Data Adapter Pivot

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 pivot feature.

Background and Motivation

Many models are provided in world coordinates, that is, many meshes are stored in a USD file such that the coordinates of the points of the mesh are at the actual coordinates in the scene, specifically they are often far away from the origin, which implies:

- Otherwise identical meshes are stored with different vertex coordinates.
- When looking at a mesh in “local coordinates”, it may be miles away from the origin and literally hard to find in the viewport, which is not user friendly.

Initially the goal was to come up with a canonical position of a mesh in its local coordinate frame to:

- Allow subsequent deduplication on RTX by ensuring meshes are identical (RTX can deduplicate bitwise identical geometry at runtime, improving performance and saving VRAM).
- Present the mesh in a user-friendly position, so the transform manipulator is displayed near the mesh

Since the first point is now solved more generally by Deduplicate Geometry, the current feature can focus solely on the issue of finding a user-friendly local placement of a mesh. That placement is defined by the choice of the pivot point (which becomes the new local origin) as well as the choice of the local frame (which becomes the new local coordinate frame).

The choice of a **reasonable pivot point** is most important as the camera will be targeted at that point and since the mesh is going to rotate around that point while the user is inspecting the mesh. However, we should also strive for a **consistent orientation** so that the user can easily recognize identical meshes even though they have been given to us in different initial placements. The choices made for these two properties are described below.

Choice of Pivot Point

This section discusses the various options for choosing the pivot point.

Centroid of AABB

One option is to use the centroid of the axis aligned bounding box (AABB) of the mesh in its initial position. This is not a good choice since the position of that point relative to the mesh varies with its initial orientation, as the shape of the AABB depends on that orientation. However, assuming that we have already computed the new orientation of the mesh, the centroid of the AABB (in this now well defined position) is a feasible choice.

Advantages:

- Easy to compute and therefore numerically stable.
- Depends on the shape of the object only. Specifically, it is independent of the distribution of points.

Disadvantages:

- It is not intrinsic to the shape of the object, as it is dependent on the chosen orientation

Average Point Position

The intuition here is that one would like to use the center of mass as the pivot point. Unfortunately, this is not well defined with respect to the represented shape as one already struggles to give a clean definition of the center of mass. Specifically, which mass: of the points? the surface? or the volume? While the volume would be the natural choice, this is often not defined or not well defined as this requires the mesh to bound a volume in the first place. We therefore simply **settled on using the average of all points (vertices)** of the provided mesh as it is always defined and the most easy to compute.

Advantages:

- Relatively easy to compute.
- Intrinsic to the given mesh or point cloud.

Disadvantages:

- Bias towards areas of high sampling rate, while the actual center of mass is somewhere else.

Choice of Orientation

While one could choose an arbitrary set of linear independent vectors to define a new local coordinate system, we restrict the discussion to basis transfers with an orthonormal basis, that is, the computed basis vectors are normalized to length one and are pairwise perpendicular. Therefore, the computed transform that puts the mesh into its new local frame is going to be a so-called rigid transform[1] that does not scale or shear the object.

Principal Component Analysis

The principal component analysis (PCA) somewhat suggests itself as it aligns the object along its symmetry axis, that is, the Eigenvectors of the inertia matrix of the point cloud. It can be computed somewhat easily, e.g., by using the well known library Eigen. However, one should be aware that this is a numerical approach that may not always yield a clear or stable result. As an extreme example, consider the case of a sphere. As there are no preferred directions, the computed Eigenvectors are solely dependent on numerical errors and therefore essentially random.

Advantages:

- The main advantage is that visually the resulting orientation is most likely rather intuitive at first glance, especially for objects with one obvious symmetry axis.

Disadvantages:

- Unstable for symmetric objects.
- As the approach computes the inertia matrix of the set of points, it may in fact not compute the actual symmetry axis of the represented object.

Define Transform via Order of Points

In order to avoid any ambiguities caused by rounding errors the idea is to simply use the order in which the points of the mesh are given to compute an orthonormal basis from the first three points that define two linear sufficiently linear independent vectors. And since we are anyway going to compute an orthonormal basis, this is already enough, as the third vector is going to be computed as the normalized cross product of the other two vectors. However, since the first points are also often from the same area of the mesh it is rather likely that with this approach the chosen vectors are just barely linear independent, which can cause issues with numerical stability.

For instance, consider the case of a highly tessellated sphere, we might have to traverse many points until we find two vectors that are sufficiently linear independent and more importantly we would likely settle on the more or less worst pair of such vectors, that is, the one that just surpasses the threshold that defines our notion of *sufficiently linear independent*.

