Efficient Representation of Triangle Meshes for Simultaneous Modification and Rendering

Horst Birthelmer, Ingo Soetebier, and Jörg Sahm

Fraunhofer IGD, Darmstadt, Germany
{horst.birthelmer, ingo.soetebier, joerg.sahm}@igd.fhg.de

Abstract. This paper introduces an efficient representation of triangulated meshes for the simultaneous modification and rendering in realtime. The modification includes the simplification and the refinement of the triangle meshes as well as the animation of their vertices. Since the vertices and the topology of the meshes are stored completely independent of each other, modifications can be applied very fast. Because the presented approach is not only applicable to modification, but also to rendering, the triangle meshes have not to be converted into application specific representations such as simplification or visualization.

1 Introduction

Realtime rendering of complex 3D scenes has become very common on todays graphics workstations. There are techniques like 3D scanners or surface approximation, that can easily create huge amounts of polygonal triangle meshes. However, even modern workstations need an efficient representation of the 3D scene to govern this increasing amount of data.

Because of their widespreaded hardware support, data representations such as indexed face sets, indexed triangle sets, or triangle strips are often used in this context. The most popular implementation, including hardware support, is the OpenGL[17] graphics library developed by SGI, which is available on all major platforms.

In this paper we introduce a new representation of triangulated meshes, which provides the fast modification of the geometry and the topology as well as the realtime rendering of the mesh. Although the implementation is OpenGL based, the approach is portable to any graphics library supporting indexed triangle sets.

2 Previous Work

Many work has been done in the area of efficient representations of triangle meshes. There are two basic approaches. The *edge-based* and the *face-based* representations.

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2657, pp. 925–934, 2003.

2.1 Edge-Based Representation

Lot of work is based on the idea to represent a mesh by representing its edges. Most edge-based representations use a sort of half-edge structure. "Winged-Edge" [1] is one of the oldest, most used and therefore well tested representations. More recent versions are those of Weiler [16] and Campagna et al. [4].

McMains et al [12] published a representation which is designed for handling very large meshes in a out-of-core manner. It offers the posibility to access geometry and topology information stored on external memory like harddisks.

The well known Library CGAL[3] for computational geometry also uses an edge-based representation for polyhedral surfaces as does Kettners data structure for polyhedral surfaces [11].

Botsch et al. recently published OpenMesh[2], a representation for polygon meshes integrated in OpenSG[10]. The representation is also edge based and very similar to that of CGAL but has some enhancements in the design.

Edge-based representations are very efficient and flexible assuming non-manifold meshes. This representation is easy to use in algorithms which presume this property. However, not all meshes have that property.

2.2 Face-Based Representations

Face-based representations are often used in CAD systems because of their flexibility. This representation stores the faces of the mesh by storing pointers to their vertices. The representation is independent of the polyhedron used for it. Triangles, quadrangles or any other polyhedron can be used. Some work on faced-based representation was presented by Muuss[13] et al.

Hoppe[8] published an efficient data structure for progressive meshes which stores wedges along with material information for every face.

3 Requirements

Although representations such as display lists in OpenGL can be rendered very fast, these data structures lack of flexibility in concern to modification. So the complete display list has to be rebuilt, if a single vertex is changed. On the other side, there are representations like half-edge structures, providing easy access to all information of the 3D model. Unfortunately these representations are not suitable for rendering. They have to be converted, which is very time consuming, especially if done several times per second in the area of animation and real-time rendering. In our application[14] we are working with large, dynamic 3D scenes. So we need a respresentation, which has to fullfill the following requirements:

- Access to topological information as well as geometrical information is needed. For example, this is necessary for calculating an error metric for mesh simplification.
- The geometry as well as the topological information should be modifiable.

- Fast insertion of triangles from a progressive structure or refinement guided by some subdivision scheme should be possible.
- The representation should allow rendering during all the steps of modification of the mesh, like simplification or refinement.
- Conversion should not be necessary between the different applications of the mesh representation.

4 Vertex Arrays and Indexed Triangle Sets

Because of the requirements listed in section 3, the client states and the respective property arrays of OpenGL offer a more promising solution in comparison to display lists. While the first memory block contains the complete geometrical information, the second memory block saves the topological information. For that reason, the client state concept of OpenGL provides a strict separation of the mesh's geometric and topological information. Additionally, OpenGL supports the storage of almost any further information per vertex either in information specific arrays or in interleaved arrays. These arrays are used in order to avoid data redundancy, what is highly recommended, if the modification of the properties is a critical demand of the representation.

