Efficient Implementation of Progressive Meshes

Hugues Hoppe

January 1998

Technical Report
MSR-TR-98-02

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Efficient Implementation of Progressive Meshes

Hugues Hoppe
Microsoft Research, Redmond, WA, USA
(to appear in Computers & Graphics, 1998)

ABSTRACT

In earlier work, we introduced the progressive mesh (PM) representation, a new format for storing and transmitting arbitrary triangle meshes. For a given mesh, the PM representation defines a continuous sequence of level-of-detail approximations, allows smooth visual transitions (geomorphs) between these approximations, supports progressive transmission, and makes an effective compression scheme. In this paper, we present data structures and algorithms for efficient implementation of the PM representation and its applications. Also, we report quantitative results using a variety of computer graphics models.

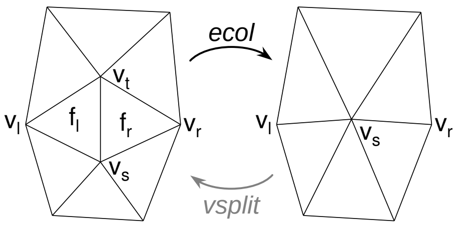


Figure 1: Illustration of the edge collapse transformation.

1 INTRODUCTION

Creating computer graphics often requires detailed geometric models for three-dimensional objects. Such models are typically created using commercial modeling and 3D scanning systems. Although some geometric models may be initially defined using high level primitives, for efficient rendering they are typically converted to their lowest common denominator form — polygonal approximations called *meshes*.

In the simplest case, a mesh consists of a set of vertices and a set of faces. Each vertex specifies the (x, y, z) coordinates of a point in space, and each face defines a polygon by connecting together an ordered subset of the vertices. Although the polygons may in general have arbitrary numbers of vertices (and even holes), we consider in this paper the special case of *triangle meshes*, in which all faces have exactly 3 vertices. However, arbitrary meshes can be easily converted to triangle meshes through a simple triangulation process.

Complex triangle meshes are notoriously difficult to render, store, and transmit. One approach to speed up rendering is to replace a complex mesh by a set of level-of-detail (LOD) approximations; a detailed mesh is used when the object is close to the viewer, and coarser approximations are substituted as the object recedes [2, 4]. These LOD approximations can be precomputed automatically using mesh simplification methods (e.g. [5, 8, 9, 10, 11, 12]). For efficient storage and transmission, mesh compression schemes [3, 13] have also been developed.

In earlier work [6], we introduced the *progressive mesh* (PM) representation, a new mesh format that provides a unified solution to these problems. In PM form, an arbitrary mesh $\hat{M}$ is stored as a coarse base mesh M^0 together with a sequence of n detail records that indicate how to incrementally refine M^0 into $M^n = \hat{M}$ (see Figure 2). Each detail record encodes the information associated with a *vertex split*, an elementary transformation that adds one vertex to the mesh. In addition to defining a continuous sequence of approximations $M^0 \dots M^n$, the PM representation supports smooth visual transitions (*geomorphs*) between these approximations, allows progressive transmission, and makes an effective mesh compression scheme.

Since the original paper [6], we have developed data structures and algorithms allowing the efficient implementation of progressive meshes. In this paper, we detail these data structures and algorithms, and present quantitative results on their performance. The remainder of the paper is organized as follows. We first review the PM representation in Section 2. Section 3 describes our basic data structures for meshes and progressive meshes. Section 4 describes the process of traversing the levels of detail within a progressive mesh. Section 5 discusses the creation of geomorphs, Section 6 addresses the issue of compression, and Section 7 summarizes the paper.

2 REVIEW OF PROGRESSIVE MESHES

To construct a PM representation [6], an arbitrary triangle mesh $\hat{M}$ is simplified through a sequence of *n edge collapse* transformations (*ecol* in Figure 1) to yield a much simpler base mesh M^0 (see Figure 2):

