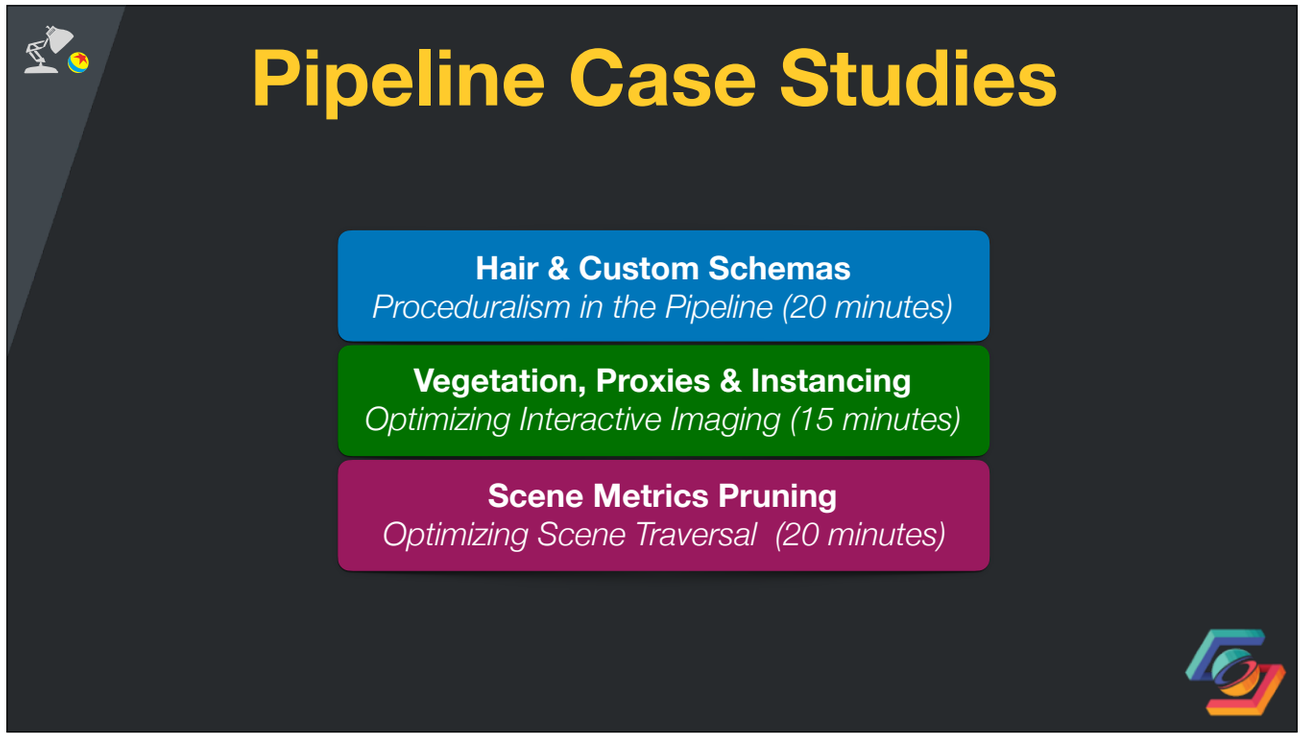
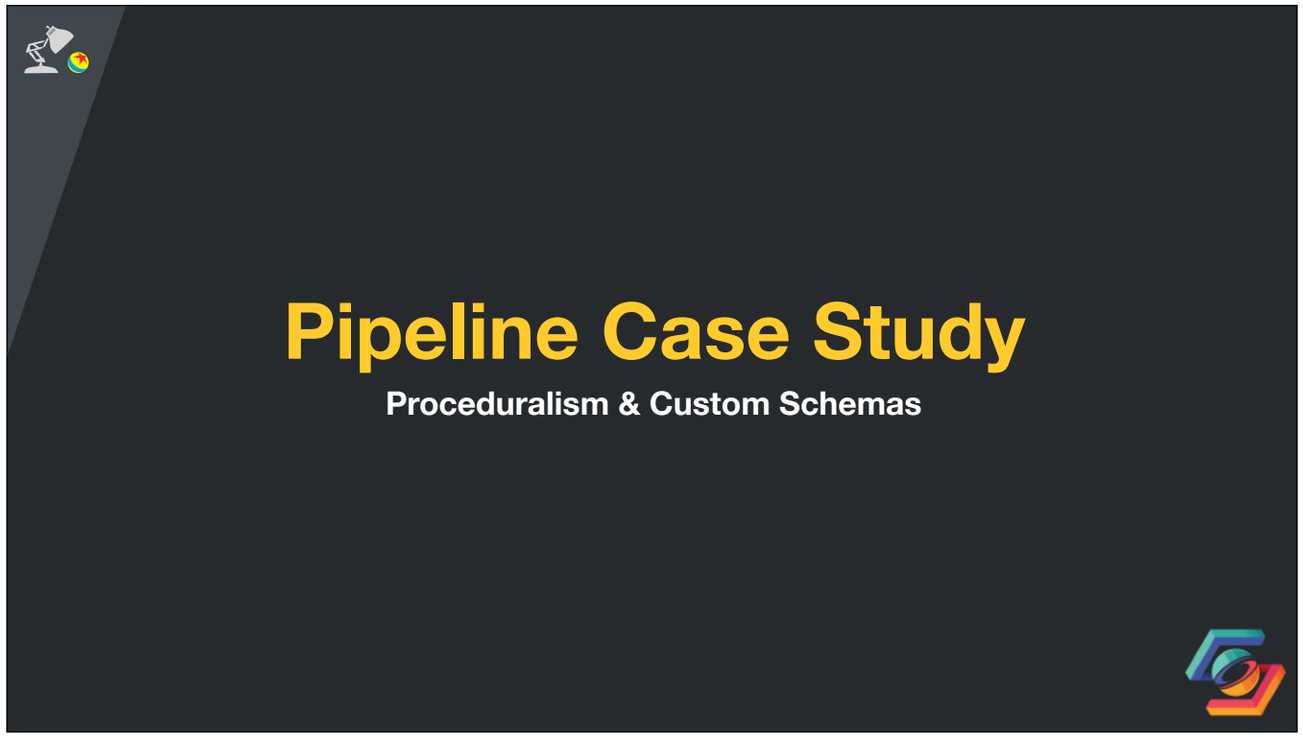

Pipeline Case Studies
Matt Kuruc
©Disney/Pixar

Goals

- Solving Scalability Problems Beyond Composition
- Learn How to Apply Strategies in Your Pipeline
- Help Us Evolve the USD Ecosystem

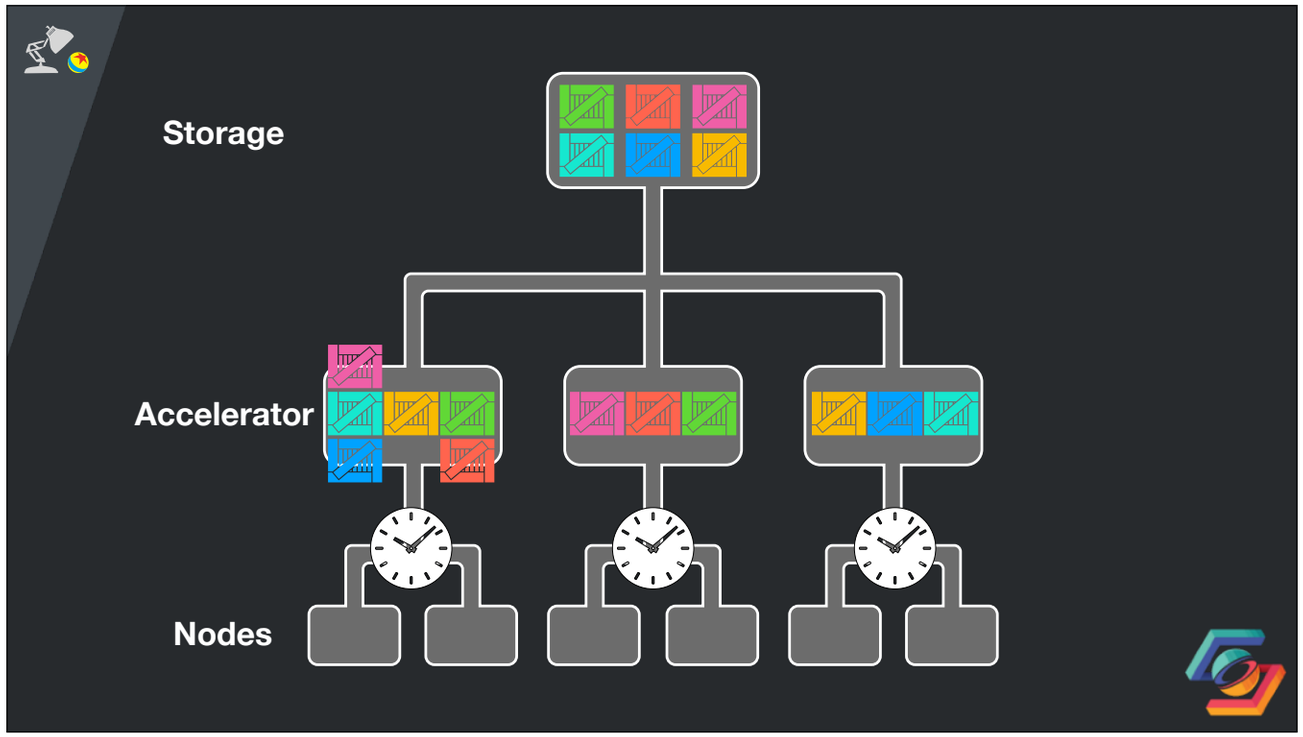


In this section, we'll be looking at three case studies from Pixar's pipeline



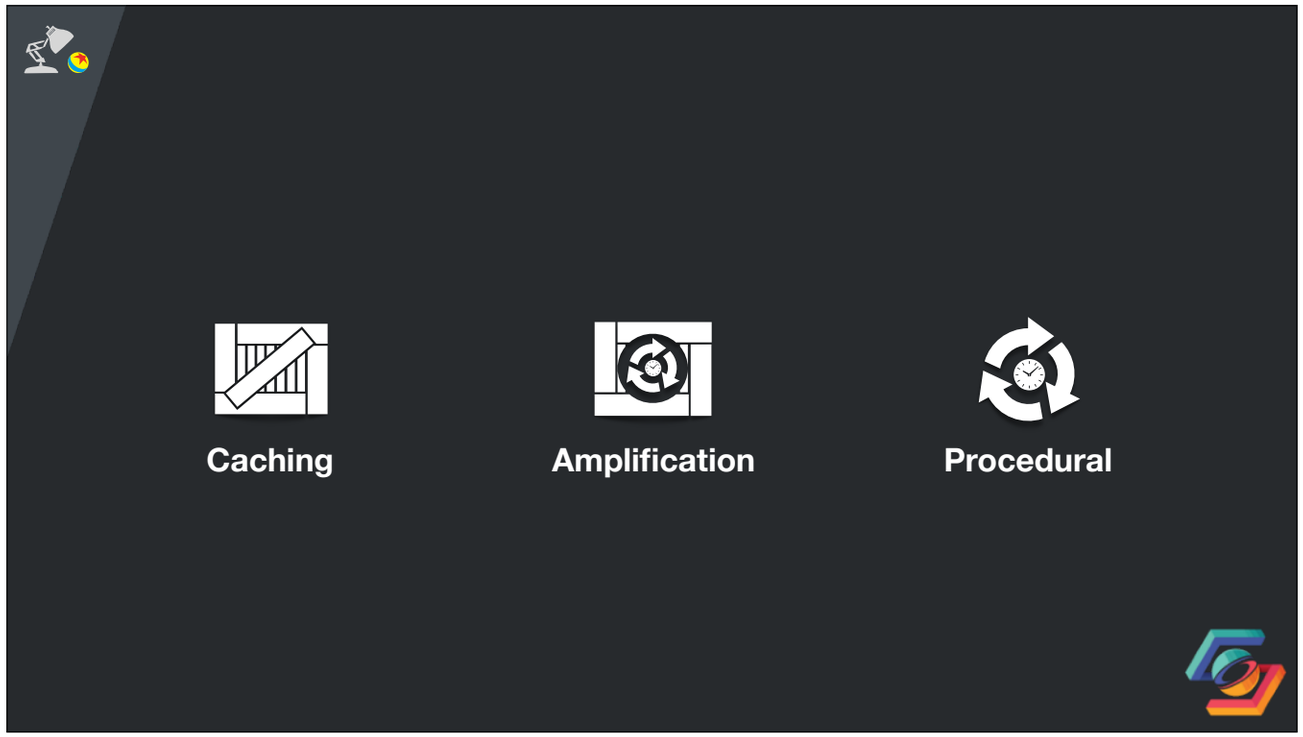
Let’s start talking about procedurals and custom schemas in the pipeline.

USD is a tool for composing cached scene graph data.
Why do we need to think about proceduralism and USD?

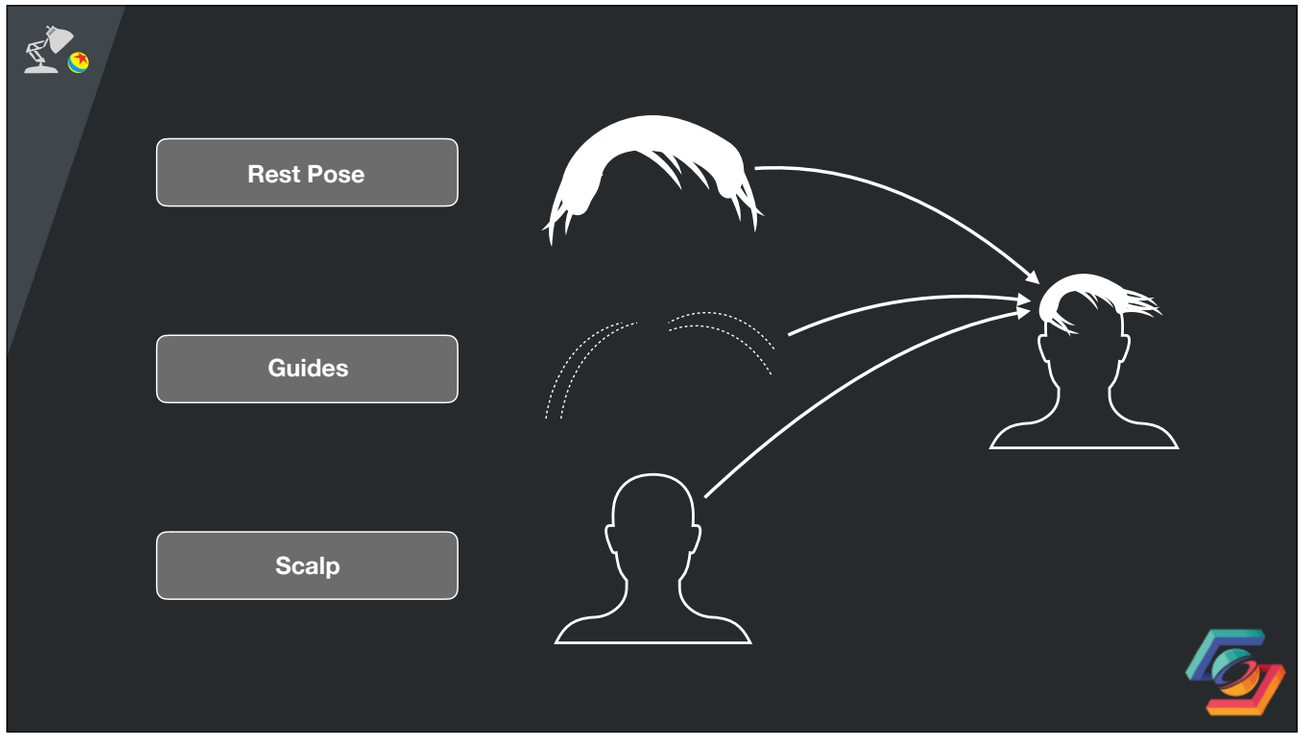


Here is a crude representation of a render farm. There's some storage, render nodes, maybe some accelerators. Nodes request data from storage. They can use the accelerator to avoid going back to the centralized repro. At some point we may overwhelm part of the system with I/O requests.

Caching works really well for most production data. Proceduralism has its own pipeline and performance issues. We're going to look at proceduralism not as a workflow, but as a targeted tool for keeping us from going off the performance cliff.

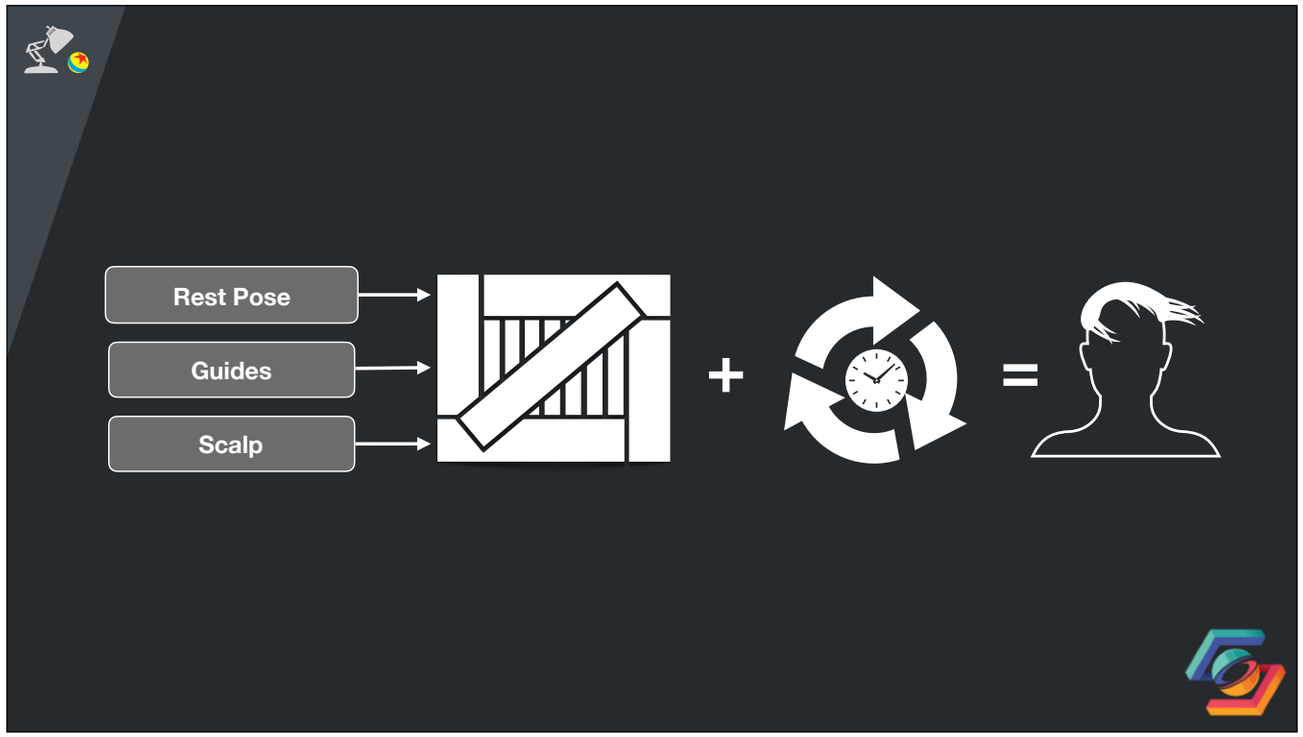


Specifically, we’re going to marry our cached data with a small amount of proceduralism to provide data amplification.

