A Mesh Data Structure for Rendering and Subdivision

Robert F. Tobler
VRVis Research Center
Donau-City Str. 1/3
1120 Wien, Austria
rft@vrvis.at

Stefan Maierhofer
VRVis Research Center
Donau-City Str. 1/3
1120 Wien, Austria
sm@vrvis.at

ABSTRACT

Generating subdivision surfaces from polygonal meshes requires the complete topological information of the original mesh, in order to find the neighbouring faces, and vertices used in the subdivision computations. Normally, winged-edge type data-structures are used to maintain such information about a mesh. For rendering meshes, most of the topological information is irrelevant, and winged-edge type data-structures are inefficient due to their extensive use of dynamical data structures. A standard approach is the extraction of a rendering mesh from the winged-edge type data structure, thereby increasing the memory footprint significantly.

We introduce a mesh data-structure that is efficient for both tasks: creating subdivision surfaces as well as fast rendering. The new data structure maintains full topological information in an efficient and easily accessible manner, with all information necessary for rendering optimally suited for current graphics hardware. This is possible by disallowing modifications of the mesh, once the topological information has been created. In order to avoid any inconveniences due to this limitation, we provide an API that makes it possible to stitch multiple meshes and access the topology of the resulting combined mesh as if it were a single mesh. This API makes the new mesh data structure also ideally suited for generating complex geometry using mesh-based L-systems.

Keywords

Mesh, Subdivision, Rendering.

1. INTRODUCTION

Subdivision Surfaces [Cat78], [Doo78] have recently been established as a very popular method for generating smooth geometrical objects in computer graphics [Cav91], [DeR98], [Kob00]. With the rise of cheap graphics hardware for consumer PCs, real-time rendering of subdivision surfaces becomes ever more important.

In order to generate the necessary geometry for real-time rendering of subdivision surfaces, complete topology information is necessary for a given mesh of input polygons. Over the years a number of mesh representations have been developed that provide this topology information, given an arbitrary input mesh. Most of these mesh representations are based on the winged-edge representation introduced by Bruce Baumgart [Bau72].

In order to facilitate access to the complete topology of a polygon mesh, winged-edge representations maintain lists of edges around each vertex, and lists of edges for each face. Thus each edge partakes in four lists: the two lists for its end vertices, and the two lists for the faces it separates (see figure 1).

This type of data-structure is very elegant from a topological point of view: as an example, the dual mesh, where each vertex is replaced by a face, and each face is replaced by a vertex, can be easily created, since vertices and faces are structurally equivalent in this representation. Also, topological modifications to the mesh can be easily implemented by changing the corresponding lists. However, this flexibility comes at a price: due to the dynamic nature of the list data structures, access to specific edges, faces or vertices often results in following a number of elements of the described lists, reducing the access performance. In addition to that, dynamic memory management is needed to maintain these structures.

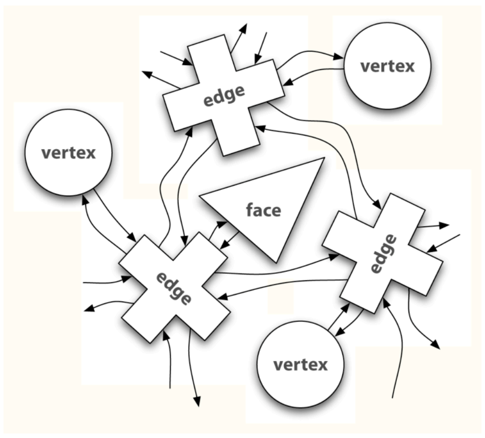

Figure 1: Winged-Edge data structure with complete topological information.

For real-time rendering, the topological information is not needed, therefore the standard approach of a number of packages is to generate a triangle representation optimized for rendering when it is needed. Although this results in optimal rendering performance, the memory cost is increased, since some information needs to be duplicated. Additionally the generation of this rendering representation requires some computation.

2. MESH REPRESENTATIONS

In order to improve the performance of the winged-edge data structure for a number of algorithms that are usually performed on meshes, various mesh representations have been developed. In the following section we will highlight two of these representations on which we have based our additional ideas.

Directed Edges

A useful adaptation of the winged-edge that avoids the original pointer data structures has been developed by Campagna et al. [Cam98]. The idea of this structure is the representation of each edge between two faces as two directed edges, each belonging to one of the faces. This is equivalent to the half edge structure described by Kettner [Ket98]. Each directed edge is part of a circle of directed edges around a face, and references its corresponding directed edge in the neighbouring face (see figure 2). In order to avoid lists in the representation, directed-edges are stored consecutively in an array, and only the base index needs to be stored in the face data structure.

