Directed Edges — A Scalable Representation for Triangle Meshes

Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel

University of Erlangen, Computer Graphics Group
Am Weichselgarten 9, D-91058 Erlangen, Germany
Email: {campagna,kobbelt,seidel}@informatik.uni-erlangen.de

Abstract

In a broad range of computer graphics applications the representation of geometric shape is based on triangle meshes. General purpose data structures for polygonal meshes typically provide fast access to geometric objects (e.g. points) and topologic entities (e.g. neighborhood relation) but the memory requirements are rather high due to the many special configurations. In this paper we present a new data structure which is specifically designed for triangle meshes. The data structure enables to trade memory for access time by either storing internal references explicitly or by locally reconstructing them on demand. The trade-off can be hidden from the programmer by an object-oriented API and automatically adapts to the available hardware resources or the complexity of the mesh (scalability).

1 Introduction

Many algorithms use triangle meshes for representing geometric surfaces. The simple triangle can serve as a basic surface primitive to adaptively approximate smooth freeform geometry. A triangle mesh naturally allows to adapt the required number of primitives to the geometric shape instead of the basic topology of an arbitrary object. Triangle meshes are efficient, since the triangle is currently the only surface patch that is directly supported by computer graphics hardware and even software rendering methods like ray tracing or radiosity prefer to use piecewise linear primitives to represent geometric objects due to gains in performance [KS97, SP94].

A triangle is uniquely defined by the geometric position of its three vertices. Thus, assuming 4 bytes float values, each 3d-triangle requires $3 \cdot 3 \cdot 4 = 36$ bytes. This way of storing *individual triangles* (cf. Fig. 1) is sufficient for several applications, e.g., when each triangle is processed separately.

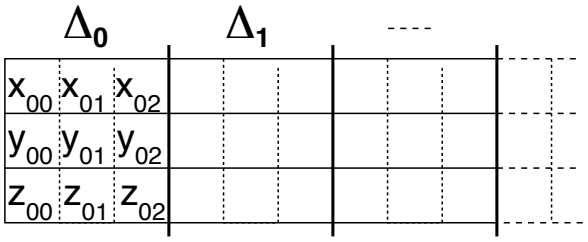

Figure 1: Storing individual triangles.

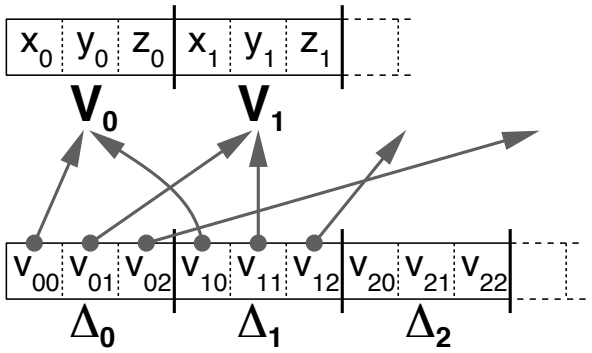

Figure 2: Shared vertex data structure.

Algorithms that perform operations on a triangle mesh, e.g., mesh editing systems [KCVS98, ZSS97] or mesh decimation algorithms [Ros97, PS97], need more information beyond the mere shape of each separate triangle. If the geometric position of one vertex is modified, this operation affects not only the shape of a single triangle that contains this vertex. Instead, all triangles containing a vertex at the same geometric position should be modified. This can be achieved by the concept of *shared vertices*: all triangles sharing a common vertex store (integer) pointers to the same physical instance. Thus, a vertex at a certain geometric position needs to be specified only once and it may be shared by an arbitrary number of triangles.

Euler's formula $V - E + F = 2$ [FvDFJ92] tells us that a manifold triangle mesh ($3F \approx 2E$) with n vertices contains $m \approx 2n$ triangles. Therefore, the shared vertex representation of a triangle mesh requires $3 \cdot 4 \cdot n = 12n$ bytes for the geometry (the geometric position of the vertices) and $3 \cdot 4 \cdot m = 12m$ bytes for the topology (the triangles referencing three vertices each), assuming 4 bytes for each reference (cf. Fig. 2). This results in $12n + 12m \approx 18m$ bytes for a triangle mesh consisting of m triangles and saves about 50 percent compared to individual triangles. This amount of memory can be considered as a lower limit for storing triangle meshes with immediate and random access to each triangle.