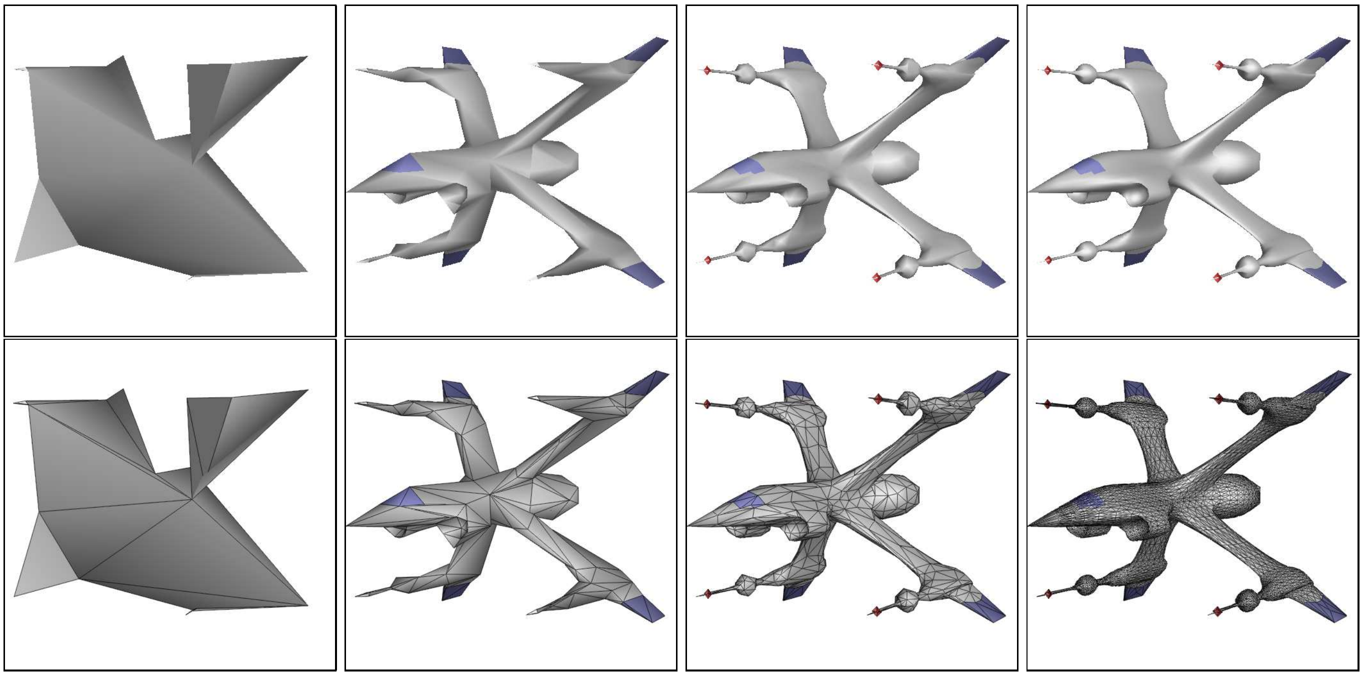
$$(\hat{M} = M^n) \xrightarrow{ecol_{n-1}} \dots \xrightarrow{ecol_1} M^1 \xrightarrow{ecol_0} M^0 .$$

The sequence of *ecol* transformations is chosen by an optimization process that seeks to preserve the appearance of the model [6]. Because each *ecol* has an inverse, called a *vertex split* transformation (Figure 1), the process can be reversed:

$$M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} \dots \xrightarrow{vsplit_{n-1}} (M^n = \hat{M}) .$$

The tuple $(M^0, \{vsplit_0, \dots, vsplit_{n-1}\})$ forms a PM representation of $\hat{M}$. Each vertex split, parametrized as $vsplit(v_s, v_l, v_r, \dots)$, modifies the mesh by introducing one new vertex v_t and two new faces $f_l = \{v_s, v_t, v_l\}$ and $f_r = \{v_s, v_r, v_t\}$ as shown in Figure 1. (We set v_r and f_r to *nil* if $\{v_s, v_t\}$ is a boundary edge.) The vertices and faces are numbered in the order that they are created, so that the indices of v_t, f_l, and f_r do not have to be stored explicitly. Of course, the vertex split must store the positions of the two split vertices v_s and v_t, as well as other appearance attributes associated with the mesh (as discussed in Section 3.1).

Email: hhoppe@microsoft.com
Web: http://research.microsoft.com/∼hoppe/



M^0 (44 faces) M^{200} (444 faces) M^{1000} (2,044 faces) M^n (17,068 faces)

Figure 2: The PM representation of an arbitrary mesh $\hat{M}$ captures a continuous-resolution family of approximating meshes $M^0 \ldots M^n = \hat{M}$.

The resulting sequence of meshes $M^0, \ldots, M^n = \hat{M}$ can be quickly traversed at runtime by applying a subsequence of *vsplit* and *ecol* transformations, and is therefore effective for real-time LOD control.

In addition, smooth visual transitions (*geomorphs*) can be constructed between any two meshes in this sequence. Given a coarser mesh M^c and a finer mesh M^f, $0 \leq c < f \leq n$, each vertex in M^f has a unique ancestor vertex in M^c, obtained by tracing back through the intervening *ecol* transformations. If all vertices in the mesh M^f are moved to the positions of their ancestor vertices in M^c, the mesh that results looks identical to M^c, because all faces in M^f missing from M^c are collapsed to degenerate (zero area) triangles. A geomorph is therefore obtained by smoothly interpolating the vertices of the mesh M^f between their original positions in M^f and that of their ancestors in M^c.

Finally, because each *vsplit* transformation can be encoded concisely, the PM representation is in fact a space-efficient representation. This paper describes data structures for achieving good space compression while maintaining time efficiency.

3 BASIC DATA STRUCTURES

In this section, we describe the basic data structures for both meshes and progressive meshes, with the aid of C++ notation. It should be noted that the C++ structures have been simplified for presentation purposes. Although we show most of the structure data members, we omit the numerous class member functions that should encapsulate these data members, as well as *public/private/friend* access declarations.

3.1 Mesh representation (Mesh)

Besides the geometric positions of its vertices, a computer graphics mesh often has numerous other appearance attributes used in the rendering of its surface. These appearance attributes can be classified into two types: discrete attributes and scalar attributes.

