Vol. 6, No. 1: 00–00

Star-Vertices: A Compact Representation for Planar Meshes with Adjacency Information

Marcelo Kallmann and Daniel Thalmann
Swiss Federal Institute of Technology

Abstract. In this paper we propose a new structure, *star-vertex*, to represent general planar meshes. The basic concept is simple, allowing constant adjacency query time, and scalability (able to trade size for speed), and under specific situations requiring less storage space than others. For simplicity, we use a generic traverse element, which resembles the behavior of oriented edges. We present implementation examples of the proposed structure and comparisons with other mesh representation schemes.

1. Introduction

Polyhedral surfaces, or planar meshes, are used for describing the boundary of solids for visualization purposes, virtual reality applications, geometric algorithms, and for many types of calculations, often using finite element methods. Desirable properties of a mesh representation include low storage space, simplicity, fast retrieval of adjacency information, easy manipulation, and scalability (trading size for performance according to the target application).

The *winged-edge* structure [Baumgart 72] pioneered the concept of storing adjacency information. Later, traverse and construction operators were introduced [Guibas, Stolfi 85], [Mäntylä 88], and extensions to n dimensions

1086-7651/01 $0.50 per page

have been addressed [Brisson 89]. However, most of these structures keep a lot of redundant information in order to provide direct access to all adjacent elements. A consequence of keeping the redundant information is that the storage space requirements and the complexity of the implementation are largely increased. We believe that specific data structures can still be designed to optimize their target applications with respect to storage and speed. Recently, the *directed-edge* [Campagna et al. 99] was developed; it is a compact and scalable representation providing constant time adjacency information. It is restricted to triangle meshes and its small version requires 32 bytes per triangle.

In this paper we present the *star-vertex* data structure, which is able to represent arbitrary planar meshes; it is simple, compact, and provides constant time adjacency information. Because of its vertex-based nature, its size depends on the mean vertex degree of the mesh being represented, and we show that under specific situations, it may require a surprisingly low storage space. Moreover, it can be further compacted for meshes with uniform vertex degree or by slowing adjacency lookup to be linear in vertex degree. We also present a traverse element to safely access the data stored in the structure.

2. Star-Vertex Data Structure

The star-vertex structure is depicted in Figure 1. It is a vertex-based structure that keeps, for each vertex v of the mesh, its three float coordinates, pointers to all neighbor vertices of v, and an index that says, for each neighbor v' of v, which is the neighbor pointer of v' pointing to the vertex v'' so that v, v', and v'' are in the same face. This index is so used to retrieve vertices and edges around a face. Figure 1 represents the pointers and indices with arrows. Dashed arrows v_{0n}, v_{1n}, and v_{2n} represent the indices used to retrieve vertices

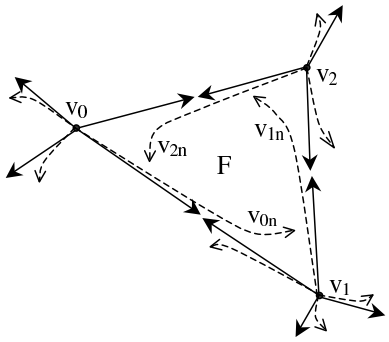

Figure 1. Connectivity diagram of the star-vertex structure.

and edges around faces. The letter n stands for the *next* edge around the face. The usage of such indices will be clearer in the example explained later with Figure 2 and Table 1.

There is a design choice when implementing this structure, between the use of pointers for direct memory access, or the use of integers as indices to positions in a user-maintained array. We have implemented and tested a hybrid approach where the design goal was the simplicity of implementation and the ease of comparison with other structures. This implementation was done as follows:

```
struct Neighbor
 { Vertex *vtx;      // pointer to the neighbor vertex
   int nxt;          // to find the next vertex in the face
 };

struct Vertex
 { float x, y, z;    // vertex coordinates
   int num_nb;       // number of neighbors
   Neighbor *nb;     // pointer to the array of neighbors
 };

struct StarVertexMesh
 { array<Vertex> vertices; // all vertices of the mesh
 };
```

As an example, consider the planar mesh shown in Figure 2. Its star-vertex representation is given in Table 1. In this table, the third column encodes the neighborhood information. For example, vertex v_0 has as neighbor array $\{(v_1, 3), (v_2, 2), (v_5, 1), (v_4, 2)\}$. Note that the first element of each pair in the array explicitly stores in counterclockwise order the neighbors of v_0: $\{v_1, v_2, v_5, v_4\}$.

