Data Adapter Merge Static Meshes

Background and Motivation

The performance of a USD Stage within the Omniverse ecosystem can often be improved by reducing the number of Prims. A common source of large Prim counts is geometry that is described using individual mesh Prims for each chunk of connected topology. In cases where individual addressability is not required, these chunks can be combined into a single mesh Prim that produces the same visual result with a smaller overall Prim count.

This can be a useful optimization if your goal is to increase playback FPS or reduce load time. However it can come with the trade-off of potentially using more GPU RAM, since the renderer has less ability to deduplicate identical geometry.

Reducing the prim count can help scene load time drastically. Composition in USD (taking the various layers and determining the current renderable state of the stage) is expensive, and the fewer prims there are, the less time that needs to be spent on this process. It also results in less data that needs to be passed to the renderer - fewer materials to process, fewer meshes to consider, fewer changes to track, etc.

The **Merge Static Meshes** process (referred to throughout as Merge) provides a function in the Create Data Adapter extension to perform this merge operation.

Assumptions

For this document it is assumed the reader is familiar with Pixar's Universal Scene Description (USD). If not, the Glossary is a good starting place to get a feel for some of the terms that are used throughout.

Caveat

In some cases Merge can improve the performance of a scene drastically. However scenes can be authored in very different ways and in some cases the merge operation may not offer much benefit, or may even make performance worse. Users may need to try different argument values to find a worthwhile result.

Strategy

The array based nature of most UsdGeomMesh schema attributes makes it possible to concatenate the values from multiple input meshes and produce a single output mesh that has the same worldspace geometric state.

The values of some attributes need to be modified during concatenation. The *points* or *normals* of an input mesh may end up in a different transform hierarchy on their output mesh and may need to be adjusted to compensate. Another example is attributes that hold indices of other attributes and need to be offset. This means that the Merge process needs to be aware of the *meaning* of attributes in order to merge them.

Attributes that are not understood by Merge are copied onto the output mesh unmodified. If the values of these attributes differ between input meshes then they cannot be merged and multiple output meshes will be created.

The overall approach of Merge is to:

- identify any meshes that can potentially be merged somewhere in the stage
- bucket them into groups of meshes that can safely be merged
- concatenate the data from all meshes in a bucket into buffers
- write that data back to the stage as the final merged output meshes
- remove the input meshes from the renderable state

These steps will be described in detail below.

The overall architecture has been designed in order to allow executing the various phases more efficiently - via both multithreading and mechanisms within the USD APIs. Generally, you can read from a USD stage from multiple threads, but can't write to it in parallel. Merge is designed around this so that it can cache information up front, process it in parallel, and then author it back at the end.

A priority for Merge was to ensure that it executes as quickly as possible. To satisfy this it is liberal with RAM usage in order to optimize access to data within the USD stage, and also to ensure output data can be generated in parallel, before being written in serial. This may potentially cause issues on scenes that are pushing RAM limits when they are opened.

Care is taken to read data from a USD stage in an efficient manner (for example, const-only access where possible to *VtArrays*, use of USD caching mechanisms, etc.). Likewise, *SdfChangeBlocks* are used where possible to reduce the number of notifications that are sent when changes are authored on a stage.

Challenges

The idea of merging meshes is fairly straightforward, however with USD comes a lot of complexity. There are countless ways to author scenes using a variety of mechanisms.

Some meshes have Attributes that are not compatible with others. Some are, but are described in a different way - for example using a different interpolation. Some meshes may live underneath an animated transform and cannot be merged with others that don't, or are animated elsewhere. A simple mesh can be defined, or an "Abstract" (non-renderable) mesh can be described that is then instanced in other places throughout the scene. Instances of prims cannot be modified, however their Prototype (the thing they are instancing) can.

USD provides various Composition Arcs, ways of putting a scene together. Separate USD files can be sublayered or referenced together. Prims can inherit from and specialize a base prim description to refine it. These layers of composition make processing scenes much more complex than a simple flat scene, as the description of a mesh in a scene may end up being derived from multiple different source files or locations.

