Guidelines for Structuring USD Assets

The Guidelines doc has moved, and will be maintained in the USD-WG Assets repository on Github.

https://github.com/usd-wg/assets/blob/main/docs/asset-structure-guidelines.md

Overview

The goal of this guide is to provide artists and pipelines a good starting point for structuring reasonable and featureful USD assets. It is not necessarily trying to provide a strict specification, but rather define some terms and expectations for building tools and workflows around the “minimum viable” USD assets. This document also hopes to accelerate some intuition around USD assets, by providing a good base to start working in USD, without requiring extensive USD research or knowledge up front. Nothing replaces experience, but we hope to accelerate your experience.

Many studios, artists, and productions will want and need flexibility in structuring their assets. It is important to provide some core concepts that artists, TDs, managers, coordinators, and developers can use when talking about USD assets. The following concepts and descriptions should be common enough for any asset structures in any USD pipeline.

- **Models** are assets in USD. Published models use the *component* or *assembly* model kind, or kinds based on those.
- **Component** models are publishable assets that keep their geometry behind a *payload*; they are generally self-contained (apart from materials referenced from a global library).
- **Assembly** models are publishable assets which reference components or other assembly assets. Assemblies that generate their own Components should payload those components and ensure correct model hierarchy. *Group* kind is used only to organize assemblies and maintain model hierarchy.
- All published component and assembly models should **inherit** from a **class** primitive.

Some of these terms may have existing meaning within pipelines and workflows. This can make discussions and decisions difficult for artists and developers. However, using these terms consistently across tools and studios will accelerate the learning curve for everyone involved.

TODO

Add images and links to example USD assets corresponding to USD snippets.

What is an Asset in USD?

An Asset, as we often use the term in the industry, is a ***model*** in USD. The term asset often means different things in different studios/contexts/tools; for example, a geometry file, zipped bundles of related files, animation caches, texture maps, and shaders are all frequently referred to as assets; even USD uses “asset” as a type for properties that point to external, non-USD files. These non-USD pieces of an asset can be stored and versioned alongside the USD data, and although that may not always be the case, it helps to consider all parts of an asset when defining or referring to USD models.

Model referring to more than just geometry is not only a USD/Pixar convention. Softimage XSI had *models*, with the similar notion that a model encompassed everything that contributed to an asset.

Asset structure in USD generally consists of two considerations: **Primitive Hierarchy** and **Composition/File Organization**.

Primitive Hierarchy

There are some rules around model hierarchy and composition, but otherwise the specifics of asset primitive hierarchy are fairly open.

- Use Scopes for organizing primitives, and where transforms should be avoided. Xforms have a bit more overhead to them, and using Scopes can reduce accidental transforms to child prims.
- Shorter names can help make inspecting prim paths less painful
- Setting Purpose on Scope primitives is cleaner as you don’t have to track individual gprim purposes. This is also easier for Hydra to digest when working live in USD.

Composition & File Organization

The specific implementation details of USD assets on disk is often driven by a studio's existing infrastructure. Whether a studio or individual, consider these points when structuring assets:

- Layers are usually divided by department/work/contribution/content
    - A payload requires an external file, but otherwise layer separation is mostly about organizing work as needed
- Storing layers as child directories/layers of the <asset>.usd, referred to with relative paths, helps with portability
- Individual element layers of a component are generally either referenced or sublayered together.
    - Referencing helps keep opinion strength a bit more predictable, as each contribution is the "R" in LIVRPS
- The bare minimum for a reasonable USD asset is the asset.usd and payload.usd
- Using the ".usd" extension is highly recommended for the "asset.usd", to avoid having to re-author reference paths when the asset layer is switched between ascii and binary. It may also be useful to carry that to individual layers as well.
- Organize each asset with subfolders, especially when there are additional files that individual layers reference or sublayer together.
    - i.e. "model", "lgt", etc...
- Within the layer folders, it can be helpful to organize the non-USD data referenced by the USD files using folders
    - i.e. "maps", "vol", etc…

Versioning the Structure

Structuring USD assets should be flexible, to adapt to different needs or priorities. With metadata on the primitives or in the files, systems can be written to understand how to flexibly edit or contribute to an asset. Referencing a model probably doesn't need to know about this as much, this is more to help with artists or tools editing assets.