In principle, every information required by an algorithm may be determined from the shared vertex representation. Neighboring triangles sharing a common edge reference two identical vertices that span this edge. Explicitly storing this neighborhood relation requires additional $3 \cdot 4 \cdot m = 12m$ bytes and further $4 \cdot n \approx 2m$ bytes for referencing one triangle attached to each vertex. The set of triangles $t_i = (v_{i0}, v_{i1}, v_{i2})$ referencing a certain vertex v describe the topological neighborhood of v: neighboring vertices v_j, adjacent edges $\overline{v\,v_j}$, triangles t_i, orbit-edges $\overline{v_a\,v_b} \subset \{t_i\}$ with $v_a, v_b \neq v$, and so on. Unfortunately, access to such and similar information requires a global search on the standard shared vertex data structure, if the neighborhood information is not stored explicitly.

Several data structures have been developed to provide fast access (ideally $\mathcal{O}(1)$) to the information that is required by different algorithms. The most popular ones are the *winged-edge* [Bau72], *quad-edge* [GS85], and *half-edge* [Mae88] data structures. Fig. 3 shows the winged-edges according to O'Rourke [O'R95]. Each vertex stores its geometric position and

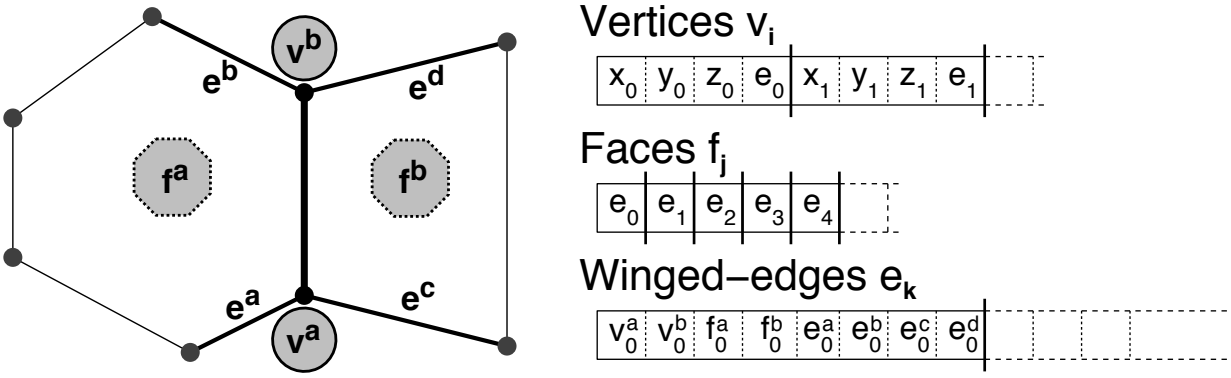

Figure 3: Winged-edge data structure.

a reference to one of its edges, while each polygonal face only points to one of its enclosing edges. An edge stores references to the two vertices v^a and v^b spanning that edge, to the two neighboring faces f^a and f^b, and to the four "wings" $e^a, \ldots, e^d$. This data structure requires $4 \cdot 4 \cdot n + 4 \cdot m + 8 \cdot 4 \cdot \frac{3}{2}m \approx 8m + 4m + 48m = 60m$ bytes for a triangle mesh with m triangles. This is about three times the memory compared to the shared vertex representation, but therefore the neighborhood and adjacency can be looked up directly. For algorithms processing on a mesh data structure this means that we trade memory for performance (even compared to the shared vertex representation with neighborhood information that still requires some search operations).

We are not going to discuss the huge variety of mesh data structures in detail, nor will we examine theoretical aspects like duality, completeness, and so on. Instead, we focus on practical requirements and present a new *scalable* data structure for triangle meshes.

