Data Adapter Deduplication

The data adapter is a Kit extension that performs scene optimization. This allows complex scenes to be converted into more lightweight representations which can be displayed and evaluated more quickly.

The adapter provides a number of optimization tools which can be used individually or in combination to optimize a scene. Examples are merging meshes, deleting specific objects from the scene, and deduplicating geometry.

This document discusses the technical details of the deduplicate geometry feature.

Motivation

The largest chunk of data within a USD file is occupied by geometry. Within the context of manufacturing datasets particularly, it is staggering how many obvious duplicates there are, although the same problem can occur in data from many industries and content creation tools. Thus, there is a lot to gain by replacing duplicate geometry with instances. Reducing the amount of geometric data in a USD file implies:

- Less data to be fetched from drive
- Less data to be sent over the network
- Less data to be uploaded to the GPU
- Less data occupying GPU Memory

To put it short, when looking at a USD file of a factory scene, it may be good if the geometric data of a specific screw, or bolt, that may exist several hundred times is stored exactly once.

This document is split into two parts, how to find duplicates and how to deal with those duplicates once we've found them.

Finding Duplicates

Deduplication In an Ideal World

For now let's assume that all duplicate meshes are in fact bitwise duplicates. This applies to the connectivity data of the mesh as well as attributes such as points, normals and uv data. For this scenario it is rather straightforward to devise a linear algorithm that detects all bitwise duplicates in a USD file.

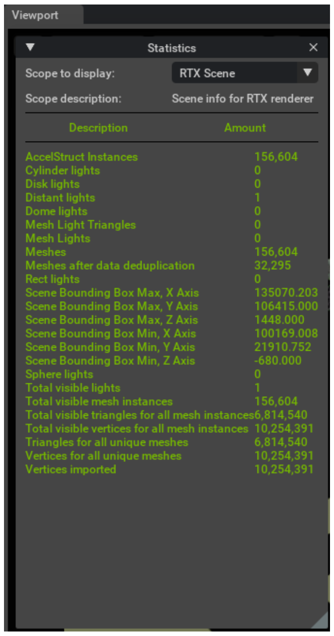
Assuming collision free hashing:

- Keep an unorderd_map
    - from size_t – the hash value
    - to a list of UsdGeomMesh – the duplicates

- For every UsdGeomMesh:
    - Compute a hash value for the UsdGeomMesh using the entire data structure.
    - Add the UsdGeomMesh to the list of duplicates associated with the computed value.

In order to account for hash collisions one just stores the pair of the hash value and the first UsdGeomMesh that established that hash value. That is, whenever a potential duplicate is found via hashing, the duplicate is verified using a bitwise comparison with the first UsdGeomMesh that established that hash value.

In fact this algorithm is essentially implemented in the RTX renderer in Create, which can potentially save a lot of memory on the GPU. The screen shot shows the RTX statistics on a rather large scene with about 156K meshes, which boils down to only 32K meshes after deduplication on RTX.



Scope to display: RTX Scene

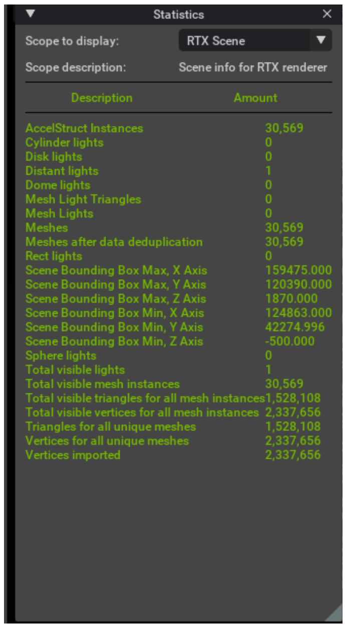
Scope description: Scene info for RTX renderer

Description	Amount
AccelStruct Instances	156,604
Cylinder lights	0
Disk lights	0
Distant lights	1
Dome lights	0
Mesh Light Triangles	0
Mesh Lights	0
Meshes	156,604
Meshes after data deduplication	32,295
Rect lights	0
Scene Bounding Box Max, X Axis	135070.203
Scene Bounding Box Max, Y Axis	106415.000
Scene Bounding Box Max, Z Axis	1448.000
Scene Bounding Box Min, X Axis	100169.008
Scene Bounding Box Min, Y Axis	21910.752
Scene Bounding Box Min, Z Axis	-680.000
Sphere lights	0
Total visible lights	1
Total visible mesh instances	156,604
Total visible triangles for all mesh instances	6,814,540
Total visible vertices for all mesh instances	10,254,391
Triangles for all unique meshes	6,814,540
Vertices for all unique meshes	10,254,391
Vertices imported	10,254,391

Deduplication in the Wild

Deep Transforms

Unfortunately this is not an ideal world and this certainly holds for the current generation of import/export tools. Specifically, the transforms that are supposed to just place an object in the scene are in fact applied to the points (as well as normals) of a mesh. We call these deep transforms[1].