This is not an exhaustive list, but serves to show that Merge must deal with much of the nuance involved in authoring scenes with USD that may not be immediately obvious to a user that is interacting with a DCC tool. It must understand these concepts so that it can do its job without breaking the intent or functionality of the scene.

Input Mesh Selection

The selection of Prims that will be considered for merging is a combination of user input and programmatic filtering. The Stage is traversed in a manner that is aware of the capabilities of Merge, along with an understanding of when meshes *can't* be grouped together.

The operation accepts a list of Prim Paths which are used as the starting points for stage traversal, if no value is provided then the traversal starts at the Stage root.

At various points during the traversal we may encounter "Merge Boundaries", points of the hierarchy we know we cannot merge meshes across. For example, we may find meshes underneath an animated transform, or inside the prototypes of a point instancer. In order to maintain the functionality of the USD stage we can't merge across these points. However, we can still merge *within* them.

As the stage is traversed, any meshes that are found are recorded against a “root prim”, which is the path at which their merged output can be created. In the case we hit a Merge Boundary then we can start a new traversal at that point, using the boundary as the new root path. This allows us to end up with a mapping of root paths to the list of Prims that can be merged there.

Traversal Preparation

Prior to stage traversal the scene is queried for any instances that may occur in the scene. We look at the Prototypes in the scene (the things that are being instanced) and record which Prims instance them. We can't determine this from the instanceable Prim as we traverse, so the information is cached up front.

Stage Traversal

The traversal seeks to match that of Hydra so that Prims are considered in the context of how the renderer will see them.

The traversal is pruned in the following cases;

- **Visibility**: If a prim is invisible we do not consider it or traverse its children. This is because invisibility is inherited and cannot be overridden on children.
- **Abstract specifier**: If a Prim has a Concrete specifier (“define”) we do not traverse child Prims with Abstract specifiers (“over” and “class”) as they are not part of the renderable state.
- **Non-imageable Prim types**: We do not consider Mesh Prims below UsdGeomCamera or UsdShadeMaterial Prims as they should not be considered standard meshes.
- **Skinned meshes:** We assume that any meshes below a UsdSkelRoot Prim are skinned so should not be considered standard meshes.

Prim Selection

Prims are excluded in the following cases;

- **Prim Type**: Prims of any type other than UsdGeomMesh are skipped.
- **Prim State**: Prims that are Abstract, Invalid or Inactive are skipped.
- **Empty Topology**: If the topology attributes (points, faceVertexCounts or faceVertexIndices) have no values then the mesh is skipped.
- **Time Sampled Topology**: If the topology attributes might be time varying then the mesh is skipped.
- **Time Sampled Xform**: If the Xform of the mesh prim itself might be time varying then the mesh is skipped.

The fact that the Stage Traversal and Prim Selection rules are separate means that, in cases where a user explicitly defines a start point for traversal (below an *over* for example) we will consider the prims, even though we would not have considered them if traversal started at the pseudo root.

Merge Boundaries

In order to maintain the functionality of the USD stage, there are parts of the hierarchy that Prims cannot be merged across. We call these “*Merge Boundaries*”. A Merge Boundary is a Prim that must be the common ancestor of the input and output mesh Prims to maintain a behavior. These Prims are identified during traversal and paired with the mergeable Prims below them. Examples of merge boundaries are listed below.

Animated Transform

When a UsdGeomXformable Prim has a time sampled Xform we treat it as a merge boundary. If we did not then we would need to bake the time sampled transformation into the points of the output mesh which has a performance cost. Having the transformation inherit from the transform hierarchy provides more value than reducing the prim count in this case.

Point Instancer

UsdGeomPointInstancer Prims interrupt the transform hierarchy and start a new hierarchy for each of the prototypes that the point instancer distributes across its points. We treat each prototype as a Merge Boundary for that reason. If we did not then the input meshes below the prototype could be merged to an output location that is not below the prototype. This would change the point instancer behavior.

