Geometry Coding and VRML

GABRIEL TAUBIN, SENIOR MEMBER, IEEE, WILLIAM P. HORN, FRANCIS LAZARUS, AND JAREK ROSSIGNAC

Invited Paper

The virtual-reality modeling language (VRML) is rapidly becoming the standard file format for transmitting three-dimensional (3-D) virtual worlds across the Internet. Static and dynamic descriptions of 3-D objects, multimedia content, and a variety of hyperlinks can be represented in VRML files. Both VRML browsers and authoring tools for the creation of VRML files are widely available for several different platforms.

In this paper, we describe the topologically assisted geometric compression technology included in our proposal for the VRML compressed binary format. This technology produces significant reduction of file sizes and, subsequently, of the time required for transmission of such files across the Internet. Compression ratios of 50 : 1 or more are achieved for large models. The proposal also includes a binary encoding to create compact, rapidly parsable binary VRML files. The proposal is currently being evaluated by the Compressed Binary Format Working Group of the VRML Consortium as a possible extension of the VRML standard.

In the topologically assisted compression scheme, a polyhedron is represented using two interlocking trees: a spanning tree of vertices and a spanning tree of triangles. The connectivity information represented in other compact schemes, such as triangular strips and generalized triangular meshes, can be directly derived from this representation. Connectivity information for large models is compressed with storage requirements approaching one bit per triangle. A variable-length, optionally lossy compression technique is used for vertex positions, normals, colors, and texture coordinates. The format supports all VRML property binding conventions.

Keywords**—***Algorithms, compression, graphics.*

I. INTRODUCTION

Although modeling systems in mechanical computer-aided design and in animation are expanding their geometric domain to free-form surfaces, polygonal models remain the primary three-dimensional (3-D) representation used in the manufacturing, architectural, geographic information systems, geoscience, and entertainment industries. Polygonal models are particularly effective for hardware-assisted rendering, which is important for video games, virtual reality, fly through, and electronic mock-up applications involving complex computer-aided-design (CAD) models.

A polygonal model with V vertices and F triangles is usually represented by a *vertex positions array*, a *face array*, and (optionally) one or more *property arrays*. The position of each vertex of the polygonal model is represented in the vertex positions array by three floating-point coordinates. Each face of the polygonal model is represented in the face array by three or more indexes to the vertex positions array.

A polygonal model may also have continuous or discrete properties, such as colors, normals, and texture coordinates, associated with its vertices or triangles. To model various discontinuous phenomena on the polygonal model, properties can also be associated with each vertex of each face. These (face, vertex) pairs are called *corners*, and the corresponding properties are called *corner* properties. These corner properties are represented in the optional property arrays.

In recent years, there has been a rapid growth in the exploitation of the Internet to serve text (hypertext markup language) and image (JPEG, GIF, PNG) content to client browsers. With faster communication links and improving personal-computer graphics, there has been an increasing interest in delivering 3-D content. The virtual-reality modeling language (VRML), where polygonal models are used as the main representation for 3-D content, is the emerging standard for the delivery of 3-D content in a networked environment. However, the bandwidth requirements for 3-D content are significant and have served as a detriment to the wide acceptance of network-delivered 3-D graphics.

In this paper, we address the reduction of the bandwidth requirements for the transmission and storage of VRML files through the use of topologically assisted compression. For example, Fig. 1 details the results of applying topologically assisted compression to a VRML ASCII model. For this example, the file is reduced in size by a factor of 41.72 with absolutely no loss of connectivity information and visually imperceptible loss of geometry information.

Manuscript received July 15, 1997; revised November 30, 1997. The Guest Editor coordinating the review of this paper and approving it for publication was A. M. Tekalp.

G. Taubin, W. P. Horn, and F. Lazarus are with the IBM T. J. Watson Research Center, Yorktown Heights, NY 10598 USA (e-mail: taubin@watson.ibm.com).

J. Rossignac is with the Graphics, Visualization and Usability Center, Georgia Institute of Technology, Atlanta, GA 30332-0280 USA (e-mail: jarek.rossignac@cc.gatech.edu).

Publisher Item Identifier S 0018-9219(98)03524-5.

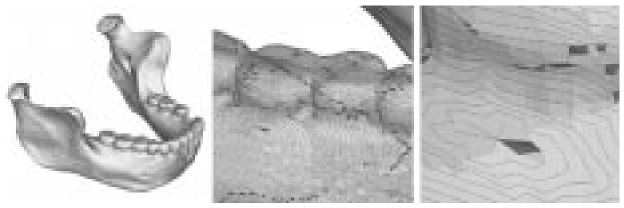
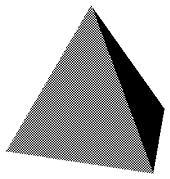


Fig. 1. This model contains 86 939 vertices, 173 578 triangles, one connected component, and no properties. In the standard ASCII VRML format, it requires 8 946 259 bytes of storage. Using 11 bits per coordinate, the file in compressed binary format occupies 214 148 bytes for a compression ratio of 41.72. 65 571 bytes are used for connectivity (3.02 bits per triangle), and 148 590 bytes are used for coordinates (6.84 bits per triangle). The remaining bytes are used to represent the scene graph structure of the VRML file. The edges of the vertex spanning tree, composed of 10 499 runs, are shown as black lines. The triangle spanning tree is composed of 18 399 runs. Leaf triangles are shown in red, regular triangles in yellow, and branching triangles in blue.



```
#VRML V2.0 utf8
Shape {
 appearance Appearance {
  material Material {
   ambientIntensity  0.5
   diffuseColor     0.1837 0.1837 0.1837
  }
 }
 geometry IndexedFaceSet {
  coord Coordinate {
   point [
     0.94  0.00 -0.33 ,
    -0.47  0.81 -0.33 ,
    -0.47 -0.81 -0.33 ,
     0.00  0.00  1.00 ,
   ]
  }
  coordIndex [
    2, 1, 0, -1,
    3, 2, 0, -1,
    1, 3, 0, -1,
    2, 3, 1, -1,
   ]
  color Color {
   color [
    1.00 0.62 0.00,
    1.00 0.00 0.00,
    0.87 0.00 0.87,
    0.37 0.37 1.00,
   ]
  }
  colorPerVertex FALSE
  colorIndex [ 0 1 2 3 ]
 }
}
```

Fig. 2. A simple VRML file.

We start by providing a brief overview of VRML. We then examine related work, concentrating on the topologically assisted compression algorithms of Taubin and Rossignac [9], [10]. Next, we show how we have adapted topologically assisted compression techniques to create our proposal for the VRML compressed binary format. We then apply the compressed binary format to more than 600 VRML models and examine the results. Last, we draw some conclusions and address future work.

II. The Virtual-Reality Modeling Language

The virtual-reality modeling language (VRML) [3], sometimes pronounced verml, is a format for describing and transmitting 3-D objects and worlds composed of geometry and multimedia in a network environment. VRML targets Web applications such as CAD, engineering and scientific visualization, multimedia products, entertainment and educational offerings, and shared virtual worlds. A small VRML file is shown in Fig. 2. VRML has a number of features that make it particularly attractive for authoring virtual worlds. These features include:

- scene graph;
- event processing;
- behaviors;
- encapsulation and reuse;
- distributed content;
- extensibility;
- interactivity;
- animation.

A. Scene Graph

The basic building block for VRML is the *node*. Nodes have *fields*, which serve as attributes and define the persistent state of the node. There are 54 different types of nodes, which can be partitioned into three classes: grouping nodes, children nodes, and attribute nodes. Grouping nodes contain, as an attribute, child nodes. Grouping nodes may contain other grouping nodes. Grouping nodes may contain other grouping nodes as children. These parent relationships define a directed acyclic graph of attributed nodes known as the *scene graph*. The leaf nodes in a scene graph are called children nodes. A grouping node defines a coordinate system for its children nodes relative to its parent coordinate system. The parent relationship provides a sequence of transformations that, when concatenated, positions children nodes in the file's world coordinate space. Attribute nodes serve as attributes for other nodes. So, for example, a material attribute node can be associated to the appearance field of the shape child node.