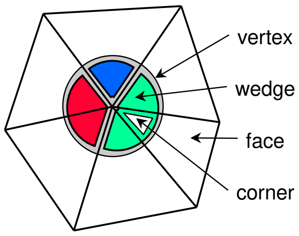


Figure 3: Illustration of vertices, wedges, and faces. In this example, the central vertex has 6 adjacent corners which are partitioned into 3 wedges.

Discrete attributes are usually associated with faces of the mesh. A common discrete attribute, the *material identifier*, determines the shader function used in rendering each face of the mesh [14]. For instance, a trivial shader function may involve simple look-up within a specified texture map.

Many scalar attributes are often associated with a mesh, including normals (n_x, n_y, n_z) and texture coordinates (u, v). More generally, these attributes specify the local parameters of shader functions defined on the mesh faces. In simple cases, these scalar attributes are associated with vertices of the mesh. However, to represent discontinuities in the scalar fields, and because adjacent faces may have different shading functions, it is necessary to associate scalar attributes not with vertices, but with corners of the mesh [1]. A *corner* is defined as a (vertex,face) tuple. Scalar attributes at a corner (v, f) specify the shading parameters for face f at vertex v. For example, along a *crease* (a curve on the surface across which the normal field is not continuous), each vertex has two distinct normals, one associated with the corners on each side of the crease.

A mesh with n vertices has approximately $2n$ faces, and thus approximately $6n$ corners. Explicit storage of attributes at all corners of the mesh would therefore require a significant amount of memory, and seems unnecessary since in general many corners adjacent to a vertex share the same attributes. One common approach to al-

```
struct VertexAttrib {          // Attributes at a vertex
    Point point;               // (x, y, z) coordinates
};
struct WedgeAttrib {           // Attributes at a wedge/corner
    Vector normal;             // (n_x, n_y, n_z) normal vector
    UV uv;                     // (u, v) texture coordinates
};
struct Vertex {
    VertexAttrib attrib;
};
struct Wedge {
    int vertex;                // vertex to which wedge belongs
    WedgeAttrib attrib;
};
struct Face {
    int wedges[3];             // wedges at corners of the face
    int fnei[3];               // 3 face neighbors
    short matid;               // material identifier
};
struct Mesh {
    Array<Vertex> vertices;
    Array<Wedge> wedges;
    Array<Face> faces;
    Array<Material> materials;
};
```

Figure 4: The mesh data structure.

```
struct PMesh {
    Mesh base_mesh;            // base mesh M^0
    Array<Vsplit> vsplits;     // {vsplit_0, ..., vsplit_{n-1}}
    int full_nvertices;        // number of vertices in M^n
    int full_nwedges;          // number of wedges in M^n
    int full_nfaces;           // number of faces in M^n
};
```

Figure 5: The progressive mesh data structure.

```
struct VertexAttribD {         // Delta applied to vertex attributes
    Vector dpoint;             // ΔVertexAttrib.point
};
struct WedgeAttribD {          // Delta applied to wedge attributes
    Vector dnormal;            // ΔWedgeAttrib.normal
    UV duv;                    // ΔWedgeAttrib.uv
};
struct Vsplit {
    int flclw;                 // a face in neighborhood of vsplit
    short vlr_rot;             // encoding of vertex v_r
    struct {
      short vs_index : 2;      // index (0..2) of v_s within flclw
      short corners : 10;      // corner continuities in Figure 9
      short ii : 2;            // geometry prediction of Figure 10
      short matid_predict : 2; // are fl_matid,fr_matid required?
    } code;                    // set of 4 bit-fields (16-bit total)
    short fl_matid;            // matid of face fl if not predicted
    short fr_matid;            // matid of face fr if not predicted
    VertexAttribD vad_l, vad_s;
    Array<WedgeAttribD> wads;
};
```

Figure 6: Vertex split data structure.

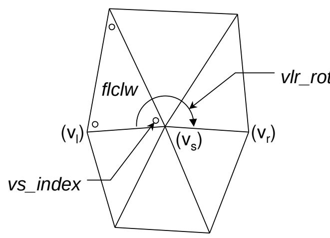


Figure 7: The Vsplit parameters *flclw*, *vs_index*, and *vlr_rot*, which identify the location of a vertex split.

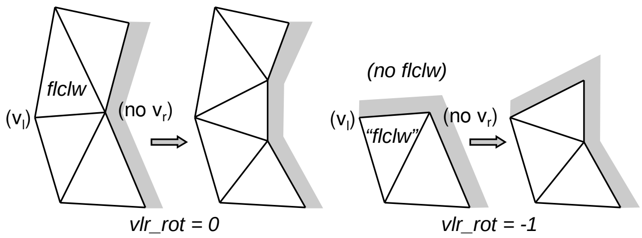


Figure 8: The parameter settings for the special cases of vertex splits in which vertex v_r or face *flclw* do not exist, i.e. next to a surface boundary.

leviate this problem is to store attributes only at vertices, and to tear the mesh apart along discontinuity curves (where adjacent corner attributes differ) by replicating some vertices. While this is satisfactory for static meshes, it makes runtime LOD and progressive transmission difficult, since modifications to the mesh structure may pull replicated vertices apart and introduce cracks in the surface.