The geometric mapping of vertices into 3D space is stored in the vertex array and can be modified from frame to frame. Unfortunatly, this structure can not deal with topological changes such as the insertion of new vertices into an existing mesh or the manipulation of triangles in the neighbourhood of a specific vertex, as it often occurs in simplification. For that reason, the representation has to be extended in order to fulfill the requirements.

5 Additional Information for Fast Modification

The presented approach differentiates between the topology representation and the representation of the properties, such as color, geometry, or texture information. As a matter of principle even data calculated by a simulation software can be inserted into the representation. As shown in figure2 the OpenGL vertex array is part of the structure, but extended by some application specific information. A very important aspect of the structure is, to keep all data necessary for fast rendering inside of one memory block in an OpenGL "friendly" manner, while the other information is allocated somewhere else. For that reason, only a pointer to this memory block has to be passed to OpenGL for rendering.

The more interesting part is the additional data from the structure shown in figure 2. The vertex pointer array is the counterpart to the vertex array of OpenGL. It does not save the vertex information but pointers to the vertex data. But in the opposite to the OpenGL array, the vertex pointer array may contain holes. As another difference to OpenGL, the entries inside of the vertex pointer array are allowed to be NULL pointers. That means, the vertex information is not saved as a block of memory, but as a block of pointers to some vertex

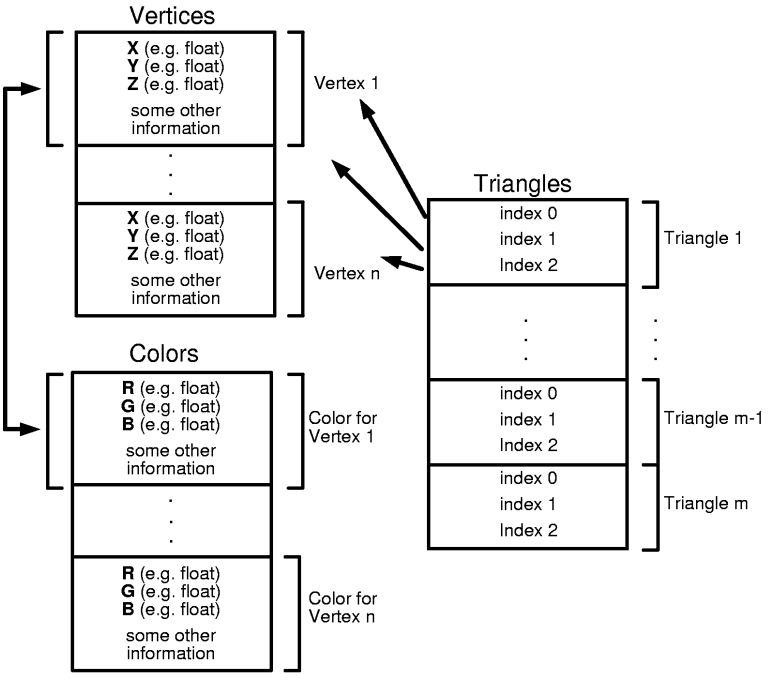

Fig. 1. An indexed triangle set with color and geometric properties as it is implemented in OpenGL

information allocated inside of the heap. So it is possible to simplify the mesh by deleting vertices.

The update process of a delete operation requires some further information. Each pointer inside of the vertex pointer array references one vertex specific structure, which consists of the vertex index in the OpenGL array and an occurrence list. Typically, the vertices addressed by the pointer array won't be in the same order as the ones in the OpenGL array. The advantage of saving the vertex array *index* in this vertex structure is quite obvious. The geometrical position of the vertex in 3D-space can be modified as well as the index of the vertex in the OpenGL array without any update of other data. A further advantage of the vertex pointer array is, that each entry always addresses the same vertex structure for the lifetime of the representation. That means, the index of an entry in the vertex pointer array never changes. For that reason, these indices can be saved in the additional information field inside of the triangle structure (see figure4).