Scope to display: RTX Scene

Scope description: Scene info for RTX renderer

Description	Amount
AccelStruct Instances	30,569
Cylinder lights	0
Disk lights	0
Distant lights	1
Dome lights	0
Mesh Light Triangles	0
Mesh Lights	0
Meshes	30,569
Meshes after data deduplication	30,569
Rect lights	0
Scene Bounding Box Max, X Axis	159475.000
Scene Bounding Box Max, Y Axis	120390.000
Scene Bounding Box Max, Z Axis	1870.000
Scene Bounding Box Min, X Axis	124863.000
Scene Bounding Box Min, Y Axis	42274.996
Scene Bounding Box Min, Z Axis	-500.000
Sphere lights	0
Total visible lights	1
Total visible mesh instances	30,569
Total visible triangles for all mesh instances	1,528,108
Total visible vertices for all mesh instances	2,337,656
Triangles for all unique meshes	1,528,108
Vertices for all unique meshes	2,337,656
Vertices imported	2,337,656

First the bad news:

- We don't know the transforms that have been applied.
- The rounding errors that occur during a deep transform deform the object slightly.
- Hashing, as applied by RTX, doesn't work anymore (see below).
- Applying pairwise shape matching is not an option due to the complexity of the algorithms and most importantly due to the quadratic nature of the pairwise comparison.

Hashing the Connectivity

The good news is that while deep transforms change the coordinates of points (and normals), they do not change the order in which the points are stored, nor do they alter the connectivity information of a mesh, that is, the **faceVertexCounts** as well as the **faceVertexIndices** usually remain unchanged**.** And so long as we consider real world scenes in the context of manufacturing, hashing just the connectivity data is sufficient to keep the linear nature of the basic algorithm discussed above.

Duplicates up to Deep Transforms

Assuming that we have already detected and verified that two meshes indeed have identical connectivity data, we are left with the task of coming up with a linear transform that maps one mesh onto the other. If we'd known the deep two transforms T_1 and T_2, this would be easy, since:

$$T_1^{-1} M_1 = M = T_2^{-1} * M_2$$

Which is equivalent to:

$$T_2 T_1^{-1} M_1 = M_2$$

However, as already stated, we neither know M, T_1, nor T_2. But, if we are able to come up with transforms T'_1, T'_2 such that

[1] This is an analogy to shallow and deep copies, in the context of shared pointers.

$$T'_1{}^{-1} M_1 = M' = T'_2{}^{-1} * M_2 ,$$

we can still map M_1 onto M_2 via

$$T'_2 T'_1{}^{-1} M_1 = M_2 .$$

Canonical Form

We call the mesh M' the canonical form of M. However, it should be noted that we have some freedom in how we choose M', and in fact we've tried several strategies, though they all leverage the same observation, namely, that the order in which the points of a mesh are given is not changed by a deep transform. And so the principal idea is to simply take the first 4 linear independent points of $M_1$and define M' as the mesh in which the first point is mapped to the origin and the other three are mapped onto the three unit vectors of the three main axes, which then allows us to define T'_1 as well as its inverse. By doing the same with M_2, we can compute the transform that should map M_1onto M_2, in case they are indeed duplicates.

Dealing with Numerical Instability

We can not simply take the first 4 points as those 4 points may only span a linear subspace. Moreover, we should also be careful to use a set of sufficiently linear independent vectors to obtain a numerically stable transform. While we can simply define a threshold for which we consider those vectors to be sufficiently linear independent, taking the first 4 such points often lets us compute a rather bad transform, namely the first that just barely surpasses the chosen threshold.

An example to keep in mind here is a highly tessellated sphere with the first points all being in the same area, which implies that they are barely linear independent. So in case this first set of points does not yet surpass our threshold we would continue our search in that area until we finally get out of that “bad zone”. However, while that would be a “sufficiently linear independent set”, it would most likely be the one of the worst combinations we can choose.

Pseudo Random Selection

The crucial observation is that we don't need to select the first 4 points so long as we choose the same points for both meshes. Thus we apply a pseudo random shuffle to the points of each mesh (using the same seed) and then choose the first 4 sufficiently linear independent points according to that new order. And while we change the canonical form, we still send both meshes to the same one. Though with the advantage that the chosen set of points is well distributed over the mesh and is therefore most likely to yield a more stable transform than the one generated via the basic strategy.