Instead, our approach is to introduce an intermediate abstraction called a *wedge*. A wedge is a set of vertex-adjacent corners whose attributes are the same. Each vertex of the mesh is partitioned into a set of one or more wedges, and each wedge contains one or more face corners (see Figure 3). As shown in Figure 4, we define a mesh to contain an array of vertices, an array of wedges, and an array of faces, where faces point to wedges, and wedges point to vertices. Our implementations of the *vsplit* and *ecol* transformations requires adjacencies between elements of the mesh, so for each face we store pointers to its three neighboring faces. (A special neighbor value of -1 indicates a surface boundary.) Finally, each face contains a material identifier that indexes into an array of materials. These materials are platform-dependent but often include material colors and texture mapping parameters.

3.2 PM representation (PMesh)

The data structure for the PM representation corresponds closely with the tuple $(M^0, \{vsplit_0, \ldots, vsplit_{n-1}\})$. As seen in Figure 5, the *base_mesh* field stores M^0 using the Mesh structure of Section 3.1, and the *vsplits* field is an array of vertex split records. Also included are three fields that store information about the original mesh $\hat{M} = M^n$; these fields are used by the PM iterator (Section 4) for efficient pre-allocation of arrays. The remainder of this section discusses the encoding of the vertex split records, that is, the internals of the Vsplit structure (Figure 6).

Because the Mesh structure has incidence information only in the direction Face→Wedge→Vertex, we identify the location of the vertex split within the mesh not with vertices (v_s, v_l, v_r) but through the index of a face *flclw*, as shown in Figure 7. The vertex v_s being split is specified as an index $0 \leq vs\_index \leq 2$ into the ordered vertices of face *flclw*. The vertex v_l is the next clockwise vertex on face *flclw*. To determine the other vertex v_r, we store the number *vlr_rot* of clockwise face rotations about v_s from v_l to v_r. The face adjacency information Face::*fnei* is used to perform these rotations. Two special symbols for *vlr_rot* are used for the cases when v_r or *flclw* do not exist, as shown in Figure 8.

In the common case, a vertex split introduces two new faces (f_l and f_r) and therefore 6 new corners (Figure 9). A field of 10 bits, *corners*,

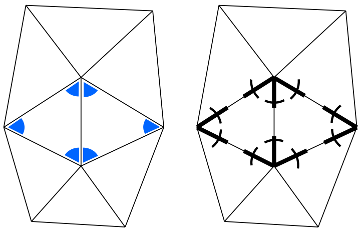


Figure 9: The 6 new corners introduced by a vertex split, and the 10-bit field *corners* used to record the continuity of corner attributes.

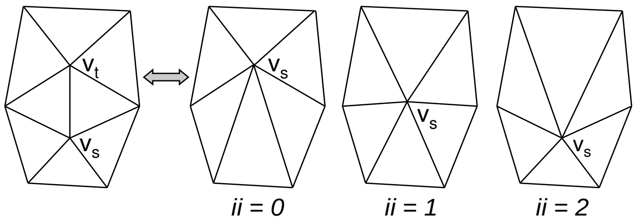


Figure 10: The Vsplit parameter *ii* used for geometry prediction.

encodes the wedges to which these new corners are assigned. Each bit in *corners* records whether a pair of adjacent corners (after the vertex split) belongs to the same wedge, as shown in Figure 9. From the bit field *corners*, one can determine how many new wedges (from 1 to 6) to introduce during a vertex split, and how to assign corners to new and old wedges.

For concise vertex and wedge attribute encodings, we predict the positions of the split vertices v_s and v_t relative to the old vertex v_s using a 2-bit field *ii* as shown in Figure 10. Specifically, we store two vertex position deltas, *vad_l* (large delta) and *vad_s* (small delta), and let the vertex split transformation modify vertex positions as follows:

$$\text{If } ii = 0, \quad \mathbf{v}_t := \mathbf{v}_s + vad\_s; \quad \mathbf{v}_s := \mathbf{v}_s + vad\_l$$
$$\text{If } ii = 2, \quad \mathbf{v}_t := \mathbf{v}_s + vad\_l; \quad \mathbf{v}_s := \mathbf{v}_s + vad\_s$$
$$\text{If } ii = 1, \quad \mathbf{v}_t := \mathbf{v}_s + vad\_s + vad\_l; \quad \mathbf{v}_s := \mathbf{v}_s + vad\_s - vad\_l$$

Similarly, the array *wads* encodes the deltas to the wedge attributes in the neighborhood. Depending on the number of wedges present in the neighborhood, the size of this array ranges from 1 to 6. For typical models, the average array size is only about 1.0–1.5 ($|wad|$ in Table 3).

The material identifier (Face.*matid*) of each new face (f_l and f_r) is predicted from an adjacent face prior to the vertex split. (The specific adjacent face is chosen based on *ii*.) A 2-bit field *matid_predict* records whether these predicted materials are correct. If incorrect, the materials are stored explicitly in the *fl_matid* and *fr_matid* fields of Vsplit.

The *code* field of Vsplit is a 16-bit mask that combines the bit-fields *vs_index* (2 bits), *corners* (10 bits), *ii* (2 bits), and *matid_predict* (2 bits).