2 Practical Requirements

In practice, many objects represented by triangle meshes contain isolated vertices or edges that are *locally non-manifold* for several reasons [CKS98]. On the other hand, many data structures are restricted to orientable 2-manifold triangle meshes. Even if there are only some few artifacts that are locally non-manifold within a large data set, these data structures are not able to represent such an object. Other data structures which are capable of storing non-manifold meshes typically require more memory due to the fact that they have to handle non-manifolds all over the whole object by using extra memory, even if most entities are in fact 2-manifold. A compromise between the two approaches can be found by using the data structure that we discuss in the next section.

As previously mentioned, several sophisticated data structures have been developed, most of them capable to represent general polygonal meshes instead of just the special instance of a triangle mesh. *Specializing* to triangles obviously allows to design more efficient data structures.

Of course, the more explicit information is stored, the more memory is required, but

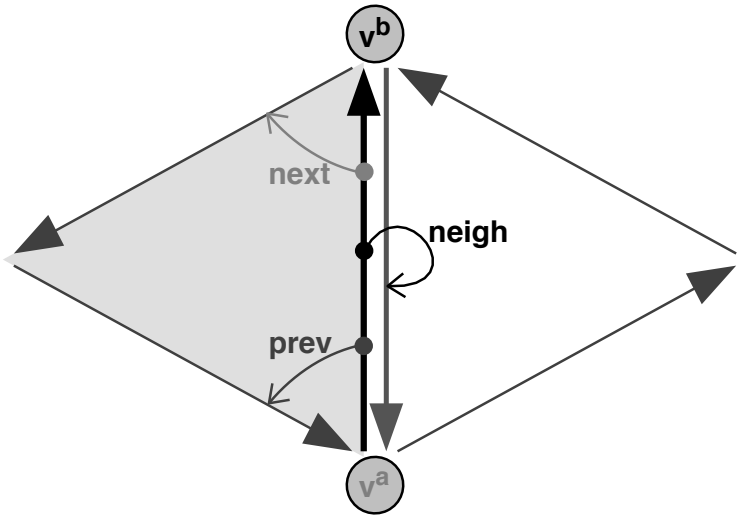

Figure 4: The data structure records two vertices v^a, v^b, the neighboring edge e^{ng} and the previous and next edges e^{pv} and e^{nx} for each directed edge. The memory for some values (gray) can be exchanged by a constant amount of additional calculations.

the faster is the access to the required data. Unfortunately, memory is a *hard limit* in computer systems. Even virtual memory does not help in most cases, since the mapping of a complex triangle mesh to the linear structure of the memory usually opposes the need for local coherence access to memory. Drastic slowdown of the performance due to extensive swapping is the result.

Performance is a very important aspect, but time is in general a *soft limit*. When doubling the amount of input data to be processed, it is acceptable to wait even more than twice the time to get a result, if this is necessary. But if doubling the amount of data implies that the data structure does not fit into the memory of a computer, this makes it impossible to work on a data set.

As a result, the implementation of a data structure can be considered as a compromise between memory requirements and performance gains. We propose a *scalable data structure*, where additional memory can be utilized to speed up the access and hence the performance as long as enough resources are available. Once the hard memory limit is reached, performance can be *traded* for memory in order to maximize the complexity of meshes which can be processed on a given system. We will present a new data structure for triangle meshes that has been motivated by this idea in the following section.

3 Directed-Edge Data Structure

We are first going to describe the *directed-edge data structure* for the case of oriented 2-manifolds. Afterwards we will present our extension to handle non-manifold triangle meshes.