This approach supports even non-uniform scaling as well as shearing.

Pseudo Maximizing Selection

Another strategy is to select a set of points using a heuristic that produces the most stable transform that can be generated from the point set that the mesh provides. Starting from a reference point (can be the first point or the centroid):

- The first pass over the point set selects the longest diff vector from the reference point
- The second pass selects the vector that is most perpendicular to the first.
- The third pass select the vector that generates the largest determinant (fabs)

Obviously, this is a greedy approach and will not generate the best triple of spanning vectors, but it will certainly generate a pretty good one.

However, the main problem with the approach described above are rounding errors that may have already occurred during the initial application of the deep transform. The issue is that for symmetric objects such as a cube there are in fact several vectors that would establish the maximum, which means that in the presence of rounding errors their selection would be essentially random. To prevent that, we give a slight preference to the vector that we found first, and only select a new one in case it is better by some relative threshold.

Due to the way we select the maximal vector, this approach does not support non-uniform scaling as well as shearing in all cases as the change of the metric prevents us from selecting the same vectors. However, since essentially all deep transforms that occur in the context of manufacturing are rigid transforms, this strategy performed slightly better as it generates stable transforms more reliably.

Handling Duplicates in USD

Once we have identified meshes that are duplicates (based on the criteria established) the next challenge is how to express those duplicates in USD so that we achieve the desired optimizations outlined in our motivation.

In all of the strategies that we use there is a common pattern. For each set of duplicate meshes nominate one to act as the “prototype”, calculate the transformations needed to take the prototype and match it to the worldspace result of each of the other duplicates, then express that in USD.

Copying Geometry Attributes

The simplest approach is to take the geometry information of the prototype (points, normals, extents) and copy them into the corresponding attributes of the duplicate, while also adding a local transform to the local transform stack of the duplicate to balance out the changes.

Thus, if M_1 is the prototype and M_2 is the duplicate, we replace M_2 with $T'_2 {T'_1}^{-1} M_1$.

To summarize, this manifests in USD as;
- Opinions on the values of points, normals, extents.
- An additional xformOp in the xformOpStack.

Because we are not changing the size or ordering of points we are able to retain primvars and other point based array properties.

By doing this we make the duplicates trivial to identify as they then have identical geometric values. Specifically via hashing, that is, the RTX internal deduplication logic identifies these duplicates, reducing GPU memory consumption.

Note that we can apply this change blindly, once a duplicate has been identified. That is, we do not have to take into account any further properties of the prims, such as primvars, **UsdGeomSubset** etc.

Duplicates via USD Composition

To reduce the number of times that duplicate data is expressed on disk and in the scene description we need to take advantage of the composition system in USD. Rather than setting identical values on all the duplicate meshes we use composition arcs to express the values in one place and reuse that data across the stage.

Choice of Composition Arcs

Although we could employ Inherits, References, Specializes, or Payloads to compose the prims we have chosen to focus on References in our implementation.

Our reasoning being;
- Inherits and Specializes are not commonly understood arcs.
- References and Payloads are somewhat interchangeable, with Payloads allowing users to defer loading of the data. Because we are placing these composition arcs near the leaves of the hierarchy (where the Mesh prims live) it is unlikely that users could meaningfully manage their loaded state.

Scene Graph Instancing aka Instanceable

In order to reduce the redundant prim count for duplicate geometry (both in the stage and the renderer) we need to enable Scene Graph Instancing. This is done by setting the “instanceable” metadatum to “True”.

Scene graph instancing has a few limitations;
1. Child prims must come from composition arcs alone.
2. The children of the instanceable Prim are the items that will become instance proxies.

This means that we cannot instance geometry itself to reduce duplication, we must instance the parent Prim of the geometry. We must also create composition arcs that provide the full description of the geometric children.

Added Complexity of Composition

Changing the composition structure of an existing USD stage poses a range of challenges. We have likely not predicted all the cases where “Deduplicate Geometry” will modify the scene state. If cases are found please report them as we strive for a robust solution to this complex problem.

After meshes with duplicate topology have been identified, we perform subsequent comparisons to ensure all other properties on the Prim and its child Prims match those of duplicates.

The exception being “inheritable” properties such a visibility, purpose, transformation, constant primvars and material bindings. These can remain on the Prim (that will become the Parent) and will still inherit to the instance proxies.

This reduces the number of duplicates that we identify but allows us to reference the source prim without having to then apply additional opinions on to it.