To traverse the vertices around a face, we start with one of its vertices, let's say, v_0. Because of the implicit counterclockwise ordering, to traverse

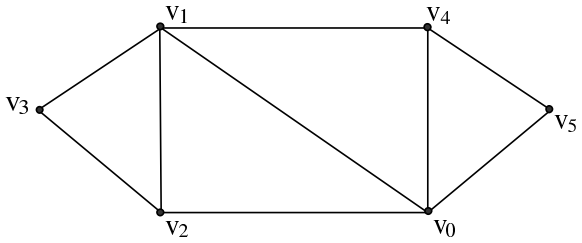

Figure 2. A planar mesh example.

(x, y, z)	num_nb	nb - list of neighbors
v_0	4	$(v_1, 3), (v_2, 2), (v_5, 1), (v_4, 2)$
v_1	4	$(v_0, 3), (v_4, 1), (v_3, 1), (v_2, 0)$
v_2	3	$(v_0, 0), (v_1, 2), (v_3, 0)$
v_3	2	$(v_1, 1), (v_2, 1)$
v_4	3	$(v_0, 2), (v_5, 0), (v_1, 0)$
v_5	2	$(v_0, 1), (v_4, 0)$

Table 1. Mesh of Figure 2 in the star-vertex representation.

the face $\{v_0, v_1, v_2\}$ we know that the edge to consider is $\{v_0, v_1\}$ which has v_0 as its first vertex. Since the first pair $(v_1, 3)$ of the neighborhood array of v_0 is the one that points to v_1, we take the index 3 that tells which pair in the neighborhood of v_1 is the one to continue the traverse. The pair with index 3 of v_1 is $(v_2, 0)$ (indices start from 0). Continuing with this process, we move to the pair $(v_0, 0)$ of v_2, which will then lead us back to the initial pair $(v_1, 3)$. In this way we have identified all vertices and edges around the face $\{v_0, v_1, v_2\}$, in an ordered way, by traversing sequentially the pairs: $(v_1, 3), (v_2, 0), (v_0, 0)$. Note also that the boundary $\{v_0, v_2, v_3, v_1, v_4, v_5\}$ is considered to be a back face and will be traversed clockwise.

3. Traverse Element

We have seen in the last section how the structure encodes edges around a vertex and vertices around a face. This information is sufficient to retrieve all kind of adjacency relations, and in order to provide a safe and convenient interface to access them, we now propose a traverse element, or *travel*.

A travel is a structure-independent generalization of concepts from edge-based structures, like the *edge-use* [Weiler 85], the *dart* [Lienhardt 89], the *half-edge* [Mäntylä 88], and the iterators defined in a recent C++ implementation [Kettner 98]. Figure 3 shows the same mesh example of Figure 2 and Table 1, but some travels are represented graphically as oriented edges. Each travel

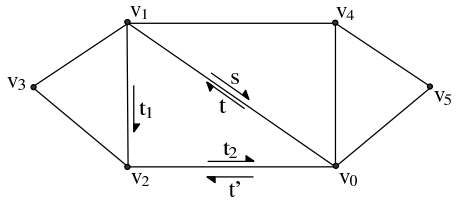

Figure 3. Some traverse elements graphically represented.

is always adjacent to one, and only one, vertex, edge, and face of the mesh. For example, in Figure 3, travel t is adjacent to vertex v_0, to edge $\{v_0, v_1\}$, and to face $\{v_0, v_1, v_2\}$. We now define two operators that can be applied to t: *nxt* and *rot*. The *nxt* operator,when applied to t, will return the travel that is adjacent to the next edge and vertex around the face adjacent to t. This operator allows traversal of the edges around a face. For example, in Figure 3, t.nxt $\equiv t_1$, t_1.nxt $\equiv t_2$, and t.nxt.nxt.nxt $\equiv t$.

Similarly, when applying the rot operator to t, the travel adjacent to the next edge and face around the adjacent vertex of t is returned. This operator allows the possibility to "rotate" around a given vertex. For example, in Figure 3, we have t.rot $\equiv t'$, and t'.rot.rot.rot $\equiv t$.