B. Event Processing

Nodes can both receive messages from and send messages to other nodes. These messages are called *events*. Input events typically alter the state of a receiving node and may trigger behavior. Output events reflect a change in the state of the transmitting node. Events are frequently associated with the setting and changing of a node's fields. The connection between a receiving node and a transmitting node is called a *route*. Routes are not nodes. Like nodes, however, they are defined in the VRML file.

C. Behavior

An author may wish to have his virtual world respond to user input using custom logic. For example, if the user selects the door and the door is not open, then open the door. In VRML, this type of custom logic is supported using a special node known as a script node. The script node is special in that a user may augment it by defining additional events and fields. The uniform resource locator (URL) field of the script node contains program logic. The program logic defines the behavior of the script node. This arrangement permits the script node to send events, receive events, and alter state using customized behavior.

D. Encapsulation and Reuse

VRML supports the definition of new node types, called prototypes, in terms of existing node types. Existing node types may be either built-in or previously defined prototypes. The combination of prototypes and script nodes provides a powerful mechanism to encapsulate content and behavior in a reusable entity.

E. Distributed Content

One of the key features of VRML is its support of the World Wide Web. VRML has several nodes that use URL's to connect the scene graph to the network. These nodes include:

- fetch on demand of additional VRML content (Inline node);
- hyperlinks to other URL's (Anchor node);
- fetch on demand of audio content in uncompressed pulse code modulation (PCM) (wavefile) format or musical-instrument digital interface (MIDI) file type 1 sound format (AudioClip node);
- fetch on demand of texture content in Joint Photographic Experts Group JPEG or PNG, with optional support of CGM and GIF (ImageTexture node);
- fetch on demand of video content in Moving Picture Experts Group (MPEG)-1 systems (audio and video) and MPEG-1 video (video only) movie file formats (VideoTexture node).

The Inline node is particularly powerful in a networked environment since it permits authors to create worlds composed of multiple VRML worlds, which may themselves be composed of multiple VRML worlds. For example, a VRML file of a car—let us call it car.wrl—may be composed of several other VRML files, for example, engine.wrl and door.wrl. The file car.wrl establishes the coordinate system and a hierarchal grouping of its component parts. The included files may also include other files. For example, engine.wrl might inline piston.wrl. Of course, an intelligent browser processing car.wrl would fetch piston.wrl only if it was required for rendering. Fig. 3 sketches the flow of information in a network environment while processing a VRML file.

F. Extensibility

In addition to enabling encapsulation and reuse, the previously mentioned VRML prototype mechanism also enables authors to extend the language by introducing what are essentially new nodes. VRML also supports external prototypes. External prototypes function much like a regular prototype, except instead of residing in the current file, they reside in another URL-identified VRML file. This feature allows developers to extend VRML with logic and content residing and possibly evolving at a specific URL. For example, an author could create an external prototype—let us call it NurbSurface—to model nonuniform rational B-spline (NURB) surfaces. NurbSurface would define fields for the necessary parameters and contain a script node with a URL reference for a Java program. The Java program would interpret the parameters and fill the standard VRML node for describing a surface, the IndexedFaceSet node. Within the scene graph, NurbSurface would appear to behave exactly as an IndexedFaceSet.

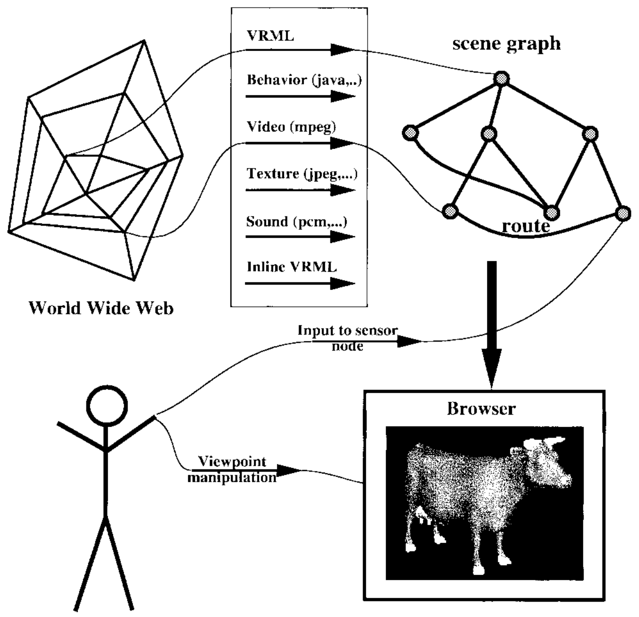


Fig. 3. Flow of information while processing a VRML file.

G. Interactivity

VRML supports a variety of sensor nodes, including environment sensors, pointing-device select sensors, and pointing-device drag sensors. Environmental sensors trigger on a variety of browsing events that might occur while viewing a world. For example, the visibility sensor will trigger an output event when a specific part of the scene graph becomes visible. Pointing-device select sensors trigger based on a user-generated button-up event. For example, an output event is triggered when a specific piece of geometry is selected. Pointing-device drag sensors trigger on a user-generated button-down/drag/button-up sequence. For example, the sphere sensor maps the drag part of the sequence into a spherical rotation output event.

H. Animation

VRML has a variety of interpolator nodes for use in creating linear key-framed animation supporting interpolation in colors, coordinates (arrays of three-tuple floats), normals, orientations, positions (three-tuple float), and scalar floats. Each interpolator node has an input event named *set_fraction*, which triggers an output event named *value_changed*. The event set_fraction defines the key, and the output event value_changed contains a keyed output value of the appropriate type.

III. AN OVERVIEW OF THE VRML COMPRESSED BINARY FORMAT PROPOSAL

The VRML format was designed to be minimal yet complete. It does not suffer from the bloat of earlier scene graph technologies [12] but is still powerful enough to describe complex, animated worlds. Unfortunately, even simple VRML files tend to be quite large (>100 kB). Our proposed VRML compressed binary format [8] addresses this problem by representing the 3-D information contained in a VRML ASCII file in a concise, rapidly parsable binary format with user control over the precision requirements for geometric and property data.

The format combines a binary encoding scheme together with the geometric compression scheme of Taubin and Rossignac [9], [10] to create compact, rapidly parsable VRML files. The format does not require any modifications or extensions to the existing VRML specification and is currently being evaluated by the Compressed Binary Format Working Group of the VRML Consortium.

The binary encoding is a direct transliteration of the ASCII format with a couple of additions to enable the geometric compression of several nodes. It was designed such that the conversion to and from the binary format need not have any effect on the structure of the scene graph. In addition to the transliteration of ASCII data, the compressed binary format supports two compression mechanisms.

One mechanism, referred to as *field compression,* provides for the compression of several multivalue and multivector fields by quantization of coordinates. Another, more interesting mechanism, referred to as *topologically assisted compression,* enables the compression of certain VRML nodes that contain geometry. Topologically assisted compression acts on both the connectivity and property fields of these nodes. The compression of connectivity information is lossless. The compression of property data (vertex coordinates, normals, colors, animated coordinates and normals, and texture coordinates) is optionally lossy and may be controlled by the user.

Several VRML nodes are designed to contain geometry. Some of the geometry nodes are completely specified with small numbers of parameters. For example, the Sphere node requires just a radius. Nodes such as these are already semantically compressed, but the range of shapes that they can describe is rather limited. The workhorse of the geometry nodes is the IndexedFaceSet node or `IFS` for short. Any polygonal shape can be described with an `IFS`. The `IFS` may also specify properties such as normals, colors, or texture coordinates. The ASCII specification of an `IFS` is expensive in terms of storage requirements. A typical VRML file containing an object of a couple hundred faces will have a size of about 30 kbytes. The description of a large virtual world can easily reach 10 Mbytes. As will be shown in subsequent sections, the most dramatic reductions in storage requirements result from the topologically assisted compression of the `IFS`.