The occurence list inside of the vertex structure is the only dynamic container in the complete representation. So it is the only data structure, which changes its size, memory consumption and number of members according to the manipulations of the mesh. It can be implemented as a list or in a more efficient

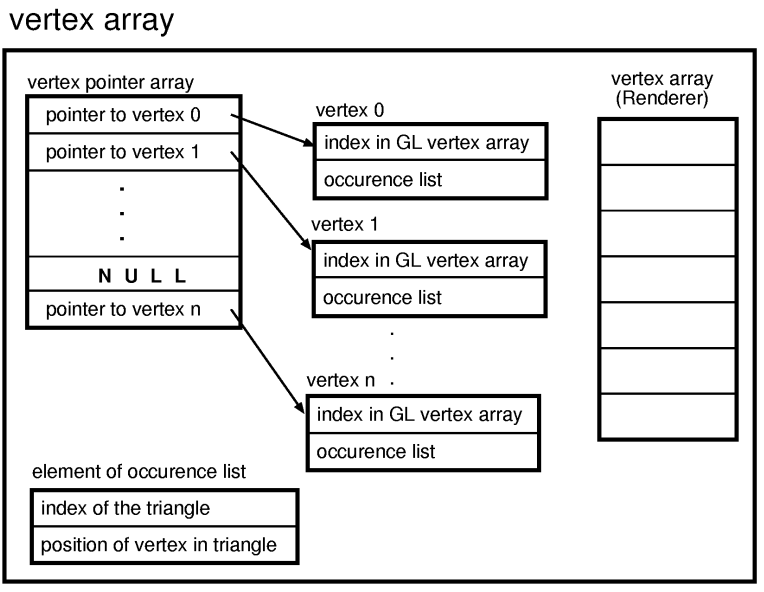

Fig. 2. The representation of the properties (in this example just the geometric information)

manner, such as a binary tree. Figure 2 illustrates the structure of an entry inside of the occurence list. This structure does not only contain the triangle, in which the vertex occurs, but also the position of the vertex in that triangle. The position varies between 0 and 2 and represents the index of this vertex in the representation of the triangle (see figure 3). It is *not* the index of the triangle in the OpenGL triangle array.

Since the triangle pointer array is treated in the same way as the vertex pointer array, these arrays are consistent as well as the information stored for OpenGL. The indices in the OpenGL representation of a triangle are indices from the OpenGL vertex array, so the mesh can be rendered without any knowledge of the additional information. The additional information stores just indices and links to the additional information of the other part. Figure 3 shows this for the topological information.
Summarizing this information, we get a representation which is fast for rendering and for modifying the mesh using simplification, subdivision, animation, etc. (Figure 4 shows an overview).

6 Changing the Mesh

Changing just the geometry of existing vertices is trivial, however some interesting animations can be done with that. The more challenging part is the changing

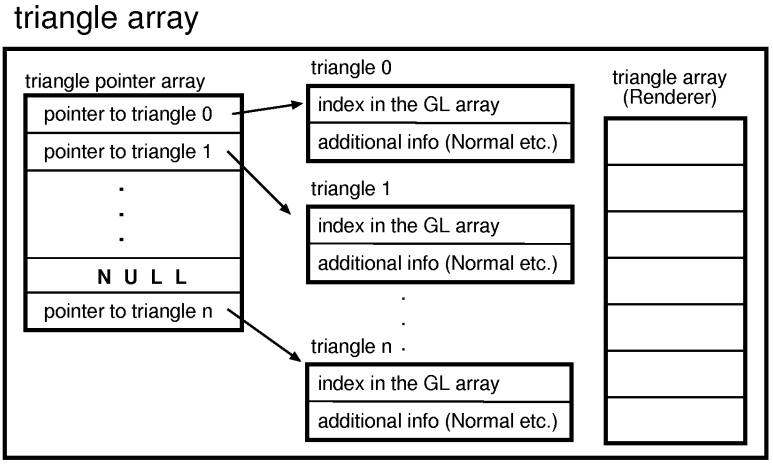

Fig. 3. The representation of the topology

of the geometry and the topology at the same time, while keeping the structure consistent and renderable.

Most metrics for error determination in simplification use topological information and geometrical information of the adjacent polygons of a vertex. Since we know the occurences of the vertex and the position of that vertex in the triangle, we can determine all the edges in the neighborhood and all the vertices, to which the specified vertex is connected. We simply iterate over the occurence list and increment the position by one. Always calculating modulo 3 we get the position of the next vertex in positive direction of the triangle, since the vertices are saved in positive order refering to the triangles normal. Garland et al. proposed the Quadric Error Metric[5] which uses besides edge information the plane equation of the adjacent triangles. This information can be extracted very easily from the structure as well. Either the information is stored directly in the additional structure of the triangles or is computed on the fly (which of course takes more time).