4 PM TRAVERSAL

4.1 PM Read Stream (PMeshRStream)

The PMeshRStream class (Figure 11) provides an interface to abstract the source of PM data. This abstraction allows PM's to be used in three different scenarios:

(1) Reading from a PM stored in memory (a PMesh).
(2) Reading from a PM received progressively over an input stream.
(3) Reading from an input stream while archiving to a PMesh.

```
struct PMeshRStream {     // read from either PMesh or istream.
    PMesh* pm;            // may be 0
    istream* istr;        // may be 0
    int vspliti;          // if pm ≠ 0, index into pm->vsplits
    Vsplit vspl;          // if pm = 0, temporary buffer
};
```

Figure 11: Progressive mesh read stream.

```
struct PMeshIter : public Mesh {
    PMeshRStream& pmrs;
    PMeshIter(PMeshRStream&);
    PMeshIter(PMeshIter&);
    int next();                // apply one vertex split
    int prev();                // apply one edge collapse
    enum Type { WANT_NVERTICES, WANT_NFACES };
    int goto(Type, int);       // go to specified # of vertices/faces
    int nextA(Ancestry*);      // for use in Section 5
};
```

Figure 12: Progressive mesh iterator.

4.2 PM iterator (PMeshIter)

The class PMeshIter is used as an iterator within a PM sequence. As shown in Figure 12, it is derived from a Mesh, and contains a pointer into a PM source (PMeshRStream).

A PMeshIter is initialized from a PMeshRStream by simply copying the PM base mesh (PMesh::*base_mesh*) onto itself. In the case that the PMeshRStream is associated with an input stream, the base mesh is read directly from the input stream. A PMeshIter can also be initialized by cloning another iterator. Once initialized, PMeshIter is used to traverse the PM sequence, and since it is a Mesh, it can be rendered as needed.

4.3 Vertex split transformation (PMeshIter::next())

The member function PMeshIter::next() applies the next vertex split transformation to the current mesh. If this Vsplit record is not found in memory (in *pmrs.pm*), it is read on demand from the input stream. The vertex split transformation works as follows. It appends 1 vertex, 1–6 wedges, and 1–2 faces to the arrays in Mesh. It traverses the old corners around the newly added vertex v_t (using the face adjacencies in Face::*fnei*) and possibly updates the corners to point to the new wedge(s) associated with v_t. It updates the local face adjacencies to reflect the introduction of the new faces. Finally, it updates the vertex and wedge attributes using the deltas stored in Vsplit.

4.4 Edge collapse transformation (PMeshIter::prev())

The member function PMeshIter::prev() moves through the PM sequence backwards by performing the edge collapse transformation that is the inverse of the previous vertex split. Since it requires accessing the Vsplit sequence backwards, the prev() function is only supported if the PM source (PMeshRStream) has an associated memory-resident PMesh, i.e. in Scenarios (1) and (3) of Section 4.1.

The Vsplit structure contains enough information to perform both the vertex split and its inverse edge collapse. One key element that makes this possible is that all changes to mesh attributes are recorded as deltas, so that they can be applied both forwards and backwards.

The edge collapse works in the reverse order of the vertex split

```
struct Geomorph : public Mesh {
    Array<VertexAttrib> vattribs[2];
    Array<WedgeAttrib> wattribs[2];
    enum Type { WANT_NVERTICES, WANT_NFACES };
    Geomorph(PMeshIter&, Type, int);
    void evaluate(float);        // takes parameter 0 ≤ α ≤ 1
};
```

Figure 13: The data structure for a geomorph.

```
struct Ancestry {
    Array<VertexAttrib> anc_vattribs;
    Array<WedgeAttrib> anc_wattribs;
};
```

Figure 14: The data structure used to track ancestral attributes of PMeshIter::*vertices* and PMeshIter::*wedges* during geomorph construction.

transformation. It updates vertex and wedge attributes, updates face adjacencies, updates corners around the old vertex v_t, and finally removes 1 vertex, 1–6 wedges, and 1–2 faces from the ends of the arrays in Mesh.

4.5 Iteration to specified complexity (PMeshIter::goto())

The function PMeshIter::goto() supports iteration to a desired level of complexity, expressed as either number of vertices or number of faces, by simply invoking next() or prev() repeatedly.

We used a number of meshes (Table 1) to measure the speed of PM iteration. Table 2 shows the iteration rates, in vertices per second, for both reconstruction (going from M^0 to M^n) and simplification (going from M^n to M^0), on a 200 MHz Pentium Pro processor. We suspect that the lower iteration rates for the larger models are due to the memory architecture of the machine.

5 GEOMORPHS

As discussed previously in Section 2, a geomorph allows the smooth visual transition between any two meshes M^c and M^f, $0 \leq c < f \leq n$, in a PM sequence. The geomorph is essentially a copy of the mesh M^f, but whose attributes at vertices and wedges interpolate between their values in M^f and those of their vertex and wedge ancestors in M^c. As shown in Figure 13, a Geomorph structure is derived from a Mesh, and in addition contains a pair of end states (*vattribs*[0..1] and *wattribs*[0..1]) for its vertex and wedge attributes.

