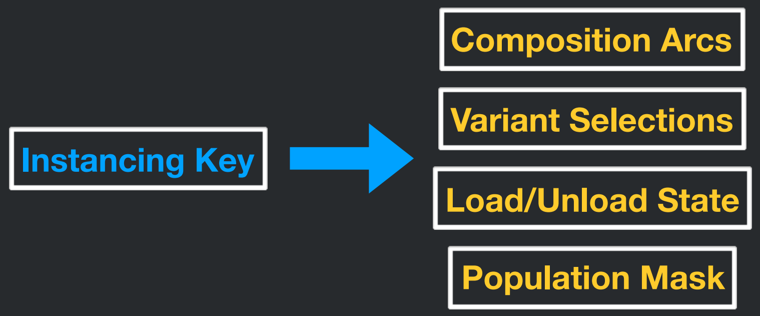
- Native support for Alembic files
  - `usdview` them or reference them into other usd files
- Write your own!

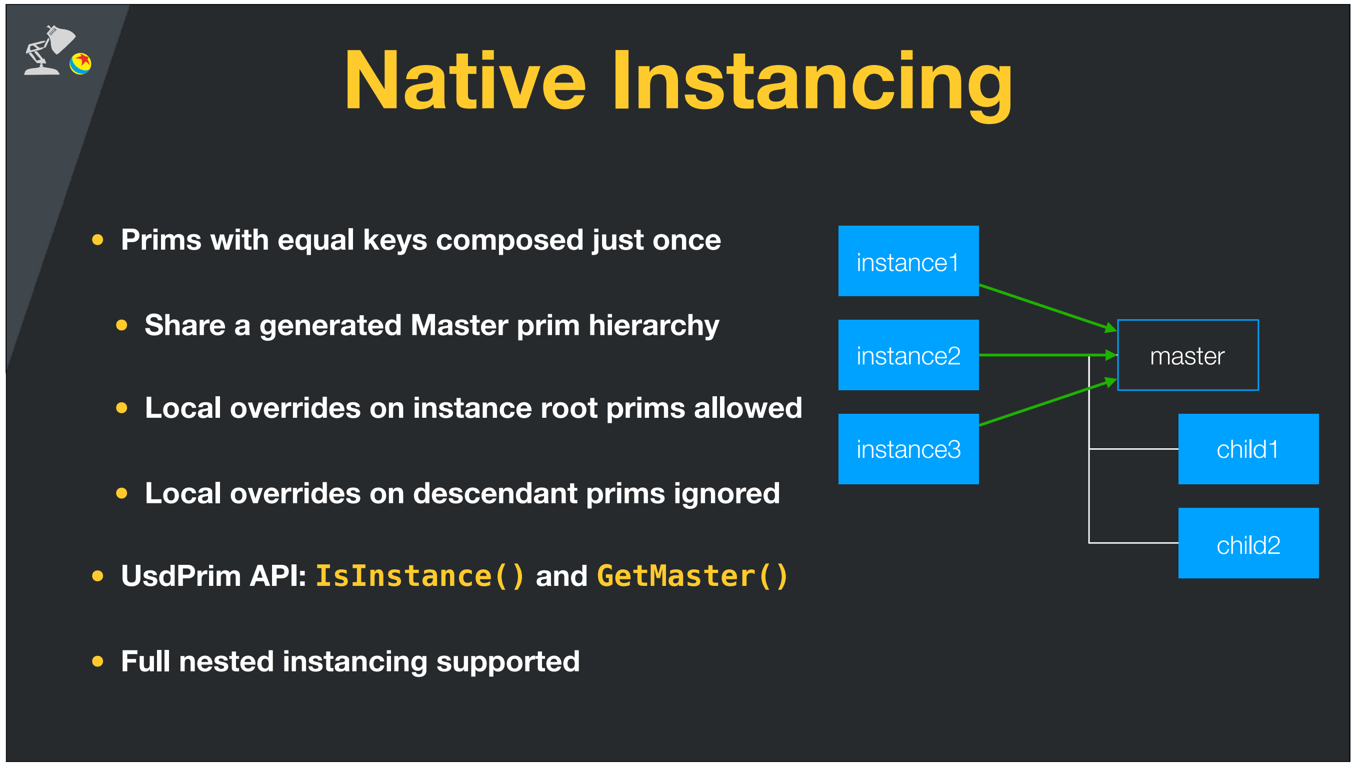
Advanced USD Features

- Native Scene Graph Instancing (*not* UsdGeomPointInstancer)
- Value Clips
- Dynamic File Formats