With these two operators defined, we can define the operator *sym*, that gives the symmetrical travel: t.sym $\equiv t$.nxt.rot $\equiv s$. And also the inverses: $t.\text{sym}^{-1} \equiv t$.sym, $t.\text{nxt}^{-1} \equiv t$.rot.sym, and $t.\text{rot}^{-1} \equiv t$.sym.nxt. As our traverse element behaves exactly as an oriented edge, we refer the reader to the half-edge structure [Mäntylä 88] for a detailed explanation of a very similar scheme of traverse operators.

Once the traverse element is equipped with operators to retrieve its adjacent elements, we are able to traverse freely through the structure, querying all adjacent relations. The following code indicates how to implement such a traverse element for the star-vertex structure, using C++ notation:

```
class Travel
 { Vertex *v;  // points to the adjacent vertex of the travel
   int     r;  // indicates the adjacent edge of the travel

   // some operators and methods :
   Travel ( Vertex *vtx, int rot ) { v=vtx; r=rot; }
   Travel rot () { return Travel(v,(r+1)%v->num_nb); }
   Travel nxt () { return Travel(v->nb[r].vtx,v->nb[r].nxt); }
   Travel sym () { return nxt().rot(); }
   float *pnt () { return &(v->x); }
   bool operator == ( Travel t ) { return v==t.v && r==t.r; }
 };
```

The travel structure keeps a pointer to the current adjacent vertex v, and the index r. This index defines the pair in the neighborhood array of v which has v_n as the vertex defining the current adjacent edge $\{v, v_n\}$ of the travel. Because of the implicitly stored counterclockwise order, the adjacent face is also defined. As an example, it is easy to verify that: Travel$(v_0, 0)$.nxt() $\equiv$ Travel$(v_1, 3)$, and that Travel$(v_0, 0)$.rot() $\equiv$ Travel$(v_0, 1)$. One consequence of using such a vertex-based structure is that faces are not stored explicitly. Thus some algorithm to retrieve faces is needed, and in many cases some

mechanism is required to mark traverse elements already visited. The following code shows how we can use the nxt index of the *Neighbor* structure to mark elements, by adding two methods to the Travel structure:

```
void Travel::mark  () { v->nb[r].nxt *= -1; }
bool Travel::marked () { return v->nb[r].nxt<0; }
```

The mark is stored by setting the index to a negative value. Note however, that this implies that the 0 index can not be used, and that it is necessary to consider the absolute value of the index. The following code gives an example of an algorithm that sends the faces of a star-vertex mesh to an OpenGL renderer. It starts with an initial face and exploits face adjacency to render all other faces:

```
render ( const StarVertexMesh& m )
   {
      array<Travel> stack;
      stack.push( Travel(m.vertices[0],0) );

      while ( !stack.empty() )
       { Travel ti = stack.pop();
         if ( ti.marked() ) continue;
         Travel t=ti;
         glBegin ( GL_POLYGON );
         do { glVertex3fv ( t.pnt() );
              if ( !t.marked() ) t.mark();
              stack.push ( t.sym() );
              t = t.nxt();
            } while ( t!=ti );
         glEnd ();
       }
   }
```

Note that in the case of a planar mesh like the one in Figure 2, the exterior border is also sent, but it will be considered a back-face as we have consistent orientations. Note also that faces need to be convex to be correctly handled by OpenGL.

Some strategies can be used to avoid unmarking all previously marked traverse elements each time such an algorithm is called. For example, each time the algorithm starts, it can determine if elements are considered marked when indices are negative or positive. However, with this technique, we cannot allow an algorithm to leave the mesh "half-marked".

The fact that faces are not explicitly stored does not slow down rendering, because nearly all systems work with optimized display lists of the polygons

to be rendered. Thus, traversal of faces would be done to update display lists only when the model changes. Moreover, the generation of display lists can make use of the encoded adjacent relations to generate optimized "connected" lists, as for example, the triangle or quad strip schemes of OpenGL.

4. Analysis

The size of the star-vertex structure is directly related to the number of edges around vertices. Let v be a vertex of the mesh. The degree of v is equal to the number of edges incident to v. Let us now define k as the mean of all vertex degrees in the mesh: $k = (\sum degree(v))/n$, where n equals the total number of vertices. The parameter k is directly related to how the mesh was created. For instance, meshes generated from parametric surfaces, as NURBS [Foley et al. 92], commonly have quadrilateral faces and $k = 4$.