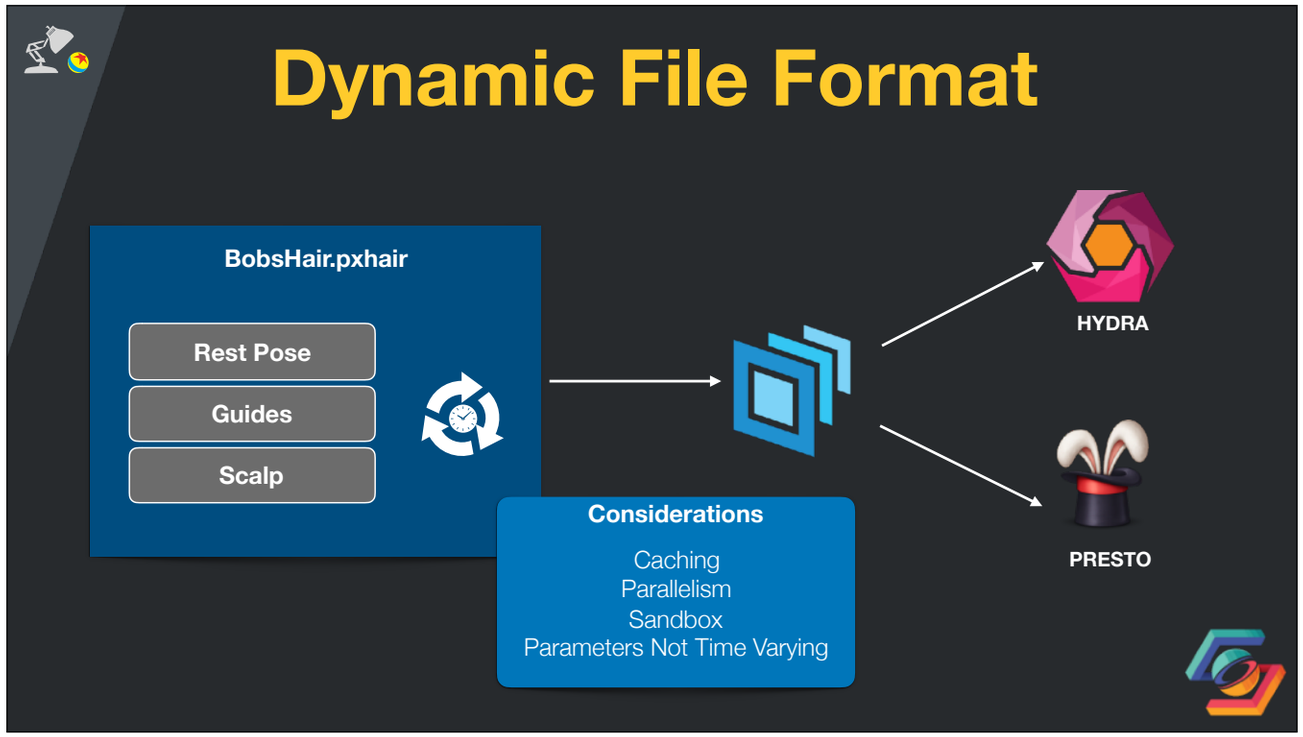


Data amplification was important on Incredibles 2 as our tools characeters team began to rework our hair pipeline. Bob’s hair was in 83,023 frames across 913 shots.

Violet has 4 times as much hair. It’s not just violet, not just the parr family, but crowds of citizens of Municiberg. The rest pose for Bob’s hair is only 100MB. We want a data amplification workflow that takes the 100MB rest pose and produces the 350GB worth of hair data needed without the I/O cost.
Most pipelines, ours included, approach the problem by caching out a rest pose of the hair, using animated guide curves and scalp mesh to pose the final hair.

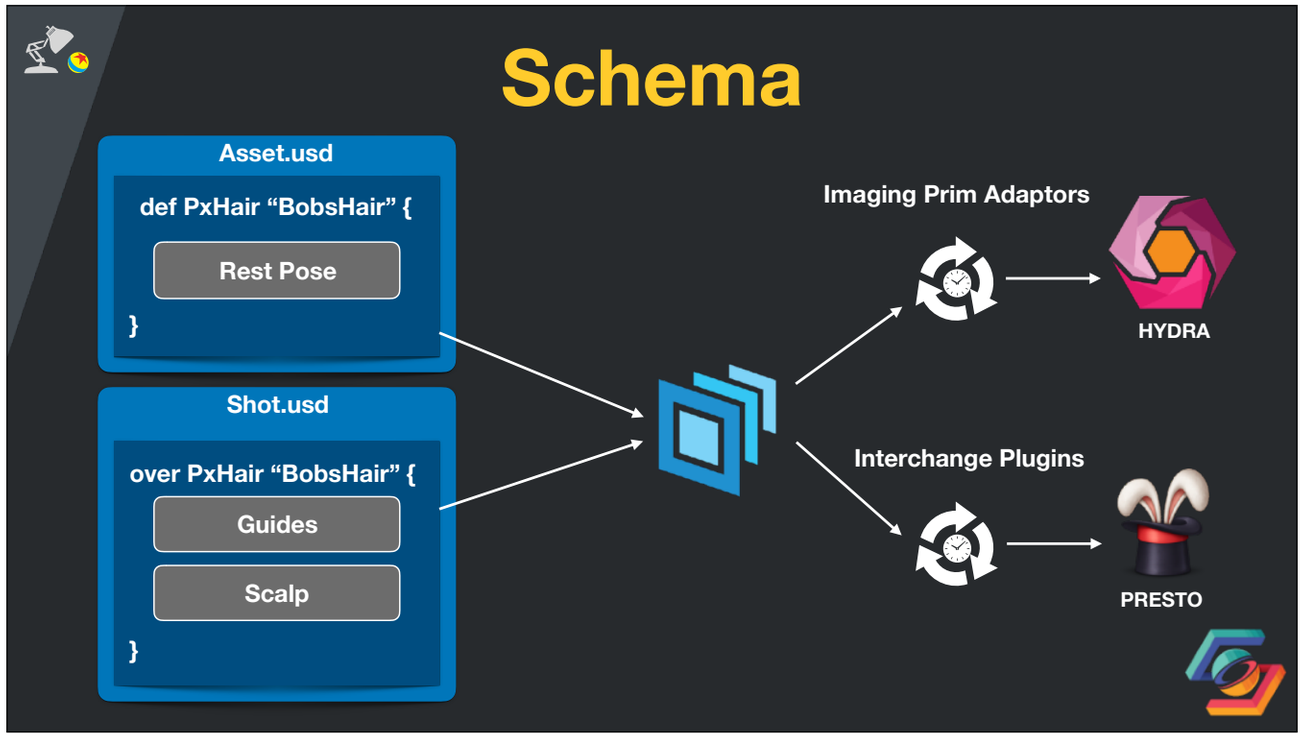


To reframe that, we want to store our rest pose, guide curves, and scalp mesh in USD and apply some computation to produce the final posed curves.

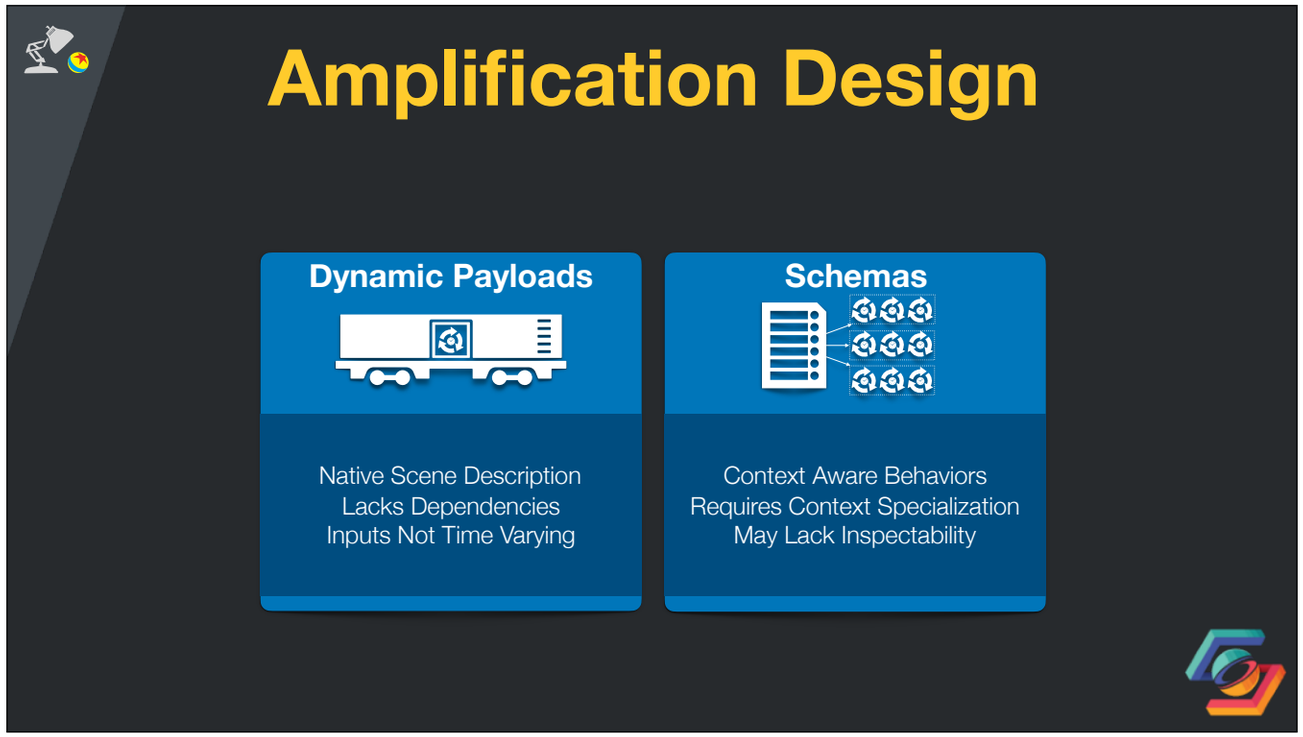


You saw dynamic file formats earlier in the talk, but we're not going to spend a lot of time exploring what this path looks like. A solution involving dynamic payloads would involve writing a file format plugin to store all of our data. The file format plugin can accept non-time varying metadata parameters to help guide procedural expansion of data. USD will compose this data just like a normal layer, and could feed our data into both Hydra and various DCCs.

There's a lot of considerations one needs to make before embarking on designing a file format plugin. Plugins are on the hook for maintaining their own caches to avoid over computation.
Threading is another consideration, USD is designed to be performant via parallelism. File format plugins are on the hook being reentrant. File format plugins exist as a sandbox, so they can't access external context from the USD stage or the imaging or interchange context.
Finally, dynamic file formats can only be parameterized by metadata.



At Pixar, we define hair as a new typed schema in Pixar’s pipeline. We can store our data in disparate layers, composing them via USD. We then can write plugins for imaging and interchange to translate our new schema into representations for Hydra and Presto.



We just introduced the two main ways to express data amplification in USD.
Dynamic payloads allow you to parameterize file format plugins to generate native scene description, at the cost of restricting the inputs to your plugin to be the layer identifier and non-time varying metadata. Schemas offer you a way to describe a procedural workflow, providing a type to key off of for translation plugins in Usd Imaging and various DCCs.

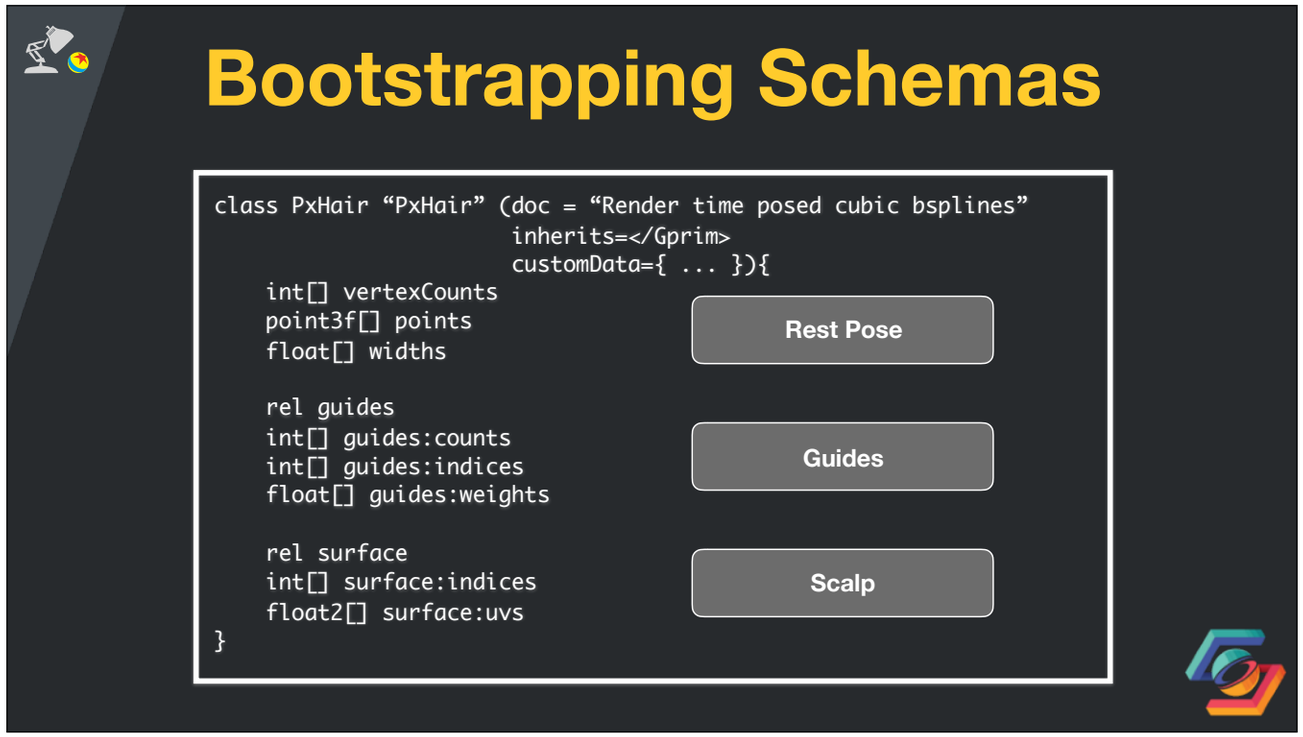
At Pixar, we use schemas to describe data amplification, so let’s do a deeper dive into defining schemas for our hair workflow.

Schemas

- Typed Schemas
- API Schemas

Typed schemas you’re probably familiar with. Things like UsdGeomMesh, UsdGeomXform
API Schemas are often used as extension schemas, like our UsdCollectionsAPI.

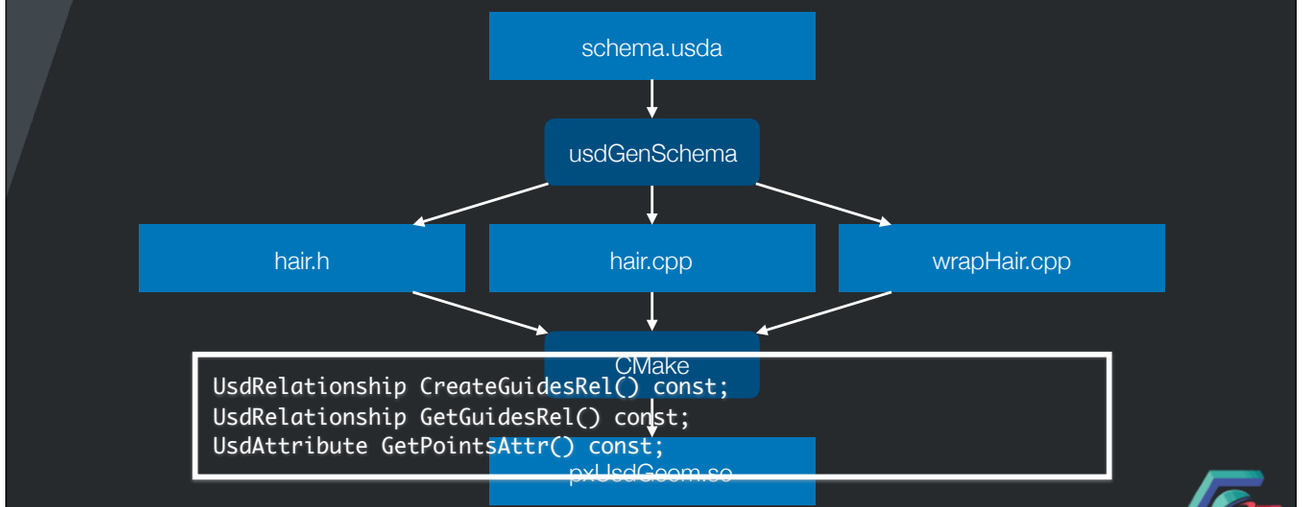
We’re going to start by looking at building a typed schema to describe our hair, and we’ll come back to API Schemas later.

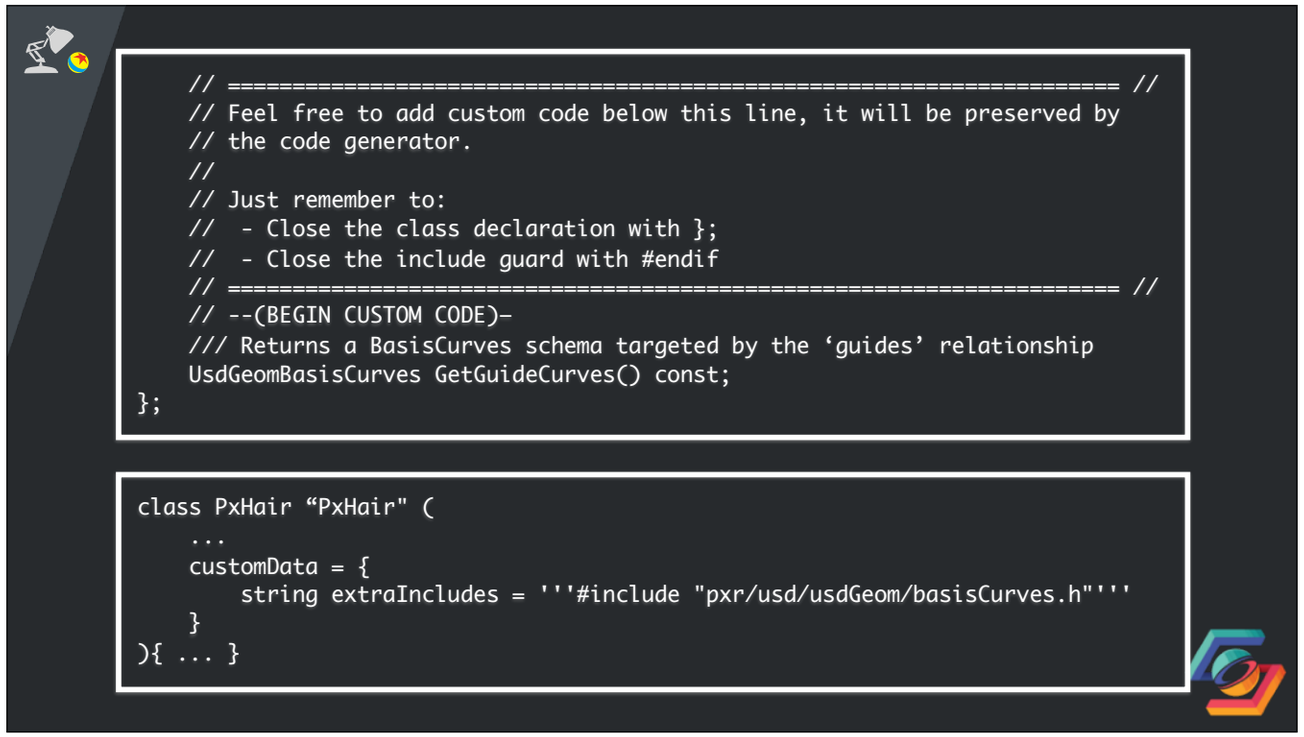


You can actually bootstrap new schemas using USD. This is how many domains including usdGeom, usdShade, and usdLux are all specified

Hair schema is broken into three parts
— Rest Pose: points, topology, and width attributes
— A relationship to guide curves / key hairs / sim curves used by simulation
— A relationship to a scalp surface to constrain hair to

Bootstrapping Schemas

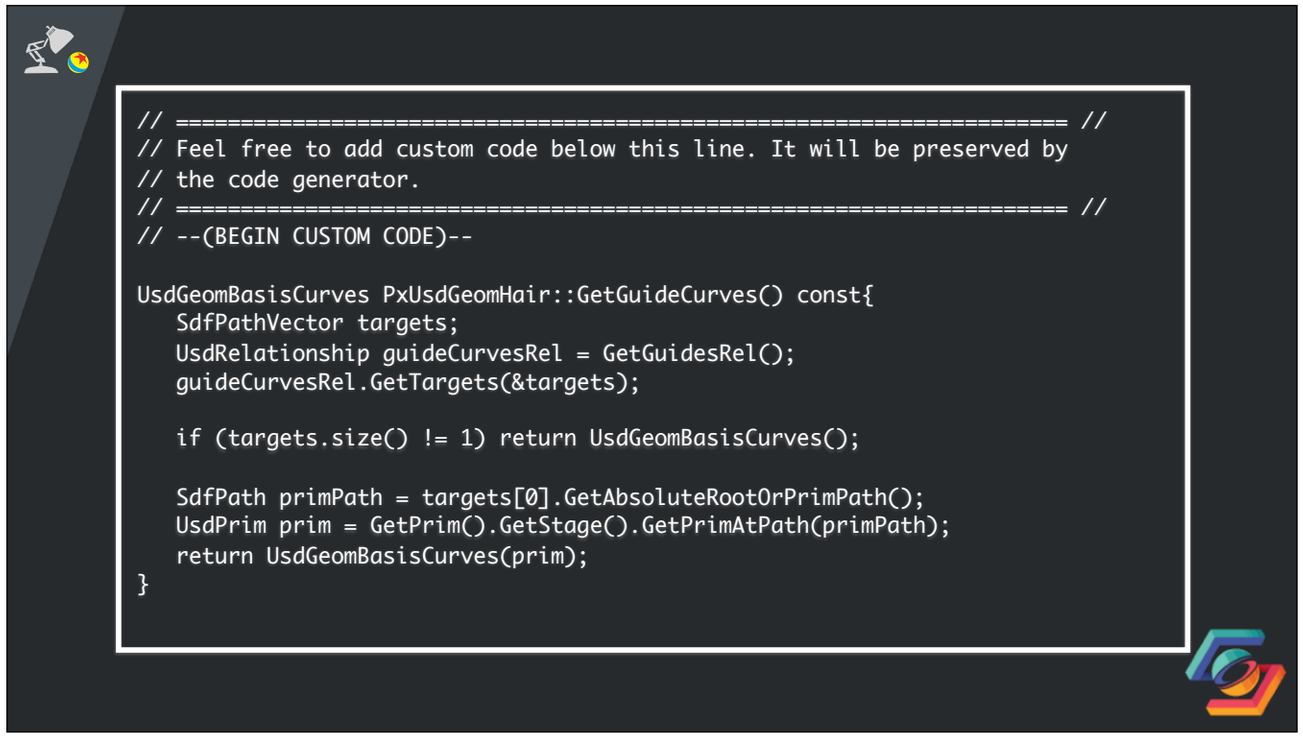




We can still add custom code to our schema class. Just look for the “BEGIN CUSTOM CODE” hint.

