DMesh: Mesh Handling Middleware

Ian Ruh
University of Wisconsin-Madison

Mentor: Abhinandan Jain (Jet Propulsion Laboratory/California Institute of Technology)
Mentor: Aaron Gaut (Jet Propulsion Laboratory/California Institute of Technology)

We developed a new mesh handling middleware for the DARTS [1] simulation engine that unifies the existing implementations currently used throughout the codebase and extends their functionality to enable additional use cases, including modeling terrain for sampling operations and driving. As meshes are an integral part of 3D simulations, we rely as much as possible on well-tested and performant open-source code, including the LibIGL [3] and CGAL [4] libraries, rather than developing and maintaining custom code. The DMesh middleware intentionally maintains a relatively low-level API and provides only the foundational operations necessary for more advanced uses. The low-level API allows the middleware to be built upon for different applications throughout the code base, including collision detection, terrain modeling, CAD parts, soil modification, and graphics.

Meshes are widely used throughout computer graphics and simulation software as a discrete representation of three dimensional objects. The DARTS simulation engine currently uses multiple different implementations of mesh classes for different applications, including graphics, collision detection, and modeling of sampling operations. The objective of this work was to design and implement mesh handling middleware that can be used for multiple applications and can unify the existing implementations.

In order to support multiple use cases, we designed the DMesh middleware with several objectives in mind including maintainability and the flexibility to allow it to be built upon. To support different specialized applications, we confined the API to the low-level operations that would enable more specialized use cases to be built on top of it rather than attempting to incorporate everything into the core implementation.

In the following sections we first describe the core implementation of the middleware, namely the mesh representation in memory and the associated data, and then we provide an overview of the DMesh API, including modification of the mesh, set operations (union, intersection, etc.), and other utilities. As of the writing of this paper, the DMesh middleware has begun to be incorporated into the rest of the DARTS codebase but is still being developed.

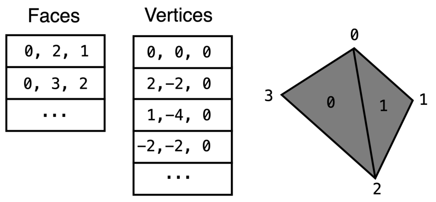
I. IMPLEMENTATION

The implementation of DMesh followed several primary design principles, including utilizing existing and tested open-source code as much as possible, providing a low-level API that allows DMesh to be utilized for different applications throughout the code base, and limiting the API size of DMesh to only providing the building blocks and relying on each use case to build the more specialized functionality they require. The specific design decisions with regard to how the meshes are represented in memory, how attributes are associated with the mesh, and how DMesh interoperates with third-party code are detailed in the following sections.

A. Mesh Storage

In order to maintain compatibility with the existing libraries we wanted to utilize, we store the triangular mesh internally as a face-vertex list, the same format used by LibIGL [3]. The face-vertex representation's simplicity makes it easy to store in memory and easy to serialize to disk, as it does not rely on pointers or references to encode the Mesh's structure. However, the representation does limit the efficiency with which some queries and operations can be carried out including removing a vertex, which becomes an $O(n)$ operation in the size of the mesh due to the need to update the indices in each face of the mesh, and determining the faces adjacent to a given vertex, which requires a search of the faces.

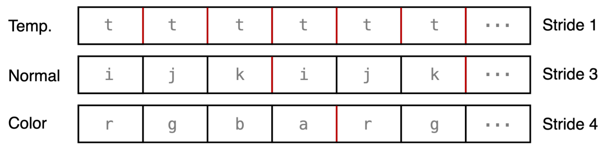
FIG. 1. Face-Vertex mesh representation example.



B. Attribute Storage

In addition to the geometry of the mesh, the Mesh class supports storing additional data about the mesh as a whole and about specific vertices. Per vertex attributes allow the user to store multiple layers of integer and double attributes with the mesh geometry. A layer consists of a vector of integers or doubles with each value, or a set of values, mapping to a single vertex. When each layer is constructed, a stride can be defined on the layer that allows multiple values to be stored for each vertex. This allows more complex structures to be built on top of the low level attributes API provided by DMesh.