IV. Related Work on Geometric Compression

There are several options for representing and storing geometric models. Options include boundary representations, constructive solid geometry (CSG) methods, voxel representations, and various polygonal models. When evaluating these alternatives for use in a particular application, several issues should be considered. For example, if the data are stored for use by a solid modeler, then perhaps the best selection is the CSG model. Alternatively, when the application primarily involves view-only access, polygonal models with their simplicity and popularity are frequently chosen. VRML is based on a polygonal model. The VRML polygonal model is simply a list of faces, with each face described by a sequence of vertex references. Each vertex reference points to a separately stored triple of floating-point numbers representing the coordinates in three space. The VRML polygonal model is simple, and, although the vertex referencing scheme is somewhat compact, there is room for a considerable reduction in model size. One easy change is to use a binary rather than an ASCII representation.

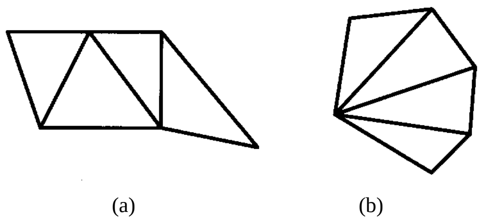


Fig. 4. (a) Triangle strip. (b) Triangle fan.

Another standard technique involves removing redundant vertex references. In VRML, several topologically adjacent faces store the same vertex index. Models composed strictly of triangles can be compressed by constructing triangle strips and triangle fans. As shown in Fig. 4, a triangle strip is a chain of triangles where each new vertex reference implicitly defines a new triangle.

The trailing edge of the previous triangle is used with the incremental vertex index to form the next triangle in an alternating, "zigzag" fashion. A triangle fan is a similar structure except the chain of triangles is constructed around one common vertex instead of alternating left and right. As shown in Fig. 5, by including a swap operation (one bit per triangle), triangle strips and fans may be combined into a single structure called a generalized triangle strip. If we assume that we are dealing with a closed triangulated surface of low genus, then if there are n vertices, there will be approximately $2n$ triangles. If we can turn such a manifold into a small number of generalized triangle strips, then the storage requirement will be approximately $2n(\log(n) + 1)$ bits.

Deering's [5] generalized triangle mesh extends the generalized triangle strip by adding a 4-bit address buffer along with a couple of new operations. The 4-bit address buffer enables the addressing of the 16 most recently visited vertices, and the new operations permit direct access to these vertices. In the compressed format proposed in [5], the topological information is lost so that it is not easy to compare the storage requirement with techniques that preserve the topology. Bar-Yehuda and Gotsman [2] have performed a general analysis on the use of buffers for rendering triangular meshes.

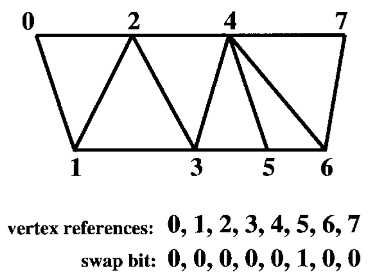

0
2
4
7
1
3
5
6
vertex references: 0, 1, 2, 3, 4, 5, 6, 7
swap bit: 0, 0, 0, 0, 0, 1, 0, 0

Fig. 5. A generalized triangle strip.

The generation of optimal generalized triangle strips and generalized triangle meshes is a challenging computational geometry problem. In fact, the generation of an optimal generalized triangle strip (a Hamiltonian path of triangles) has been shown to be an NP-complete problem [1]. Heuristic approaches have been proposed for the generalized triangle strip [6]. However, we are not aware of any published work on the generation of generalized triangle meshes.

The main goal in both [2] and [5] is to design a storage format that minimizes the amount of computation required for the rendering process. Assuming that most of the time is spent processing vertex coordinates for projection and clipping operations, the rendering cost is proportional to the number of vertices sent to the graphics engine. Optimally, each vertex should be sent once and only once. In [2], it is proved that this requires a buffer of size at least $1.649\sqrt{n}$ and at most $12.72\sqrt{n}$ for a triangle mesh with n vertices. The topology can be preserved, but addressing a pushed vertex still requires $O(\log n)$ bits and, as a result, storing a whole mesh with this format will require $O(n \log n)$ bits.

Another class of methods, including the one used in the current work, starts with a vertex spanning tree. Turán [11] shows that a planar graph and, consequently, faces and edges of a closed surface with genus zero can be encoded using $12n$ bits. As will be shown in Section VII, the method used in the current work encodes the topology for large triangulated polygonal models with storage requirements approaching 1 bit per triangle.

V. Topologically Assisted Compression

In this section, we briefly describe the method of Taubin and Rossignac for representing triangular meshes in compressed form (Fig. 6) [9], [10].

To develop the basic concepts, we will first concentrate on *simple* triangular meshes, that is, triangulated connected oriented manifolds without boundary, of Euler characteristic two, and without properties (normals, colors, or texture-mapping coordinates). Examples of simple and nonsimple meshes are shown in Fig. 7.

A. Simple Mesh

Let us assume that we have a simple mesh composed of V vertices, E edges, and T triangles. If a *vertex spanning tree* is constructed on the graph defined by the vertices and edges of the mesh, and if the mesh is cut through the edges of the vertex spanning tree, the result is a triangulated simply connected *polygon*. We make the following observations.

1) The result of cutting through the vertex spanning tree is a connected oriented triangular mesh.
2) The boundary of the mesh forms a single *bounding loop* of edges and there are no internal vertices.
3) Each edge of the vertex spanning tree corresponds to exactly two boundary edges of the new mesh.
4) A spanning tree of V nodes has exactly $V-1$ edges, and so the bounding loop has $2V-2$ edges and vertices. Therefore, the resulting mesh has $2V-2$ vertices, $E+V-1$ edges, and T triangles. The Euler characteristic is equal to $(2V-2)-(E+V-1)+T = (V-E+T)-1 = 1$.

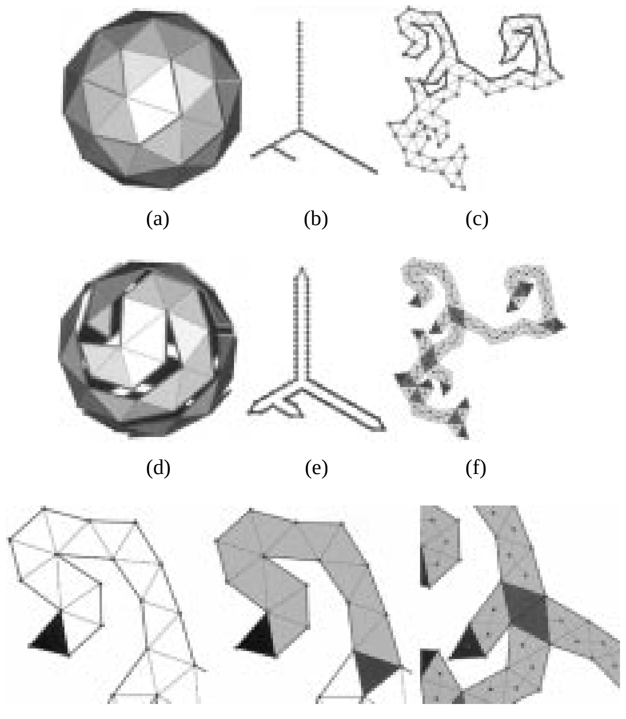


Fig. 6. Representation of a simple mesh in compressed form. The vertex spanning tree (a), (b) is composed of vertex runs. Cutting through the vertex tree edges produces a topological simply connected polygon (c), (d). The bounding loop (e) is the boundary of the polygon. The dual graph of the polygon is the triangle spanning tree (f). Triangle runs end in leaf or branching triangles. The triangle spanning tree has a root triangle (g). Marching edges (h) connect consecutive triangles within a triangle run. Each branching triangle has a corresponding Y-vertex. Two consecutive branching triangles are represented as a run of length one (i).