A Geomorph between M^c and M^f is constructed by providing both a PMeshIter pointing to M^c and the complexity (number of vertices or number of faces) of M^f. During the geomorph construction, the PMeshIter is advanced through the PM sequence using the special member function PMeshIter::nextA(). This function nextA() behaves just like PMeshIter::next(), except that it tracks the ancestral attributes of vertices and wedges using the Ancestry structure shown in Figure 14. Once the PMeshIter has been advanced to M^f, the current vertex and wedge attributes of PMeshIter::Mesh are copied to *vattribs*[1] and *wattribs*[1], and the ancestral attributes in Ancestry are copied to *vattribs*[0] and *wattribs*[0]. In our current implementation, the creation of a geomorph requires approximately twice as much time as simple iteration through the PM sequence.

The Geomorph::evaluate() function uses the floating-point parameter $0 \leq \alpha \leq 1$ to interpolate its vertex and wedge attributes between the pair of end states. Points and texture coordinates are linearly interpolated, but normals are interpolated over the unit sphere. If the fraction of vertices and wedges that require interpolation is small, a sparse data structure can replace *vattribs*[0..1] and *wattribs*[0..1] to reduce memory use and speed up geomorph evaluation.

6 COMPRESSION

In this section, we compare the memory space required for the Mesh and PMesh structures, and also compare how well these structures can be compressed for storage and transmission.

6.1 Memory-resident representation

The two columns labeled "memory" in Table 2 show the average number of bits per vertex for the Mesh and PMesh data structures (Figures 4 and 5) using our test meshes. To make the comparison fair, we omitted the Face::*fnei*[3] field when computing the memory required for Mesh, since face adjacency information is unnecessary for rendering static models. All coordinates (for points, normals, and texture) are represented as 32-bit floating-point numbers; integers are 32-bit, and shorts are 16-bit.

We observe that the PMesh structure is in fact slightly more compact than the standard Mesh structure, even though it encodes not just M^n but the entire PM sequence $M^0 \ldots M^n$. Of course, the PMesh structure cannot be rendered directly, since a PMeshIter must first traverse it to construct a Mesh. However, in a complex scene, only a fraction of the scene objects require a high level-of-detail, and thus the memory overhead of maintaining these dynamic Mesh structures may be small.

6.2 Compressed representation

For storage and transmission of meshes, it may be worthwhile to compress the data structures. While compression may typically be performed off-line, the time overhead for decompression must be considered against storage scarcity and transmission bandwidth. We analyze two types of compression schemes, for both Mesh and PMesh. In both compression schemes, we quantize position coordinates to 16-bit, normal coordinates to 8-bit, and texture coordinates to 16-bit. The compression is therefore lossy, but these quantization levels seldom result in significant visual artifacts. The first compression scheme, labeled "gzip" in Table 2, applies Lempel-Ziv coding [17] to the binary data structures, as implemented by GNU "gzip". The second scheme, labeled "arith.", performs arithmetic coding [15], where the coding probability distributions are optimized on a per-mesh basis.

As shown in Table 2, PMesh compresses significantly better than Mesh. There are two main reasons for this.

First, the Face→Wedge→Vertex incidences are more concisely represented in PMesh than in Mesh. For arithmetic coding in particular, the Face::*wedges* and Wedge::*vertex* fields in Mesh require a total of more than $n(7 \log_2 n)$ bits, where n is the number of vertices, whereas the corresponding Vsplit fields *flclw*, *vs_index*, *vlr_rot*, and *corners* use approximately $n(\log_2 n + 5)$ bits (see Table 3).

Second, PMesh uses deltas to encode mesh attributes (positions, normals, and texture coordinates). These relative deltas compress better since they tend to be smaller in magnitude than the absolute values. When using arithmetic coding, we use variable-length delta encoding as described in [3, 6].

Table 3 shows how many bits are required on average to encode each field of the Vsplit records using arithmetic coding and variable-length delta encoding. As noted in the table, only one of our test meshes had non-zero texture coordinates.

A number of changes can be made to further improve the compression results of Table 3, to obtain the results of Table 4.

- The field *vad_s* is set to zero by restricting each edge collapse to place the new vertex at the position of one of the old vertices $\{\mathbf{v}_s, \mathbf{v}_t\}$ or at their midpoint. We can in fact perform this as a post-process on an existing PMesh, by first constructing the original mesh M^n and then traversing the Vsplit array backwards, and finally updating the vertex positions of the base mesh.

Model	Original mesh M^n			Base mesh M^0			n
	#vertices	#wedges	#faces	#vertices	#wedges	#faces	
garethman	801	1,207	1,586	31	84	46	770
cessna	6,795	9,533	13,546	46	75	48	6,749
bigship	8,536	8,847	17,068	24	59	44	8,512
dunebuggy	11,322	11,674	22,444	513	568	826	10,809
gameguy	21,412	25,095	42,712	31	50	27	21,381
drumset	34,794	59,834	68,776	963	2,192	1,114	33,831
chandelier	36,627	55,289	72,346	2,140	4,930	3,372	34,487
bunny	34,835	34,835	69,473	13	13	18	34,822
dragon	429,753	429,753	859,586	259	259	598	429,494
buddha	517,924	517,924	1,036,260	942	942	2,296	516,982
gcanyon	360,000	360,000	717,602	3	3	1	359,997

Table 1: Statistics for the various data sets.