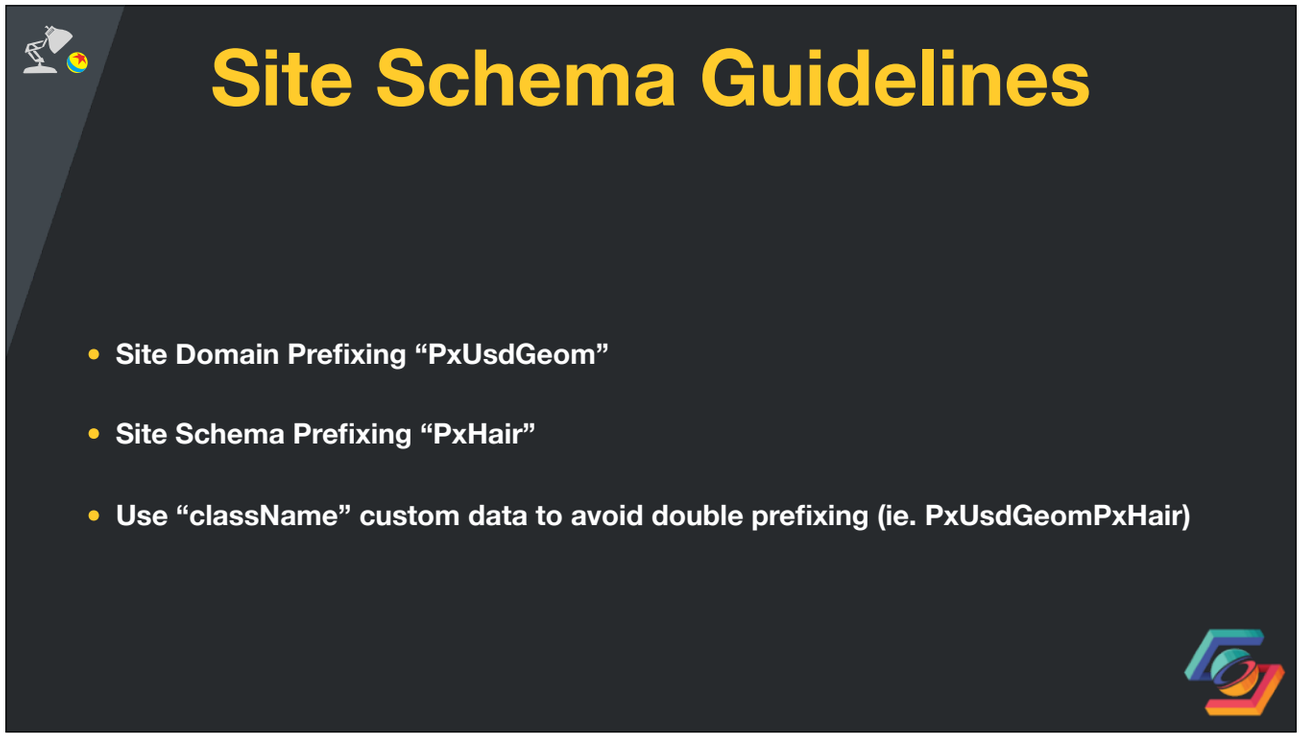
Our guide curves are defined via relationship to a Basis Curves prim. There may be some boiler plate validation we may want to extract away in converting that relationship target to a BasisCurves schema. So we can drop in GetGuideCurves method to help schema users out.

As we have added a dependency to another header with this change, we have to update our schema’s custom data as well.

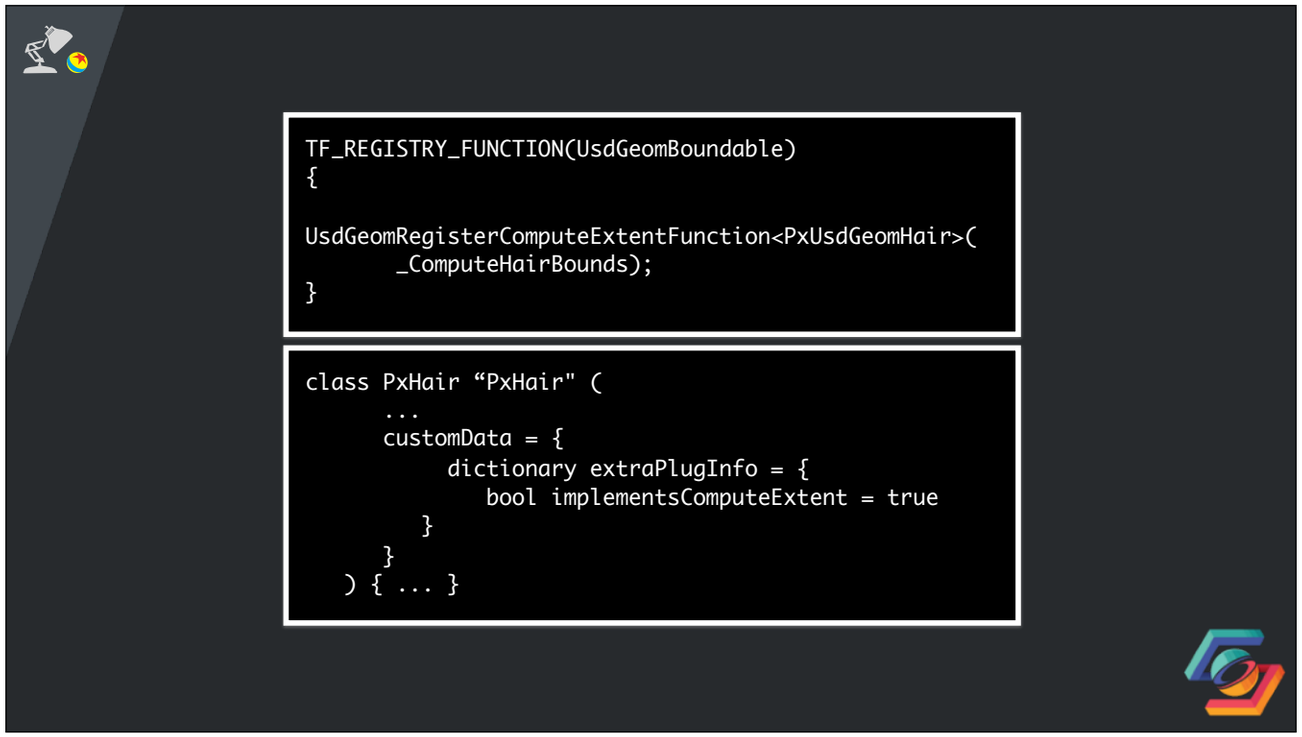


What does the implementation of this method look like?

First we get all the relationship targets.
We make sure there is one and only one targeted prim.
And finally, we return a schema bound to the prim at the targeted path.



Pixar internal schemas are prefixed to avoid collisions and make migrations to open source easier.



One important part of defining new geometric schemas is boundability.

It’s important for clients of USD to be able to spatially track the a gprim in the scene. Bounds provide a common spatial representation used in many optimization workflows including pruning and level of detail pipelines. We want to allow tools to make decisions about whether or not its important to deal with a prim without having to know specifics of the type.

For many workflows, the bounds can be computed at export time, but the extent of many custom schemas have dependencies on other prims.

We provide bounds computation plugins as a way for tools to track and spatially analyze the scene in a schema agnostic way.

Bounds Computation

- Bounds of external prims
  - No “bounds dependency graph”
  - Don’t trust cached values are up to date
- Fast Loose Bound
  - Correct, Not Necessarily Tight
  - Minimize Dependencies

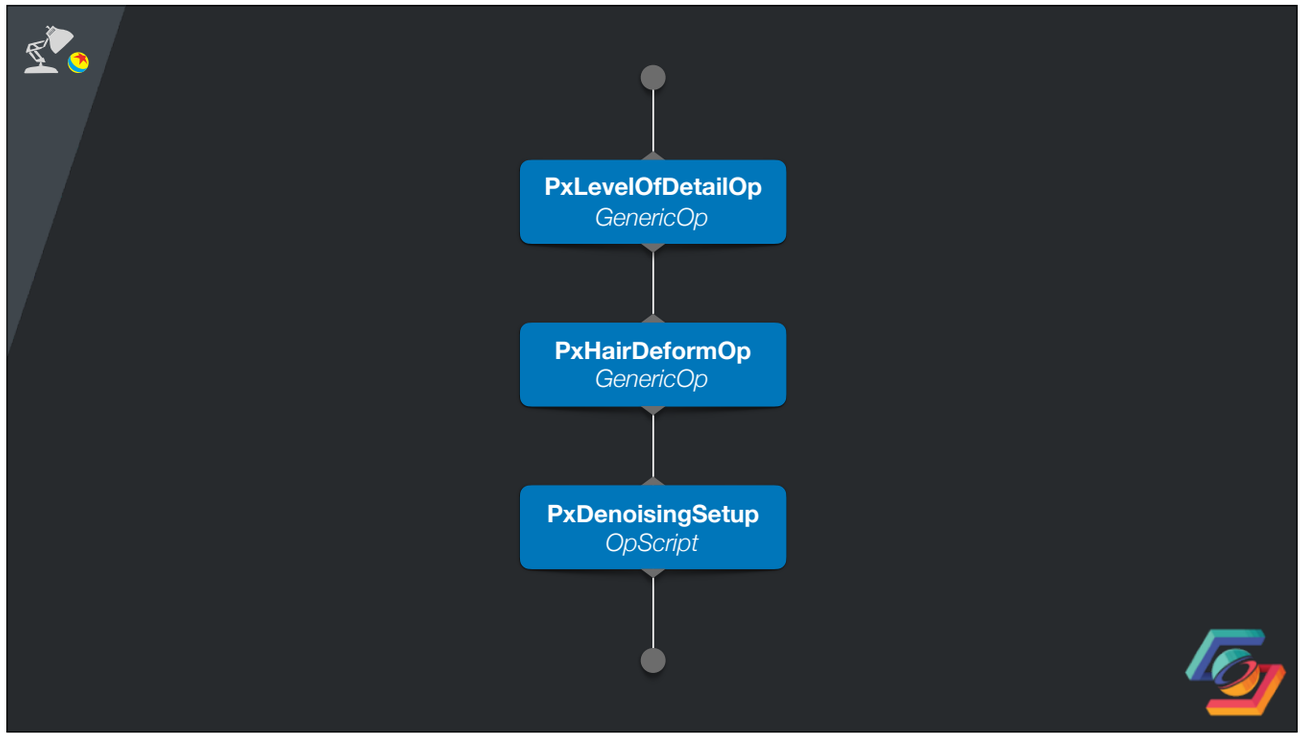
TIP#1 - Recompute the bounds of external prims.

TIP#2 - These bounds are not the bounds you would send to a ray intersection bounding volume hierarchy. Bounds computation of a subdivision surface is just a min/max on the cage points. It doesn’t link against OpenSubdiv and run Catmull-Clark subdivision. Keep your bounds plugins simple.

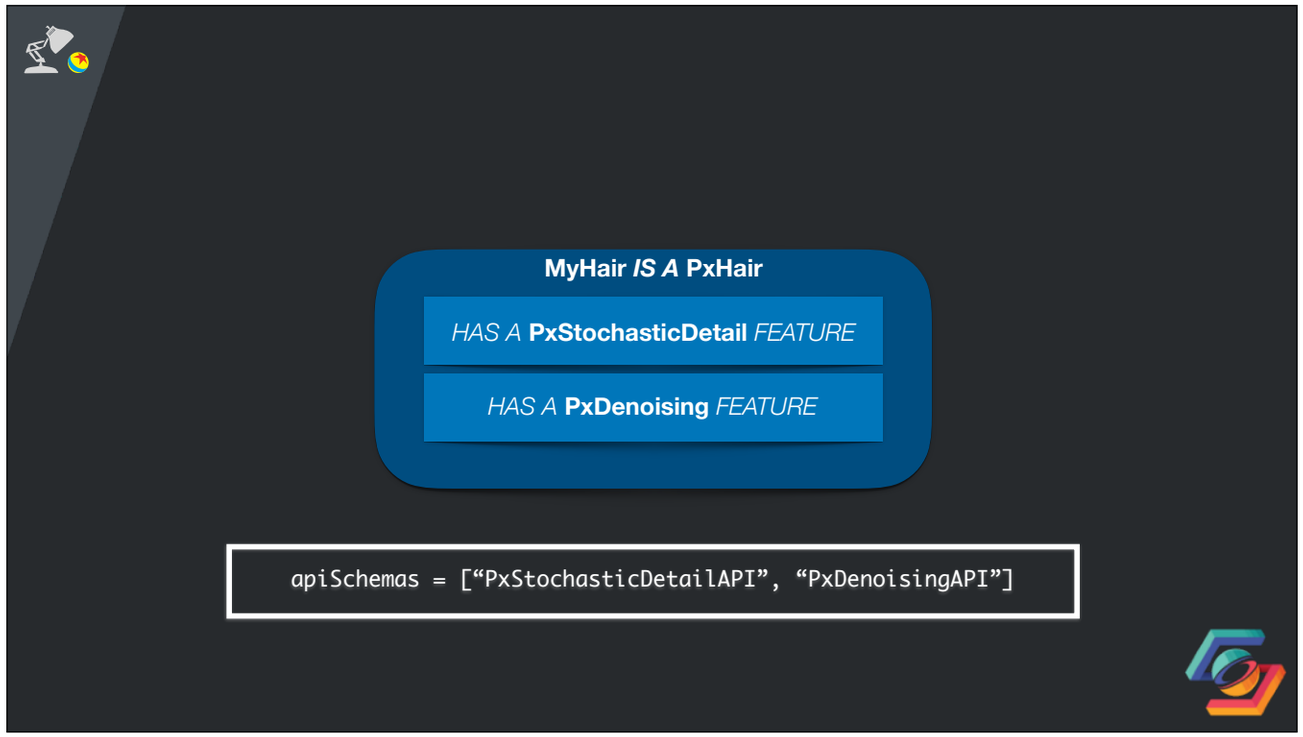
Now that we've defined our schema, and we know how to bound our schema, it's time to bring this data into a tool. Let's look at what it takes to bring PxHair into katana. Luckily, usdKatana offers helpful macros and utilities to writing an interchange plugin easier.

Plugins like the PointInstancer translator op are only 113 lines of code (including the USD copywrite notice). The UsdKatana utility library makes writing your own interchange plugins for your pipeline straight forward. Similar libraries exist for other DCCs like Maya.

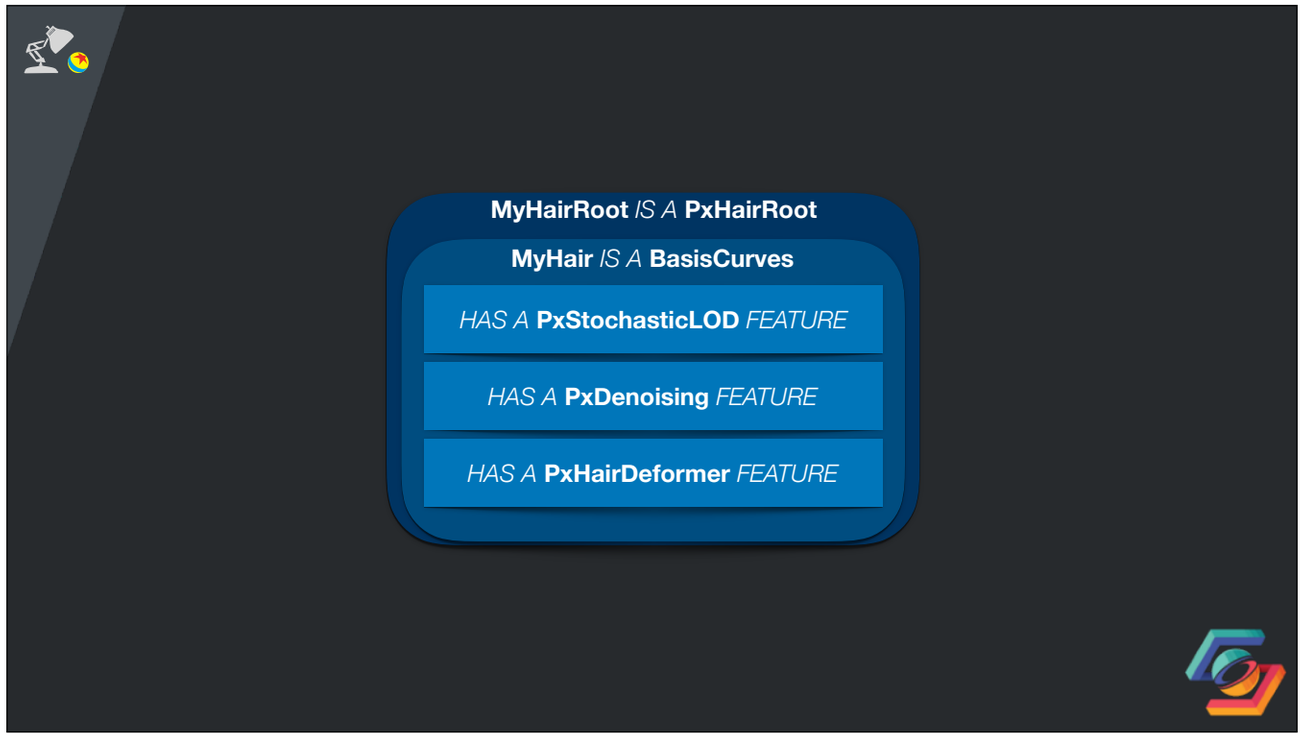
You have multiple options for running deformation.
— You could run the deformation on import.
— You could attach a child op that’s resolved by katana.
— We’re going to put our deformation into a katana op in the katana node graph.
— We like this approach because it makes it really easy to see, debug, and modify the operation order.



— Hair isn’t just deformation. There’s also level of detail. There’s also denoising.
— We could just keep growing our Hair schema, but we don’t want a swiss army knife
— We want to be able to add and remove features on demand, and separate functionality



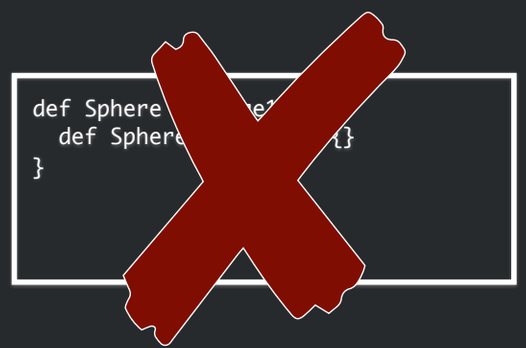
Objects can only have one type in USD, but they can subscribe to many API schemas.



You may wonder if we could've expressed our hair deformation as an API schema. Many parts of the USD schema key off of only the type of a prim. However, we can introduce a parent typed schema as a way of getting around that restriction.