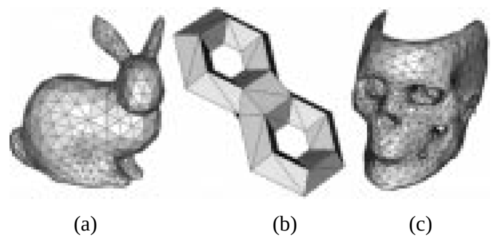


Fig. 7. Examples of (a) simple and (b), (c) nonsimple meshes.

5) Every connected oriented manifold triangular mesh of Euler characteristic equal to one is homeomorphic to a topological disk [7].

1) Vertex Spanning Tree: The branching nodes and the leaf nodes of the vertex spanning tree decompose the tree into *vertex runs*. A vertex run is a sequence of edges connecting a starting leaf or branching node to subsequent regular nodes and ending in a leaf or branching node. The vertex spanning tree is represented as an array of triplets, each triplet is composed of a *length,* a *branching bit,* and a *leaf bit,* and describing a vertex run. These triplets are ordered according to when the corresponding runs are visited during depth-first traversal of the tree starting from a root leaf node. Runs with a common first node are ordered according to the global orientation of the mesh. The length is the number of edges of the run, the branching bit indicates whether or not the run is the last run sharing the *current branching node,* and the leaf bit indicates if the run ends in a leaf or branching node. This representation efficiently encodes the structure of the vertex spanning tree. To increase the compression ratio, the compression algorithm attempts to build vertex spanning trees with the least number of runs

2) Bounding Loop: The *bounding loop* is constructed during the recursive traversal of the vertex tree and is represented by a table of $2V-2$ vertex indexes. References to vertices encountered going down the tree are added to the table during the traversal. Except for leaf vertices, these references are also pushed onto a stack. The two bits (*branching bit* and *leaf bit*) that characterize each run of the vertex tree are used to control the tree traversal and the popping of the stack. When a leaf is visited, references are popped from the stack and added to the bounding loop table until the reference to the branching vertex where the next vertex run starts is popped or until the stack is exhausted. Since it can be derived from the structure of the vertex spanning tree, the bounding-loop lookup table is not included in the compressed representation of the mesh. However, both compression and decompression algorithms must construct the lookup table.

3) Triangle Spanning Tree: The dual graph of the polygon forms a binary spanning tree of the triangles of the mesh, which can also be decomposed into runs. This *triangle spanning tree* is encoded in the same way as the vertex spanning tree. Because the triangle spanning tree is binary, however, it is sufficient to store the length of each triangle run and the leaf bit. The root triangle of the triangle spanning tree is identified by the bounding loop index of its tip. Together, the vertex and triangle spanning trees permit the recovery of the length and boundary of each triangle run and the vertices that bound each triangle.

Traversing a triangle run along the direction that corresponds to a top-down traversal of the triangle spanning tree defines the left and the right boundaries. Because the left and right boundaries of each triangle run form connected subsets of the bounding loop, the boundary of each run can be recovered if the two starting vertices (one on each side) and the number of vertices along the left and right boundary of the run are known.

4) Marching Edges: The internal edges of the polygon are called *marching edges*. Within each triangle run, each marching edge shares a vertex with the previous marching edge in the run. That shared vertex could lie on the left or on the right boundary. A single bit of information per marching edge is used to encode the correct side. These bits are concatenated in the order in which the corresponding marching edges are visited by the decompression algorithm. They form what we call a *marching pattern* of left or right steps. $N-1$ marching bits are needed to encode the triangulation of a triangle run of length N.

5) Y-Vertices: If a triangle run ends at a branching triangle, the next vertex is not adjacent to the marching edge along the loop. However, the bounding-loop indexes that identify these *Y-vertices* need not be stored. They are derived by a simple preprocessing step of the decompression algorithm. Indeed, the distance along the loop from either the left or right vertices to a Y-vertex can be derived from the triangle spanning tree independently of the marching pattern.

For computational convenience, the Y-vertices are identified not by the absolute index in the bounding-loop lookup table but by their offset (topological distance along the bounding loop) in that table from the reference to the last vertex of the left boundary of the corresponding triangle run. These offsets are precomputed and stored in the *Y-vertex lookup table*.

For each branching triangle, the distance along the loop from either the left or the right vertices to the Y-vertex, the *left branch boundary length* and *right branch boundary length*, respectively, can be computed by recursion. The length of the boundary of a branch starting with a run of length n is equal to $n+n_L+n_R-1$, where n_L and n_R are both equal to one if the runs end at a leaf triangle and equal to the left and right branch boundary lengths of the branching triangle if the run ends at a branching triangle.

The branch boundary lengths are computed for each branch as a preprocessing step of the decompression algorithm and stored in a table. When a branching triangle is encountered during the triangle reconstruction phase, the identity of the corresponding Y-vertex can be determined by adding the left branch boundary length to the loop index of the left vertex. Because of the circular nature of the bounding loop table, this addition is performed modulo the length of the bounding loop.

6) Compression of Vertex Positions: Because proximity in this vertex spanning tree often implies geometric proximity of the corresponding vertices, we can use ancestors in the tree to predict vertex positions and thus only need to encode the difference between predicted and actual vertex positions.

When vertex coordinates are quantized by truncating the coordinate to the nearest number in a fixed-point representation scheme, these corrective vectors have on average smaller magnitude than absolute positions and can therefore be encoded with less bits. Furthermore, the corrective terms are then compressed by entropy encoding using Huffman coding [4].

Within the vertex tree, there is a unique path from each vertex to the root. The *depth* of a vertex is the length of this path, with the depth of the root vertex equal to zero. A bounding box containing all the vertex positions is used to define the quantization. If v_n denotes the result of quantizing to B bits the normalized relative position of a vertex of depth n within the bounding box, then each vertex position v_n is defined by

$$v_n = \epsilon(v_n) + \sum_{j=1}^{K} \lambda_j \, v_{n-j} \tag{1}$$

where $\epsilon(v_n)$ is the *vertex position correction* associated with that vertex, the sum is the vertex positions predictor function, K and $\lambda = (\lambda_1, \cdots, \lambda_K)$ are parameters for the predictor, and $v_{n-1}, \ldots, v_{n-K}$ are the K ancestors of the vertex along the unique path to the vertex tree root. Note that since the top vertices of the tree may not have K ancestors, we define vertex positions corresponding to negative depth as equal to the position of the vertex tree root. The vertex position corrections (integer values) are represented concatenated according to the vertex tree preorder and, as mentioned above, further entropy encoded.

7) Compression Algorithm: Compressing a simple mesh is performed with the following steps:

1) constructing the vertex and triangle spanning trees;
2) encoding the vertex tree;
3) compressing the vertex positions;
4) encoding the triangle tree;
5) computing and encoding the marching pattern.

8) Decompression Algorithm: Decompressing a simple mesh proceeds using the following steps:

1) decoding the vertex tree;
2) reconstructing the table of vertex positions;
3) constructing the bounding loop (lookup table pointing to the vertex position table);
4) computing the relative indexes for Y-vertices in the order in which they will be used;
5) reconstructing and linking of triangle runs.