The only part of USD specifically designed to help with version control, is Asset Resolver plugins. Other resources for writing AR plugins can be consulted for more details.

Component

A Component is the most basic asset published in a USD pipeline. A reasonable, basic component has the following features:

- Component kind on the model's root prim
- Payload makes geometry deferrable
- Self-contained/Portable, no references to external geometry or models
    - Materials referenced from a global library would be a common exception.
- Inherits from at least one Class prim, allowing context specific overrides to be broadcast to every asset inheriting (or "subscribing to") the same class.
- A transform ("Xformable") prim at the root, which is set as the defaultPrim of the asset's root file.

Component models may or may not have DCC-specific rigs, geometry variants, etc… The non-USD data (i.e. rigs) are generally published alongside the core USD layers of an asset, and often have metadata pointing at these sidecar files.

The primitives within a component are organized with Scopes. This helps avoid accidental transforms of primitives which aren't the component root, or subcomponents. Shorter names such as 'geo' and 'mtl' help make investigating primitive paths a bit easier. However, the general organization is more important than specific names.

Simple Component

```
/apple          (xform)  Component Kind
    /geo        (scope)
    /mtl        (scope)
/__class__      (class)
    /apple      (class)  Inherited by /<component>
```

Scopes

A component should have at least two scopes at a minimum: for geometry and materials. Additional scopes may be added for specific purposes, but it's reasonable to assume all will start with these.

Geometry

The ***geo*** scope and layer represents the visible geometry in the scene. Any downstream contributors will overlay their work on the existing geomtry of the asset.

Materials

The ***mtl*** scope contains the materials and shaders used by the component. These can be defined in the asset, referenced from a global material library, or some combination of both. Keep in mind the material layer almost always has opinions on the geometry of a model; at the very least material bindings, but opinions like new UV sets or primvars for shader signals are also likely.

Purpose

Display purpose can be used to curate a light-weight representation for GL viewers or other uses. When present, it is recommended to name scopes underneath ‘geo’ when using display purposes. There is no need to mark every primitive with its purpose; marking the parent scope is sufficient. Not only is per-prim management of purpose difficult, it seems to cause instabilities for Hydra.

Simple Component - With Purpose Scopes

```
/apple          (xform)  Component Kind
    /geo        (scope)
        /proxy  (scope)  Purpose: proxy
        /render (scope)  Purpose: render
    /mtl        (scope)
/__class__      (class)
    /apple      (class)  Inherited by /<component>
```

A few other notes about scene graph structure for components:

- Organizing class primitives underneath a class prim isn’t necessary, but it can make working in primitive hierarchies a bit easier
- Materials may be referenced from an external material library, but any texture maps for materials defined with the component should live with the component.
- Materials can get heavy in their own way (prim count), so it helps to keep them behind the payload. If tweaking materials with payloads unloaded is desirable, assets could move materials outside of the payload in these cases.
- Variants can be used for LOD or geometry variations of the same asset. While variant sets are super flexible, it helps to define at least one or two common/standard variant sets (i.e. geometry and materials).
- Asset Info metadata on the root prim can store more useful information about an asset

Hopefully the basic component structure will help provide a good foundation for specialized component structures in the future.

Subcomponents

Sub components are Xformable prims that live within a component model, and which are intentionally made available in USD for transformation. Rather than rely on pivots, it is recommended that subcomponents be positioned where the pivot would be. This can avoid requiring a single type of xformOpOrder convention.

Subcomponents can be nested, and where applicable, each subcomponent can mirror the overall structure of a simple component. For some improved efficiencies, a geometry and material scope within each subcomponent could be referenced into place, and made instanceable. This is likely most useful for environment props which only use rigid transforms, and rely heavily on scene graph instancing.