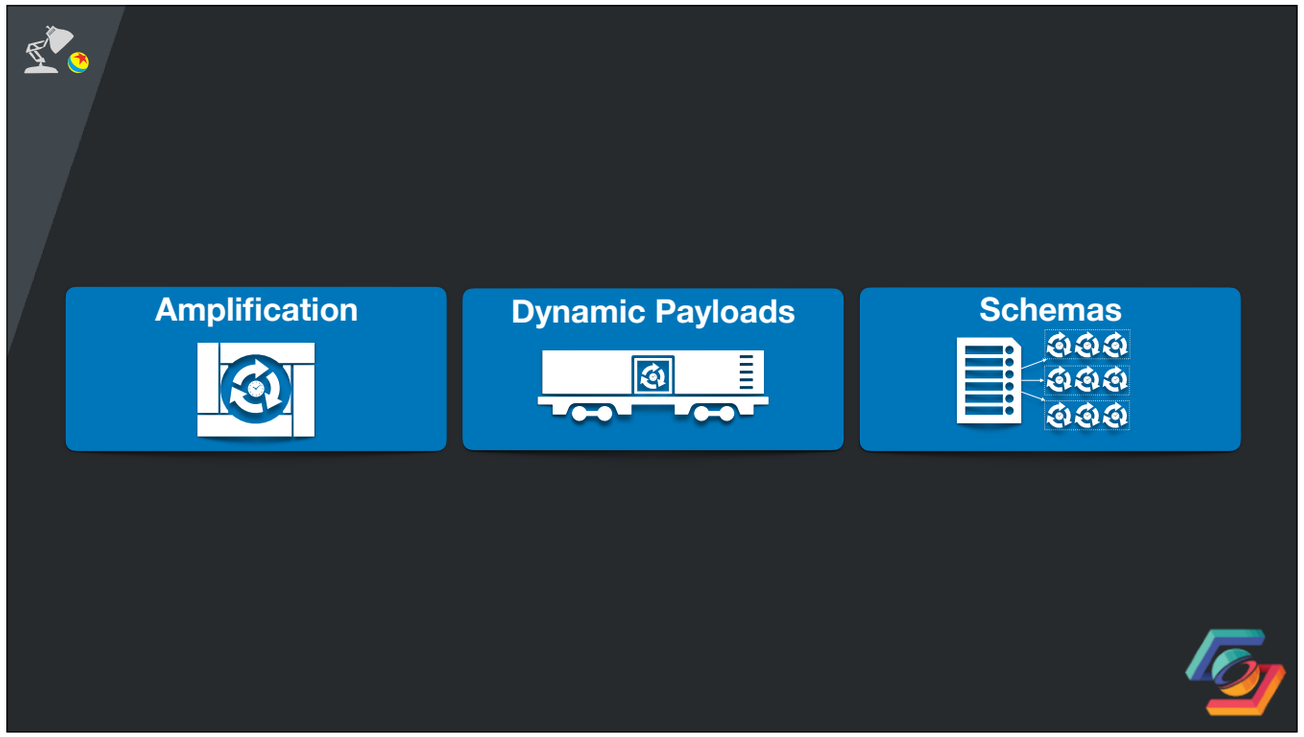
Boundable Root Pattern

```
def PxHairRoot “MyHairRoot” {
  def BasisCurves “MyDeformingHair” (
     apiSchemas = [“PxHairDeformerAPI”, “PxStochasticDetailAPI”]){}
}
```



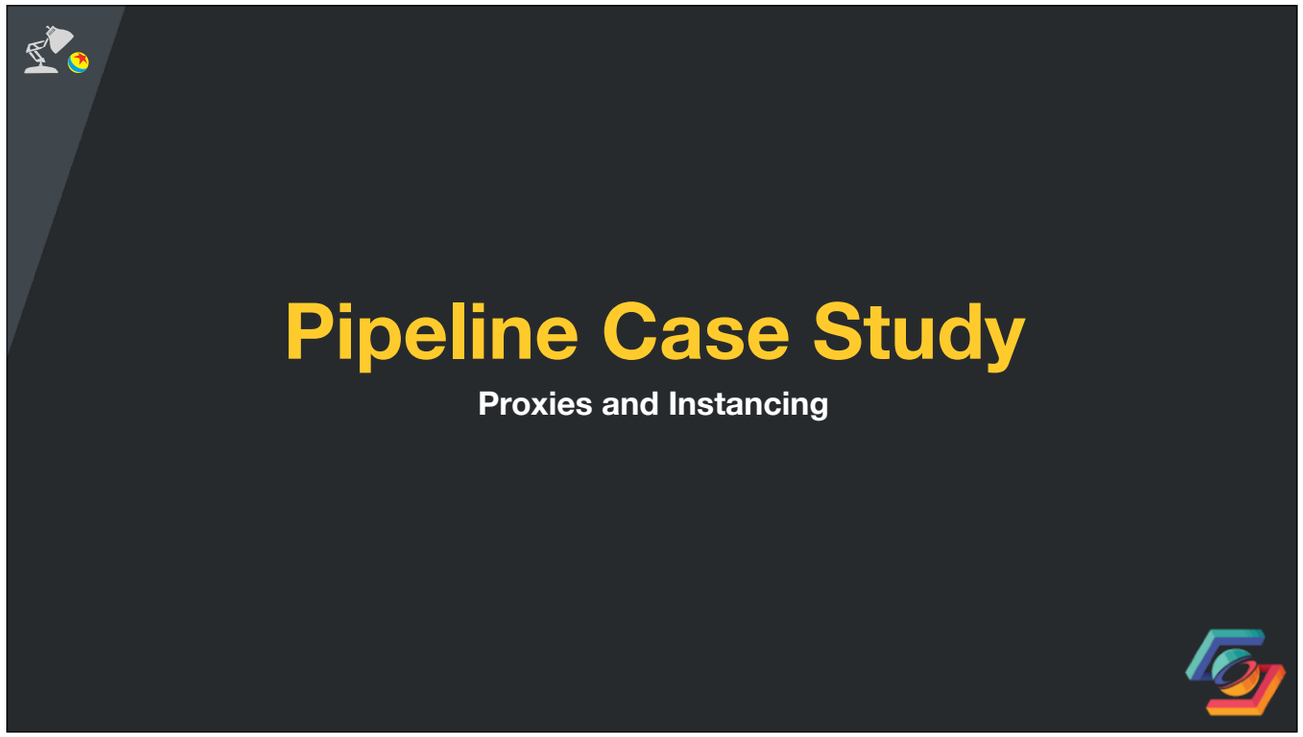
We call this the boundable root pattern. We have a root prim which we key off for things like bounds computation and interchange, allowing the rest pose be easily expressed as a standard USD Gprim.

We call this the boundable root pattern because PxHairRoot must be of type boundable.
The schema definition of a gprim defines them as leaf level scene elements and thus cannot be nested. What does it mean for a Sphere to be parented to a sphere?
Boundables exists as a way of annotating a boundable subgraph of the scene that isn’t leaf geometry.

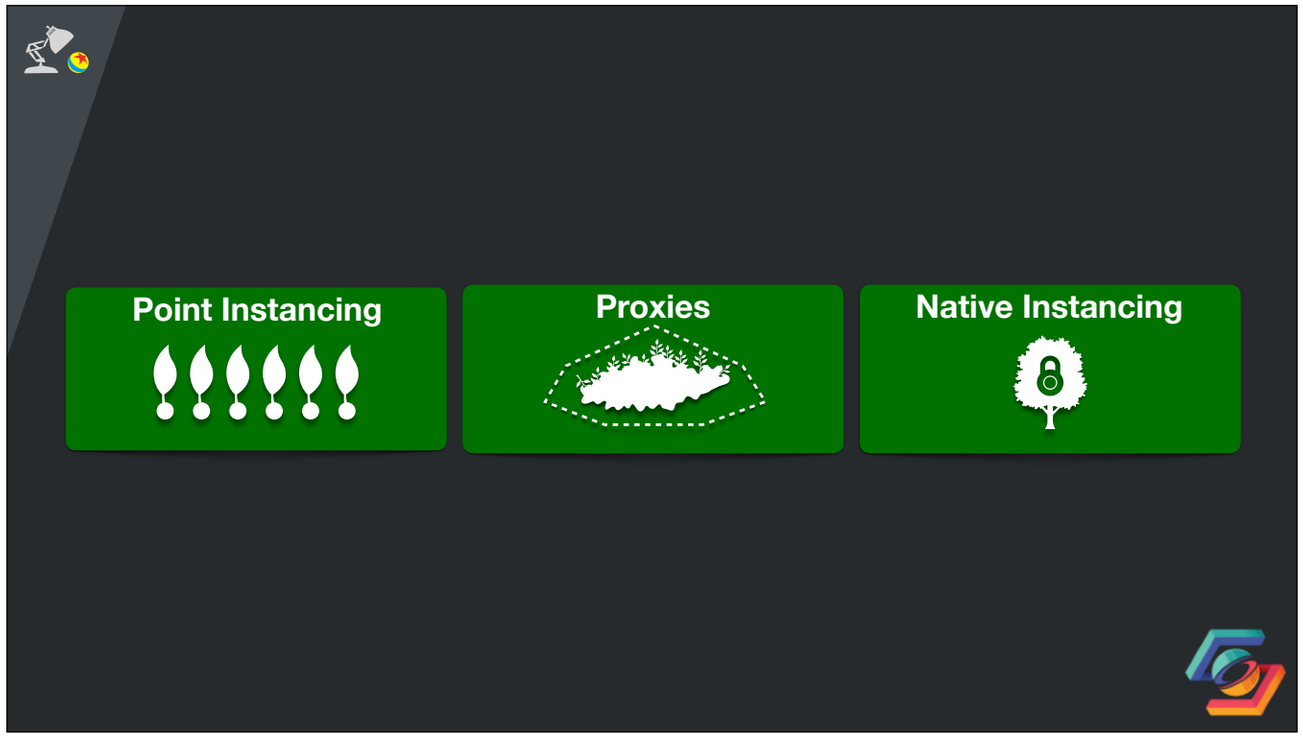


We’ve talked about how when approaching production scalability problems, we can use targeted procedurals to amplify cached USD data.

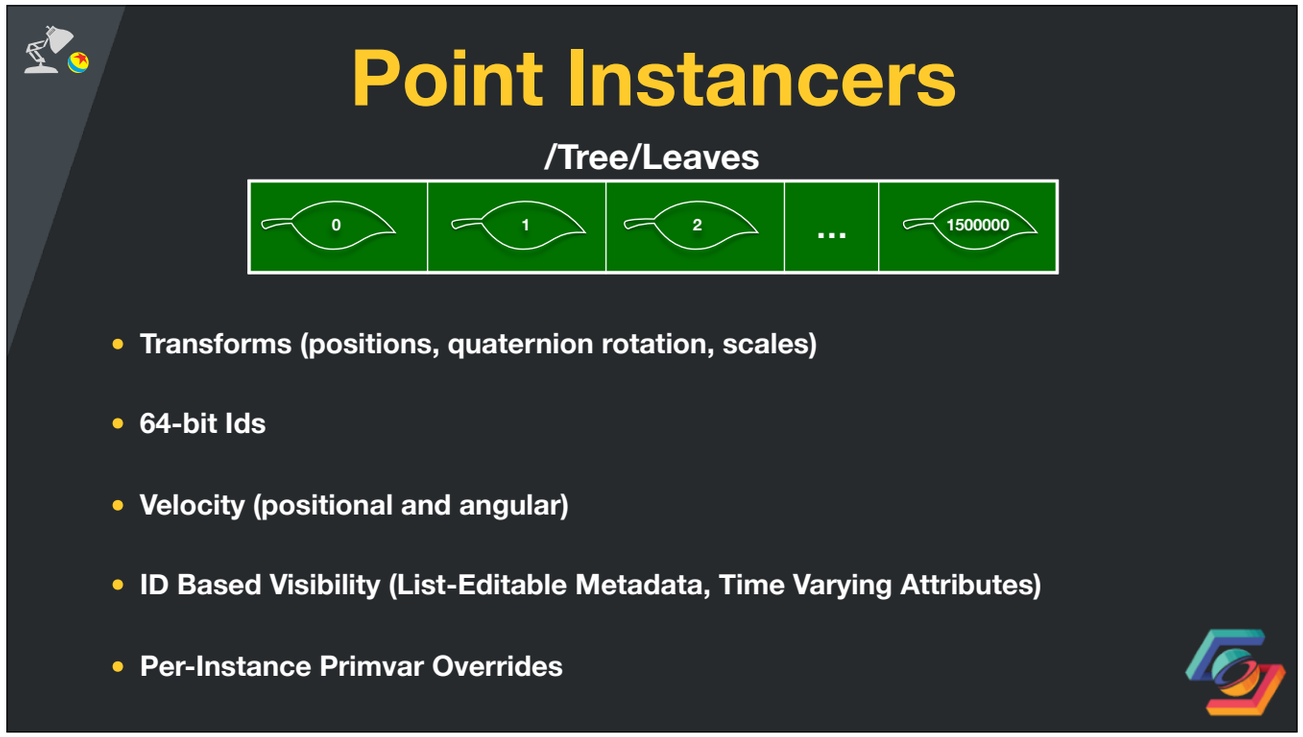
We explored dynamic payloads as one vehicle for expressing this.
We then did a deeper diver into expressing Hair as schemas, focusing on interchange with Katana. We explored typed schemas and API schemas and saw the boundable root pattern where we can leverage them in tandem.



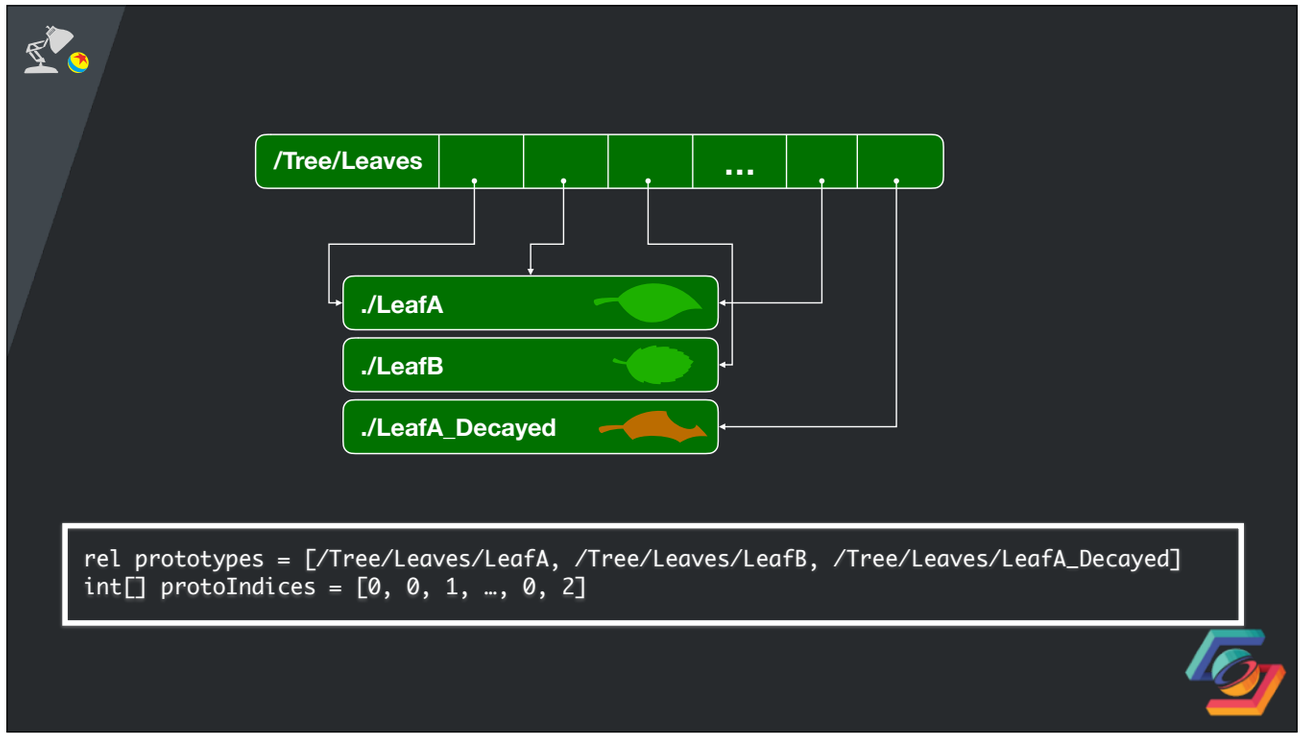
GOAL— We want to compose and interactively image as much vegetation as possible as fast as possible and with as little memory as possible.



— Point instancers to reduce the number of unique prims required to describe a tree
— Proxies to reduce what we’re drawing interactively
— Native instancing can be used as an accelerator for composition and imaging

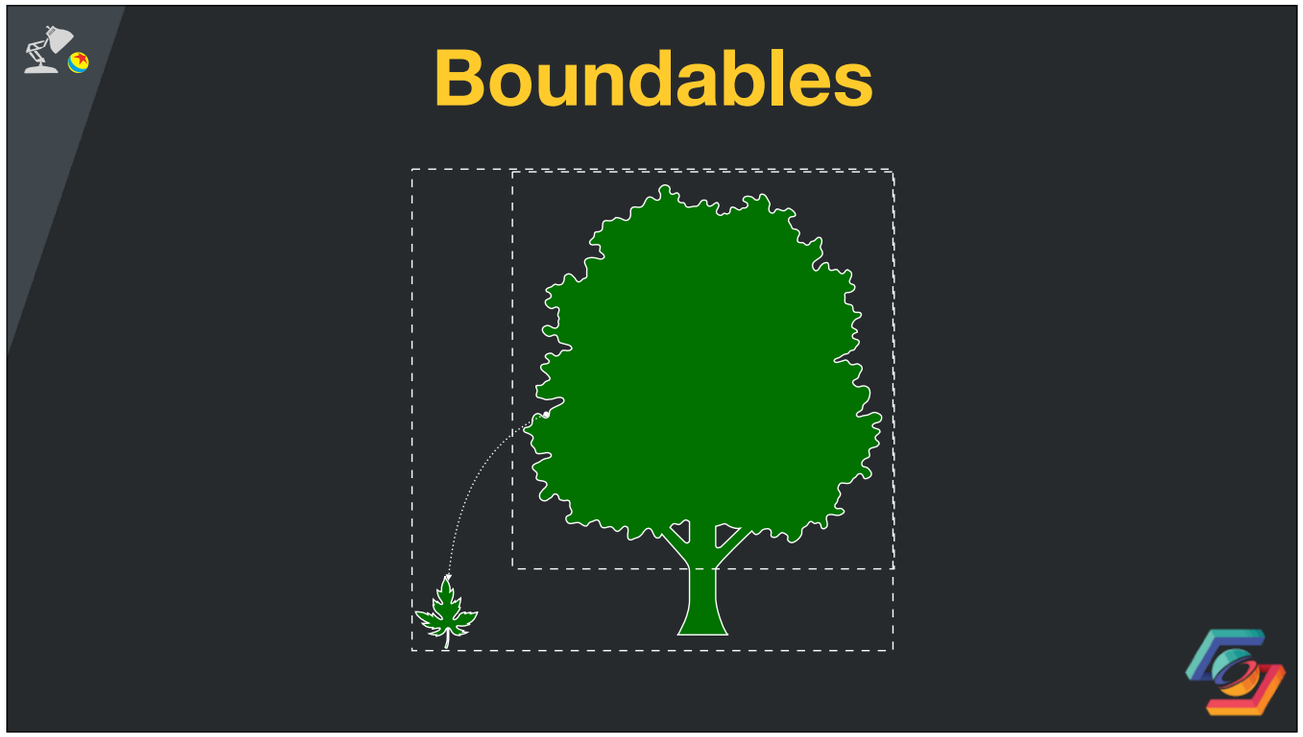


We want an efficient USD representation that allows us to represent all 1.5 million leaf instances as a single prim, with each element addressable by not by path but by index.



The point instancer representation is great for describing granular elements like leaves on a tree. But how do we describe what we instance? We’e going to nest the geometry we want to instance underneath the point instancer prim. We’ll call these prims prototypes.