9) Construction of Spanning Trees: Several methods for constructing the spanning trees are described by Taubin and Rossignac [9], [10]. The one that produces the best results performs a layered decomposition of the mesh and an incremental construction of both trees. Intuitively, this process mimics the act of peeling an orange by cutting concentric rings, cutting the rings open, and joining them as a spiral. This process is illustrated in Fig. 8(a), (c), and (e). A vertex is chosen as the root of the vertex tree. The singleton consisting of the root vertex is the first boundary. The nth triangle layer is the set of triangles that are incident to one or more vertices of the nth boundary but do not

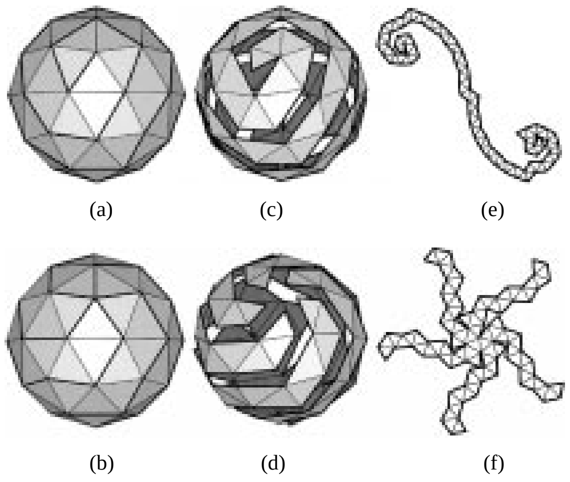


Fig. 8. Two ways of peeling an orange. (a), (b) The thick edges are the edges of the vertex tree constructed on the mesh. (c), (d) The mesh is cut through the vertex tree edges (the vertex positions have been modified here only to illustrate the creation of the cut). (e), (f) The result is a topological simply connected polygon. The dual graph of this polygon is the triangle tree.

belong to a previous triangle layer. The $(n+1)$st boundary consists of all the edges of triangles of the nth layer with neither of the two end vertices belonging to the nth layer. The boundary edges do not constitute a tree, but most typically each boundary is composed of one or more cycles. The layers are also typically composed of cyclical triangle paths. This construction can incrementally generate both trees by converting the rings into a spiral. Let us assume that a vertex tree has been constructed with all the vertices included in the first n boundaries and that a triangle forest has been constructed with all the triangles included in the first $n-1$ layers. For each connected component of the $(n+1)$st boundary, one edge connecting that component to a vertex of the nth boundary is chosen and added to the vertex tree. All these cross edges are chosen minimizing the number of new branches added to the two trees. Then, the edges of the $(n+1)$st boundary are included in the vertex tree after removing a minimum number of edges to maintain the tree structures. These edges are also chosen minimizing the number of new branches. Fig. 9 illustrates this construction.

B. More General Meshes

Triangular manifold meshes of Euler characteristic other than two, nonorientable, and with boundaries require minor extensions to the representation, compression, and decompression algorithms. The compressed representation of meshes with multiple connected components consists of the concatenation of the compressed components, perhaps with common compression parameters (bounding box, number of bits per vertex coordinate, number and value of predictor coefficients, and Huffman encoding tables).

1) Arbitrary Euler Characteristic: When a connected oriented manifold without boundary is cut through the edges of the vertex tree, the resulting mesh is a triangle graph and not necessarily a simply connected polygon. However,

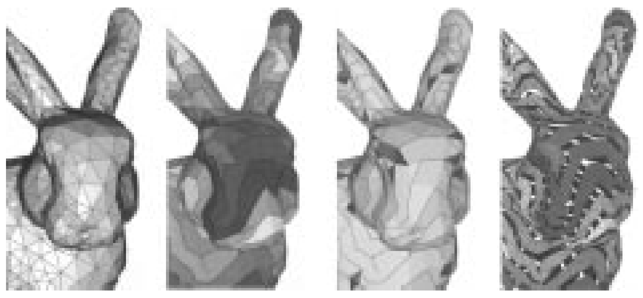


Fig. 9. Construction of spanning trees. (a) Triangular mesh. (b) The topological distance from a chosen vertex defines the layers. (c) Vertex tree and triangle tree are constructed by traversing the layers in order. (d) Polygon resulting of cutting along cut edges with artificial gap introduced. Triangles are color coded according to their corresponding layer.

a simply connected polygon can be obtained by making $2-\chi$ extra cuts along *jump edges*, where $\chi = V - E + T$ is the Euler characteristic of the original mesh.

To find the jump edges, a triangle spanning tree is constructed on the triangle graph. The jump edges are then found by selecting the edges not crossed by this triangle spanning tree.

The representation defined for simple meshes is extended to account for the jump edges by using a new table with one entry per jump edge. Each entry indicates the number of edges in the bounding loop it short-circuits. In this extended representation, regular (nonbranching and nonleaf) triangles of the triangle tree incident to a jump edge are treated as branching triangles with one run of length zero starting at the jump edge. Leaf triangles, including the root triangle, may be incident to zero, one, or two jump edges. Furthermore, in the case of one jump edge, it may be the one incident to either the left or the right vertex. This is encoded with two extra bits per leaf of the triangle tree in the marching pattern.

2) Meshes with Boundary: The representation is the same as for connected oriented manifold without boundary described above, except that some edges of the bounding loop will have no incident triangles. To ensure that after cutting through the vertex spanning tree the resulting mesh is connected, it is sufficient to include all but one of the edges of each connected boundary component in the vertex spanning tree and to treat the remaining boundary edges as jump edges.

3) Nonorientable Meshes: In the orientable case, when a jump edge is crossed, the loop path encountered after the jump is traversed in the same direction as the one before the jump. In the nonorientable case, the direction of loop traversal may or may not change across a jump edge. An extra bit per jump edge is added to the marching pattern to represent changes of direction.

VI. Topologically Assisted Compression and the VRML Compressed Binary Format

The discussion in the previous section was restricted to surfaces described by triangular meshes. In this section, we extend the compression technique to handle polygons. Also in this section, we address the property-binding models found in VRML. The encoding of VRML property data is complicated by the existence of several options for binding property data. These options include indexed, not indexed, per face, per vertex, or per corner. For each of the bindings, we define an ordering for the storage of property values. Last, we introduce a new scheme to encode corner properties compactly for corners sharing both a common vertex and property value.

A. From Triangles to Polygons

The topology of an `IFS` is stored in its coordIndex field. The coordIndex field contains a sequence composed of indexes and "-1"'s. The -1's partition the sequence into subsequences. Each subsequence of indexes describes a simply connected polygonal face composed of three or more indexes.

We refer to an arbitrary triangulation of a face as a *topological triangulation*. A topological triangulation does not take into account the geometric position of the component vertices. We refer to the additional edges used to triangulate a face as *nonpolygonal edges*. The other edges are called *polygonal edges*.

To extend the compression algorithm to polygonal surfaces, we construct a spanning tree on the polygonal surface. A spanning tree constructed in this manner is simply a vertex tree consisting of only the polygonal edges. We then topologically triangulate each polygonal face to obtain a triangular surface. This construction insures that every nonpolygonal edge will be an interior (marching) edge. Moreover, the interior edges can be implicitly ordered by a depth-first traversal of the triangle trees. Therefore, we can use a bitstream with as many bits as the number of interior edges to differentiate between polygonal and nonpolygonal edges. This surface can be compressed using the scheme discussed in the previous section. The compressed triangulated surface, together with this bitstream, provides sufficient information to recover the polygonal surface.

Polygons are recovered by first reconstructing the triangular surface and subsequently removing the extra nonpolygonal edges. Removing extra edges means merging adjacent triangles. We refer to two triangles in a triangle tree as being *polygon connected* if they share a common nonpolygonal interior edge. Each polygon-connected component corresponds to a polygon to be recovered.

As shown in Fig. 10, a polygon may spread over several triangle runs. As explained in Section V, the reconstruction of the triangular mesh is accomplished using a depth-first traversal of the triangle runs. Within a run, triangles are recovered by advancing the current vertex index on the left or on the right according to the marching pattern. Since a polygon may spread over several triangle runs, we cannot bound the recovery of a polygon to an individual run. However, we will show that it is still possible to reconstruct all the polygons in one linear traversal of the triangle tree.