Instanceable

When a prim is instanceable, its children are instance proxies and cannot be edited. We therefore do not treat them as input meshes as we cannot deactivate them after merging. We can however identify the prim that the proxies are an instance of and merge them within that scope. We do this by computing a list of the Prims that are instanced and when we encounter them during traversal we consider them Merge Boundaries, redirecting the traversal with a new root path

User Input

As well as maintaining the functional structure of the stage we also use Merge Boundaries to retain the *meaningful* structure of the stage. The Usd Kind system can express places in the hierarchy that are meaningful. We allow the user to specify which Kinds should be considered Merge Boundaries.

While we do not support arbitrary Kinds we do support
- Assembly
- Group

- Component
- Model
- Subcomponent

We also support other basic Merge Boundaries to ensure the output meshes have a logical parent:
- Pseudo Root
- Leaf Xform

Limitations

- The only composition arcs that we currently treat as a merge boundary are References where the prim is “instanceable=true”. This is because we can bake the composed state of other arcs into the current layer.

Mesh Bucketing

After the set of input meshes has been established we still need to determine which ones can safely be merged together. The Bucketing process analyzes the inputs and further places them into “Buckets” of meshes that can have their data concatenated into one output mesh. This process involves generating a hash for each Prim and then grouping them based on it. The hash consists of the value and/or presence of various attributes.

Value Attributes

Attributes that cannot be concatenated during merge must be equal on Prims that are bucketed together, so that the attribute can be set on the output Prim and match that of the input Prims.

The bucketing logic calls these “Value Attributes”. The attribute name and its value at the default time determines the Bucket that an input Prim is placed into.

Inherited Values

Some attributes (such as visibility, purpose and material binding) in USD have inherited values, so getting the attribute from the Prim itself is insufficient. The effective value can only be computed by inspecting the authored value on parent Prims.

Due to Merge Boundaries we know the common ancestor of the input and output Prims, we therefore only need to consider attribute values authored between the input Prim and the common ancestor as any inherited attribute values authored on ancestors above that will also be inherited by the output Prim. This limited scope of inherited attribute values requires us to perform our own computation rather than using the schema functions provided by USD but is faster.

Fallback Values

Attribute values should be considered equal if one has no authored value, but the fallback value is the same as the authored value of the other attribute. This reduces the number of unique buckets and therefore the number of output meshes.

Cascading Attributes

Some attributes (such as normals and primvars:normals) have multiple names under which they can be stored, but only one effective value. In these cases the bucketing process must be aware of the different attribute names and the order of precedence in order to compute the effective value.

The cascading attributes for Merge are;

- *normals -> primvars:normals*

Presence Attributes

Some attributes (such as normals and texcoord primvars) do not have logical default values, so even though the values can be concatenated if authored, we cannot merge a Prim that does not have an authored value with one that does not.

The Bucketing logic calls these “Presence Attributes”. The presence of an authored value determines the Bucket that an input Prim is placed into.

The presence attributes for Merge are;

- *primvars:st*
- *primvars:normals*

Data Volume

There is a limit to the number of input Prims that should be merged into a single output Prim. The current metric that we use is the number of faces that the output mesh will have. Once a Bucket contains enough input Prims that adding the next would take the output face count over 10 million we start a new Bucket. This avoids buffers from exceeding both GPU hardware limits and software limits within USD.

Output Mesh Construction

Prim Definition

The output Prim will be of type “Mesh” and use the “def” specifier. We ensure that the specifiers of all parents remain unchanged.

The output Prim will have the same “Applied API Schema” as the input prims.

Prim Naming

Once we know the number of output meshes required (based on bucketing) we can define the names that will be given to the output meshes. We need unique names for each output Prim and do not want to reuse any existing Prim paths. Therefore we add a unique numeric suffix to the output Prims where needed.

The user can specify an “Output Name”. By default we use “merged”. If the output name is a Prim path (such as “geometry/mesh” we will create any additional Prims with a type of “xform”. The unique suffix follows the pattern “_#” and will start at 1.

The naming pattern is: *<Merge Boundary Path>/<Output Name><Unique Suffix>*

Pivot and Orientation

To make the output mesh easy to work with we set the pivot at the center of the bounds. We compute the centroid of the points of all the input meshes and set that as the xform pivot of the output Prim, we then compute the points attribute for the output mesh relative to that.