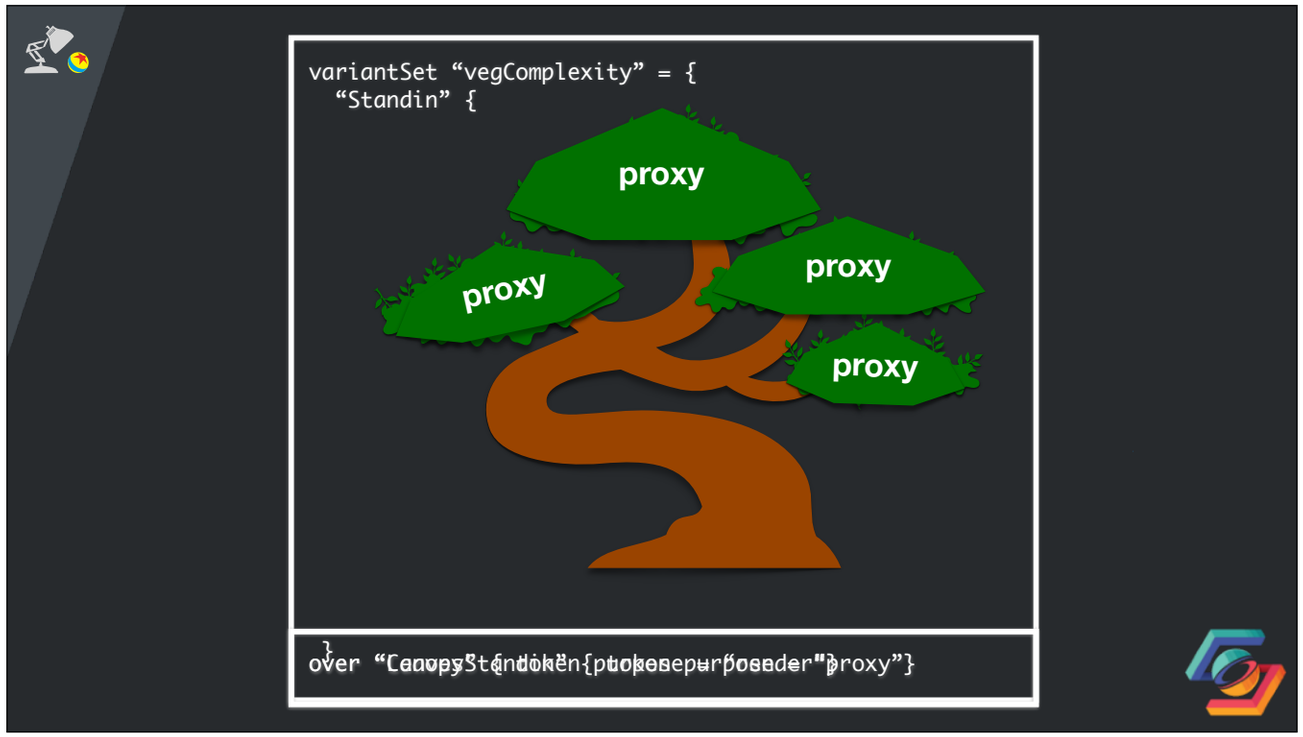
Variants of the same model must be defined as new prototypes



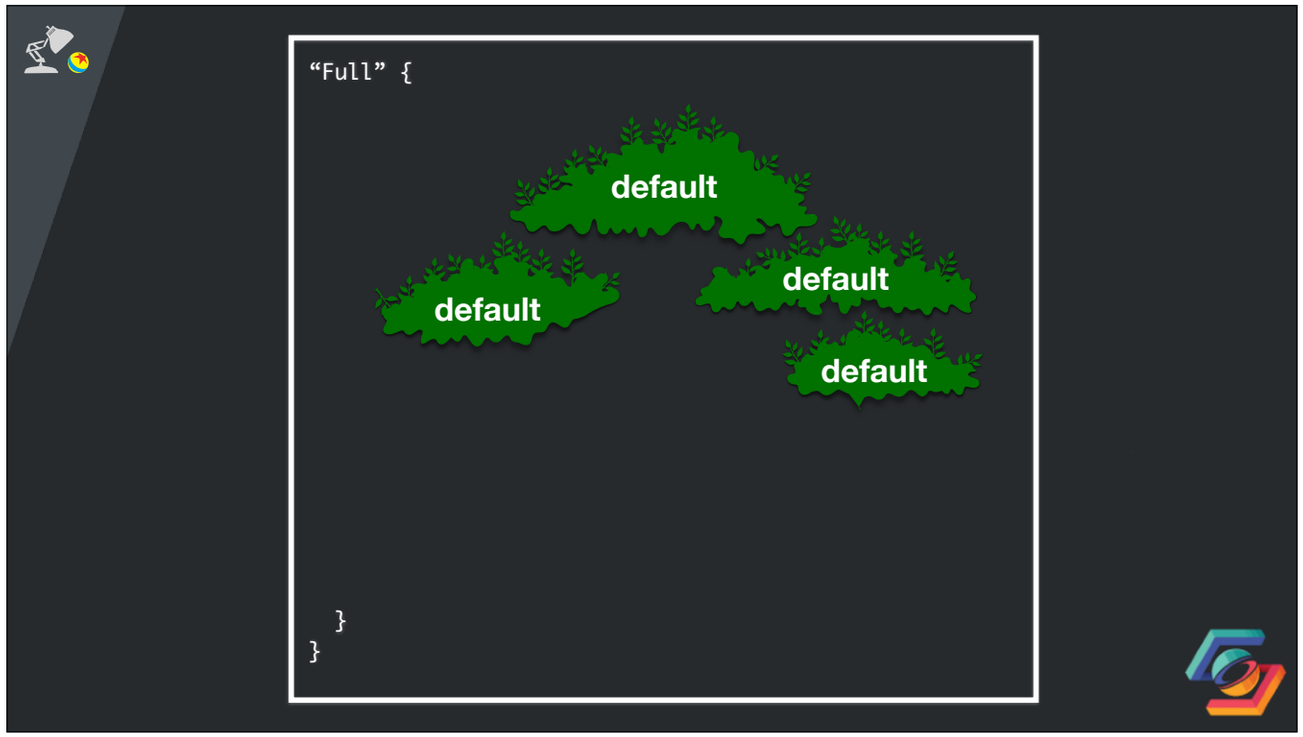
Point instancers are another example of a boundable. It significantly transforms its descendent geometry, in this case multiplying the number of instances of that object in the scene.

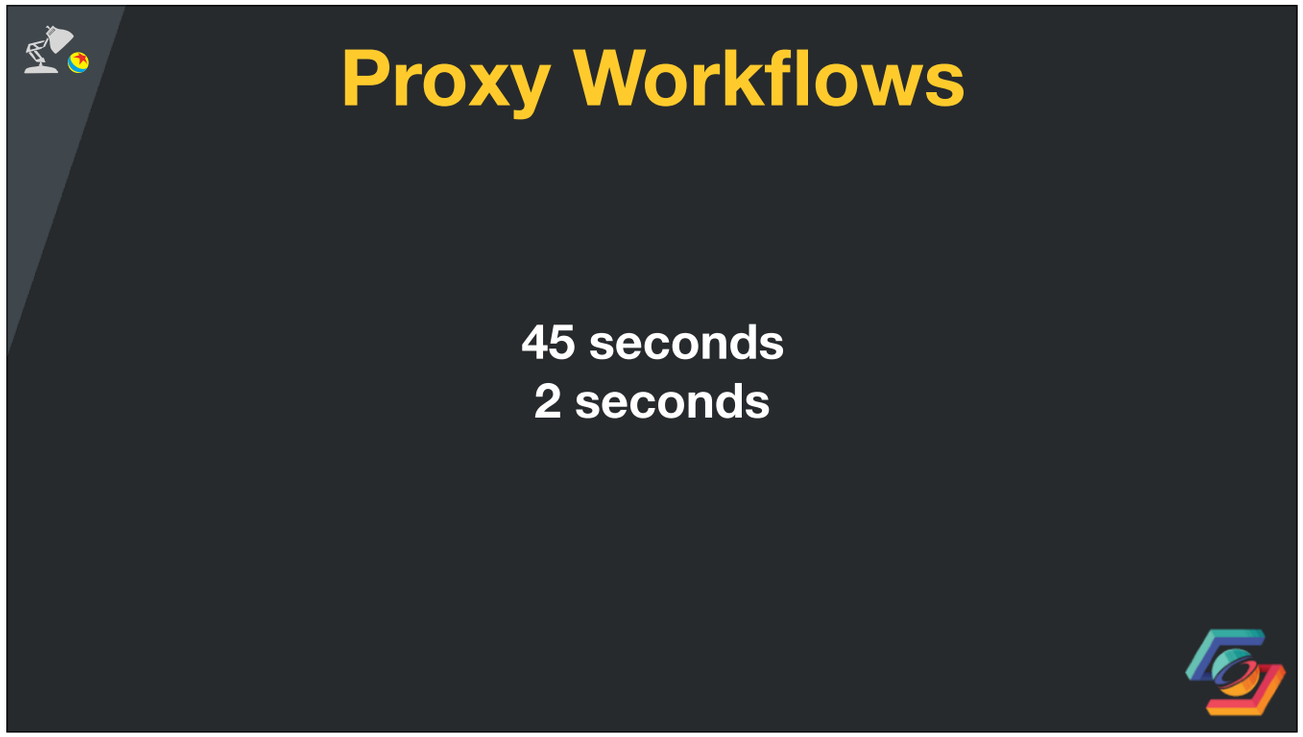
How do you compute the bounds of a model? If you naively merge the extent of all the boundables, you may include the bounds prims that are not directly imaged.

It’s often useful to prune computations like bounding at the root boundable, unless your tool or script has schema specific handling code.



Consider a sample test scene with 400 trees. Time to first image is 45 second. Can we make this better? We can replace the leaves with an interactive proxy representation.


“Full” {
default
default
default
default
}
}



With proxies, we can take our time to first image from 45 seconds down to 2 seconds for 400 trees.

Instancing

4000 Trees

	Compose	First Image	Stage	Imaging
Reference Only	4.973s	16.865s	977MB	1867MB
Instanceable	0.560s	0.801s	25MB	287MB

```
def “TreeWhitePine” (
    references = @TreeWhitePine.usd@){}
```

```
def “TreeWhitePine” (
    instanceable = True
    references = @TreeWhitePine.usd@){}
```

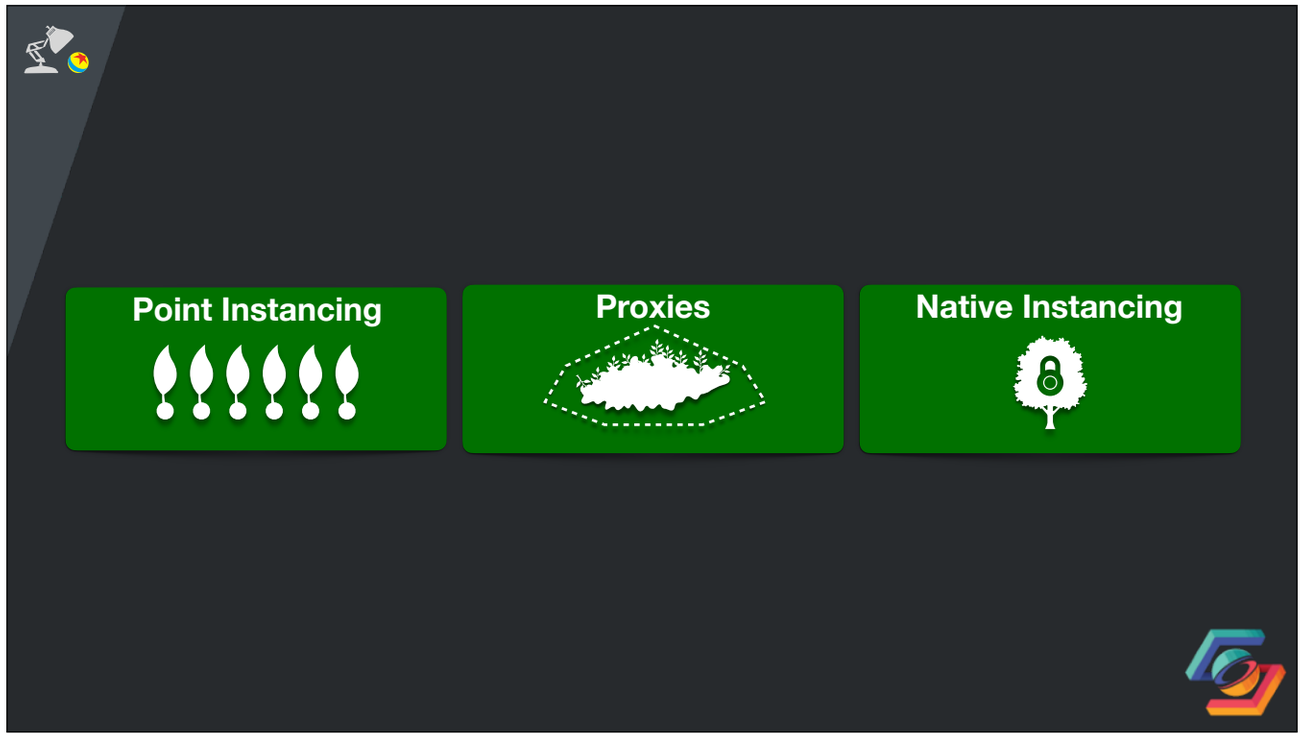
But let’s push the scalability even further. First, lets up our count of trees from 400 to 4000.

Using native instancing, we can get 6.5x memory savings in imaging and a 39x memory savings in stage composition.

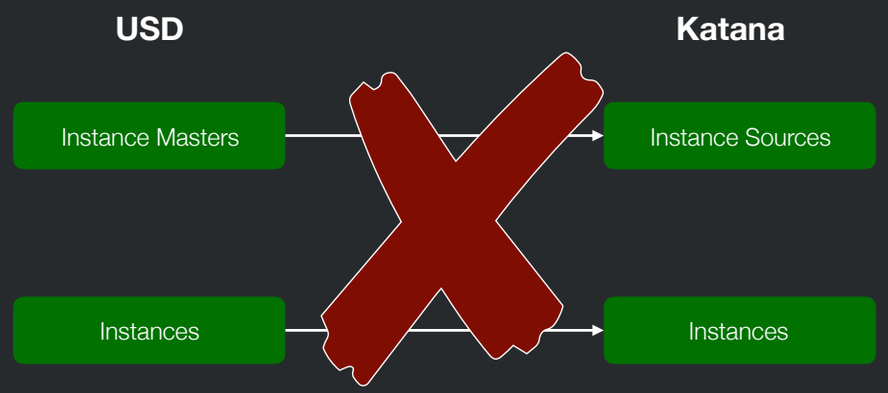
Profiling USD

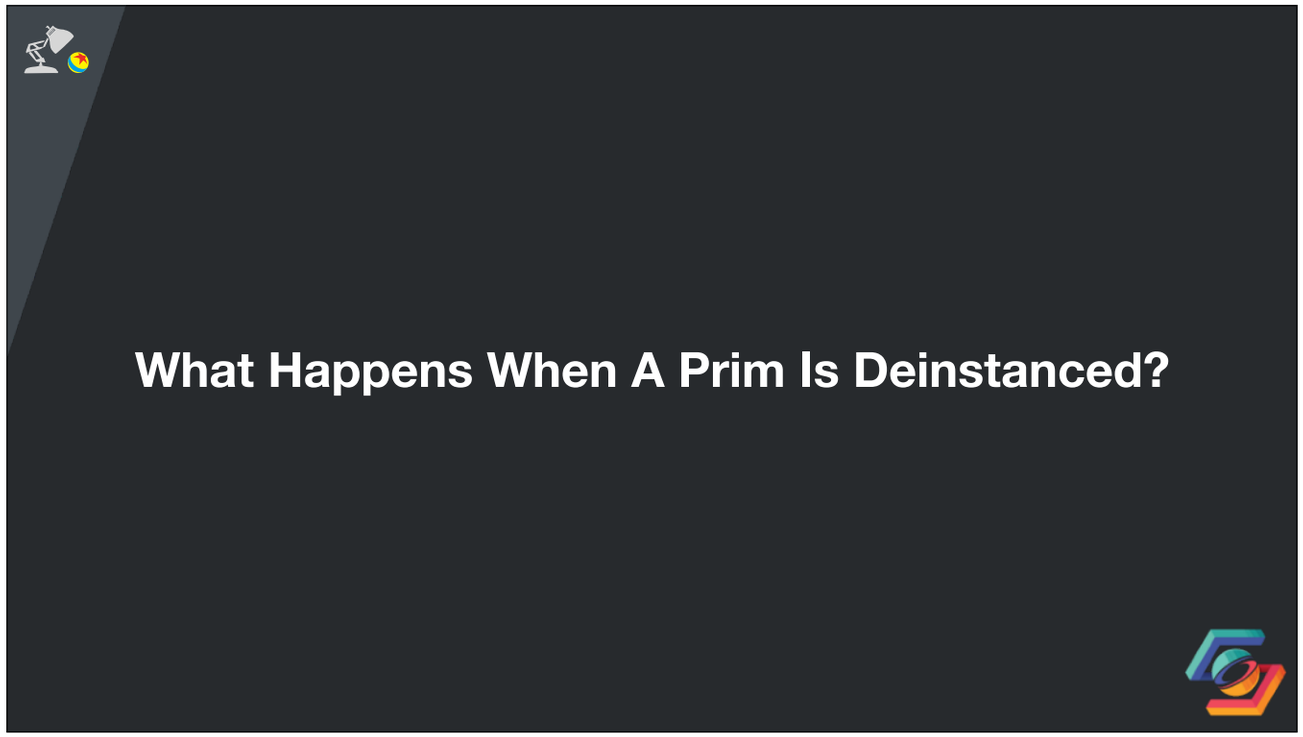
```
usdview --timing MyForrest.usd
usdview --memstats stage MyForrest.usd
usdview --memstats stageAndImaging MyForrest.usd
```

timing and memstats should be run separately


Point Instancing
Proxies
Native Instancing

Interchange



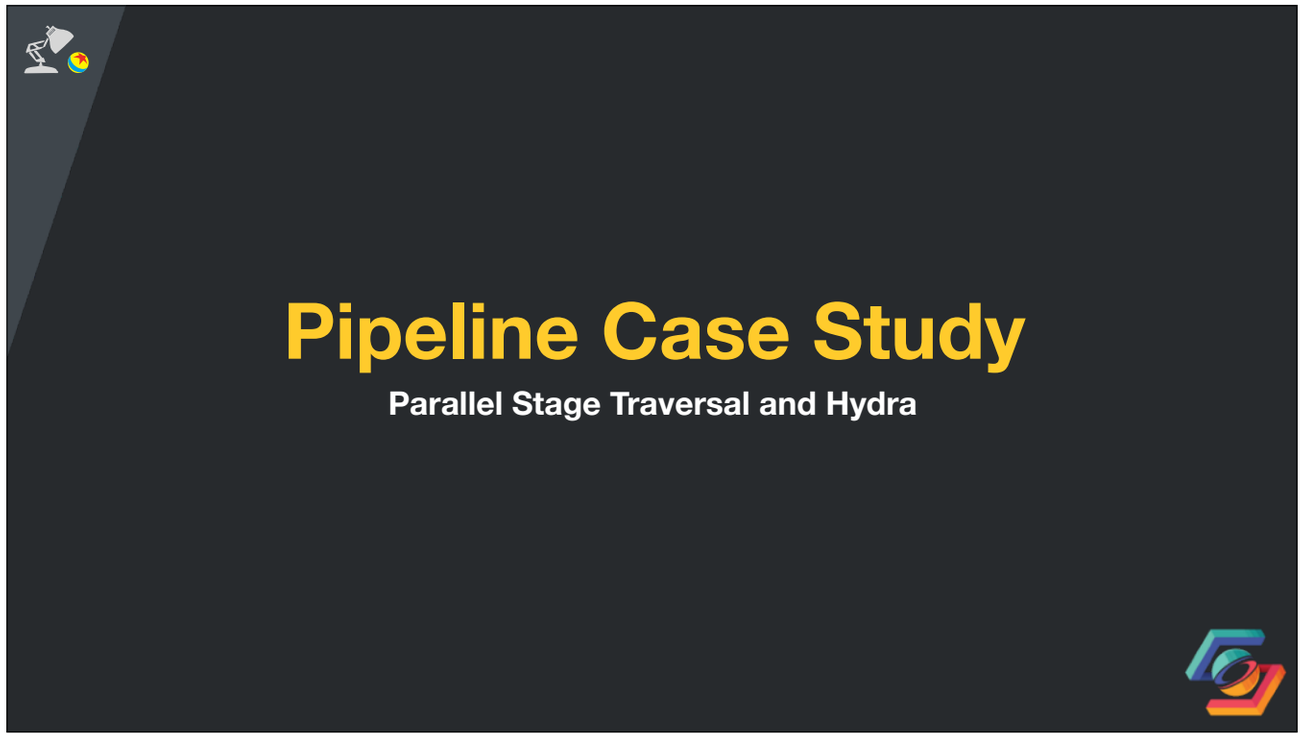

What Happens When A Prim Is Deinstanced?