Native Instancing

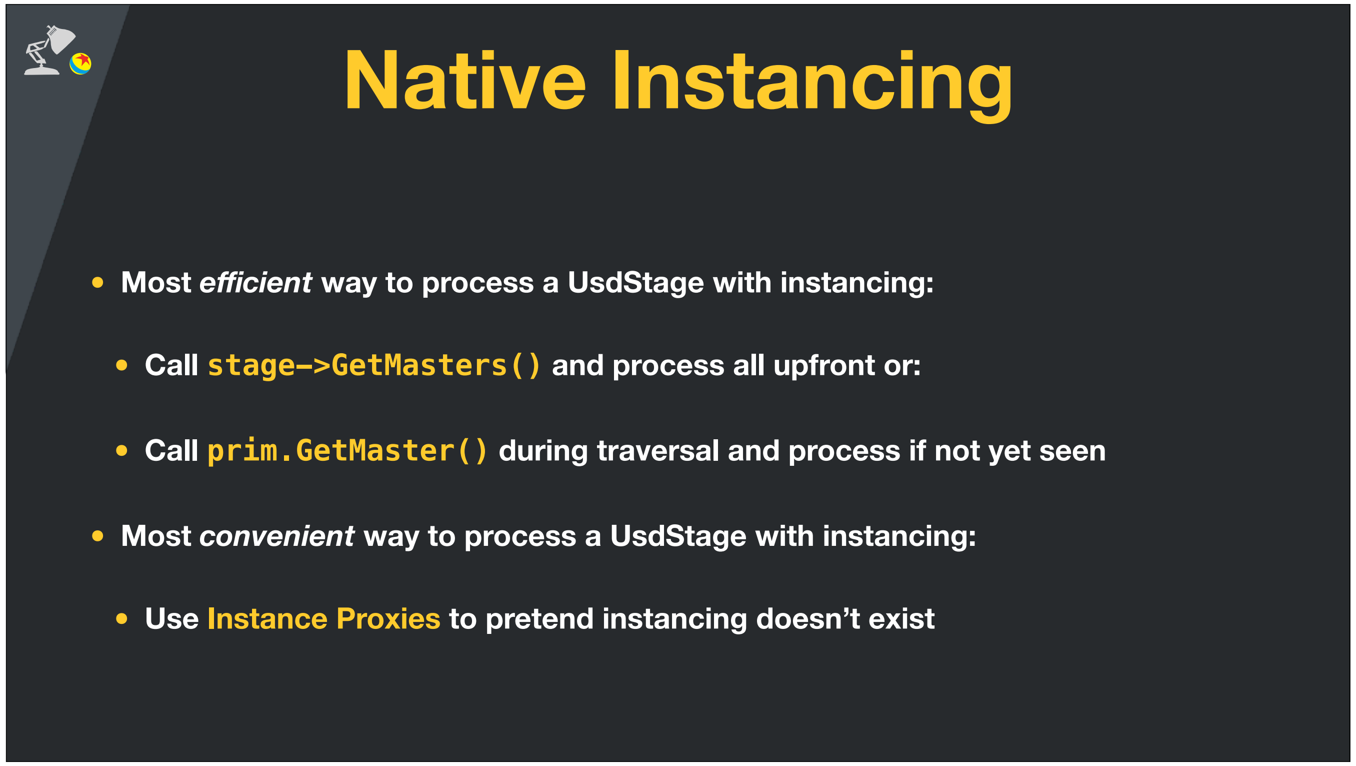
- Declare Prims intended to be instanced with (instanceable = true) metadatum
- Usd runtime determines which ‘instanceable’ prims can be shared