We refer to a connected subset of the boundary of a polygon as a *polygon segment*. A polygon partially covering

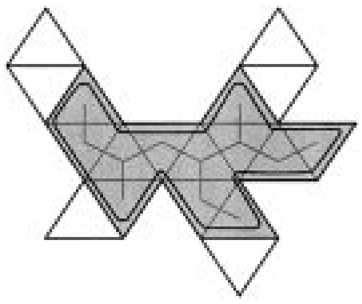


Fig. 10. One can think of a polygon as an amoeba constrained to stay in the interior of the triangle tree.

a run intersects the left (or right) side of the run in a single polygon segment. Using this construction and paying special attention to branching and leaf triangles, the boundary of a polygon is partitioned into polygon segments.

We describe our polygon reconstruction algorithm using left and right polygon segment stacks to store partially recovered polygons. In the actual implementation, the polygon segments are reconstructed using two stacks: a vertex stack and a flag stack. The vertex stack is used to store the polygon vertex indexes. The flag stack is processed in parallel with the vertex stack. The flag stack is used to indicate the beginning and the end of the polygon segments.

Using a depth-first traversal, the algorithm processes the triangle tree run by run, pushing vertices onto, as appropriate, the left or right stacks to form left and right polygon segments. When a new polygon is encountered, the first vertices are marked for future reference. If a polygonal edge is encountered, then the right polygon segment is popped from the right stack onto the left stack. If the popped right segment contains a marked vertex, then a polygon has just been completed and the left stack is popped to form a new polygon. Fig. 11(a)–(k) illustrates the recovery of polygons contained in the triangle tree of Fig. 11(l). The triangle tree is composed of three runs, and its root triangle is marked "root." In Fig. 11, arrows are marked $\mathbf{Li}$ or $\mathbf{Ri}$, where $\mathbf{L}$ and $\mathbf{R}$ indicate an extremity of the left or right edge for the $\mathbf{i}$th event. The events correspond to start a new run, pop a stack, and push a new polygon segment. The ordering is consistent with the depth-first traversal. By convention, the first visited edge is the right edge of the root triangle.

The reconstruction sequence begins with Fig. 11(a). Here, one vertex is pushed on both left and right vertex stacks. The vertices are represented as unfilled circles to emphasize the fact that a new polygon is starting. We call such vertices *start vertices*. In the actual implementation, this event would be recorded by pushing a start flag on both the left and right flag stacks. When we encounter polygon edge $(\mathbf{L2}, \mathbf{R2})$, the polygon segment in the right stack is popped onto the left stack. This event is represented by the arrow marked "1" in Fig. 11(b). Since the right polygon segment ends with a start vertex, we know that a polygon has just been completed. The new polygon is formed by popping a segment off the left stack. This event is represented by an arrow marked with a "2" on Fig. 11(c). Fig. 11(d) shows the state of the stacks after processing the remainder of the first run. Two new (blue) polygon segments, one on the left and one on the right, have been pushed onto the stacks.

Fig. 11(e) shows the initial processing of edge $(\mathbf{L4}, \mathbf{R4})$. Here, one vertex is pushed on the left stack to augment the left segment, while a new segment with a single vertex is pushed on the right stack. Since the right edge of the branching triangle has not yet been visited, we do not know at this time whether or not the vertex forms a contiguous segment with the previous right segment in the stack, and we classify the vertex as a separate segment. Since $(\mathbf{L4}, \mathbf{R4})$ is a polygon edge, the right top segment (a single vertex) is popped onto the left stack. This event is shown in Fig. 11(f).

Next, in Fig. 11(g), we start the pink polygon by processing edge $(\mathbf{L5}, \mathbf{R5})$. Since the run ends in a leaf triangle, we can connect the current left and right segments. As shown in Fig. 11(h), this is accomplished by pushing the tip of the leaf triangle onto the left stack and then popping the right stack onto the left stack, as indicated in the figure by an arrow marked with a "1." Since popping the right stack yields a start vertex, we can reconstruct a polygon. In the same figure, the arrow marked "2" indicates the popping of the left stack to reconstruct the polygon.

A new run starts with edge $(\mathbf{L6}, \mathbf{R6})$. This edge is a polygon edge, and, once again, we connect the top left and right segments. Since the right segment contains a start vertex, we also reconstruct a polygon as shown in Fig. 11(i). Fig. 11(j) and (k) shows the final steps in the reconstruction algorithm.

The ordering of faces in the decompressed `IFS` is important for property recovery. We always order the faces according to the order in which the face's starting edge is encountered during a depth-first traversal. However, the reconstruction algorithm outputs a face when its last edge is visited. Fortunately, only minor modifications are required to modify the ordering. Fig. 12(a) shows the corner ordering in the output coordIndex for the example used in Fig. 11.

B. Geometry and Property Encoding

Geometry encoding refers to the encoding of vertex coordinates (x, y, z). As such, geometry can be considered as a particular property, that is, floating-point values attached to vertices. We support three types of property bindings. Properties may be attached to the vertices, the faces, or the corners of the polygonal surface. Furthermore, there are two options for specifying a property value. One option is to explicitly encode the property value; the other is to specify an index into a palette of values. Last, there are four basic types of property values: floating-point values, integer values, color values (a tuple of three floating-point values in $[0, 1]$), and normal values (three-dimensional unit vectors).

To establish a connection between property data and the recovered topology, we have to define an implicit ordering on the occurrences of the appropriate feature: vertices, faces, or corners. By storing the properties using

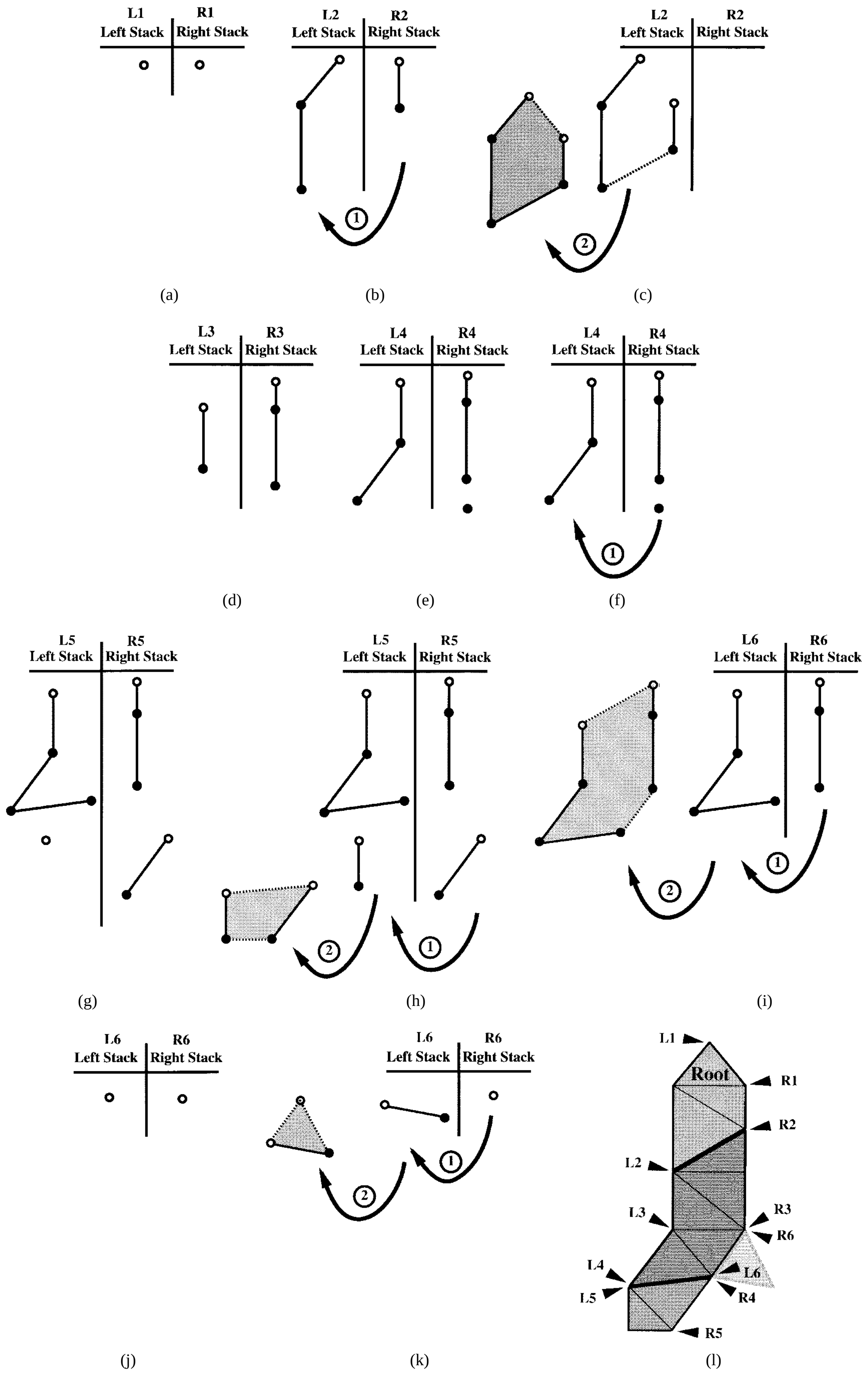
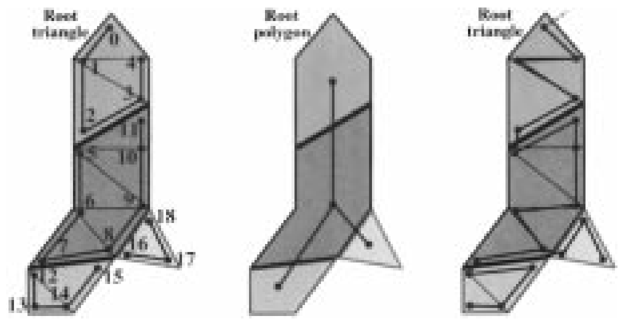