Interchange

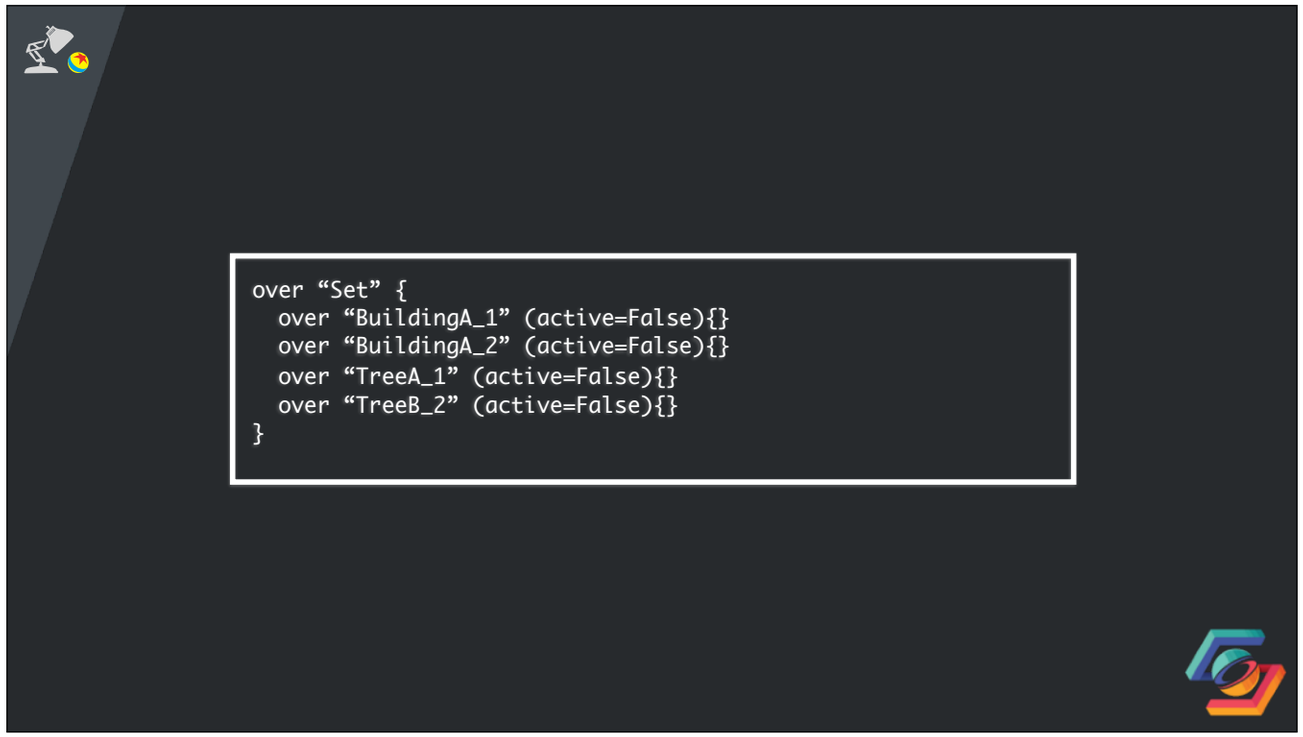
- USD
  - Masters Are Read Only
  - Edits Happen Via Composition
  - Values Are the Same Regardless of Instancing
- Katana
  - Sources Are Locations
  - Edits Are Not Restricted
  - Source Edits Are NOT Applied to Deinstanced

Interchange

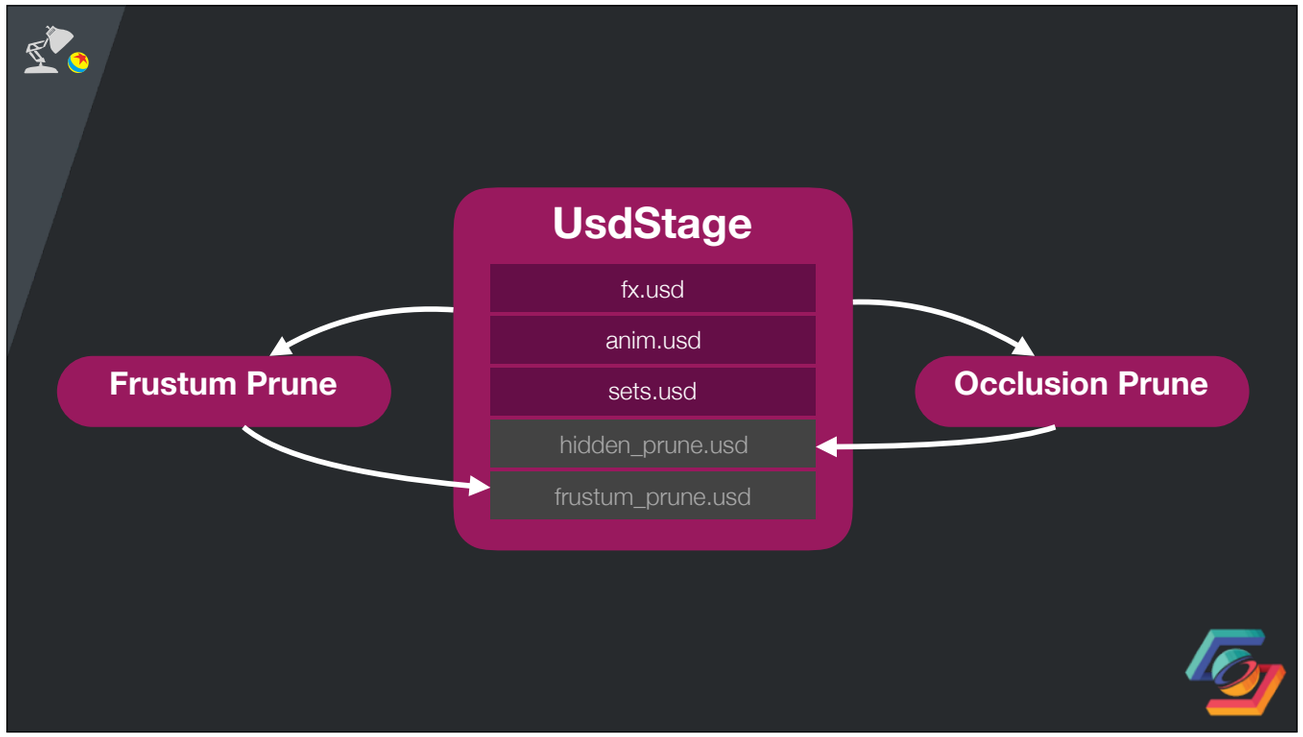
- Tag Katana Locations Read Only
- RenderMan Instance ONLY If Both USD Instanceable and Read Only



For our last case study, we’re going to use our scene metrics pipeline as a way of talking about stage traversal strategies as well as how we leverage hydra’s storm renderer for scene optimization.



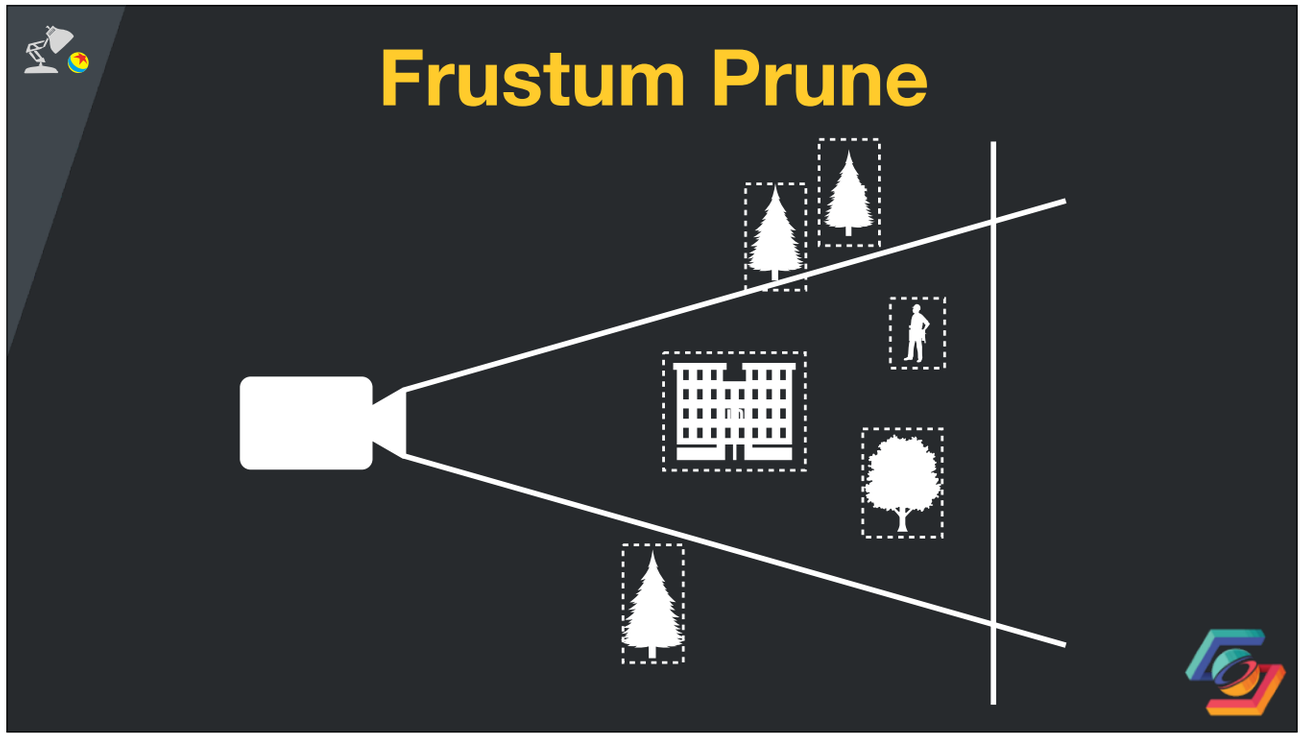
Our goal is author deactivation opinions. Deactivations are a non destructive version of deleting prims.



We have layers in usd which represent deactivation opinions. These are our prune layers. Our scene metrics pipeline is about quickly generating these two layers to respond to artist edits. We run scene metrics nightly on every shot on the show, as well as when artists request.

For our pipeline, it is KEY that the automatically generated layers are weaker than the user defined layers. A user needs the ability to reactivate and undo any mistakes the automated systems make.

The first stage of scene metrics is to mute the existing prune layers. Objects may moved, cameras may have changed, we need to assume the old prune is out of date otherwise, we wouldn’t be generating a new one. We then compute our frustum prune using information that the USD stage provides us. We then run a visibility prune, identifying objects that are hidden behind other objects.

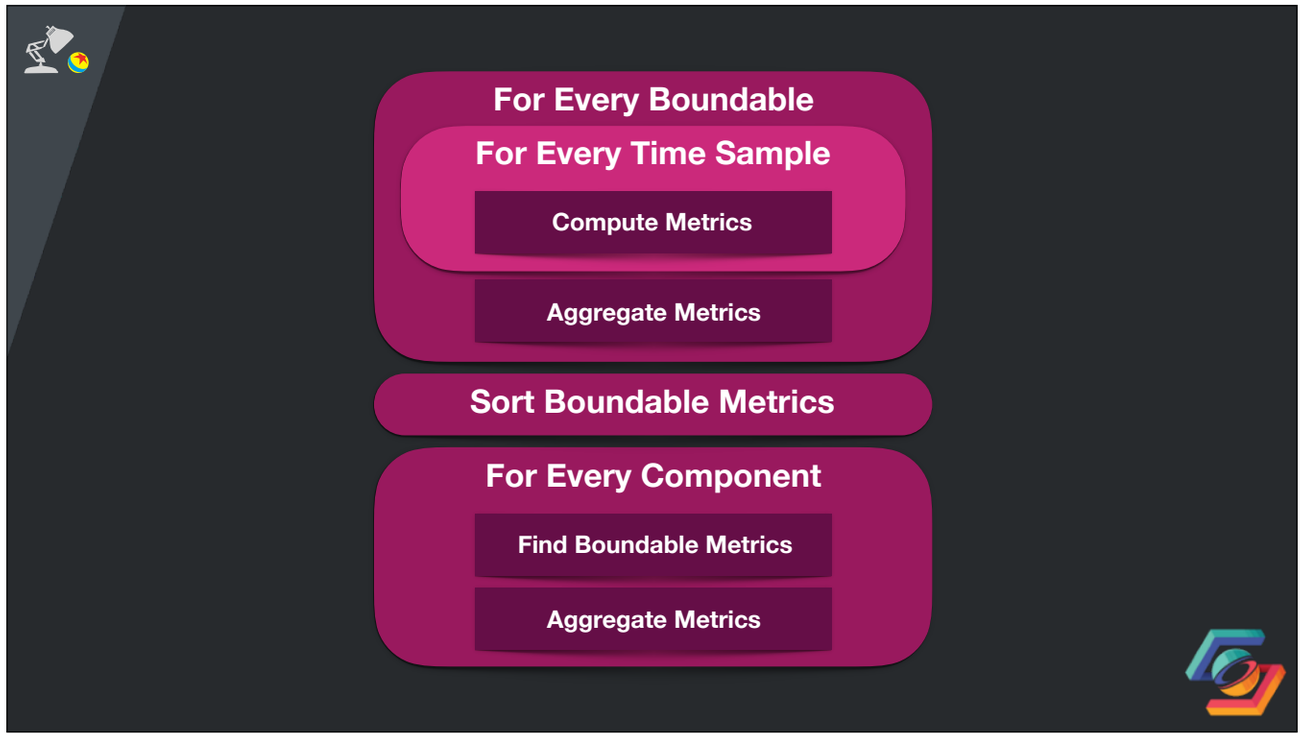


For the frustum prune, we’re going to look at every boundable prim’s bounding box to make a determination as to whether or not that object is in frustum or out of frustum.

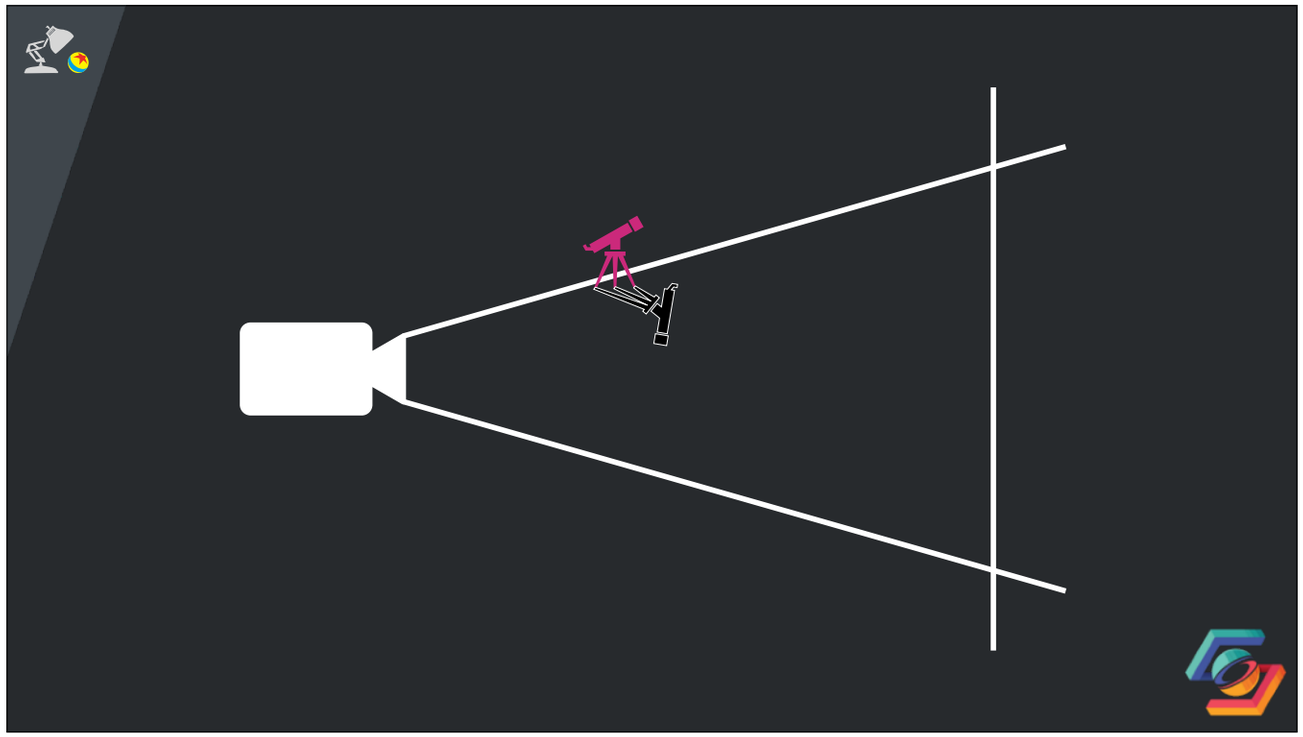
Frustum Prune

Frame	101	102	...	201	Aggregated
	in	in		in	always
	in	in		out	sometimes
	out	out		out	never
	in	in		in	always

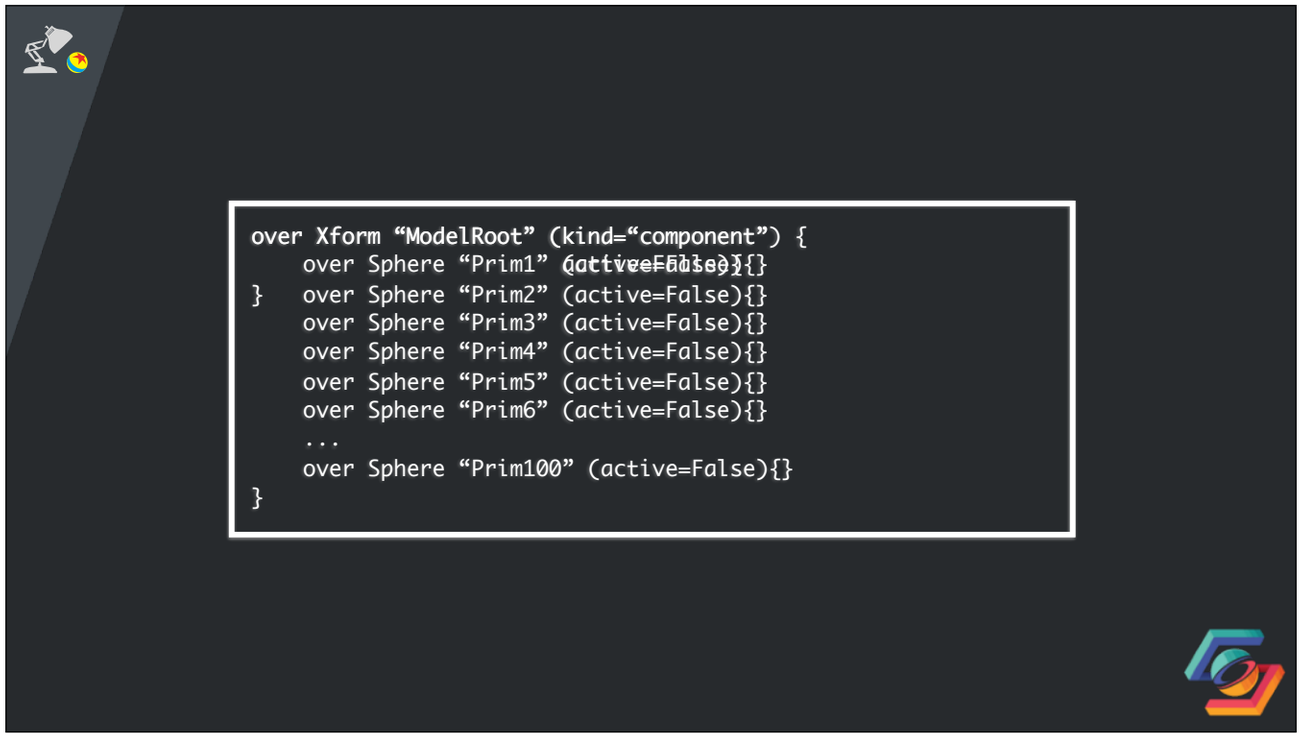
We'll compute this on every frame for every object, ultimately aggregating frustum visibility into one discreet value for the entire frame range. We only want to remove things that are out of frustum for the entire frame range to prevent things like shadowing pops.



We’ll be saying bondable and component a lot this section so I want to make sure the ideas are clear. So when you hear boundable, in most scenes boundables are meshes, but they can be other things like curves and point instancers and hair. Component model definition is certainly pipeline specific, but just think of them as props, architecture, characters, but not the sets which contain component models.