Model	Iteration rates (verts/sec)		Space for M^n (bits/vertex)					
	goto(M^n)	goto(M^0)	Mesh			PMesh		
			memory	gzip	arith.	memory	gzip	arith.
garethman	n/a	n/a	607	257	214	541	221	111
cessna	105,000	149,000	589	232	227	517	152	86
bigship	112,000	158,000	519	241	199	455	189	105
dunebuggy	97,000	135,000	516	230	208	461	168	88
gameguy	92,000	126,000	544	240	223	477	158	80
drumset	79,000	108,000	648	272	276	572	179	100
chandelier	81,000	112,000	607	249	257	542	170	98
bunny	80,000	107,000	511	247	209	448	148	74
dragon	76,000	101,000	512	248	235	448	132	64
buddha	75,000	100,000	512	248	237	449	132	65
gcanyon	71,000	94,000	511	223	233	448	97	58

Table 2: PM iteration rates and space requirements.

Model	Avg. $\|wad\|$	*flclw*	*vs_index*	*vlr_rot*	*corners+ii+ matid_pred*	*fl_matid+ fr_matid*	VertexAttribD *vad_l*	*vad_s*	WedgeAttribD Δ*normal*	Δ*uv*	Σ
garethman	1.49	8.4	1.6	1.6	4.6	0.1	36.2	21.7	31.4	0.0	105.5
cessna	1.42	11.3	1.6	1.9	4.1	0.1	29.1	12.5	24.3	0.0	85.0
bigship	1.02	11.6	1.6	2.0	1.2	0.0	30.2	18.0	18.3	22.2	105.1
dunebuggy	1.03	12.2	1.6	2.0	0.6	0.0	27.5	20.0	20.6	0.0	84.5
gameguy	1.18	13.0	1.6	1.7	2.5	0.0	26.3	13.7	21.2	0.0	80.1
drumset	1.74	13.8	1.6	2.2	4.6	0.7	25.3	15.6	32.0	0.0	95.8
chandelier	1.46	14.0	1.6	2.1	1.9	0.0	23.9	15.1	28.6	0.0	87.3
bunny	1.00	13.6	1.6	1.4	0.1	0.0	28.2	15.3	13.7	0.0	74.0
dragon	1.00	17.3	1.6	2.0	0.0	0.0	21.6	8.9	12.9	0.0	64.2
buddha	1.00	17.6	1.6	2.0	0.0	0.0	21.1	8.4	13.7	0.0	64.4
gcanyon	1.00	17.0	1.6	1.7	0.1	0.0	21.5	6.4	9.7	0.0	58.1

Table 3: Space of Vsplit fields (bits/*vsplit*), with arithmetic coding and variable-length delta encoding.

Model	Avg. $\|wad\|$	Δ*flclw*	*vs_index*	*vlr_rot*	*corners+ii+ matid_pred*	*fl_matid+ fr_matid*	VertexAttribD *vad_l*	*vad_s*	WedgeAttribD Δ*normal*	Δ*uv*	Σ
garethman	1.49	6.1	1.6	1.6	4.6	0.1	36.2	0.0	0.0	0.0	50.1
cessna	1.42	6.8	1.6	1.9	4.1	0.1	29.2	0.0	0.0	0.0	43.7
bigship	1.02	6.5	1.6	2.0	1.2	0.0	30.2	0.0	0.0	22.2	63.6
dunebuggy	1.03	6.8	1.6	2.0	0.6	0.0	27.3	0.0	0.0	0.0	38.3
gameguy	1.18	6.8	1.6	1.7	2.5	0.0	26.3	0.0	0.0	0.0	39.0
drumset	1.74	6.8	1.6	2.2	4.6	0.7	25.2	0.0	0.0	0.0	41.1
chandelier	1.46	6.9	1.6	2.1	1.9	0.0	23.9	0.0	0.0	0.0	36.5
bunny	1.00	6.3	1.6	1.4	0.1	0.0	28.2	0.0	0.0	0.0	37.7
dragon	1.00	6.7	1.6	2.0	0.0	0.0	21.5	0.0	0.0	0.0	31.8
buddha	1.00	6.8	1.6	2.0	0.0	0.0	21.1	0.0	0.0	0.0	31.5
gcanyon	1.00	6.6	1.6	1.7	0.1	0.0	21.4	0.0	0.0	0.0	31.5

Table 4: Space of Vsplit fields (bits/*vsplit*) using three additional enhancements (reordering of *vsplit* records and encoding of Δ*flclw*, setting *vad_s* = 0, and computing normals based on wedges).

- Since normals are constrained to lie on the unit sphere, the Δ*normal* field could be encoded more succinctly using 2 degrees of freedom instead of 3 as it is now.

  Better yet, since discontinuities in the normal field are more important than the precise normal directions, the Δ*normal* field is omitted entirely, and normals are computed based on the vertex positions and the wedge information (which indicates the presence of creases).

- Finally, instead of storing the index of the face *flclw*, we store Δ*flclw* with respect to that in the previous Vsplit record, and permute the sequence of Vsplit records to make these deltas small.