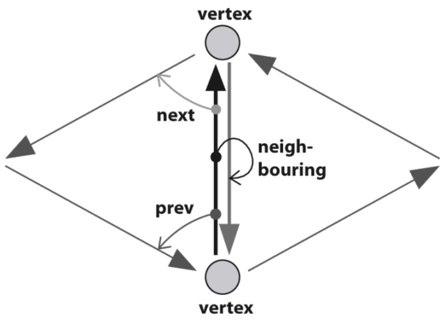

Figure 2: Directed edges representing triangles.

OpenMesh

OpenMesh [Bot02] is a C++ implementation of the ideas presented in the directed edge data structure paper. Due to the use of templates, the exact mesh data-structure that is used by an application can be adapted to the algorithms that need to be applied to the mesh. With the help of the template mechanism, arbitrary attributes can be attached to each geometric object (face, vertex, edge) in the mesh. On first glance, this data structure seems to solve all possible needs, however due to the use of templates, it is necessary to instantiate specific mesh data-structures for each algorithm that needs private attributes. Therefore, if two algorithms with different attribute needs are run on the same mesh, the mesh data structure has to be cloned.

3. MESH REQUIREMENTS

The two main algorithms that our mesh data structure has been designed for, are *subdivision* and *real-time rendering*. Although this seems limiting for a data structure, it turns out, that our resulting design is still very general and can be used for a wide variety of algorithms. The following section summarizes the base requirements that are necessary for efficient implementation of our two main algorithms.

Requirements for subdivision

As already mentioned, full topology information is necessary for subdivision algorithms. In order to optimize these algorithms it is beneficial if topology information is not stored in lists, that have to be traversed, but in arrays that can be directly indexed. Since the resulting meshes of subdivision algorithms can be stored in different meshes, it is not necessary to modify the topological information once it has been created.

Requirements for real-time rendering

Current graphics hardware supports a number of ways to supply geometric data. Among these are: sending the coordinates of each triangle separately, sending so-called triangle strips, where shared coordinates of neighbouring triangles are not retransmitted, or indexed structures, with separate coordinate arrays and triangle arrays containing vertex indices.

Since triangle strips or indexed structures are normally more efficient than sending each triangle separately, they are the methods of choice for sending geometric data. Triangle strips require an additional preprocessing step for finding the longest strips in a given mesh of triangles. Thus the indexed structures often represent the optimal choice for storing geometric information.

In real-time rendering applications a specific triangle mesh is normally rendered multiple times in subsequent frames. In order to prevent this, it is possible to explicitly store the transmitted data directly in the memory of the graphics hardware using so called *vertex buffer objects*. By using the indexed structures that separate coordinates and vertex indices, some of the information stored in the graphics hardware can be reused between different meshes.

For these reasons the use of indexed structures is often the preferred method for storing geometric data and sending the data to the graphics hardware.

4. THE NEW DATA STRUCTURE

Our data structure for rendering and subdivision maintains rendering information and topological information separately. The topological information can be created if it is needed, but need not be present for rendering.

Geometric information for rendering

In order to facilitate fast rendering, the memory layout of all geometric information has been chosen to correspond to the indexed face set data structure as used in VRML and Inventor. As current graphics hardware supports explicit commands for supplying triangles and quadrangles, we reflect this by maintaining separate arrays for these geometric primitives. Some popular types of geometric data, such as heightfields use quadrangles and therefore our explicit support in noticeable memory savings (as opposed to splitting everything into triangles). Figure 3 shows the data structure with additional normal and colour attributes for each vertex. In order to simplify our first implementation we do not explicitly support general polygons, and directly split these into triangles.

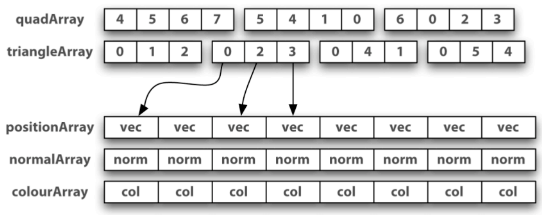

Figure 3: Mesh data structure for rendering. Shown with normal and colour attributes for each vertex.