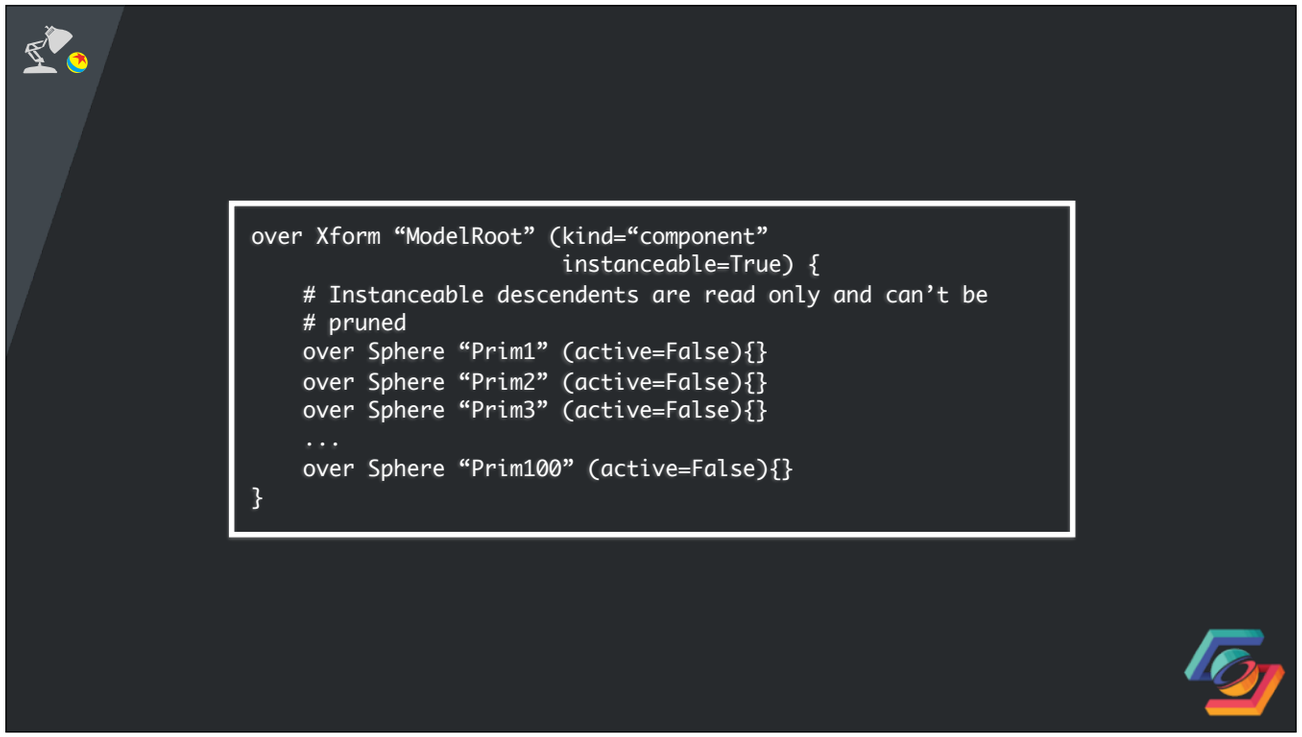


Why do we say we want to aggregate our pruning at the component model level?
Take this telescope component model. Its stand is in frustum, but the scope itself is out.
However, depending on lighting, the entire model’s shadow may be in frustum. If we pruned the boundables, our telescope shadow may look like a stool.

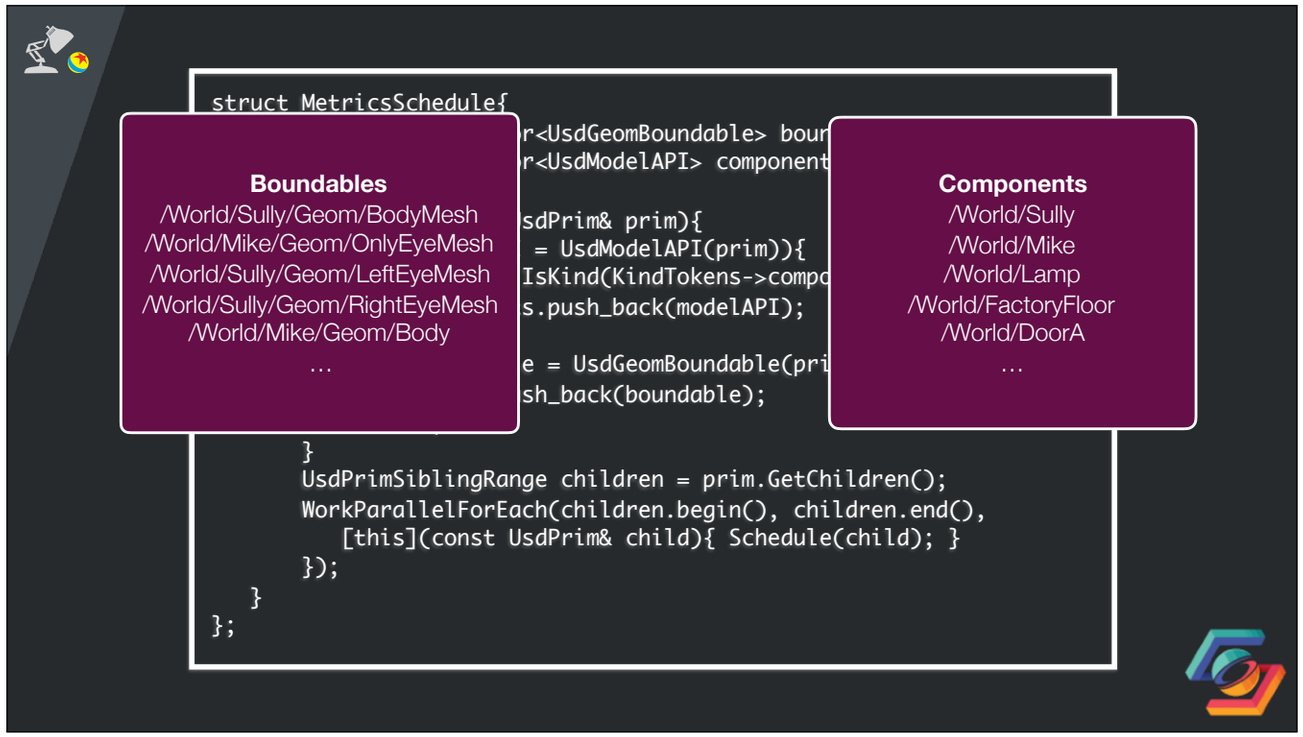


Now, for some models or pipelines, we may say that those shadowing artifacts are okay. It’s still valuable to aggregate your deactivate to the component model level. Otherwise composition has to compose your entire model full of deactivations.

It’s also generally better from a UI perspective to show one deactivated model rather than force users to navigate down to find 100s of deactivated prims.



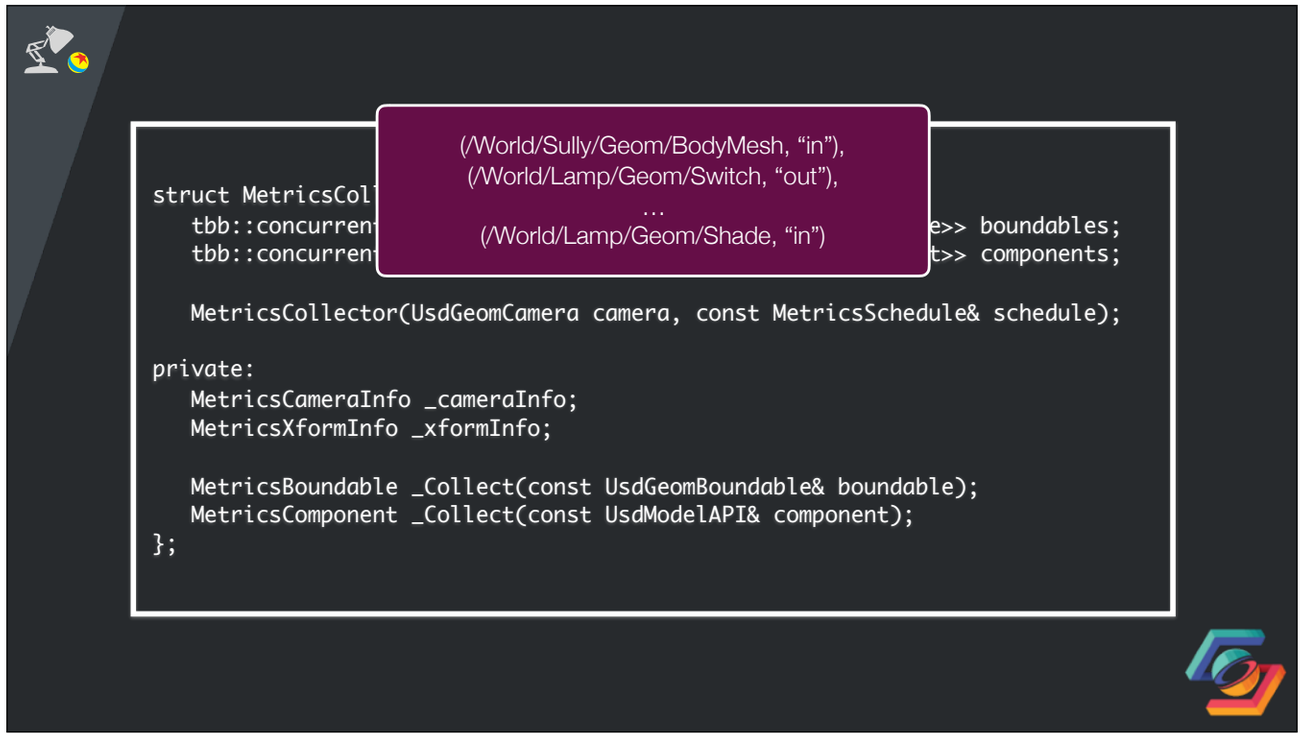
Lastly, we need to think about instancing. If we applied our deactivations below the model root, we couldn’t instance it.



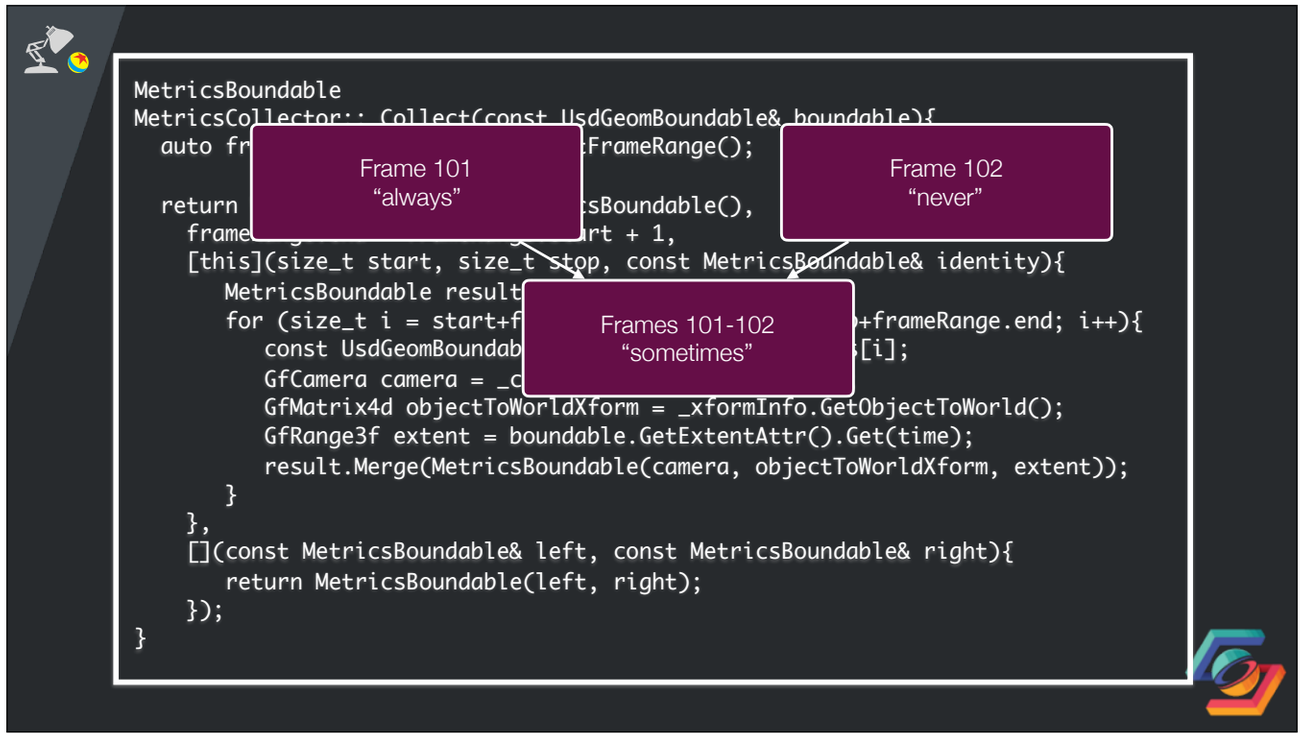
It’s important to use thread safe containers. We’re going to traverse the stage in parallel, accumulating component models and the boundaries they contain.

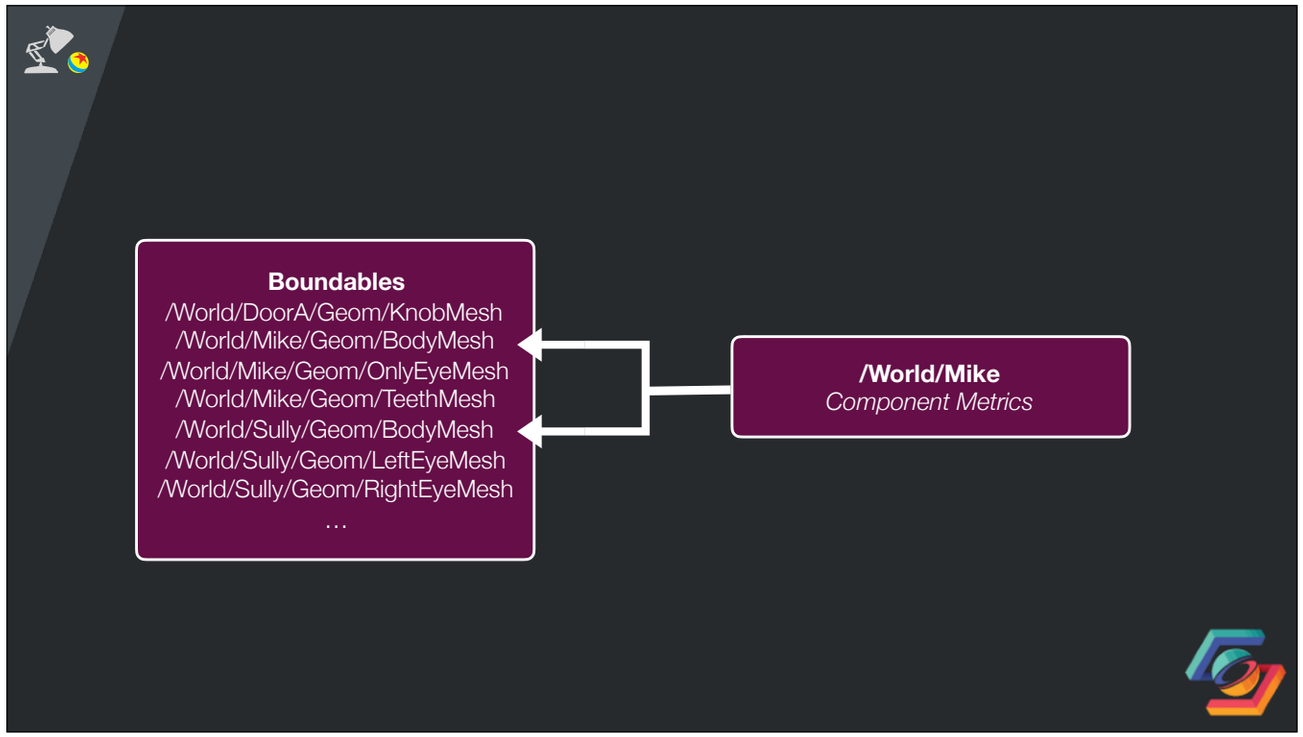
- Find All Component Models
- Find all Boundables
- Then Schedule Children

Work is a wrapper for Intel’s TBB. We prefer using Work whenever possible. It ensures you’re using the same constructs and thread limits that USD is using. It also ensures you’re using the same version of TBB as USD

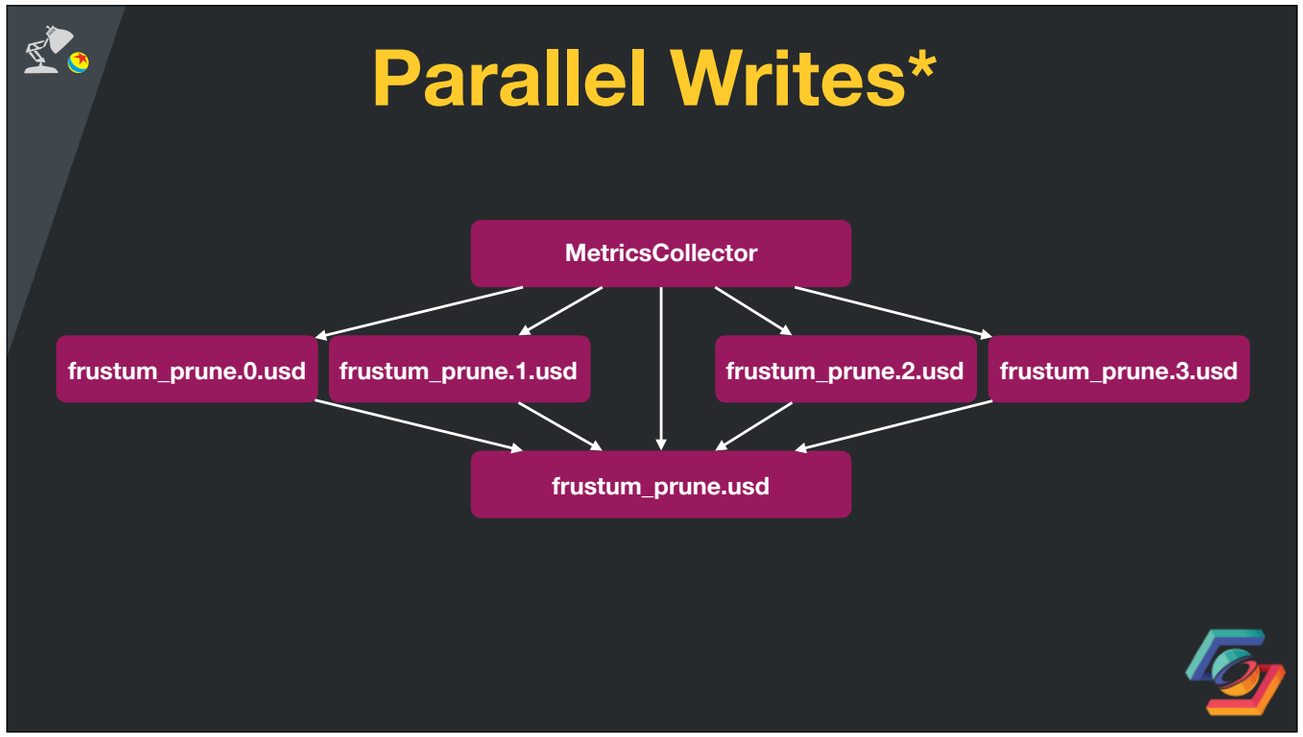

(/World/Sully/Geom/BodyMesh, “in”),
(/World/Lamp/Geom/Switch, “out”),
…
(/World/Lamp/Geom/Shade, “in”)
struct MetricsCol
tbb::concurren
tbb::concurren
e>> boundables;
t>> components;
MetricsCollector(UsdGeomCamera camera, const MetricsSchedule& schedule);
private:
MetricsCameraInfo _cameraInfo;
MetricsXformInfo _xformInfo;
MetricsBoundable _Collect(const UsdGeomBoundable& boundable);
MetricsComponent _Collect(const UsdModelAPI& component);
};

```
MetricsCollector::MetricsCollector(
   UsdGeomCamera camera, const MetricsSchedule& schedule) : _cameraInfo(camera){

   WorkParallelForEach(schedule.boundables.begin(), schedule.boundables.end(),
     [this](const UsdGeomBoundable& boundable){
       boundables.push_back({boundable.GetPrim().GetPath(), _Collect(boundable)});
   });

   WorkParallelSort(&boundables);

   WorkParallelForEach(schedule.components.begin(), schedule.components.end(),
     [this](const UsdGeomModelAPI& component){
       components.push_back({component.GetPrim().GetPath(), _Collect(component)});
   });
}
```


MetricsBoundable
MetricsCollector:: CollectConst UsdGeomBoundable& boundable){
auto fr
FrameRange();
return
sBoundable(),
frame
rt + 1,
[this](size_t start, size_t stop, const MetricsBoundable& identity){
MetricsBoundable result
for (size_t i = start+f
)+frameRange.end; i++){
const UsdGeomBoundab
[i];
GfCamera camera = _c
GfMatrix4d objectToWorldXform = _xformInfo.GetObjectToWorld();
GfRange3f extent = boundable.GetExtentAttr().Get(time);
result.Merge(MetricsBoundable(camera, objectToWorldXform, extent));
}
},
[](const MetricsBoundable& left, const MetricsBoundable& right){
return MetricsBoundable(left, right);
});
}
Frame 101
“always”
Frame 102
“never”
Frames 101-102
“sometimes”



Given a sorted list of boundables, we can use STL’s lower_bound and upper_bound with custom compare functions.