Meshes with $k = 3$ have good properties and methods exist for their generation [Delingette 94]. A cube, a cylinder and a tetrahedron are examples of models with $k = 3$. However, for some objects, it is not possible to have an accurate representation with $k = 3$. One example is the polyhedral approximation of a cone with a polygonal base of b vertices. All vertices in the cone base have degree 3, but the peak will have degree b, giving $k = (3b+b)/(b+1)$, which approaches $k = 4$ for large values of b.

For triangular meshes, k approaches 6 when the number of vertices grows. Figure 4 gives examples of meshes with $k = 3$, $k = 4$, and $k = 6$.

We now count the space requirements of our structure. From Euler's formula [Foley et al. 92], we have $V - E + F = 2$ for a general manifold mesh, and also that $F \approx 2V$ ($F = 2V - 4$) if the faces are all triangles. Consider now that we have a mesh composed of n vertices and m faces. A mesh represented by the star-vertex structure will occupy $4 \times 5 \times n$ bytes for the vertex structure, plus approximately $4 \times 2 \times k \times n$ bytes for the list of neighbors, assuming four-byte integer and float types.

For comparison purposes, let us assume we are applying the star-vertex to a triangle mesh, so that we can apply the $m \approx 2n$ property. The whole

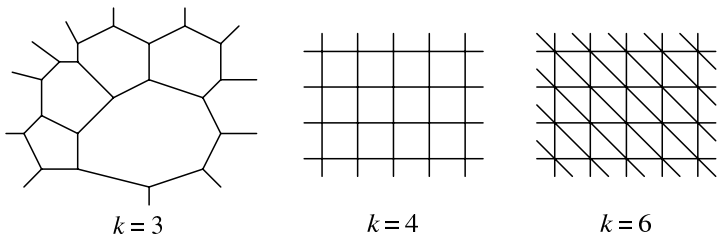

Figure 4. Example of meshes with different vertex degrees.

structure will then take $(4 \times 5 + 4 \times 2 \times k)n \approx 10 + 4k$ bytes per triangle. This gives us approximately 22, 26, 30, or 34 bytes per triangle when describing a triangle mesh with k equal to 3, 4, 5, or 6, respectively. We can still decrease this storage space for some specific cases, and we will now show two simplifications for the given "general" star-vertex structure.

An initial simplification is accomplished when the mesh has a constant vertex degree for all vertices. In this case, we can drop the pointer to an array of variable length and the number of neighbors per vertex. Doing so, we can economize 1 integer and 1 pointer per vertex, making an economy of 8 bytes per vertex, or 4 bytes per triangular face. The result is $6 + 4k$ bytes per triangle for this "uniform" star-vertex, that can only represent meshes with constant vertex degree.

Another type of simplification reduces the required storage space even more, but no longer has constant time execution for the nxt operator. This simplification simply drops the nxt index of the Neighbor structure. The nxt operator will then take time $O(d_{max})$, where d_{max} is the maximum vertex degree in the mesh. This happens because we need to search among all edges incident to the neighbor vertex, the one that correctly produces the result of the nxt operator. The implementation of the nxt operator would then look as follows:

```
Travel Travel::nxt ()
  { Travel t (v->nb[r].vtx,0);
    while ( t.v->nb[t.r].vtx!=v ) t=t.rot();
    if ( --t.r<0 ) t.r = t.v->num_nb-1;
    return t;
  }
```

In this compact version, the structure will occupy $4 \times 5 \times n$ bytes for the vertex structure, plus $4 \times k \times n$ bytes for the list of neighbors, ending up with $(4 \times 5 + 4 \times k)n \approx 10 + 2k$ bytes per triangle. It is also possible to have a structure with both compact and uniform simplifications, leading us to $(4 \times 3 + 4 \times k)n$ bytes $= 6 + 2k$ bytes per triangle.

5. Comparison

We will now compare the star-vertex with several structures: directed-edge [Campagna et al. 99], a simplified version of quad-edge [Guibas, Stolfi 85], and with simple polygon lists. Even if some of the structures can represent general meshes, in order to compare them, we will consider that they are representing triangular meshes, so that the $m \approx 2n$ relation can be applied.