We do not currently set a meaningful orientation, so its orientation will match that of the parent.

Common Attributes

Attributes that cannot be merged will have contributed to the bucketing process, ensuring that the values are common across all the input Prims in that bucket. We can therefore safely set those attributes on the output Prim.

Concatenated Attributes

Attributes whose values need to be concatenated from the input Prims follow a common pattern. The count is calculated in advance during the bucket process, the array is reserved, then values are copied from the input Prim attributes.

Attributes that hold the indices of values in other attributes (faceVertexIndices for example) are offset with a running total that matches the attribute they are indexing into. Attributes that are not defined on some input Prims but have logical default values (primvars:displayOpacity for example) have that value inserted to pad the value.

The attributes supported are;

- Face Vertex Counts

- Face Vertex Indices
- Points
- Normals
- Hole Indices
- Corner Indices
- Corner Sharpness
- Crease Lengths
- Crease Indices
- Crease Sharpness
- See supported Primvars

Primvar Merging

The Data Adapter has special handling for merging primvar attributes. Primvars can be described with various different interpolations, for example *constant* or *uniform*. Differing interpolations can't be merged by directly concatenating them, but we don't want this to prevent being able to merge these meshes. For example we may also have two meshes that both have a *constant* primvar, however the value is different. In this case we also can't directly concatenate as it would no longer be constant. Fortunately we can convert between interpolations in a number of cases, which we call "*Expansion*".

After the primary geometric data has been merged we're able to process the primvars, as we now have some useful information such as the total number of faces and points. This is required if we are expanding to *uniform* or *faceVarying* interpolation.

Primvars can also be stored in two different ways. Either just the values can be authored, or the values and a set of indices into those values can be authored. The Data Adapter indexes all primvar values. This can result in reduced file size in cases where it can be cheaper to author integer indices plus unique float arrays rather than float arrays containing duplicate values.

Processing Primvars

Following is a description of the general primvar process that happens during a merge.

1. Preprocess

First each input mesh is iterated. We check the interpolation of each primvar on the mesh so that we can work out what the final output interpolation of that primvar should be. For example, we check each mesh that has a *displayColor* primvar. If we find one that is *constant* and then one that is *uniform* we know that the output interpolation must be *uniform* to represent both of those sets of data. As described above, we can also compare the value of *constant* primvars to work out whether we need to expand to *uniform* for that primvar if the values differ.

2. Prepare

After working out the target interpolation for each primvar we can "prepare" them, which resizes the output indices vector based on the interpolation, and the number of points or faces where required. This is done before concatenating or expanding the data for performance reasons.

3. Expand From Mesh

Next we can traverse the meshes in parallel and store the output primvar values and indices. For each primvar on each mesh we know its current interpolation and we know the target output interpolation. We do the actual "expansion" of values here. Each value of the primvar is indexed and then those indices are written directly into the previously sized vectors. A prerequisite of this is that we know the offset in the output vector at which to write the data.
If the primvar interpolation and target interpolation match then it's a simple copy into the output. If we have to expand then they are expanded between the various interpolations. For example if we are expanding a *constant* primvar to *uniform* we can simply index the value and then author it for each face, instead of just once.

4. Authoring

At this point we have prepared all of the final output values required for each valid primvar. During the Output Mesh Construction stage of Merge we can create the primvars and author the final values/indices.

Normals

Handling *normals* is also done via primvar merging. In USD, *normals* can be described as a standard attribute on a mesh or they can be described as a primvar. Describing them as a primvar allows them to be indexed, so the Data Adapter always authors them as an indexed primvar. There is some special-case handling to query them from either of these sources, and also to transform them if required, for example their orientation may need to be adjusted based on transform hierarchy differences between the source and destination prims.

Overrides

The primvar merging process has internal support for "Overriding" the value of a primvar. This allows us to use a set of predetermined values in place of querying a mesh directly for its authored values for any given primvar. We currently use this in two situations.

The first is for normals. We may need to adjust the normal values if the merged mesh will be in a part of the stage with a different transform hierarchy, so this lets us process the normals separately and then use the new overridden values in place of the authored normals. If no adjustment is required then we can copy them directly which is more efficient.