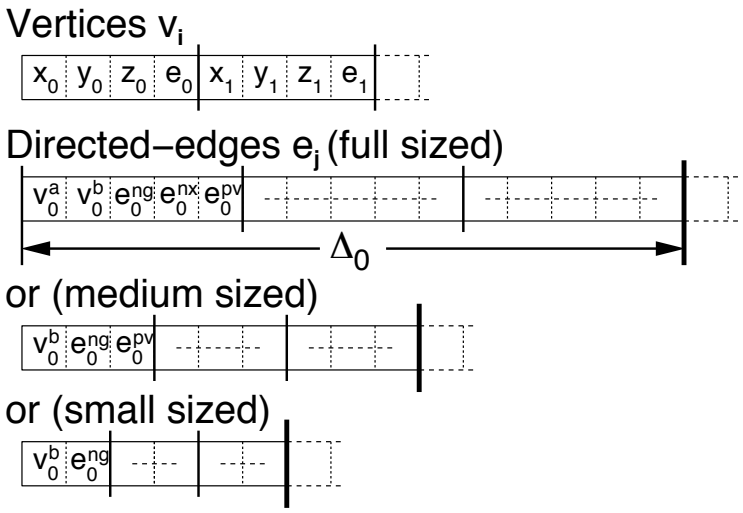

Figure 5: Three variants with different memory requirements of the directed-edge data structure for a triangle mesh.

A single triangle is represented by three directed edges (cf. Fig. 4). Thus, the common edge $\overline{v^a\, v^b}$ of two neighboring triangles corresponds to two directed edges $v^a \rightarrow v^b$ and $v^b \rightarrow v^a$. This is very similar to the concept of half-edges [Mae88]. For each directed edge the following information is stored: a starting vertex v^a, a target vertex v^b, the previous, next, and neighboring edges e^{pv}, e^{nx}, and e^{ng}. We store all directed edge data structures in an array, such that the i'th triangle is represented by the directed edges at the entries $3i$, $3i+1$, and $3i+2$. Thus, we do not have to store explicit references from each triangle to an edge and vice versa. Instead, we derive these references from the given algorithmic context.

Storing a reference to a single directed-edge for each vertex provides fast access to the local neighborhood of such an entity. As described so far, a total amount of $4 \cdot 4 \cdot n + 5 \cdot 4 \cdot 3 \cdot m \approx 8m + 60m = 68m$ bytes is required for a triangle mesh with n vertices and $m \approx 2n$ triangles (cf. Fig. 5). Not explicitly storing the references between triangles and edges saves $4 \cdot m + 4 \cdot 3 \cdot m = 16m$ bytes.

Since we restricted to triangle meshes, we can further substitute memory by computation time. Access to v^a of an edge e can be replaced by v^b of the previous edge of e. The inquiry to the next edge of e can be answered with the previous edge of the previous edge of e (cf. Fig. 4). Thus, we can substitute the memory for the entries v^a and e^{nx} by some few additional computations. Please note that these additional calculations are of constant time and do not require a search (as would be necessary for general polygons). Of course, we could replace v^b and e^{pv} by a similar strategy instead. But these entries are required more frequently and thus should be available directly. The simplified structure requires $4 \cdot 4 \cdot n + 3 \cdot 4 \cdot 3 \cdot m \approx 8m + 36m = 44m$ bytes of memory.

So far, we considered the entries v^a, v^b, e^{pv}, e^{nx}, and e^{ng} as general references that could be implemented either as pointers or as array-indices. If we decide to use only array-indices

as references we can save further memory and shift some load to the CPU instead. Given an edge e, where e is an array-index, then the previous and next edge can be determined by the following expressions (in C-style pseudocode):

```
prev( e ) = (e%3 == 0) ? e+2 : e-1;
next( e ) = (e%3 == 2) ? e-2 : e+1;
```

Thus, the data structure for a directed edge needs to explicitly store values only for v^b and e^{ng}. This results in a total memory requirement of $4 \cdot 4 \cdot n + 2 \cdot 4 \cdot 3 \cdot m \approx 8m + 24m = 32m$ bytes. This is optimal compared to the shared vertex representation with additional neighborhood information, but the new data structure is more convenient to access the required information from an algorithmic point of view. As an example, the following pseudo-code returns the indices `idx[0]`, ..., `idx[n-1]` of the `n` neighboring vertices for the `v`'th vertex:

```
int get_neighbors( int v, int idx[] )
{
   int ee = vertex[v].e(); // each vertex points to one emanating edge
   idx[0] = edge[ee].v1();   // store the first vertex-index
   int n = 1;                // one index in the array

   int e = edge[ee].prev();  // the previous edge of ee ...
   int e = edge[e].neigh();  // ... and the respective neighbor

   while( e != ee ) {        // until all emanating edges visited
      idx[n] = edge[e].v1(); // store the current target vertex ...
      n++;                   // ... and update the count
      e = edge[e].prev();    // update the previous edge ...
      e = edge[e].neigh();   // ... and the respective neighbor
   }

   return n;                  // n indices have been found
}
```