For each edge, our simplified quad-edge implementation keeps pointers explicitly giving the result of the nxt, rot, nxt^{-1} and rot^{-1} operators, plus two pointers for the two vertices of the edge. This representation requires

data structure	rot operator time	nxt operator time	mesh type	bytes/Δ				
				Any k	$k=3$	$k=4$	$k=5$	$k=6$
polygon lists	-	O(1)	-	22	22	22	22	22
triangle lists	-	O(1)	Δ	18	18	18	18	18
simplified quad-edge	O(1)	O(1)	-	42	42	42	42	42
small directed-edge	O(1)	O(1)	Δ	32	32	32	32	32
star-vertex	O(1)	O(1)	-	$10+4k$	22	26	30	34
uniform star-vertex	O(1)	O(1)	deg cte	$6+4k$	18	22	26	30
compact star-vertex	O(1)	O(d_{max})	-	$10+2k$	16	18	20	22
minimal star-vertex	O(1)	O(d_{max})	deg cte	$6+2k$	12	14	16	18

Table 2. Comparison of the several data structures. In the rot operator column, "-" indicates that its computation is not possible with only a local search in the data structure. In the mesh type column, "-" indicates that there are no restrictions on the mesh to be represented. Variables k and d_{max} represent, respectively, the mean and the maximum vertex degree of the mesh.

$3 \times 4 \times n = 12n$ bytes for the vertex coordinates, plus $6 \times 4 = 24$ bytes per edge. From Euler's formula, $E = V + F - 2$, and so this structure requires $12n+24\times(n+m-2) \approx 6m+12m+24m = 42m$ when representing triangular meshes.

Simple polygon or triangle lists are very popular and most commercial scene graphs implement them using the name *indexed face set*. A triangle list is based on arrays of vertex coordinates and vertex indices forming the triangles, usually requiring $3\times4\times n = 12n$ bytes for the coordinates, and $3\times4\times m = 12m$ for the indices. ($12n+12m \approx 18m$). This benchmark of 18 bytes per triangle has been considered a lower limit for storing triangle meshes. For polygon lists, the face indices array will include a 1 index to indicate when each face has finished, resulting in $12n + 16m \approx 22m$, i.e., 22 bytes per triangle.

Table 2 compares the different data structures. We have listed the time required for the determination of the rot and nxt operators. When these two operators are provided in constant time, all adjacent relations also can be retrieved in constant time. As expected, our structure can achieve very low memory requirements when k is small. In particular, when the uniform or compact versions can be applied, the star-vertex can even achieve less storage space than representations based on polygon lists, while still storing adjacency relations. In such cases, the star-vertex structure becomes the most compact structure available.

It is important to note that the star-vertex structure requires many pointer indirections in order to retrieve the adjacencies, which can slow down algorithms. To give an example of a real application, we have re-written an incremental Delaunay triangulation algorithm [Guibas, Stolfi 85] using the directed-edge, quad-edge and the star-vertex structure in its general and com-

data structure	time (seconds)			size (Kbytes)		
	n=32000	n=48000	n=64000	n=32000	n=48000	n=64000
simplified quad-edge	27.6	51.2	79.6	2625	3938	5250
small directed-edge	29.2	56.3	93.7	2000	3000	4000
star-vertex	34.7	66.0	109.8	2125	3188	4250
compact star-vertex	42.3	81.7	134.5	1375	2063	2750

Table 3. Performance comparison. The variable n stands for the number of vertices inserted in the Delaunay triangulation.

pact form. We have defined an abstract traverse element class, with all operators required by the algorithm implemented as virtual methods, and then we have derived this class for each different data structure. This abstract class inclusion slows our original algorithm by a factor of 2.5.

The incremental Delaunay algorithm is divided into two phases: the first phase finds the triangle containing a random point to be inserted by just jumping through adjacent triangles, and the second phase inserts the point with local edge flip operations. In this way we are measuring both the performance of adjacency queries and structure updates.

Table 3 shows the obtained times and sizes. As expected, the simplified quad-edge was the fastest structure as it explicitly stores the result of adjacency operators; on the other hand, it was the structure consuming more memory. The star-vertex consumed approximately 20% less memory but was 30% less efficient. Compared to the directed-edge, the star-vertex was approximately 20% slower and consumed 6% more memory. This gives an idea how our structure behaves for one of its worst cases, i.e., a triangulation with $k \approx 6$.