  The vertex split transformations can be reordered as long as they preserve some dependency conditions [7, 16]. We encode these conditions by constructing a dependency graph. We then iteratively select vertex split transformations among the set of legal candidates, and use the dependency graph to update the candidate set. To obtain small values of Δ*flclw*, we store the candidate set as a balanced binary tree, sorted by *flclw*, and always select as the next vertex split the one with the next highest value of *flclw* (in circular sorted order). Our empirical evidence suggests that the size of the candidate set is roughly proportional to the size of the model reconstructed so far, so that the size of the variable-length encoded Δ*flclw* field is independent of model size; it is approximately 7 bits. The connectivity of the mesh is thus encoded in approximately $10.4n$ bits (the sum of the Δ*flclw*, *vs_index*, and *vlr_rot* fields in Table 4), and is now $O(n)$ instead of $O(n \log n)$.

  Note that the reordering of the Vsplit records modifies the progressive mesh sequence, so that the appearance of intermediate approximations ($M^i, i < n$) may deteriorate. However, this may be acceptable if storage of the detailed mesh M^n is the primary goal.

Table 4 shows the compression results when these three compression enhancements are performed. Figure 15 shows visual comparisons of the original meshes and the compressed meshes.

For the gzip-encoded PMesh stream, we measure a decompression rate of 86,000 *vsplit*/sec on a 200 MHz Pentium Pro processor. Since gzip-encoding saves about 300 bits per *vsplit*, gzip decompression is worthwhile if the transmission bandwidth is less than about 26 Mbit/sec. We unfortunately do not yet have a similar analysis for arithmetic decompression, but are confident that it would be beneficial over modem connections, which are $\leq$ 56 Kbit/sec.

7 SUMMARY AND FUTURE WORK

We have described an efficient implementation of the progressive mesh technology introduced in earlier work. This implementation is the basis for the progressive mesh feature available in Microsoft's DirectX 5.0 product release. Efficient data structures and algorithms permit fast iteration through the PM family of approximations, at speeds of approximately 100,000 vertices per second (or equivalently, 200,000 faces per second). Since reconstruction rates exceed the bandwidth of many networks, the progressive transmission of meshes benefits from data compression. We have shown that arithmetic coding, together with variable-length delta encoding, offers an effective compression scheme, and demonstrated further opportunities for compression.

ACKNOWLEDGMENTS

I wish to thank Viewpoint DataLabs for the "cessna", "dunebuggy", "gameguy", "drumset", and "chandelier" meshes; the meshes "bunny", "dragon", and "buddha" are courtesy of the Stanford University Computer Graphics Laboratory; the "gcanyon" mesh is from the United States Geological Survey. I also wish to thank John Miller for helpful discussions on compression issues.

REFERENCES

[1] APPLE COMPUTER, INC. *3D graphics programming with QuickDraw 3D*. Addison Wesley, 1995.

[2] CLARK, J. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM 19*, 10 (October 1976), 547–554.

[3] DEERING, M. Geometry compression. *Computer Graphics (SIGGRAPH '95 Proceedings)* (1995), 13–20.

[4] FUNKHOUSER, T., AND SÉQUIN, C. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics (SIGGRAPH '93 Proceedings)* (1993), 247–254.

[5] GARLAND, M., AND HECKBERT, P. Surface simplification using quadric error metrics. *Computer Graphics (SIGGRAPH '97 Proceedings)* (1997).

[6] HOPPE, H. Progressive meshes. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996), 99–108.

[7] HOPPE, H. View-dependent refinement of progressive meshes. *Computer Graphics (SIGGRAPH '97 Proceedings)* (1997).

[8] HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J., AND STUETZLE, W. Mesh optimization. *Computer Graphics (SIGGRAPH '93 Proceedings)* (1993), 19–26.

[9] RONFARD, R., AND ROSSIGNAC, J. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum (Proceedings of Eurographics '96) 15*, 3 (1996), 67–76.

[10] ROSSIGNAC, J., AND BORREL, P. Multi-resolution 3D approximations for rendering complex scenes. In *Modeling in Computer Graphics*, B. Falcidieno and T. L. Kunii, Eds. Springer-Verlag, 1993, pp. 455–465.

[11] SCHAUFLER, G., AND STÜRZLINGER, W. Generating multiple levels of detail from polygonal geometry models. In *Virtual Environments '95 (Eurographics Workshop)* (January 1995), M. Göbel, Ed., Springer Verlag, pp. 33–41.

[12] SCHROEDER, W., ZARGE, J., AND LORENSEN, W. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proceedings) 26*, 2 (1992), 65–70.

[13] TAUBIN, G., AND ROSSIGNAC, J. Geometry compression through topological surgery. Research Report RC-20340, IBM, January 1996.

[14] UPSTILL, S. *The RenderMan Companion*. Addison-Wesley, 1990.

[15] WITTEN, I., NEAL, R., AND CLEARY, J. Arithmetic coding for data compression. *Communications of the ACM 30*, 6 (June 1987), 520–540.

[16] XIA, J., AND VARSHNEY, A. Dynamic view-dependent simplification for polygonal models. In *Visualization '96 Proceedings* (1996), IEEE, pp. 327–334.

[17] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3 (May 1977), 337–343.