FIG. 2. In memory storage of vertex attributes.



C. Library Interoperability

To utilize the code provided by third-party libraries we need to be able to convert to and from the formats used by the external code. DMesh currently utilizes three open-source libraries for much of its functionality, LibIGL [3], CGAL [4], and Embree [5]. LibIGL is a general purpose 3D mesh library that itself relies on CGAL and Embree for some of its functionality. Because LibIGL already provides the functions to convert from its face-vertex mesh format to the half-edge structure used by CGAL and the mesh data structure used by Embree, we chose to store the mesh geometry in DMesh in the face-vertex format.

Each function that utilizes CGAL or Embree has the added overhead of converting the mesh's geometry to their respective format if it has changed. For some operations, including taking the union, intersection, etc., this isn't as important as they are not expected to be called in real time, but for operations such as ray casting, which may be called hundreds or more times per second, the added overhead is significant. To alleviate the cost of converting we maintain the data structure used by Embree and lazily rebuild it only when the mesh's geometry has changed.

The majority of DMesh's implementation that deals with the geometry of the meshes utilizes one of the external libraries above, which both simplifies and relieves the maintenance burden of our code.

II. MESH API OVERVIEW

In the following sections we provide an overview of the API provided by DMesh and discuss the relevant implementation details. The API can be broken up into four primary parts: functions for dealing with attributes and non-geometric mesh data, functions for modifying the mesh in place, for modifying the mesh out of place, and sub-meshes.

A. Attributes, Texture Coordinates, and Materials

Vertex attributes are an integral part of meshes, allowing non-geometric data to be mapped onto 3D parts. Ideally, DMesh would support arbitrary vertex attributes, but to facilitate serialization and simplify the implementation, only two basic attribute types are supported, integers and doubles. The intent with the API's construction is that a use case that requires more complex attributes, such as modeling soil properties, can build on top of the provided API to store and reconstruct the data structure that it requires, as demonstrated with normal vectors and colors in the following section.

The attributes and the texture coordinates are both implemented as vectors with elements that map to the vertices in the mesh. As attributes also support strides other than one, allowing multi-dimensional vectors to be stored in a single attribute layer, the mapping is not necessarily 1-1 between attribute and vertex, but may actually be a multiple, depending on the stride.

Each mesh has a single material associated with it. The Material class implementation is based on the Assimp [2] material implementation, and supports arbitrary key-value storage for integers, doubles, strings, vectors, and colors. To facilitate retrieval of common material properties, it has a static API to construct the keys need to access common properties in the key-value store.

Vertex Normals and Colors

The vertex normals and colors are actually implemented as normal vertex attributes. Their implementation demonstrates the intended usage of the low level attributes API to allow the building of more complex attributes on top. Vertex normals consist of a attribute layer of doubles with a stride of three, while the colors consist of an attribute layer with a stride of four (including an alpha value). Although both normals and colors have dedicated functions in the API because of how common their usage is, they can be accessed and modified using the same attribute functions as normal attribute layers.

Attribute Interpolation

DMesh requires that every attribute layer specify an interpolation mode when created. This allows vertices to be added without requiring the user to specify all of the attribute data. There are two interpolation modes supported for all attributes, nearest neighbor and linear, and a third mode, spherical linear interpolation (SLERP), that interpolates between two three dimensional unit vectors. Because SLERP only supports three dimensional vectors, it can only be used on double attribute layers with a stride of three. When the stride is not equal to one, both nearest neighbor and linear interpolation perform element wise interpolation of the attribute.