One consideration around nested subcomponent: when scaling up for larger scenes of many instances, if your subcomponents are shallow and not nested, it may be easier to represent their motion using Point Instancers, as each subcomponent can be a prototype.

GeomSubsets

GeomSubsets can be extremely useful, where mesh prims can be combined and instead defined as GeomSubsets. It’s recommended they be used, however GeomSubsets cannot be activated/invis’d like individual mesh prims can, nor can they be transformed. If you have variants which need to adjust visibility/activation or tweak transforms, you need to use mesh prims.

Assembly

An assembly is a published asset which aggregates other published component or assembly models together. Intermediate xforms used to group and organize models use the *group* kind. Assemblies either directly instance the references to other models, or provide those references as Prototypes to PointInstancers. Payload arcs don’t provide any benefits when they already exist within the component models; and in fact they can complicate loading /unloading stages, and add additional overhead while working in USD.

Simple Assembly

```
/appleBowl      (xform)  Assembly Kind
    /apples     (xform)  Group Kind
        /apple1 (reference to apple component)
        /apple2 (reference to apple component)
        /apple3 (reference to apple component)
        /apple4 (reference to apple component)
        /apple5 (reference to apple component)
    /bowl       (reference to bowl component)
/__class__      (class)
    /appleBowl  (class)  Inherited by /<assembly>
```

Assemblies may override materials on individual components, to better integrate the various assets together. These cases can store these assembly-level materials in a ‘mtl’ scope like this:

Simple Assembly - With Materials

```
/appleBowl       (xform)  Assembly Kind
    /apples      (xform)  Group Kind
        /apple1  (reference to apple component)
        /apple2  (reference to apple component)
        /apple3  (reference to apple component)
        /apple4  (reference to apple component)
        /apple5  (reference to apple component)
    /bowl        (reference to bowl component)
    /mtl         (scope)
/__class__       (class)
    /appleBowl   (class)  Inherited by /<assembly>
```

While an assembly generally contains only references to other models, production often presents unique requirements. For cases where an assembly needs to generate new geometry, publishing that to its own component model and referencing back into the assembly is awkward or cumbersome. In these cases, the new geometry should be stored behind a payload and have the component kind.

Simple Assembly - With Materials and New Geometry

```
/appleBowl       (xform)  Assembly Kind
    /apples      (xform)  Group Kind
        /apple1  (reference to apple component)
        /apple2  (reference to apple component)
        /apple3  (reference to apple component)
        /apple4  (reference to apple component)
        /apple5  (reference to apple component)
    /bowl        (reference to bowl component)
    /geo         (scope)  Component Kind, Payload
    /mtl         (scope)
/__class__       (class)
    /appleBowl   (class)  Inherited by /<assembly>
```

Group

It is recommended that the kind group not be used directly on the root of published assets. Groups are used simply to help assembly models obey model hierarchy rules.

Details

Instancing

The easiest way to instance an asset is to make it instanceable when referenced into a scene, or to use a PointInstancer to scatter the model many times. In both cases, the root primitive of the model can be used to transform or edit the asset, but none of the child prims are directly editable while the model is still instanceabe.Referenced components are often instanceable in assemblies. PointInstancers are useful in both components and not just large-scale assemblies. Here is a component example of what a tree could look like:

Assemblies often mark their referenced components as instanceable, though this isn’t always required.This is a modeling comment, but component model makers should see both point instancers and instanceable primitives as tools in the modeler’s bag of tricks. It may end up being easier/cheaper in some cases to simply combine into one large mesh, but instancing shouldn’t be regarded only for scene layouts. You may find it useful to have bolts, leaves, or small pieces of food use instances.

Variants

USD variant sets provide a lot of flexibility, which can make it overwhelming at first, to settle on what to do. It is recommended that USD assets have two variants to start: one for geometric variation, and another for materials. Components may leverage variant sets to describe different geometry and /or materials combinations. Assemblies too can use variants to store different layouts, material overrides, etc… But in the name of simplicity, a lot of mileage can be had from just geometry and/or material variant sets.