Topology information for rendering and subdivision

As the data structure for geometric information already supplies references from each face to its vertices we designed the topological information in such a way that this information is reused. The general structure of our topological information is shown in figure 4. Note that almost all other mesh data structures (e.g. [Hop98]) do not explicitly store the references from faces to vertices. Thus for rendering all these other data structures, either these references have to be created, or a somewhat slower traversal of the mesh data structure, together with the creation of a chaced representation with additional memory requirements, needs to be performed.

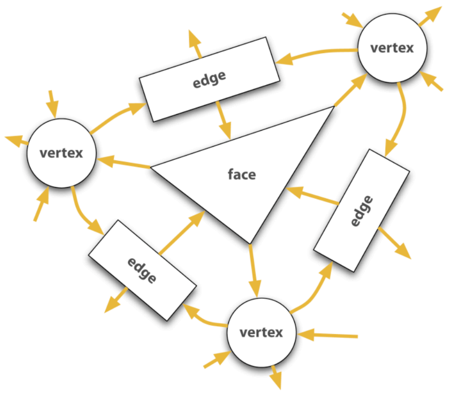

Figure 4: General structure of topological information in our mesh data structure. Note that the references from faces to vertices are already part of the rendering information.

In order to store this information as tightly as possible we use some of the ideas presented by Campagna et al. [Cam98]. A representation of the resulting memory layout can be seen in figure 5. Note that the face array shown in this figure is not stored explicitly, but consists of the concatenated triangle and quad arrays that are part of the rendering information.

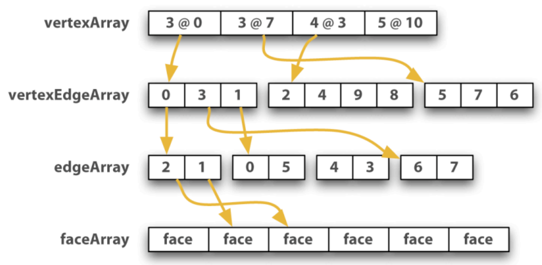

Figure 5: Memory Layout of the topological information.

To complete the topological information, as shown in figure 4 and 5, each of the references must contain more than just the index of the target object: it is necessary to know the exact orientation of the referenced object as well (see figure 6) If e.g. a face is indexed, the index of the face alone does not include the information, which side of the face is referenced. Thus for each array that contains indices of faces, vertices, or edges we maintain a parallel array that contains the orientation (or side) of the referenced face, vertex or edge. For edge references this orientation array could be packed to only contain one bit per reference. For face and vertex references, the memory consumption depends on the maximal number of vertices per face, and edges meeting in a vertex. In order to simplify our implementation we chose to use the same size for this orientation array for all three object types: 8-bit integers. Assuming 32-bit integers for all indices, this results in a tolerable memory increase of 25% for all references.

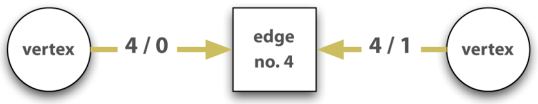

Figure 6: For each reference the orientation (or side) of the target object has to be maintained.

As the topological information for a mesh is created, the mesh is split into multiple 2-manifold sheets. Each vertex in a mesh is only allowed to be member of a single sheet, so vertices are replicated for representing non-manifold objects.

5. ADVANCED CONCEPTS

As we use indexed arrays to maintain vertices, edges, and faces, modifying the topological information would entail writing customized memory management algorithms for these arrays. Although this is possible in principle, and we may want to implement this in future, for the sake of simplicity we currently do not allow modification of the topological information, once it has been created. This sounds like a severe restriction, however any algorithm that modifies the topology of a mesh can easily be implemented in such a way, that it creates a new mesh with the modified topology.

Per object attributes

As all geometric objects (faces, vertices, and edges) can be identified by their index within a mesh, additional attributes can be maintained as parallel attribute arrays. Thus multiple algorithms that need different sets of attributes can be run on the same mesh without replicating the geometric and topological information. Based on the frequency of each attribute different implementation strategies can be chosen for each attribute:

- **dense attributes**: ie. attributes that are present for nearly every object are stored in standard arrays
- **sparse attributes**: ie. attributes that are present for very few objects are stored in hash tables

As the attributes for different algorithms can be maintained in parallel for the same geometric and topological mesh structure, and chosen to be represented according to their density, the proposed data structure achieves a near optimal memory utilization.

Stitching of multiple meshes

In order to alleviate the restriction that the topology of a mesh cannot be modified, we introduce a different concept – *stitching* of multiple meshes. Again we try to keep the concept very simple, so that it can be easily used to build complex meshes out of simpler constituting part meshes: we allow that single faces of a mesh can be set to be equivalent with single faces of other meshes (see figure 7). As we assume, that only few of the faces of a mesh are stitched with other meshes, we maintain this information as a sparse attribute of the mesh faces.