Fig. 11. Schematic view of the polygon reconstruction.



(a) (b) (c)

Fig. 12. Implicit ordering and parent relationship from the triangle tree. (a) The corner ordering. (b) The face tree. (c) The corner tree.

this implicit ordering, it is possible to attach property values to their proper location. Hence, the property binding dictates the order in which property values are stored. Since the ordering is derived from the topology encoding, we say that our compression technique is *topologically assisted*. The ordering for:

- vertices is obtained from the depth-first traversal of the vertex tree;
- faces is the same ordering as used in the output coordIndex field during recovery;
- corners is the same ordering as used in the output coordIndex field.

1) Floating-Point Value Encoding: The standard binary encoding for IEEE floating points uses 32 bits. For some property/topology combinations, it is possible to use as few as 4 bits per value with only minor losses in accuracy. To compress floating points, we combine a delta encoding scheme [5] with a predictor/corrector model. Basically, we integerize the floating-point values and then apply a linear predictor/corrector. As was discussed in Section V-A, we use an evenly subdivided bounding box enclosing the set of values to integerize the floats. When the property is a tuple of values rather than a single value, we use a bounding box of the same dimension as the tuple (three for the coordinates). For coordinate data, our experience has been that a precision of 8–12 bits is sufficient. An 8-(12)-bit precision corresponds to a bounding box with 256 (4096) elements per side.

The predictor/corrector model requires a parent relationship. The parent relationship is implicitly defined using orderings defined for each binding type. Basically, for each binding type, the spatial coherence of nodal ancestors derived from the parent relationship is used to predict values.

When the property is bound to the vertices, the parent relationship is directly obtained from the vertex tree. When the property is bound to the faces, the face parent relationship is derived from the dual of the triangle tree after removing the nonpolygon edges, as shown Fig. 12(b). This ordering is the same face ordering used for the coordIndex field. When the property is bound to the corners, we use a slightly different scheme. The path of the corner ordering defined by the coordIndex field will, in general, not lend itself to the predictor/corrector model. This could lead to poor results. Instead, we use a corner ordering on corners in the triangle tree to induce an ordering on the face corners. When all faces are triangles, the corner ordering is based on the following pattern:

When the mesh contains nontriangular faces, we use the same scheme and contract the corner ordering by skipping over "redundant" or previously visited corners. An example showing the ordering along with our conventions for root and branching triangles is shown in Fig. 12(c).

2) Integers: When the integer values possess some spatial coherence, we use a predictor/corrector scheme identical to the one defined for the floating-point numbers. Otherwise, we encode the values using $\log n$ bits per value, where n is the greatest integer value.

3) Color Encoding: Colors are integerized and stored with a user-specified fixed number of bits.

4) Normal Encoding: Normals are quantized with a subdivision scheme that produces an optionally lossy compression of three-dimensional unit length vectors.[1] A normal is encoded into a sequence of $3 + 2n$ bits, where n is the number of discrete normals in one quadrant. The first three bits of the coded normal determine the normal's octant. Each octant is spanned by a base triangle defined by the three canonical vectors. An octant is discretized by recursively subdividing the base triangle n times, as shown Fig. 13(a)–(c). A normal is encoded using the triangle's number, where the triangles are enumerated as shown in Fig. 13(c).

5) Compact Storage of Corner Properties: Since each face has three or more vertices, the number of corners on a surface is at least three times the number of faces. Furthermore, from Euler's formula, the number of faces is at least twice the number of vertices (up to the characteristic of the surface). Therefore, the number of corners is at least six times the number of vertices. Consequently, corner-based properties require significantly more storage than vertex-based properties:

- from previous triangle, cross over edge to adjacent corner;
- traverse cross-over edge to second corner;
- proceed to third corner;
- cross over edge into next triangle.

To alleviate the storage requirements for corner properties, we have devised a scheme to efficiently handle corner properties shared around a common vertex. Operating around the star of a vertex, the scheme stores one property for each collection of connected corners sharing a common property with an additional cost of one *discontinuity* bit per corner. The discontinuity bits are put into a bitstream in the order defined by the coordIndex field. The discontinuity bits are assigned as follows. First, we give an orientation to each vertex star. We then cyclically visit every corner in the star of a vertex, associating a "1" to a

[1] This encoding scheme was developed with F. Pettinati.

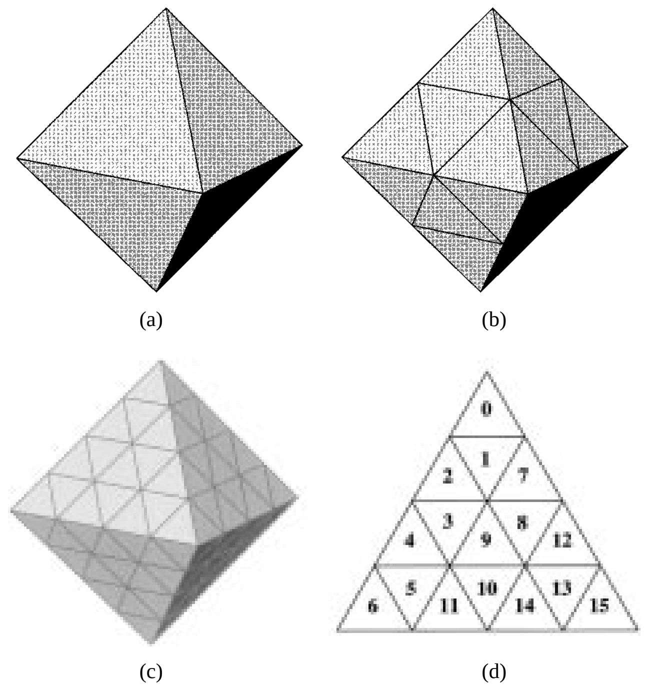


Fig. 13. Three subdivision levels of the base octahedron. (a) Subdivision level 0. (b) Subdivision level 1. (c) Subdivision level 2. (d) Triangle enumeration for level 2.

corner whenever the prior corner has a different property. Otherwise, we assign a "0." Fig. 14 shows a vertex shared by six faces with different properties. The "0"'s and "1"'s in the figure are values from the discontinuity bitstream. A "1" indicates that a property value is associated with this corner. A "0" indicates that a corner should obtain its property value from the first "1" corner encountered by cycling counterclockwise.

