A Generic Halfedge Mesh Data Structure for .Net

Alexander Kolliopoulos

November 13, 2005

Introduction and Motivation

Polygonal meshes are used extensively in computer graphics. A polygonal mesh represents a shape, usually in 3D, by using a set of points, the *vertices*, connected by *edges*. Edges form the boundaries of *faces*. A mesh is convenient to work with because it explicitly represents the surface that one is working with—vertices, edges, and faces are discreetly enumerated and may be manipulated directly.

Meshes are often stored by simply keeping a list of points with a list of faces, where each face is a list of vertices, and each vertex of a face is an index into the list of points. While this is simple to implement and manage, many common operations are slow and awkward. It is common to iterate over the neighbors of a mesh element—for example, to compute smooth contours in non-photorealistic rendering, it is necessary to examine faces adjacent to a face to trace contour chains. Another problem is that this mesh representation does not explicitly represent edges. For polyhedral contour rendering, it is necessary to iterate over all edges in a mesh. It is not immediately clear how one would do this without visiting the edges of every face, which means that each edge of a closed manifold mesh would be visited twice.

An elegant solution to these problems is the halfedge mesh data structure. This represents connectivity directly by means of a *halfedge*. Each edge is made up of two halfedges, one pointing to each vertex of the edge. Each of the halfedges also has a reference to an adjacent face, the next halfedge adjacent to this face, and the previous halfedge. A vertex or face only needs a reference to a single halfedge to find all adjacent halfedges in its neighborhood. Finding all faces adjacent to a face is efficient—starting at the face's halfedge, the first adjacent face is that on the opposite halfedge. Then, we only need simple lists for the faces, edges, and vertices, and halfedges. We move to the next halfedge, and its opposite halfedge has a reference to the second adjacent face. This process may be continued until we have returned to the starting halfedge.

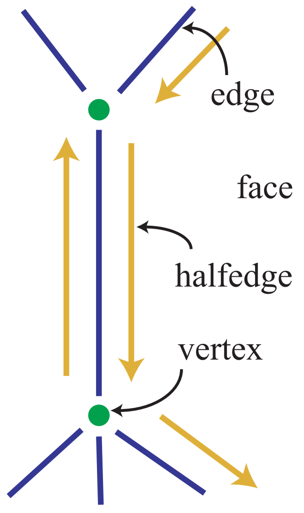

Figure 1: The elements of a halfedge mesh data structure.

While all connectivity information of the mesh is available, it can be awkward to traverse the data structure. This can be alleviated by providing iterators and methods that hide much of the internal design of the data structure. OpenMesh is a library that provides such an abstraction for C++ [1]. This presents the user with classes corresponding to all the basic mesh elements with a number of iterators. An attractive feature of this library is its use of C++ templates to parameterize the data that is associated with each element, called *traits*. A user can, say, add curvature information to vertices in the mesh type definition, and this adds curvature data members to every vertex of a mesh. Once a user is familiar with the concepts, this library provides a very powerful interface to querying mesh topology.

When I finished my masters thesis in non-photorealistic rendering, it became clear to me that the C++ design of my project had just about reached its breaking point. The C# 2.0 betas offered a compelling development framework, but this would require a completely new mesh data structure, as none existed that satisfied the needs of my work. Hence, I decided to implement a generic halfedge mesh data structure in C# 2.0 with features similar to those of OpenMesh.

Design and Implementation

The .Net design guidelines suggest a number of coding practices to assist in keeping an external interface simple and consistent, ranging from naming of types to error handling with exceptions [3]. Where it makes sense, these guidelines are followed, but instances where the guidelines are too limiting or awkward for the design of such a complex data structure are noted.

The library is based on an extensible mesh topology class that does not explicitly implement any geometric data types or methods. Using the generics feature of C# 2.0, this class parameterizes traits for each of the four mesh elements—edge, face, halfedge, and vertex. This immediately presents a problem: due to the extensive use of references, each of the element types must be aware of all the traits of the mesh. The alternative is to cast element types to some base type and back, which would be exposed to a user in a very unsatisfactory fashion. However, it is also troubling to consider declaring and using types such as `Halfedge<EdgeTraits, FaceTraits, HalfedgeTraits, VertexTraits>`, since very simple operations, such as instantiating a new halfedge object, become an unreadable mess of generic type specifiers. Each of the classes—edge, face, halfedge, vertex, and mesh—could have their names shortened with the `using` keyword, but this would require five lines that must be kept consistent at the top of every source code file for every type of mesh being used. A solution is to fold all the mesh element types into the main mesh class as public nested classes. While the .Net guidelines discourage the use of public nested classes, in this case it is a much more elegant solution than the alternative. Not only does this reflect the fact that an edge is an element that belongs to a mesh, rather than being a class at the same level as the base mesh class, but it also allows us to use a single alias with C#'s `using` statement—only at the top of a file would one need to declare `using MeshType = Mesh<EdgeTraits, FaceTraits, HalfedgeTraits, VertexTraits>;`. Then, a user can simply use types such as `MeshType.Edge`, `MeshType.Face`, and so on.