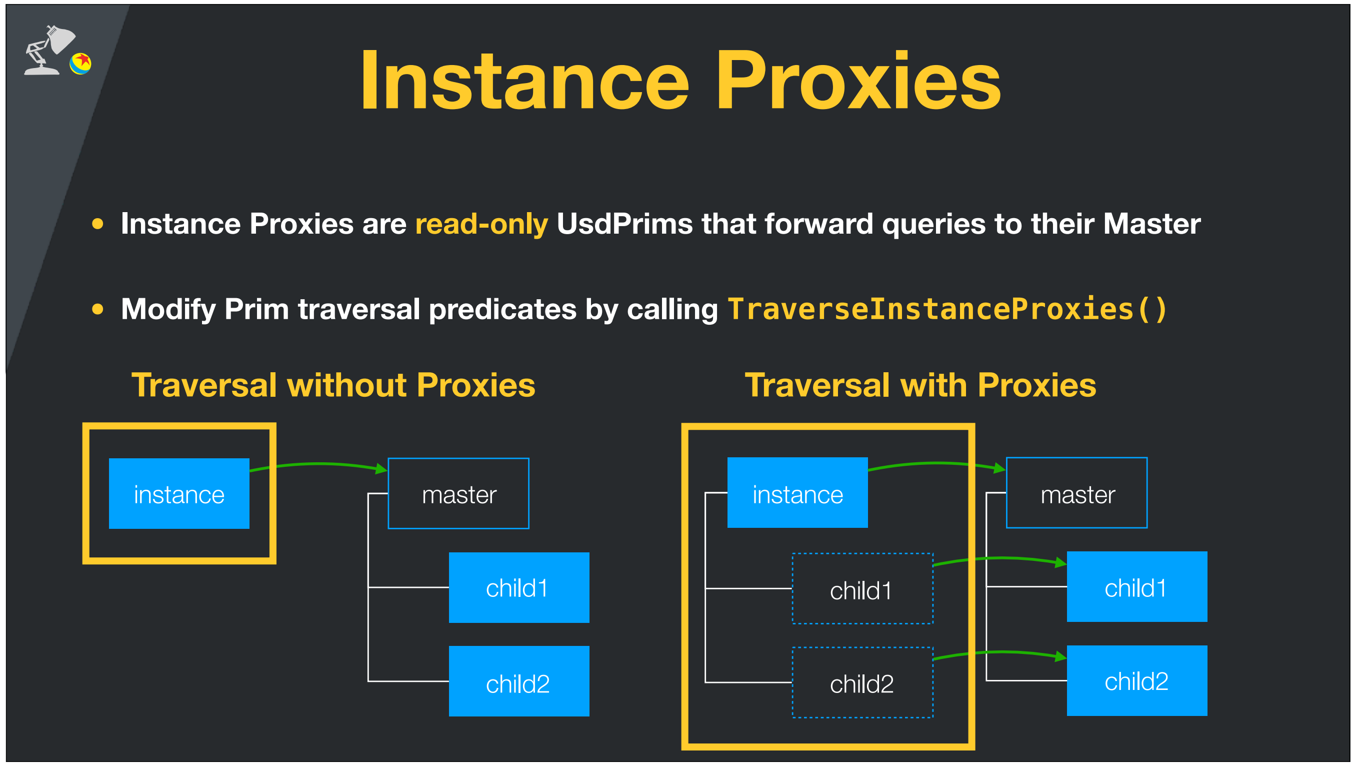
Native Instancing

- Prims with equal keys composed just once
  - Share a generated Master prim hierarchy
  - Local overrides on instance root prims allowed
  - Local overrides on descendant prims ignored
- UsdPrim API: `IsInstance()` and `GetMaster()`
- Full nested instancing supported



Native Instancing

- Most *efficient* way to process a UsdStage with instancing:
  - Call `stage->GetMasters()` and process all upfront or:
  - Call `prim.GetMaster()` during traversal and process if not yet seen
- Most *convenient* way to process a UsdStage with instancing:
  - Use Instance Proxies to pretend instancing doesn’t exist