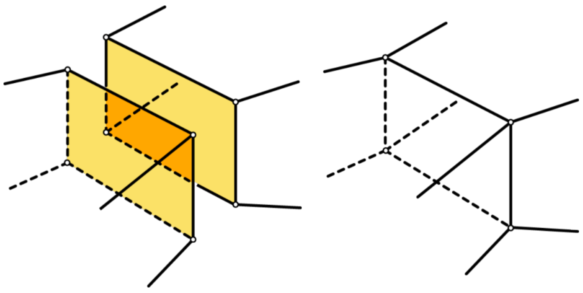

Figure 7: Stitching two meshes by setting faces of different meshes to be equivalent.

Topology API

One important aspect of our mesh data structure is an API to access topology across multiple stitched meshes, as if they were one single mesh. This API is implemented by creating simple handle objects for faces, vertices, and edges. These handle objects that are depicted in figure 8 contain a pointer to the mesh that contains the respective object, its index within that mesh, and its orientation.

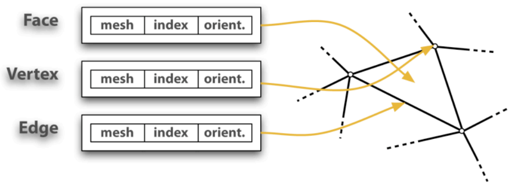

Figure 8: Handle objects are general references to faces, vertices, and edges.

The API of the handle objects allows access to all connected geometric objects of a given object, and returns handles to these connected objects. These handle objects are normally allocated on the stack, and filled with the reference to the object upon creation.

- **face handle objects**: allow access to all vertices and edges of a face, as well as all neighbouring faces
- **vertex handle objects**: allow access to all faces and edges meeting in a vertex, as well as to all neighbouring vertices
- **edge handle objects**: allow access to the two end vertices and the two coinciding faces of an edge

6. RESULTS

Memory consumption

In order to analyze the memory consumption of our mesh data structure we computed the memory requirements for the geometric and topological information of an infinite regular triangle mesh as well as an infinite regular quad mesh. We assume that 32-bit IEEE floats are used for the vertex coordinates and normals, and that for each vertex the position, the normal and a 32-bit colour is stored. Table 1 shows the resulting memory consumption per triangle/quad.

	geometric information	**topological information**
infinite triangle mesh	26 bytes	37 bytes
infinite quad mesh	44 bytes	52 bytes

Table 1. Per triangle/quad memory consumption for infinite triangle/quad meshes.

Example meshes

We used our data structure for subdivision and rendering of a few example meshes as seen in figure 9 and 10.

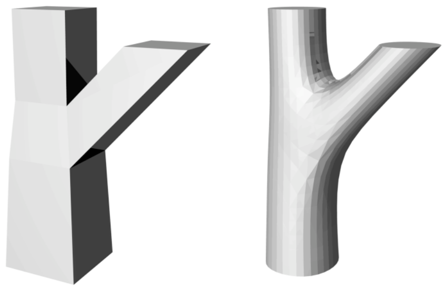

Figure 9: The input and subdivided mesh of a branching structure.

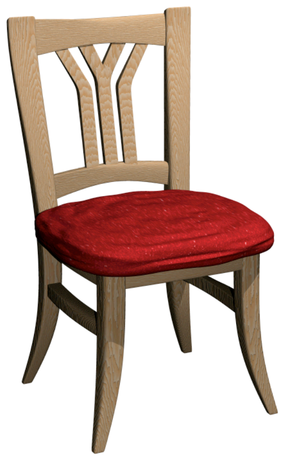

Figure 10: A chair built from multiple sheets created using Catmull-Clark subdivision [Cat78].

7. APPLICATION OF THE NEW MESH DATA STRUCTURE

The mesh data structure we presented is suitable for a number of applications, however our main application is real-time rendering of large vegetation scenes. One important aspect of this application is, that a typical scene contains more geometry than can be explicitly represented in memory.

Levels of detail

In order to overcome this problem, a level of detail approach is employed in our application. Due to the nature of current graphics hardware, the so called *chunked level-of-detail* approach has been chosen, i.e. the granularity of the geometry for which a level-of-detail is chosen, is rather coarse. Thus we represent a plant or a tree as a collection of meshes for branches, twigs, leaves and trunk parts, that are stitched to represent a single large mesh. For each part-mesh, the topology API is used to generate smoother, more detailed variants of the geometry using subdivision. In the rendering application, the currently optimal mesh is sent to the graphics hardware.