The second is for Display Color and Display Opacity. See Display Color & Opacity for details.

Limitations

Inheritance

Currently the Data Adapter does not support inherited primvars. That is, only primvars authored directly on a mesh are considered, and not primvars authored somewhere in the parent hierarchy.

Expansion

We do not support expanding to *varying* or *vertex* interpolation. If all primvars for a set of meshes use one of those interpolations then the output will remain the same, as this is a simple direct concatenation, but if there are mixed interpolations (eg, a *constant* primvar and a *vertex* primvar) then the primvar will be expanded to *faceVarying*.

Supported Primvars

Only a small set of primvars are currently considered for merging. These are:

- displayColor
- displayOpacity
- normals
- st

Material Handling

Subset Binding

UsdGeomSubset based material bindings on input meshes will be recreated for the output meshes and the face indices for each subset will be offset to match that of the new mesh. A direct material binding will be made to the original material.

Material bindings on the input meshes themselves are handled one of two ways based on the “Consider Materials” argument:

- The bound material is considered during Bucket and a unique output mesh will be created for each of the unique materials.

- The bound material is ignored during Bucket, but a UsdGeomSubset is created for each unique material and the material is bound to that. No material is bound to the output mesh itself.

Display Color and Opacity

Compute Display Colors

Display Color and Display Opacity can be handled as standard primvars however Merge has an additional option, *Compute Display Colors*. This option tells it to determine a color value based on the bound material of input meshes, and to skip binding those materials on the resulting output mesh.

When this option is enabled, Merge will use a default base gray color of (0.2, 0.2, 0.2), with opacity set to 1.0 (opaque). Each input mesh is checked to determine its bound material. For any bound material it then attempts to determine a constant color to represent it. The current heuristic for this is very simple, it checks for three color attributes and one opacity attribute on the material itself, and checks the computed surface shader if they are not found there.

These attributes are

- inputs:displayColor
- inputs:diffuseColor
- inputs:diffuse_color_constant
- inputs:opacity

Either the determined color/opacity or the default values are then used as an override for *primvars:displayColor* and *primvars:displayOpacity* on the input mesh, with the interpolation set to *constant*. The result on the merged mesh is indexed colors authored as a primvar and no material bindings. These constant values still go through the standard Primvar Merging process, so may be expanded to uniform if there are different colors on different meshes.

Limitations

- Textures are not currently considered.
- Material subset bindings are not currently supported.

Input Mesh Removal

When Merge creates an output mesh users can choose what to do with the original input meshes. They can be ignored and left as is, deleted, deactivated or hidden.

Ignore, Deactivate and Hide are straightforward. Delete is slightly more complex. It is not always possible to delete Prims from a stage due to complexities with the composition system within USD. If a Prim can't be deleted, it will instead be deactivated. This means the Prim still exists but is inactive and not rendered. If the stage is then exported it will be stripped from the result.

Limitations

Time Samples

Currently time-sampled meshes are not supported. Anything with time samples will not be considered for merging.

Might Be Time Varying

The condition for "might be time varying" is the presence of more than one time sample. For performance reasons we do not actually compare the values at those time samples, they could in fact be identical values meaning the the value does not actually vary over time.

Primvar Inheritance

As mentioned in the Primvar Merging section, inherited primvars are not currently supported.

RAM Usage

As mentioned elsewhere in this document, the Data Adapter makes liberal use of RAM in order to ensure it remains performant. It is possible that this may cause problems when attempting to merge a very large scene that is open when a machine is running out of free memory.

Variant Sets

Variant sets are not currently considered. The composed state of the scene will be used.

Metadata

We do not copy Prim metadata authored on input Meshes to the output Mesh.
We do not copy metadata authored on input Mesh attributes to output Meshes attributes.

Geom Subsets

When merging meshes with a large number of materials, and using the option considerMaterials=false, UsdGeomSubsets will be created on the merged mesh to allow defining the material per-face. There is a limit of 65535 UsdGeomSubsets per mesh, and currently there is no solution in place to prevent this limit from being broken. This may end up causing failures to display the resulting merged mesh.