So far, we considered only orientable 2-manifold triangle meshes. As discussed in the previous section, non-manifold vertices or edges should also be tolerated by a data structure when used in practical applications. Since we do not want to spend any (or only few) additional memory for this purpose and we assume that the number of non-manifold entities is typically neglectable compared to the complexity of the whole mesh, we use the following simple but effective strategy.

Using integer values as array-indices for the references to neighboring edges allows us to use the sign-bit without any additional implementation-effort. The value for the neighboring edge e^{ng} is zero or positive for every pair of directed edge mates that represent a 2-manifold interior edge. A directed edge on a topological boundary is marked by a certain

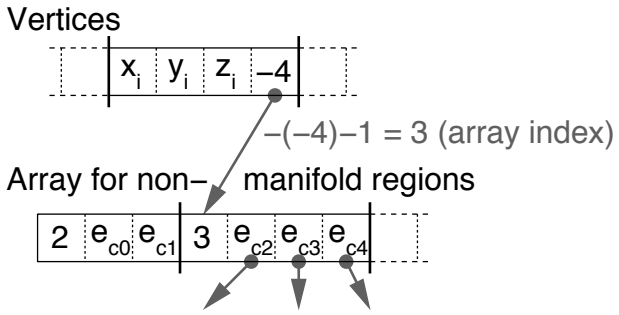

Figure 6: The entry -4 is used to determine the array-index $-(-4)-1=3$ of a separate array that lists the connected components attached to non-manifold vertices by referencing one edge of each component.

negative entry for e^{ng}, e.g. -1. A non-manifold edge or an edge whose two triangles are of opposite orientation may be marked by -2.

A 2-manifold vertex references one of its emanating edges. For non-manifold vertices the same strategy can be used as for edges. A negative array-index indicates such a node. Removing the negative sign provides a positive value that points to a separate array which lists either all edges emanating from that vertex or the connected components attached to that vertex. Fig. 6 shows an example for the second strategy. This simple trick saves memory (compared to linked lists for each vertex where only one list-entry is used for most entities) and is easy to implement.

4 Results

We used the presented data structure for our mesh decimation algorithm [KCS98]. Using object-oriented techniques allowed us to simply switch between the described strategies: a full, medium, and a small sized data structure. Currently, we may use only one of these three implementations at the same time within a single executable. By using further object-oriented techniques it is possible to provide all data structures at the same time and to decide at runtime, which one should be used, e.g., considering the memory requirements of a certain model.

Figure 7 shows the results when using the three different strategies for decimating different meshes (measured on a MIPS R10000 CPU at 250 MHz) when using our highly optimized mesh decimation algorithm (implementation optimizations are described in [CKS98]). The dashed line shows the results for the mesh that is shown in Figure 8. The other two lines show the results for decimating a mesh at two different target resolutions (it is a property of the mesh decimation algorithm that the average performance decreases with the total number of removed triangles). As expected, the smallest data structure provides the slowest performance since we exchanged speed for memory. Spending additional memory enhances the performance of the algorithm in this case. The performance-results do not differ in a broad range, since we exploit local caching of intermediate calculations within

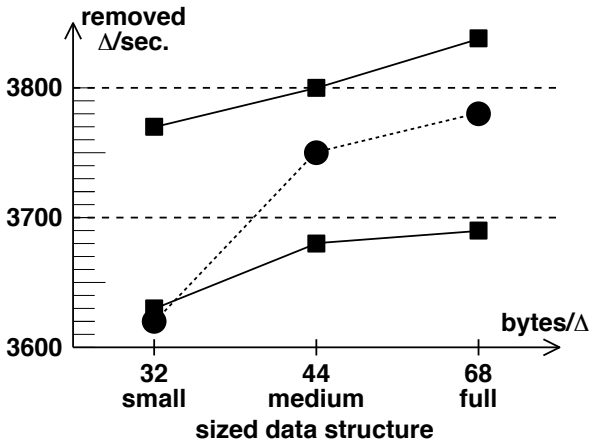

Figure 7: Using three different instances of the scalable data structure within a mesh decimation algorithm. The dashed line shows the result for the mesh shown in Figure 8, while the solid lines show the results for another mesh at two different resolutions.

the mesh decimation algorithm and thus reduce access requests to the underlying mesh data structure to a minimum.