Generating complex geometry

For generating the vegetation models in our application, the mesh data structure is highly useful due to its efficient use of memory. We used the mesh-based L-system [Tob02a], [Tob02b] that employs generalized subdivision in order to introduce detail at finer subdivision levels. Some results of our implementation can be seen in figures 11 and 12.

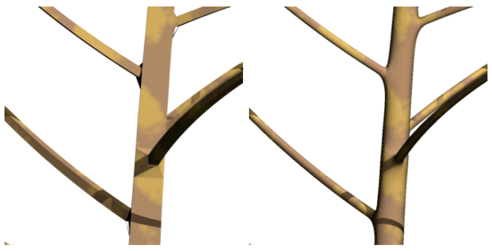

Figure 11: A grown structure consisting of stitched meshes, and a smoothed result using subdivision.

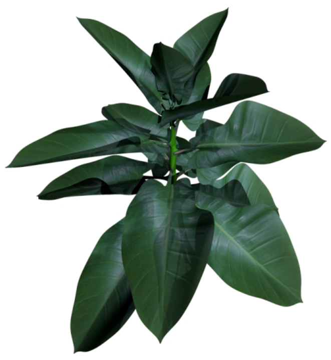

Figure 12: A complete rubber tree built using stitched meshes and subdivision.

8. CONCLUSION AND FUTURE WORK

We presented a new mesh data structure that has been optimized for both real-time rendering and subdivision. Due to the widely differing requirements of these two applications the new data structure is sufficiently general for a wide variety of algorithms.

For our prototype implementation a number of simplifications in our implementation have been made, and the only subdivision scheme we have implemented so far is the one by Catmull-Clark. In the future we expect to generalize our implementation and test our mesh data structure by implementing additional subdivision schemes, as well as other mesh based algorithms such as quadric-based surface simplification [Hec99].

9. REFERENCES

[Bau72] Baumgart B. Winged edge polyhedron representation. Artificial Intelligence Project Memo AIM-179 (CS-TR-74-320), Stanford University 1972.

[Bot02] Botsch M., Steinberg S., Bischoff S., Kobbelt L. Openmesh – a generic and efficient polygon mesh data structure. OpenSG Symposium, 2002.

[Cam98] Campagna S., Kobbelt L., Seidel H.-P., Directed edges – A scalable representation for triangle meshes. Journal of Graphics Tools, JGT 3,4, 1-12, 1998..

[Cat78] Catmull E., Clark J., Recursively generated B-spline surfaces on arbitrary topological meshes. Computer Aided Design 10, pp. 350-355, Sept. 1978.

[Cav91] Cavaretta A., Dahmen W., Micchelli C., Subdivision for Computer Aided Geometric Design. Memoirs Amer. Math. Soc. 93, 1991.

[DeR98] DeRose T., Kass M., Truong T., Subdivision surfaces in character animation. Computer Graphics 32, Annual Conference Series, pp. 85-94, Aug. 1998.

[Doo78] Doo D., Sabin M., Behaviour of recursive division surfaces near extraordinary points. Computer-Aided Desin 10,pp.g 356-360, Sept. 1978.

[Hec99] Heckbert P.S., Garland M., Optimal triangulation and qadric-based surface simplification. Computational Geometry 14, pp. 49-65, 1999.

[Hop98] Hoppe, H., Efficient implementation of progressive meshes, Computers & Graphics, Elsevier, Vol. 22, No. 1., pp. 27-36, Jan-Feb 1998.

[Ket98] Kettner L., Using generic programming for designing a data structure for polyhedral surfaces. 14th Annual ACM Symp. On Computational Geometry, 1998.

[Kob00] Kobbelt L., Sqrt(3) Subdivision. In SIGGRAPH 2000, Computer Graphics Proceedings, Akeley K. (Ed.), Annual Conference Series, ACMSIGGRAPH, pp. 103-112, 2000.

[Tob02a] Tobler R. F., Maierhofer S., Wilkie A. Mesh-based parametrized L-Systems and generalized subdivision for generating complex geometry. Journal on Shape Modelling 8, 2, pp. 173-191, Dec. 2002.

[Tob02b] Tobler R.F., Maierhofer S., Wilkie A., A multiresolution mesh generation approach for procedural definition of complex geometry. Shape Modeling International, Banff, Canada, pp. 35-42, 2002.