For the generic topology class to be of much use in graphics, traits must be defined on at least some of its elements. At the very least, a user is likely to require vertex positions, which are not provided at the topological level, as this allows a greater degree of customization. A trait exposes itself as a single public data member on each mesh element. Here again, it is necessary to break from the .Net guidelines that strongly discourage public instance fields. It would seem that a public property would trivially offer the same capabilities, however, this approach introduces its own difficulties. In the case of using a value type trait, accessing its corresponding property would copy the entire data structure to a new memory location, which is usually unnecessary. Even worse, it is impossible to assign a value to a single property or field of the trait, since a copy of the trait is being accessed rather than the trait itself. This means a user has to make a copy of the trait, change the one item in question, and replace the entire trait. Use of a public field eliminates such problems without introducing any complications. As an example, suppose the vertex trait is a `Vector3` structure. To change the sign of the Z value of a vertex using a property would require the following code:

```
MeshType.Vertex v = GetVertexFromSomeMethod();
Vector3 position = v.Traits;
position.Z *= -1;
v.Traits = position;
```

Using a public field instead of a property for the vertex trait, this code simplifies to the following:

```
MeshType.Vertex v = GetVertexFromSomeMethod();
v.Traits.Z *= -1;
```

While a trait can be as simple as an integer or a vector structure, in practice it is helpful to wrap traits in their own structures so that they will have identifiable names and new traits can be added later without breaking existing code. It is much clearer to access `v.Traits.Position` than simply `v.Traits`. One of the drawbacks to this design is apparent when a mesh element doesn't require a trait. The solution is to define an empty structure and use it as the trait type: `struct NullTraits { }`. This can incur a one or more byte overhead for each traitless element, even though it contains no data. While it may not be elegant, it isn't a significant waste of memory considering each mesh element will typically require tens of bytes for references to other mesh elements and bookkeeping information.

Traits provide a simple way to associate data with features of a mesh, but the real strength of a halfedge mesh structure is the ability to locally traverse a mesh quickly and easily. In addition to providing direct access to halfedge links, a number of iterators allow users to quickly query element neighborhoods. Unlike C++ iterators, C# 2.0 provides coroutines so that iterator implementation is very concise and clear, while the user needs never directly create or manage an iterator object. To call a `Visit` function on each face adjacent to a vertex, one only needs the code `foreach (MeshType.Face f in vertex.Faces) { Visit(f); }`, which uses the `Faces` property of a vertex object, hiding the iterator implementation.

There are often times when it is useful to have some data attached to a type of mesh element for only a short period of time during program execution. For example, to calculate normals on vertices, it is useful to calculate normals on faces first; but once the vertex normal calculation is complete, the face normals are no longer necessary. Rather than defining these face normals in the face traits and living with the memory penalty when the trait is not needed, OpenMesh provides classes that allow one to add and remove data from mesh elements at runtime, which they call properties. To avoid confusion with C# properties, we use the term *dynamic traits*. In the base mesh class, dynamic traits are implemented as nested classes, one for each type of mesh element. A dynamic trait class takes a single generic type parameter that defines the type of data to associate with its mesh element. It initializes an array of the trait type to the appropriate size, so it is only valid for elements currently in the mesh. A dynamic trait is accessed by passing the item for which the value is required to an indexer. For example, a face trait is accessed by `face.Traits.`*`Member`* while a dynamic trait is accessed by `faceDynamicTrait[face]`. When a user is finished with a dynamic trait, it can be set to null, so the memory it occupies will be freed the next time the garbage collector examines it.