Advantages:

- Easy, clear definition.

[1] As opposed to the approach taken for Geometry Deduplication in which we can use more general transforms to map one mesh onto the other.

Disadvantages:

- Issues with numerical stability for some inputs.

Pseudo Maximizing Linear Independent Vectors (implemented)

Like the previous approach, we need to find two linearly independent vectors to define an orthonormal basis. However, this time we try to choose the most stable set of linear independent vectors to compute that basis from. Choosing an initial reference point, we first compute the most extreme, that is, longest difference vector from the reference point and one of the vertices of the mesh. In a second step we compute the most perpendicular diff vector, measured by the length of the cross product between the current normalized vector and the previously chosen vector.

This approach also has some pitfalls, since for symmetric meshes, like a cube there might be several vectors that establish that extreme. Thus, in the presence of rounding errors, the final choice among those vectors would again be essentially random. To prevent that, we again rely on the order in which the points are given. That is, we give preference to the first vector that establishes the extreme and while we iterate over all candidates, we update our current choice if and only if the current candidate is able to surpass the maximum established so far by some relative threshold.

Advantage:

- Stable choice of orientation even in the presence of symmetries

Implemented Strategies

- **Pivot**: Center of Mass of the Vertices
- **Orientation**: orthonormal basis via Pseudo max linear Independent Vectors

Modification to USD files

While we apply the rigid transform computed above to the stored point attribute as well as the normal attribute (if present) of the **UsdGeomMesh**, we add the inverse transform as the most local transform to the local transform stack of the **UsdGeomXformable**, using **AddTransformOp**. While this adds a new transform to the local transform stack this allows us to be oblivious to the other transforms in the stack, that is, we implicitly handle the case in which a **TransformMightBeTimeVarying** by not overriding them in the first place.

The added transform has the suffix “`DataAdapterPivot`”. This makes the name distinguishable from other transforms in the local transform stack and guarantees that we are always able to add the transform. It also allows us to detect and skip primitives for which we may have already applied the pivot operation in a previous round. Specifically, this ensures that we can safely apply pivot a second time to the same USD file without errors or changing the file, that is, the pivot operation is **idempotent**. In fact the pivot operation is in itself idempotent, but we should avoid stacking the identity matrix onto the local transform stack over and over again, for obvious reasons.

Example of change to USD stage after Pivot

Before

```
def Xform "World"
{
    def Mesh "worldCube1"
    {
        int[] faceVertexCounts = [4, 4, 4, 4, 4, 4]
        int[] faceVertexIndices = [0, 1, 3, …]
        normal3f[] normals = [(0, 0, 1), …] (
            interpolation = "faceVarying"
        )
        point3f[] points = [(4.3954496, -0.5, 0.5), …]
        double3 xformOp:rotateXYZ = (0, 0, 0)
        double3 xformOp:scale = (1, 1, 1)
        double3 xformOp:translate = (0, 0, 0)
        float3 xformOp:translate:pivot = (4.8954496, 0, 0)
        uniform token[] xformOpOrder = [
             "xformOp:translate:pivot",
             "!invert!xformOp:translate:pivot",
             "xformOp:translate",
             "xformOp:rotateXYZ",
             "xformOp:scale"]
    }
}
```

After

```
def Xform "World"
{
    def Mesh "worldCube1"
    {
        float3[] extent = [(-0.86602545, -0.8164966, -0.70710677), …]
        int[] faceVertexCounts = [4, 4, 4, 4, 4, 4]
        int[] faceVertexIndices = [0, 1, 3, …]
        normal3f[] normals = [(0.57735026, 0.4082483, 0.70710677), …] (
            interpolation = "faceVarying"
        )
        point3f[] points = [(0.86602545, 0, 0), …]
             double3 xformOp:rotateXYZ = (0, 0, 0)
                    double3 xformOp:scale = (1, 1, 1)
             matrix4d xformOp:transform:DataAdapterPivot = ((-0.577, -0.577, 0.577, 0),...)
             double3 xformOp:translate = (0, 0, 0)
             float3 xformOp:translate:pivot = (4.8954496, 0, 0)
             uniform token[] xformOpOrder = [
                    "xformOp:translate:pivot",
                    "!invert!xformOp:translate:pivot",
                    "xformOp:translate",
                    "xformOp:rotateXYZ",
                    "xformOp:scale",
                    "xformOp:transform:DataAdapterPivot"]
    }
}
```