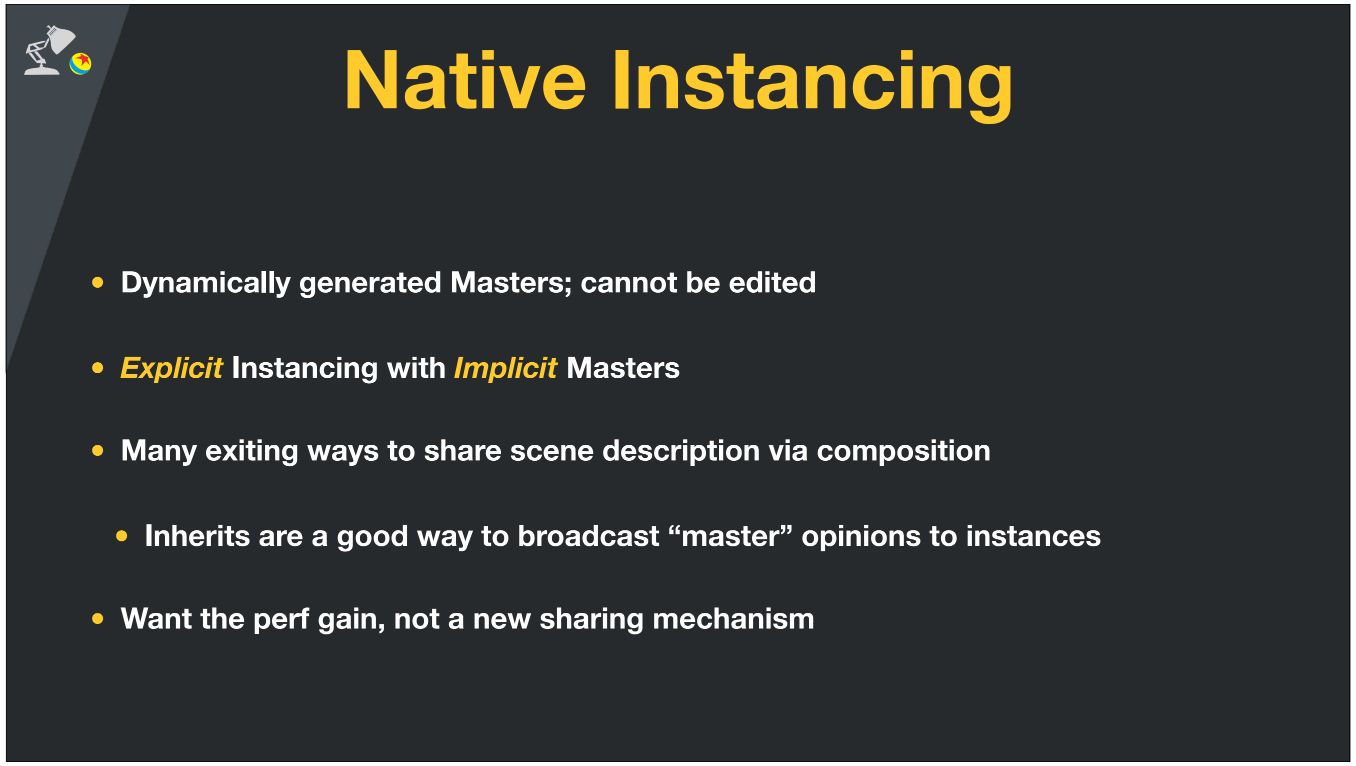


Instance Proxies

- Instance Proxies are read-only UsdPrims that forward queries to their Master
- Modify Prim traversal predicates by calling `TraverseInstanceProxies()`