With the generic topological halfedge mesh class and its element classes in place, it is only a matter of defining trait structures to produce a fully functional 3D mesh class. In fact, one can inherit from the topological class directly to hide the details of the trait definitions and associate algorithms with class methods. A `TriMesh` class has been implemented with members based on the Sharp3D.Math library to handle vectors [2], and it restricts instances to have triangular faces since some of its algorithms require this. Complex operations, such as reading a mesh from a file or computing principle curvatures, are implemented as methods. Furthermore, the traits are almost completely hidden away ("almost" hidden because the compiler will sometimes generate messages that show the full definition of the base class), but otherwise, a user need not even be aware of the generic base class. Mesh elements have simple names like `TriMesh.Face` and `TriMesh.Vertex`, and all the nested classes and methods of the base class are defined for free in the derived class.

Tests and Tools

The base mesh class alone is made up of 12 nested classes in addition to a number of new exception types, and many of the nested classes are closely tied in behavior. The `TriMesh` class introduces a number of algorithms on top of this library. With the large number of interdependent components, small changes to one can have non-obvious effects in other places. To a degree, the refactoring capabilities of Visual Studio 2005 have helped prevent problems early on [5]. A small number of tests have been written so far to test some of the basic behaviors and the more complicated parts of the library. These tests are written for NUnit, which takes advantage of C# attributes to simplify test design and execution [7]. While many more tests need to be written to reflect the scale of the library, the current collection helps give one confidence in some of the more complicated methods.

Following the .Net coding style guidelines ensures code consistency and sometimes helps prevent mistakes in judgment, but it is inevitable to occasionally overlook some good programming practices. FxCop is an exceptional tool to help find possible problems with design and performance that a programmer might miss or forget [4]. The tool analyzes compiled code, for which it indicates warnings according to a library of rules. The rules range from checking that members are cased correctly to ensuring arguments to public methods are validated before being used. Although many warnings are raised due to the atypical design of the library, those that correspond to conscious design decisions can be excluded by the user, and the others can be dealt with case by case.

This library would be of little use to most other developers without good documentation, as there are a number of particular features that diverge from the design of standard class libraries. All publicly exposed members already have C# XML comments in place; this can be easily checked since warnings can be enabled in the compiler for public members that lack documentation. Unfortunately, tools for turning these comments into a more useful form of documentation are currently lacking. NDoc is a great tool for generating human-readable documentation from XML comments, but it still does not support C# 2.0 [6]. When a tool does become ready to support the new documentation features of C# 2.0, the library will be completely ready though.

Results and Conclusion

Features of C# 2.0 have greatly reduced the amount of time spent managing memory and debugging compared to what I am used to, while the various free tools for testing and checking code have been greatly helpful. In the future, profiling tools that support C# 2.0 might be useful for optimizing performance as well. A coverage tool could also help ensure that as much code as possible is being tested.

Interestingly, OpenMesh has recently abandoned support for their implementation of traits in favor of only what roughly corresponds to dynamic traits. This simplifies extending a base mesh type—all nonessential data members exist in objects separate from the mesh itself. All of the generic type parameters could be removed from the mesh class if this approach were taken in the C# implementation, and users would need only learn one method of accessing data on mesh elements. However, this comes at the cost of a level of indirection when accessing traits, since they must be accessed by passing a mesh element to another object rather than being bound to the element itself. It would also complicate classes deriving from the topology class—all dynamic traits would be exposed as members of the mesh class, rather than hidden only where they are needed as members of the mesh elements. For these reasons, the current design of the library seems satisfactory and will remain as it is.

The library is currently being used in a number of programs within my personal research projects and will be made available online under an open source license soon. The use of generic programming for this data structure has proven to be flexible enough for my research. As the project matures, hopefully others might find it useful as well.

References

[1] Mario Botsch, Stephan Steinberg, Stephan Bischoff, and Leif Kobbelt. OpenMesh – A Generic and Efficient Polygon Mesh Data Structure. In *OpenSG Symposium*, 2002. `http://www.openmesh.org/`.

[2] Eran Kampf. Sharp3D.Math. `http://www.ekampf.com/Sharp3D.Math/`.

[3] Microsoft Corporation. Design Guidelines for Class Library Developers. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp`.

[4] Microsoft Corporation. FxCop. `http://www.gotdotnet.com/team/fxcop/`.

[5] Microsoft Corporation. Microsoft Visual Studio. `http://msdn.microsoft.com/vstudio/`.

[6] The NDoc Team. NDoc. `http://ndoc.sourceforge.net/`.

[7] The NUnit Team. NUnit. `http://www.nunit.org/`.