There may be circumstances, where the above behavior need not be true, since lower memory may sometimes result in higher CPU-performance due to the architecture of modern computer systems (first and second level caches, main memory, swap memory). The number of additional calculations can be less than the time required for waiting for data that has to be passed through the memory hierarchy. This behavior depends on the specific application and the used computer architecture.

As shown in Figure 8, a mesh may also contain additional attributes, e.g. colors, surface normals, or texture coordinates [CB97]. Of course, such attributes may also be added to the directed-edge data structure at the cost of additional memory.

Finally, Table 1 shows a comparison of the mesh data structures considered in this article. Please note that the individual triangles and shared vertex representation do not provide any immediate neighborhood information which is crucial for many applications.

5 Conclusion

We presented a new scalable data structure for triangle meshes that allows to trade performance for memory, depending on the specific application or the size of the data that has to be processed. Implementation details can be hidden from the programmer by an object-oriented API. Nonetheless, implementation and use of this scalable data structure is very simple.

We have verified our design goals by using this new data structure as a core component of our mesh decimation algorithm. Especially memory efficient handling of meshes with only few local non-manifold entities has proven to be very important when using such a data structure in practice.

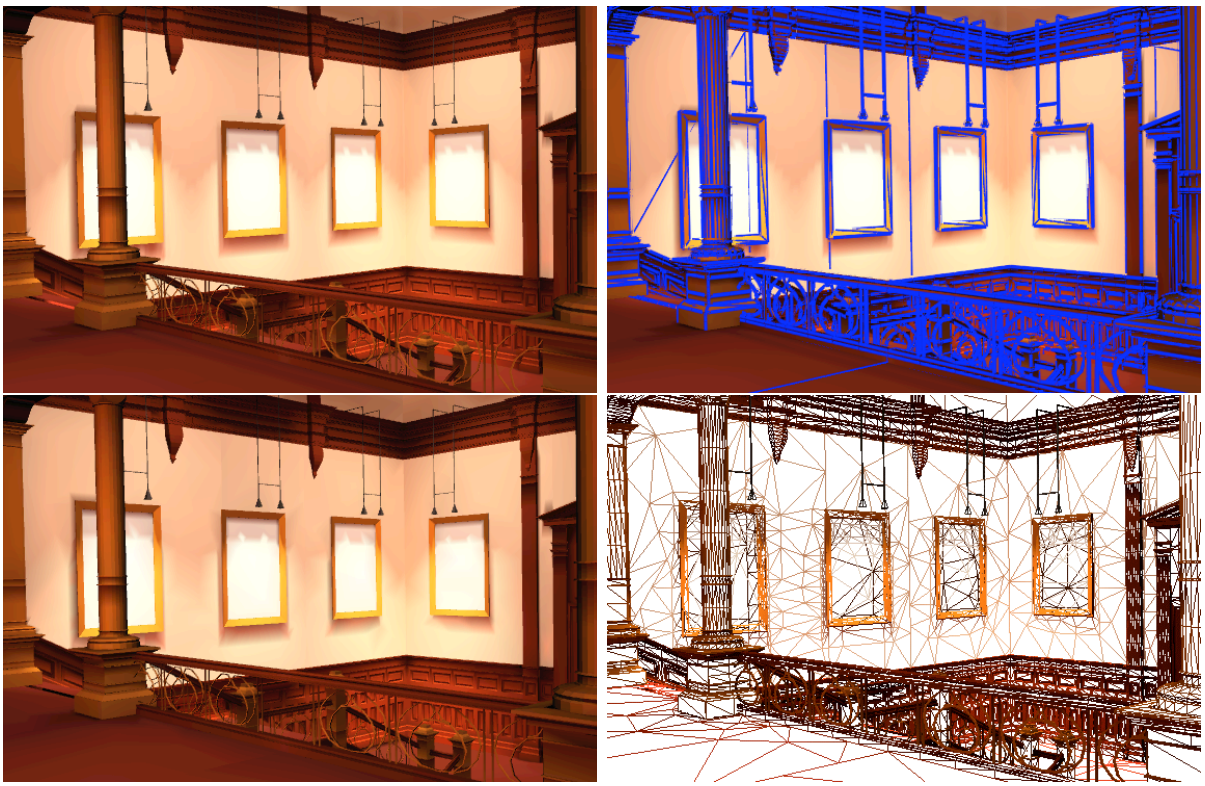

Figure 8: The upper row shows the original mesh consisting of 770,436 triangles (topological boundaries in blue). The mesh in the lower row has been reduced to 286,103 triangles.

data structure	bytes/$\triangle$	supports non-manifold	neighborhood	point reference	prev. edge	next edge
individual triangles	36	yes	-	-	(M)	(M)
shared vertex	18	yes	-	M	(M)	(M)
shared vertex with neig.	32	no	M	M	(M)	(M)
winged-edge	60	no	M	M	M	M
full directed-edge	68	yes	M	M	M	M
medium directed-edge	44	yes	M	M,C	M	C
small directed-edge	32	yes	M	M,C	C	C