B. In Place Modification

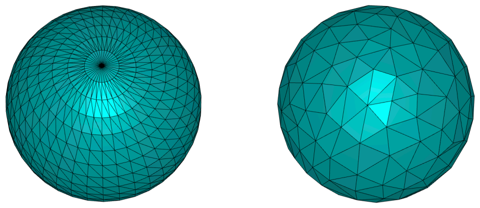
The DMesh API supports modification of the mesh's geometry and attributes in place, though not all modifications have a constant complexity. In addition to the position of each vertex, the attributes and texture coordinates associated with each can be modified in constant time. New vertices and faces can both be added in near constant time (in reality new vertices are linear in the number of attribute layers). The largest penalty, and the one a user should be most aware of, is the cost of removing a vertex from the mesh. As a consequence of the face-vertex representation of the mesh used in memory, when a vertex is removed, every subsequent vertex is shifted forward, forcing every face to be updated with the new vertex index.

C. Out of Place Modification

Operations that involve significantly altering the mesh are instead done out of place and return a new mesh with the changes. This includes decimating the mesh, which lets the user specify the target number of faces, and four basic math operations that take two meshes: union, intersection, difference, and symmetric difference.

In each case, an entirely new copy of the mesh is returned with all attributes, texture coordinates, and materials copied over, with all new vertices interpolated, with all extraneous vertices removed, but without modifying the original mesh.

FIG. 3. Decimation of sphere from 1600 faces to 400.

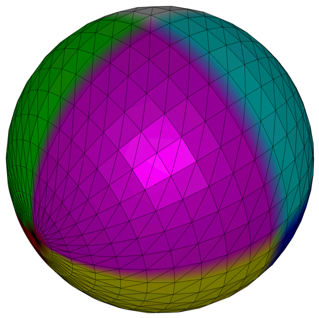


D. Sub-Meshes

In many cases the meshes being worked with are far larger than may be immediately relevant, for example a mesh modeling a terrain hundreds of meters across is not all relevant to a rover on the terrain. To increase the efficiency of some operations on the mesh, sub-meshes can be used to isolate only the locally relevant section of the mesh.

By maintaining a representation of the geometry of the mesh that only includes the isolated section, all geometric queries are sped up significantly, including ray casting, volumetric set operations, and point queries. The sub-meshes provide a window into the parent mesh, as modifications to the sub-mesh are propagated back to the parent mesh. As many of the modification operations are near constant complexity to begin with, this does not incur a significant cost.

FIG. 4. Tiled sub-meshes with each sub-mesh's color modified.



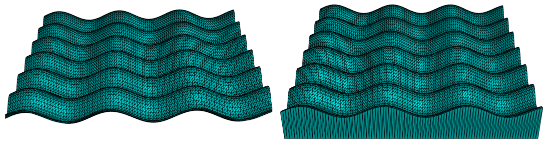
III. MESH UTILITIES

Some utility functions have also been included in DMesh to support common uses, but were not deemed essential for inclusion in the Mesh class itself. These consist primarily of functions for creating different geometric primitives in memory, including spheres, ellipsoids, planes, cylinders, and prisms.

A. Closing a Mesh

One of DMesh's intended applications is for modeling terrains for both sampling operations and for driving on. However, the meshes that model terrains are often not water-tight, meaning they extend out some distance on the top surface, but then do not close on the underside. Some operations, including union and intersection, can only be performed on meshes that actually have a volume, so we have included a utility that identifies the boundary of non-water-tight meshes and closes them by connecting the boundary to a plane provided by the user.

FIG. 5. Closing of a 2D mesh with waves to 3D volume.



IV. IMPORT / EXPORT

The import and export API is based off of the API used by Assimp [2] and consists of base Importer and Exporter classes that get derived from to implement importers and exporters for different file formats. This allows additional file formats to be easily incorporated in the future. Initially, we have implemented derived importers and exporters that support all the formats supported by Assimp.

V. CONCLUSIONS

The DMesh API should provide the necessities that allow it to be adopted in the DARTS codebase in place of the existing implementations and to enable to to be used for future applications. The design of the API should let it be built upon and extended without major modifications to the implementation.

[1] DARTS `https://dartslab.jpl.nasa.gov/`

[2] The Open-Asset-Importer Library `https://www.assimp.org/`

[3] LibIGL `https://libigl.github.io/`

[4] The Computational Geometry Algorithms Library `https://www.cgal.org/`

[5] Intel Embree `https://www.embree.org/`