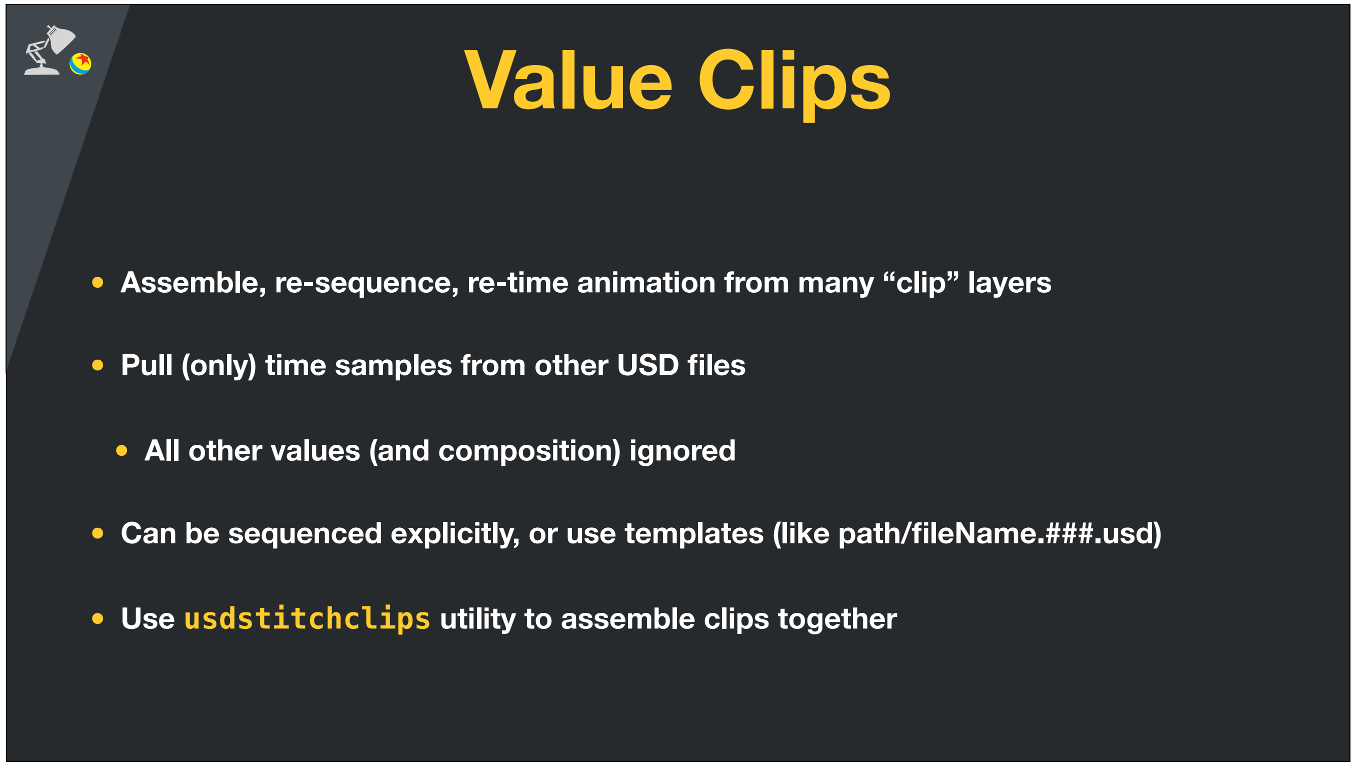
Traversal without Proxies

Traversal with Proxies



Native Instancing

- Dynamically generated Masters; cannot be edited
- *Explicit* Instancing with *Implicit* Masters
- Many exiting ways to share scene description via composition
  - Inherits are a good way to broadcast “master” opinions to instances
- Want the perf gain, not a new sharing mechanism

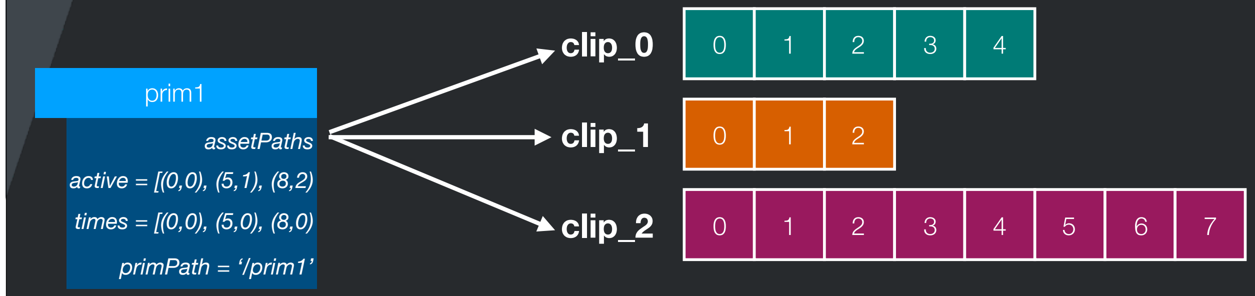


Value Clips

- Assemble, re-sequence, re-time animation from many “clip” layers
- Pull (only) time samples from other USD files
  - All other values (and composition) ignored
- Can be sequenced explicitly, or use templates (like path/fileName.###.usd)
- Use `usdstitchclips` utility to assemble clips together