Assuming we want to remove a vertex by edge collapsing. Dereferencing the pointer behind that index in the vertex pointer array will give us the index in the OpenGL array and the occurence list. This list contains every occurence of this vertex in the triangle pointer array and the position, at which this vertex is situated in every triangle. Knowing the target, to which this vertex will be collapsed, we have to change the entries in the OpenGL triangle array to the target's OpenGL index and add the modified triangles to the occurence list of the target. Some triangles will be deleted during this operation. This triangles have to be moved to the end of the OpenGL triangle array and the pointers to this triangles have to be set to NULL. Moving it to the end of the OpenGL array is necessary, because we cannot have holes in the OpenGL vertex or triangle

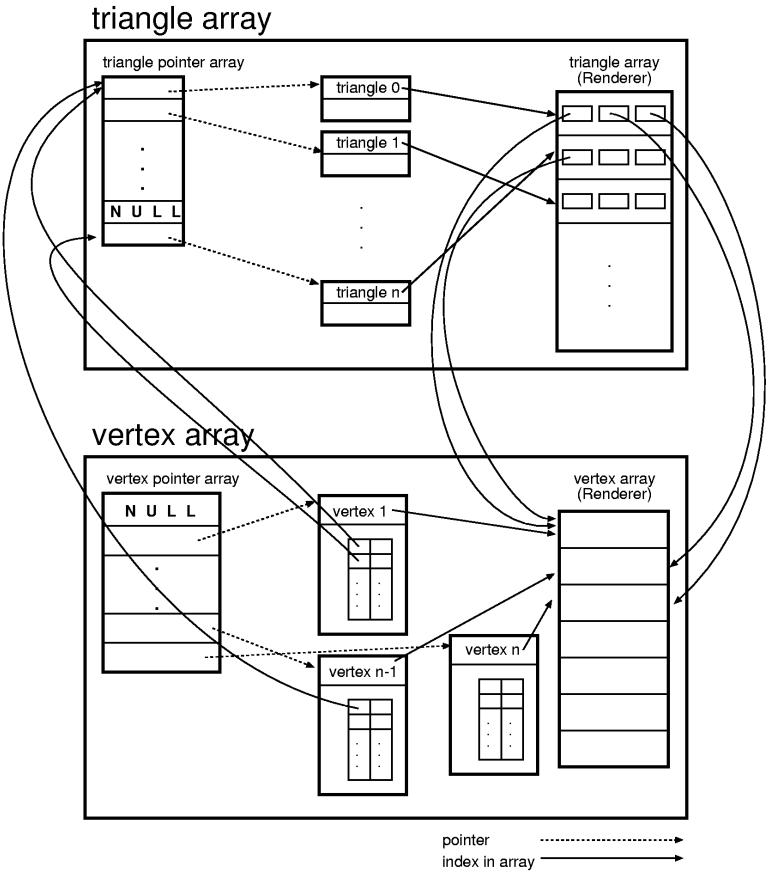

Fig. 4. An overview of the representation

arrays. Since it does not matter, in which order triangles are rendered the change in the order of the triangles will not matter during rendering. After moving these triangles to the end, the number of triangles in the array is changed and these triangles will not be rendered any more. The index in the triangle pointer array will not change so that all the references from the occurence lists of the other vertices in the mesh will remain valid.

The vertex pointer array is treated the same way. The vertex to be removed is moved to the end in the OpenGL vertex array and then removed either by doing a real realloc or by just avoiding to refere to it any more in the triangle array. The pointer to the removed vertices and triangles in the vertex and triangle pointer arrays are set to NULL and the memory can be freed. Any other changes to the mesh, e.g. geometrical or topological changes, can be treated the same way.

Fig. 5. Scenes taken from the visualization of the german federal state of Hessen presented at InterGeo in Frankfurt/Main, Germany

7 Implementation

We implemented the vertex array (see figure 2) and triangle array (see figure 3), as shown above as a C++ template library. To achieve the maximum flexibility we used the OpenGL representation of a vertex respective triangle as well as the additional information as template parameters. This gives us the ability to add information anywhere in any representation as we need it. The class representing the mesh is also template driven and uses the types of the vertex array and the type of the triangle array as template parameters. The algorithms for simplification and refinement etc. can therefor be implemented similar to the algorithms from the STL [15]. The algorithm does not contain any data types. They are given by template parameters. The algorithm just defines the way to process the data.

Our simplification, as well as the refinement, are done in separate threads. It only has to be synchronized during access to the OpenGL array entries and while the OpenGL indices of the triangle arrays are changed. In realtime rendering memory consumption and rendering speed are always an issue. The memory footprint of this representation is also very small since no information is stored redundant. However, the memory footprint depends heavily on the topology of the represented mesh.