However, the compact form consumed 45% less memory at the performance cost of approximately 44%. Note also that the directed-edge is a specific

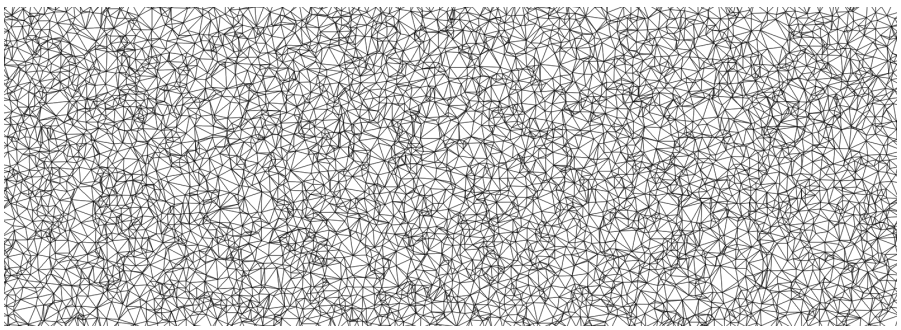

Figure 5. A zoom view of a generated Delaunay triangulation of random points used to test the star-vertex structure.

structure for triangulations while the star-vertex is a general one. These tests were performed on a Pentium III computer with 410 Mbytes of memory without any kind of optimization in the implementation of the data structures.

We have also noticed that the compact form of the star-vertex was faster than the general version up to $n = 2000$; for larger values of n the general implementation was faster. This shows that having $O(d_{max})$ time for the nxt operator is compensated by simpler structure updates and can be acceptable in many cases. As an illustration, one generated Delaunay triangulation is shown in Figure 5.

Acknowledgments. The authors are grateful to R. Farias, A. Aubel and R. Barzel for many valuable suggestions. This research was supported by the Swiss National Foundation for Scientific Research and by the Brazilian National Council for Scientific and Technologic Development (CNPq).

References

[Baumgart 72] B. G. Baumgart. *Winged-Edge Polyhedron Representation.* Technical Report STAN/CS/320, Stanford University, 1972.

[Brisson 89] E. Brisson. "Representing Geometric Structures in d-Dimensions: Topology and Order." In *ACM Symposium on Computational Geometry*, pp. 218–227, New York: ACM Press, 1989.

[Campagna et al. 99] S. Campagna, L. Kobbelt, and H. P. Seidel. "Directed Edges – A Scalable Representation for Triangle Meshes." *journal of graphics tools* 3(4): 1–12 1999.

[Delingette 94] H. Delingete. "Simplex Meshes: A General Representation for 3D Shape Reconstruction." In *Proceedings of the International Conference on Computer Vision and Pattern Recognition, CVPR '94* Los Alamitos: IEEE Press, 1994.

[Foley et al. 92] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice,* 2nd edition. Reading MA: Addison Wesley, 1992.

[Guibas, Stolfi 85] L. Guibas and J. Stolfi. "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams." *ACM Transaction on Graphics* 4: 75–123 1985.

[Kettner 98] L. Kettner. "Designing a Data Structure for Polyhedral Surfaces." In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry, Minneapolis, Minnesota, USA*, pp. 146–154, New York: ACM Press, 1998.

[Lienhardt 89] P. Lienhardt. "Subdivisions of N-Dimensional Spaces and N-Dimensional Generalized Maps." In *ACM Symposium on Computational Geometry*, pp. 228–236, New York: ACM Press, 1989.

[Mäntylä 88] M. Mäntylä. *An Introduction to Solid Modeling.* Baltimore, MD: Computer Science Press, 1988.

[Weiler 85] K. Weiler. "Edge-based Data Structures for Solid Modeling in Curved-Surface Environments."*IEEE Computer Graphics and Applications* 5(1): 21–40 January 1985.

Web Information:
The C++ source code of the examples shown in the paper is available at
`http://www.acm.org/jgt/papers/KallmannThalmann01`

Marcelo Kallmann, Computer Graphics Lab, Swiss Federal Institute of Technology, Lausanne, CH-1015, Switzerland (marcelo.kallmann@epfl.ch)

Daniel Thalmann, Computer Graphics Lab, Swiss Federal Institute of Technology, Lausanne, CH-1015, Switzerland (daniel.thalmann@epfl.ch)

Received May 16, 2000; accepted in revised form April 22, 2001.