This is really useful for certain workflows, like FX and crowds

Value Clips Example



prim1

assetPaths

active = [(0,0), (5,1), (8,2)

times = [(0,0), (5,0), (8,0)

primPath = '/prim1'

clip_0

0	1	2	3	4

clip_1

0	1	2

clip_2

0	1	2	3	4	5	6	7

prim1's timeSamples on the Composed Stage

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

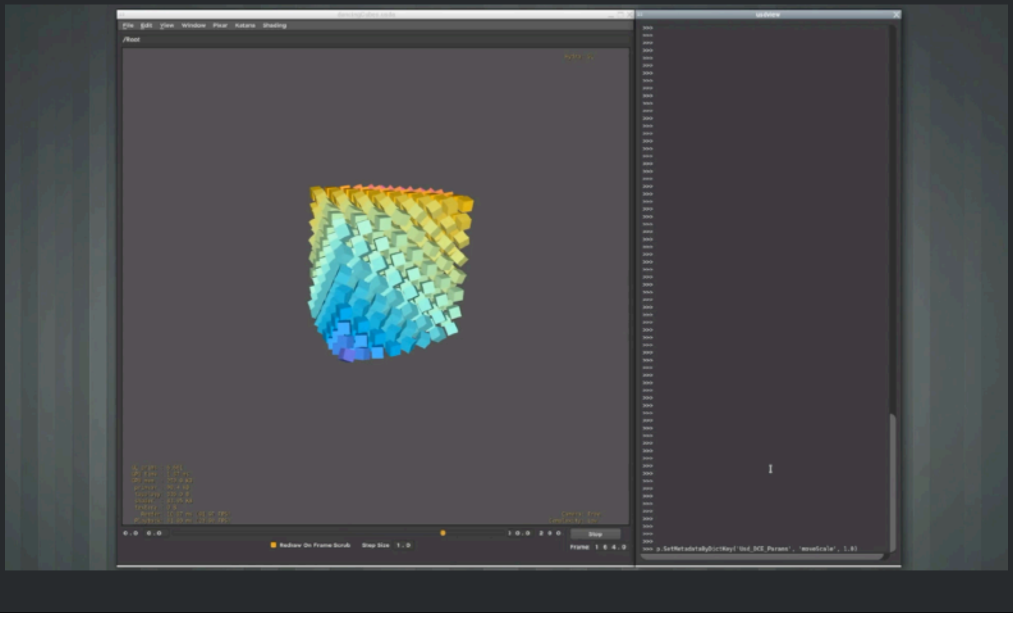
Dynamic File Formats

- Generate scene description parameterized by scene inputs
- Get composed “argument” values into your file format plugin
  - Currently restricted to custom plugin-registered metadata
- File format invoked to regenerate content when values change
- Careful! Must be “pure” and be thread-safe for concurrent readers

Example

```
#usda 1.0
(
    endTimeCode = 200
    startTimeCode = 0
)

def "Root" (
    # Dictionary value metadata field that provides all the parameters to
    # generate the layer in the payload. Change these values to change the
    # contents of the file.
    Usd_DCE_Params = {
        int perSide = 15
        int framesPerCycle = 36
        int numFrames = 200
        double distance = 6.0
        double moveScale = 1.5
        token geomType = "Cube"
    }
    # Payload to the dynamic file. The file must exist but its contents are
    # irrelevant as everything is generated from parameters above.
    payload = @./empty.usddancingcubesexample@
)
{
}
```

Example



```
>>> p.SetMetadataByDictKey('Usd_DCE_Params', 'moveScale', 1.0)
```