8 Results and Conclusions

We developed and implemented a data structure for representing large triangle meshes with various additional properties, like color or metric information. This data structure is suitable for rendering as well as modifying the stored data without conversion between these two applications. We used this data structure

in implement two demonstrators. The first demonstrator simply loads a 3D scene in a progressive file format. It inserts triangles while the scene is rendered. On a standard PC with 1.3 GHz CPU and 512 MB main memory, about 100000 triangles can be inserted per second while the scene is rendered. The second applications visualizes geographical data of the complete german federal state of Hessen. The geometry consists of a height field using a mesh aperture of 40 meters, satelite pictures as texture and the buildings of the city of Darmstadt as separate meshes. For the buildings levels of details are used performing mesh modification every frame.

There are mesh representation which are also very flexible for the user. The OpenMesh Library[2] is an example for a flexible mesh representation. It provides random access to vertices, edges and faces. Contrary to our data structure, the internal half-edge data structure of OpenMesh can not be rendered directly. Our data structure can be passed directly to a renderer for visualization. There are other mesh representations, like Normal Meshes[6] presented by Guskov et al, which implement aspects of multiresolution meshes, or Directed Edges[4] presented by Campagna et al, which is also a half-edge data structure. But those representation can also not be used directly for rendering. Display lists in OpenGL are data structures which can be rendered very fast, because they are stored in the memory of the graphics hardware. The disadvantage of this representation is, that they have to be rebuilt every time something has changed. Our data structure combines both properties. All properties of the mesh can be accessed and the data structure can be directly passed to a renderer.

References

1. Baumgart, B. "A Polyhedron Representation For Computer Vision", *National Computer Conference*, pp. 589–596, 1975.
2. Botsch, M., Steinberg, S., Bischoff, S., Kobbelt, L. "OpenMesh – a Generic and Efficient Polygon Mesh Data Structure", *OpenSG Symposium 2002*, 2002.
3. Brönniman, H., Fabri, A., Schirra, S., Veltkamp, R., Editors, "CGAL Reference Manual Part 2: Basic Library", CGAL R1.0, 1998. `http://www.cs.ruu.nl/CGAL`
4. Campagna, S., Kobbelt, L., Seidel, H.-P. "Directed Edges - A Scalable Representation for Triangle Meshes", *Journal of Graphics Tools*, 3(4), pp. 1–12, 1998.
5. Garland, M., Heckbert, P. "Surface simplification using quadric error metrics", *Proceedings of SIGGRAPH '97*,pp. 209–216, 1997.
6. Guskov, I., Vidimce, K., Sweldens, W., Schröder, P. "Normal Meshes", *Proceedings of SIGGRAPH 2000*, pp. 95–102, 2000.
7. Heckbert, P., Garland, M. "Survey of polygonal surface simplification algorithms", *Tech. Rep. CMU-CS-95-194, Carnegie Mellon University*, 1995.
8. Hoppe, H. "Efficient implementation of progressive meshes", *Computers & Graphics*, 22(1), pp. 27–36, 1998.
9. Hoppe, H. "Progressive meshes", *Proceedings of SIGGRAPH '96* , pp. 99–108, 1996.
10. OpenSG, `http://www.opensg.org`
11. Kettner, L. "Designing a Data Structure for Polyhedral Surfaces", *Fourteenth ACM Symp. on Computational Geometry*, pp. 146–154, Minneapolis, Minnesota, 1998.

12. McMains, S., Hellerstein, J. M., Séquin, C. H. "Out-of-Core Build of a Topological Data Structure from Polygon Soup", *6th ACM Symposium on Solid Modeling and Applications*, pp. 171–182, 2001.
13. Muuss, M. J., Butler, L. A. "Combinatorial Solid Geometry,Boundary Representations and Non-Manifold Geometry", *State of the Art in Computer Graphics: Visualization and Modeling D. F. Rogers, R. A. Earnshaw editors*, Springer-Verlag, pp. 185–223, New York, 1991.
14. Sahm, J., Birthelmer, H., Soetebier, I. "A Client-Server Architecture for the Cooperative Visualization of Large, Interactive and Dynamic 3D Scenes", *Proceedings of VisSim 2003*, Magdeburg, 2003.
15. Standard Template Library, Documentation, `http://www.sgi.com/tech/stl/`
16. Weiler, K. "Edge-Based Data Strucures for Solid Modeling in Curved-Surface Environments", *IEEE Computer Graphics and Application*, 5(1), pp. 21–40, 1985.
17. Woo, M., Neider, J., Davis, T., Shreiner, D., OpenGL Architecture Review Board "The OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL", Version 1.2, Addison-Wesley Pub Co; ISBN: 0201604582; 3rd edition, 1999.