Table 1: Comparison of several data structures. "M" indicates entities that are explicitly stored, "C" means that the information may be calculated in $\mathcal{O}(1)$, and "-" marks entities that require computations of complexity $\gg \mathcal{O}(1)$.

References

[Bau72] B.G. Baumgart. Winged-edge polyhedron representation. Technical report, STAN-CS-320, Stanford University, 1972.

[CB97] Rikk Carey and Gavin Bell. *The Annotated VRML 2.0 Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1997.

[CKS98] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. Enhancing Digital Documents by Including 3D-Models. *Computers & Graphics*, 22(6):655–666, 1998.

[FvDFJ92] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Jughes. *Computer Graphics - Principles and Practice.* Addison-Wesley, Reading, Massachusetts, 2nd edition, 1992.

[GS85] Leonidas Guibas and Jorge Stolfi. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[KCS98] Leif Kobbelt, Swen Campagna, and Hans-Peter Seidel. A General Framework for Mesh Decimation. In *Proceedings of Graphics Interface '98*, pages 43–50, 1998.

[KCVS98] Leif Kobbelt, Swen Campagna, Jens Vorsatz, and Hans-Peter Seidel. Interactive Multi-Resolution Modeling on Arbitrary Meshes. In *Proceedings of SIGGRAPH '98 (Orlando, Florida, July 19–24, 1998)*, Computer Graphics Proceedings, Annual Conference Series, pages 105–114, July 1998.

[KS97] Krzysztof S. Klimaszewski and Thomas W. Sederberg. Faster Ray Tracing Using Adaptive Grids. *IEEE Computer Graphics and Applications*, pages 42–51, January-February 1997.

[Mae88] Martti Maentylae. *An Introduction to Solid Modeling.* Computer Science Press, 1988.

[O'R95] Joseph O'Rourke. *Computational Geometry in C.* Cambridge University Press, Cambridge, 1995.

[PS97] Enrico Puppo and Roberto Scopigno. Simplification, LOD and Multiresolution Principles and Applications, 1997. Tutorial Eurographics '97.

[Ros97] Jarek Rossignac. Simplification and Compression of 3D Scenes, 1997. Tutorial Eurographics '97.

[SP94] François Sillion and Claude Puech. *Radiosity & Global Illumination.* Morgan Kaufmann, 1994.

[ZSS97] Denis Zorin, Peter Schröder, and Wim Sweldens. Interactive Multiresolution Mesh Editing. In *Proceedings of SIGGRAPH '97 (Los Angeles, California, August 3–8, 1997)*, Computer Graphics Proceedings, Annual Conference Series, pages 259–268, August 1997.