It is important that a model's variant sets are accessible on the root prim of the asset. This allows users to make variant selections even if the component is marked as instanceable, or if the payload is unloaded. Variant sets might live in a lower primitive, but generally they should be accessible on the model's root primitive for users to set.

Separate Files per Variant

Variants can be stored in separate files, or they can all live within a single layer. Keeping variants in one file simplifies file management and reduces the number of layers USD has to keep track of. On the other hand, layers for each variant may be easier for some tools or pipelines to manage. The ability to manage variants as layers hinges on well-defined variant set(s), as it would be challenging to enforce a pipeline where variant sets can change files on disk so arbitrarily.

Unless there is a reason to enforce one or the other, having the flexibility to pack and unpack variants would be one option that an asset structure could adopt.

Fallback Variant Selections

A pipeline feature, which allows for automatic variant selection(s) in the absence of an authored variant selection. This allows for artists to automatically work in a variant tailored to their needs.

See Colin Kennedy's USD Cookbook for more info and examples.

Variation-Only Models

Variants can be an unusual concept for some pipelines or productions. If geometry/model variants are difficult to bid/manage/etc… it might be useful to publish an "uber" component, whose variants are only references to separately published component models.

Payloads

It is important that heavy geometry and large prim counts are kept behind payloads. Components should have their own payloads, and assemblies would not need payloads, as the referenced components would already have their own payloads. Some shot-centric contributions like FX may introduce a lot of new geometry. These artists and departments need to ensure these new contributions are unloadable, and that they preserve model hierarchy, even if they are authored in a shot.

Nested payloads can add performance issues, as well as add tooling complexity. Defining components as publishable assets with a payload helps avoid nested payloads, and provides some expectations around what will be possible when opening an unloaded scene.

Alembic

Pipelines adopting USD tend to have a lot of existing assets and tooling around Alembic. During this early-adoption period, you can payload or reference the Alembic files directly, without converting the data into .usdc right away. There are advantages to using USDC over Alembic, but it should be possible to build a functional asset structure even with some data stored as Alembic.

See Alembic USD Plugin for more details.

Inherits and Classes

Inheriting from class prims is a way to set up USD assets for flexibility down the road. There are a lot of ways that inherits can be used to build powerful assets in USD, but it is recommended that models inherit from just one to start. Examples of edits made via inherit arcs could be variant selections, material overrides, or even new primitives. Broadcasting overrides via inherits is one way to make changes to scene graph instances.

Inheriting from one class is useful, but it may also be valuable for models to inherit from other class prims. In particular, the groups or subgroups used to organize assets might be good candidates for additional class prims to inherit from.

Specializes can behave similarly to inherits, but when defining an asset the opinion strength of inherits is generally better for broadcasting overriding opinions.

Draw Mode

Models in USD can improve viewport performance using Draw Mode; you can even use draw mode in conjunction with unloaded payloads.

There are 5 possible options:

- **origin** - Draw the model-space basis vectors of the replaced prim.
- **bounds** - Draw the model-space bounding box of the replaced prim.
- **cards** - Draw textured quads as a placeholder for the replaced prim.
- **default** - An explicit opinion to draw the USD subtree as normal.
- **inherited** - Defer to the parent opinion.

See UsdGeomModelAPI - Draw Modes for more details.

Additional Layers/Contributors

Models which incorporate work from other departments or disciplines, can follow the materials pattern and reference their layer into the asset's root file layer. It may also be helpful to define a scope for any unique/new contributions to the asset.For example, in the case of a campfire, which has asset-based lighting and volumetric effects:

Files and Composition

Directory named by the asset, and is stored within a Root asset layer is also named following the asset.Some studios find they like to use the extension based on whether the layer is ascii or binary, as it removes ambiguity and provides users/artists expectations of "heaviness" of a file. The advantage of using .usd, especially for the root file, is you can change between ascii and binary without breaking references.