Now we've assembles metrics for every boundable AND every component —
Let's start writing data.

It's not safe to write a USD layer across multiple threads. But let's try and use composition to our advantage. What if we write to multiple layers, and then sublayer them into our frustum prune layer.

This is a great way to break up tasks across multiple threads, processes, or even machines.

Xform Cache Usage

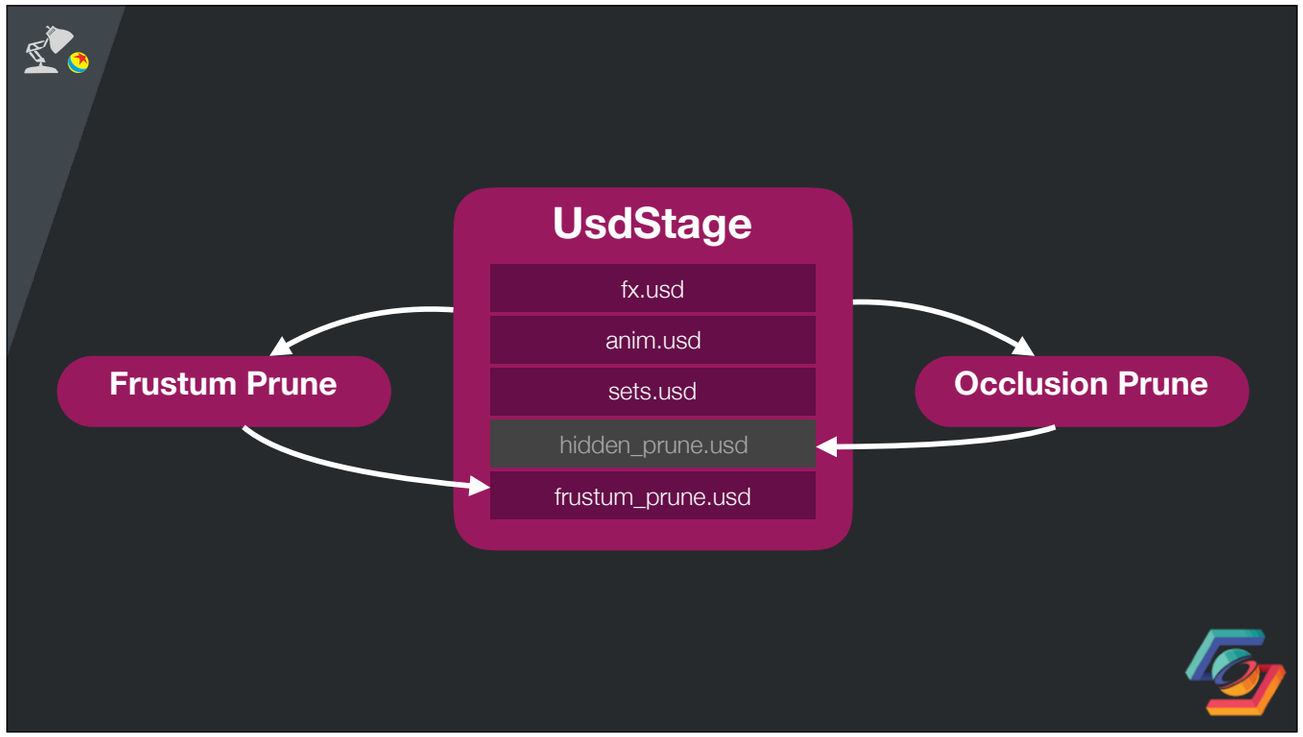
```
class MetricsXformInfo{
private:
    typedef std::map<UsdTimeCode, UsdGeomXformCache> _Cache;
    typedef tbb::enumerable_thread_specific<_Cache> _PerThreadCache;

public:

    …

};
```

Instancing

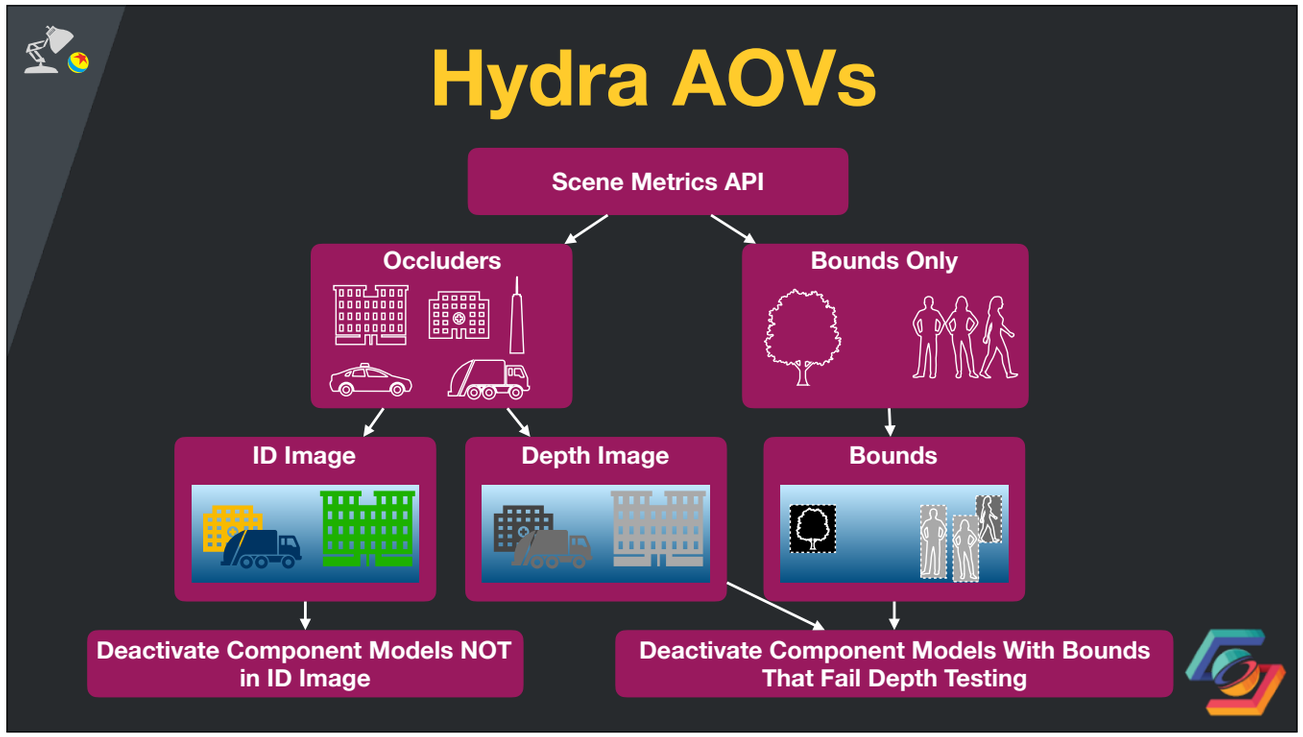
- Native Instancing
  - Instance Aware (Faster Traversal)
  - Instance Proxies (More Accurate Metrics)
- Point Instancers
  - Components of Point Instancers
  - Point Instancers of Components

Parallel Traversal

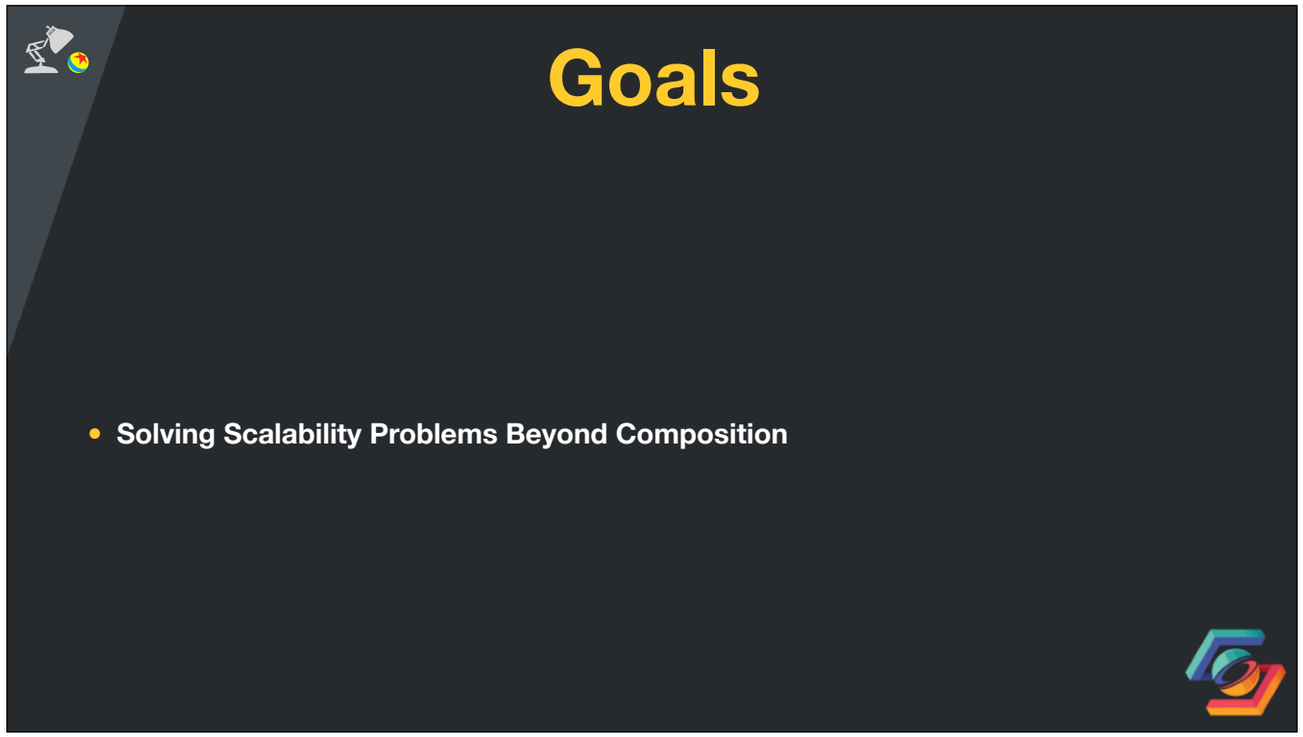
- Use Work for Parallel Scheduling
  - TBB *Should Be* Equivalent
- Thread Safe Container(s) — tbb::concurrent_vector
- Thread Enumerable Caching (UsdGeomXformCache, UsdGeomBBoxCache)



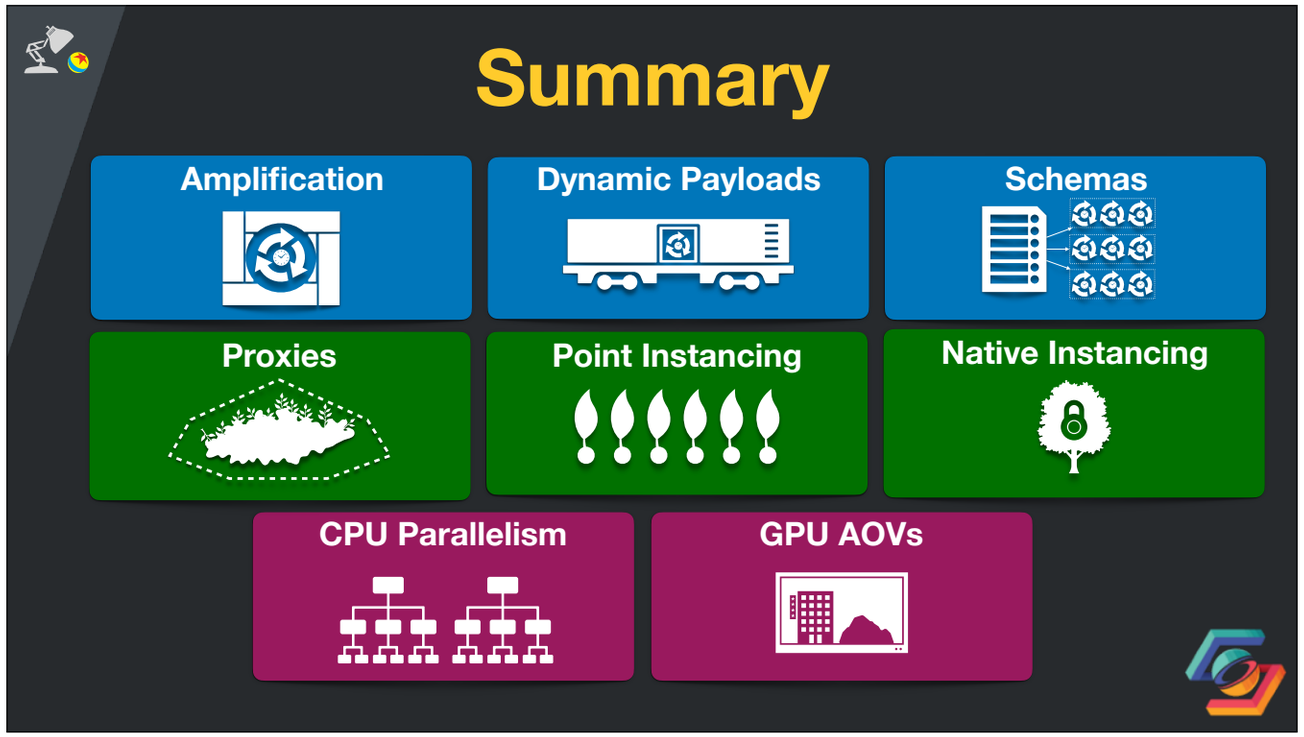
Using libwork and Intel’s TBB are great for CPU parallelism for our frustum prune. But for our occlusion prune, we’d really like to advantage of GPU Parallelism via Hydra’s Storm renderer.



Hopefully, our scene is paired down enough to be rendered by Hydra Storm.



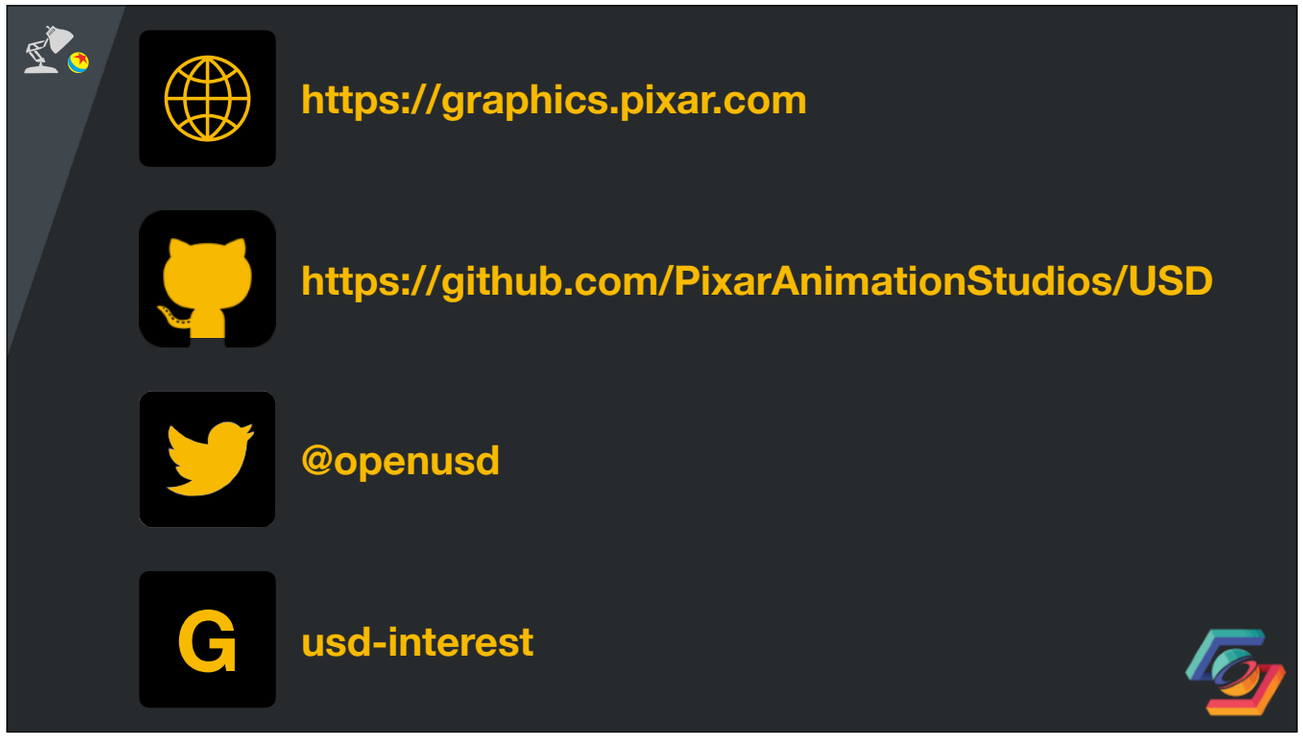
This brings us to the end of this section of the course. We've spent the last hour focusing on solving scalability problems in USD.



— Describing Data Amplification in USD
— Dynamic payloads are one toolset for describing procedural amplification
— How we use custom schemas for describing procedural workflows hair

— Proxies as a way to reduce what we image interactively
— Point instancers as a way of reducing the prim count for granular objects like leaves on a tree
— And the scalability wins in composition and imaging by using native instancing

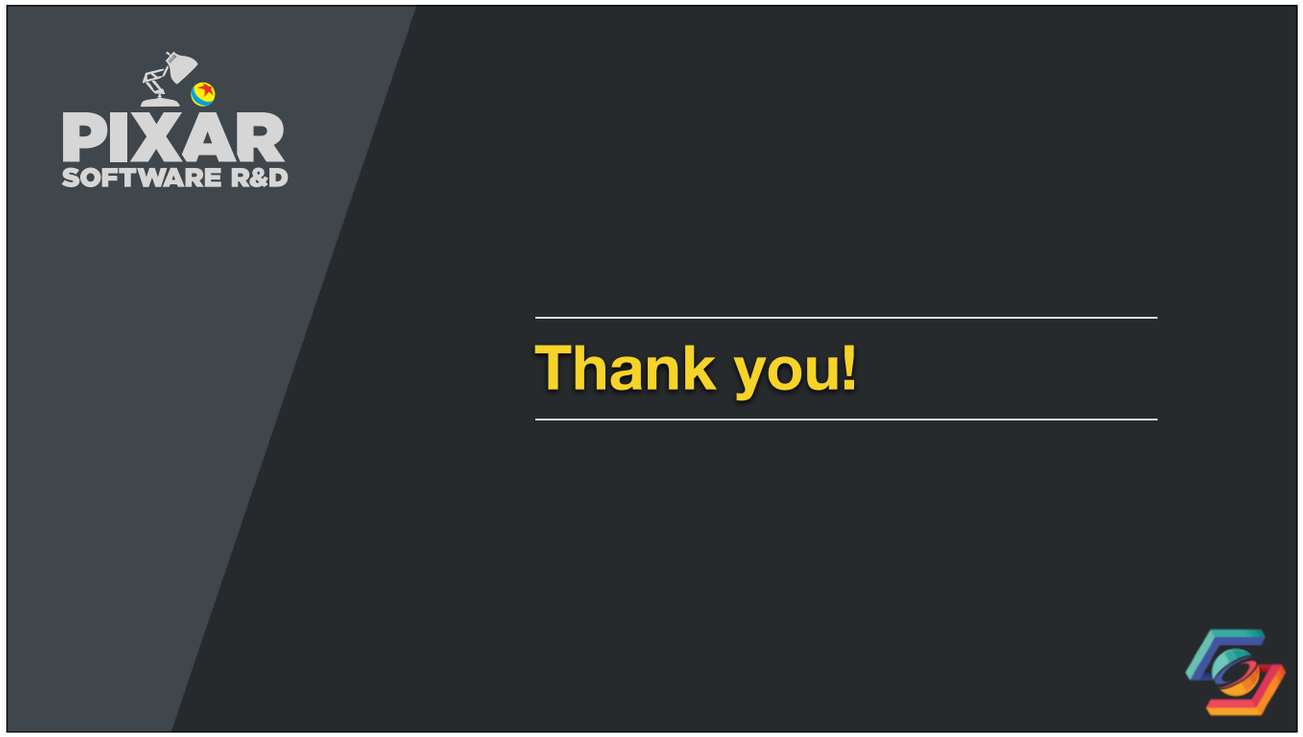
— Lastly, we provided an examples on how to process your stage in parallel
— And how to leverage Hydra as a tool for generating AOV that aid in processes like pruning



To join the discussion, find us on the usd-interest google group

Make sure you to use Pixar in your search terms.
Googling USD Interest by itself will send you to lots of great information but mostly about the US dollar’s LIBOR rates.

Thank you. We’ll be around for questions.


PIXAR
SOFTWARE R&D
Thank you!

https://graphics.pixar.com

https://github.com/PixarAnimationStudios/USD

@openusd

usd-interest