All other layer contributions to an asset should be crate files (binary .usd or .usdc) and named after their role/contribution, which get referenced into "payload". The ordering of these extra layers depends on their function, and their place in the workflow-timeline. Generally, the layers build on one another, so that later disciplines/contributors can use and/or override existing work. This is why geometry is at the bottom, then materials. And is why "lighting" is usually one of the last layers to contribute to USD assets/shots.

> ⚠ **TODO**
>
> When should individual layers be sublayered vs referenced?

It may be helpful to place related, non-USD files in subdirectories of the asset; each directory would describe the contents. For example, texture maps or rigs:

Any related files that don't go into a directory for that "type" of file, could instead be organized by folders according to the layer that uses these files.

Additional, helpful bits like thumbnails could be added to an asset like this:

Materials

USD assets must *encapsulate* all of the geometry and materials, in order to be efficiently referenced in a pipeline or workflow. The materials themselves may be referenced from some global library, but their reference point must be underneath the asset's root primitive.

The look development may happen at either the component or in an assembly. To materialize instances, particularly in larger assembly, the inherited class prims can broadcast material bindings to the relevant models.

> ⚠ **TODO**
>
> - What are the considerations around shading networks?
> - How are folks targeting multiple render delegates? MaterialX-only?

Crowds and UsdSkel

> ⚠ **TODO**
>
> - What are the considerations around crowds and USD asset structures?
> - How does UsdSkel impact structuring assets? Are BG characters structured different from "hero" models?
> - Are there considerations regarding Value Clips?

Physics or Other Schemas/Purposes

For some use cases, describing physics in models may be necessary. Where rigid body simulations are the only requirement, it may be possible to provide physics-driven interactions to instances.

If the physics needs don't call for being interspersed with the model's gprims, it may be useful to have a dedicated scope for the physics primitives.

Games

Much of this document is directed to structuring USD assets for animation and VFX. This section will outline where real-time needs require different approaches.

⚠ **TODO**

- Bidirectionality
- Simultaneous editing
- LOD
  - Game LOD often has a camera-based component to it; this could be challenging to implement satisfactorily without some sort of prim adapter, as variants cannot be animated.
- Do any game engines consume and interact with USD as-is, or do they all convert to their internal formats?

Related Workflows and Considerations

Backlots

- Two cases: archival and kitbashing
- USD and Digital Backlots - Eliot Smyrl
  - ASWF - WGUSD Presentation
  - Siggraph 2021 Talk
- Digital Backlots and Libraries are Hard - Davide Pesare

Reviews and Variants

- Does one create review-specific variants?

Custom Schemas

- When does one need them? Concrete vs ApiSchemas?

Naming

- Discussions/recommendations around different conventions (by studios and artists)

Blind Data

- Dealing with unknown schemas or undefined schemas
  - Stash to not lose them during interchange between DCCs which are converting between native and USD formats
  - Maybe something for DCC-specific schemas to consider
    - As an example, Houdini has some ApiSchemas which have no functionality outside of Solaris. So unless they are actually removed from a USD file, the data should persist but remain dormant until brought back into Houdini.
  - Schemas should persist in USD, even if the plugins aren't detected, so this concern is largely related to conversion between USD and DCC-native formats
  - Fallback Prim Types can also be defined, if there is alternative behavior that a pipeline may wish to have: https://graphics.pixar.com/usd/release/api/_usd__page__object_model.html#Usd_OM_FallbackPrimTypes

Example Structures

Representations of assets, possibly used in real productions. May or may not necessarily follow every suggestion of this document, but generally implement the most important ones, and are just useful and interesting to study.

ALab by AnimalLogic

- https://animallogic.com/usd-alab/
- https://usd-alab.s3.amazonaws.com/documentation.html

Blue Sky Studios

- Apples (usd-interest)

OpenUSD

- https://graphics.pixar.com/usd/release/dl_downloads.html
  - Kitchen Set

- City Set
- UsdSkel Examples