VII. Results

In this section, we will examine the application of our compression algorithm to a test suite of VRML models. As mentioned previously, there are several parameters available to control the degree of lossiness during compression of coordinates, colors, normals, and texture coordinates. Since the selection of these parameters affects the visual quality of the final model, it is difficult automatically to choose optimal parameters. In the absence of good heuristics, it appears that the best solution is interactively to select the compression parameters at the time a world is saved in compressed form. Additionally, options are available for sharing common bounding boxes and specifying predictor coefficients.

The compression and decompression times are not the same. The compression algorithm is significantly more time consuming than the decompression algorithm, but not prohibitively so. The time required to compress an ASCII VRML file and to store the result as a compressed binary file is comparable to the time needed to parse the ASCII VRML file and to generate the corresponding in-core representation. The time to decompress and to build the corresponding in-core representation is not only

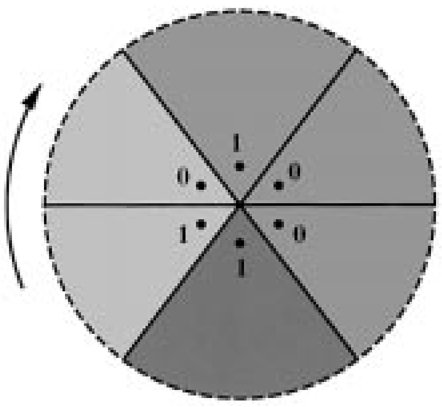


Fig. 14. The property star of a vertex and the discontinuity bits associated to the corners.

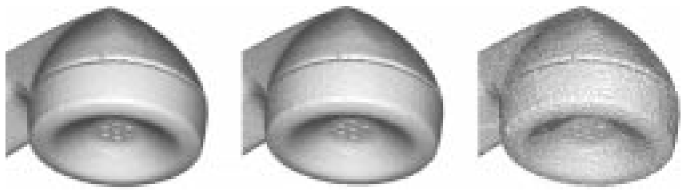


Fig. 15. (a) The original model. (b) The same model quantized to 11 bits per coordinate. (c) The same model quantized to 9 bits per coordinate.

significantly smaller than the compression time but also significantly smaller than the time required to transmit the corresponding compressed binary file across an ISDN telephone link. Since the latter can be easily computed from the compressed binary file size, we have not included statistical results in this paper.

A. Coordinate Quantization

The sequence of images on Fig. 15 shows how the geometry of one particular model (without properties) changes as a function of the number of bits per vertex coordinate. Similar effects can be observed by varying the number of subdivision levels per normal, the number of bits per color component, and the number of bits per texture coordinate.

B. Bounding Boxes

Another option involves selecting a strategy for grouping geometry with respect to bounding boxes. In our current implementation, the user may choose either to use a single bounding box to quantize all the `IFS` vertex coordinates contained in a file or to use one bounding box per `IFS`. In general, but not always, the first option produces a smaller compressed file. In Fig. 16, the teeth of the crocodile are grouped into two `IFS`, one for the upper teeth and another for the lower teeth.

The side of the smallest bounding box for the whole object is about four times larger than the side of the bounding box for the teeth. By using the same number of bits per vertex coordinate but with one bounding box per `IFS`, the teeth vertex coordinates are specified with four

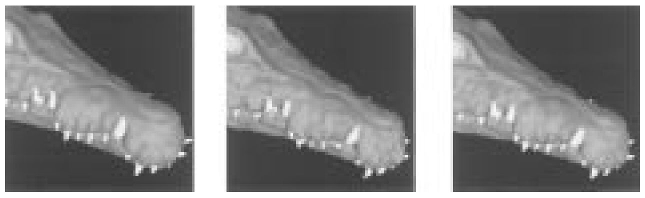


Fig. 16. (a) The original model. (b) The same model quantized to 9 bits per coordinate using one common bounding box for the head and the teeth. (c) The same model quantized to 9 bits per coordinate using a separate bounding box for the head and the teeth.

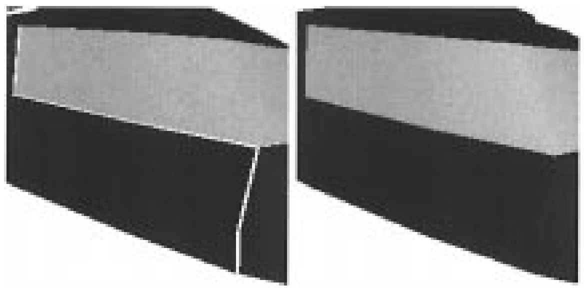


Fig. 17. (a) Model quantized with one bounding box. (b) The same model quantized with multiple bounding boxes.

times the precision. Because the overall number of vertices in the teeth is much smaller than the total number of vertices in the rest of the body, the cost is not significant. In fact, for this particular case, the compression ratio is slightly better for the second case. This is most likely due to a discrepancy in the distribution of prediction errors between the teeth and the rest of the body. This example shows that there is opportunity for optimization that is not exploited in our current implementation.

Care should be taken to prevent the creation of cracks when choosing to use one bounding box per `IFS`. The bounding box, along with the number of bits per vertex coordinate, defines a rectangular grid in 3-D. The quantized vertices reside at the nodes of this grid. To prevent the creation of cracks, the individual bounding boxes and the number of bits per coordinate should be chosen such that the bounding-box grids match in 3-D. In our current implementation, however, this is not possible. Currently, the user can only specify one bounding box per file or one bounding box per `IFS`. In the former case, the smallest bounding box containing all `IFS` coordinates is chosen; in the latter case, the smallest bounding box containing the coordinates of the individual `IFS` is selected. Fig. 17 shows cracks caused by using one bounding box per `IFS`. Cracks are created only when two vertices that coincide in 3-D in the original uncompressed geometry are quantized using combinations of bounding boxes and numbers of bits per coordinate that produce nonmatching grids.

C. Predictor/Corrector Options

The number of ancestors used to predict vertex coordinates does not have any effect on the quality of the geometry, but it does affect the compression ratio. Ideally, the compression algorithm should estimate the optimal number of ancestors. Our current implementation does not perform this optimization; instead, the number of ancestors may be specified by the user and defaults to two. In general, using two or more ancestors reduces the compressed binary size by 10–15% with respect to using only one ancestor, but there may be exceptions. Using more ancestors may decrease or increase the size of the compressed file. Usually, using more than four or five ancestors does not produce a significant enough reduction in size to warrant the extra computation effort on decompression.

A possible explanation for this behavior is that, in our implementation, the compressor does not attempt to calculate the predictor coefficients that would minimize file size. Instead, it uses a suboptimal strategy of minimizing the average square prediction error. This strategy was chosen because selecting the number of ancestors to minimize the file size is a difficult combinatorial optimization problem, while minimizing the average square prediction error involves a simple explicit solution based on matrix computations. Again, there is potential for improvement.

D. Examples

Our geometric compression algorithms require each `IFS` to be represented by a manifold surface with one or more connected components. A significant number of VRML files do not meet this requirement. To overcome this problem, we have developed a method to convert a singular (nonmanifold) `IFS` into a manifold surface without modifying the geometry. This work will be presented in a forthcoming paper.

We have tested the compression algorithm on more than 600 VRML files. The models came from four sources:

- *http://www.microsoft.com/vrml/stack:* more than 200, relatively small, Viewpoint 3-D models in four categories: nature, architecture, transportation, and accessories;
- *http://www.acuris.com/free_25.htm:* some 23 models, both VRML1 and VRML2.0, from the Acuris Web site;
- *http://www.3dcafe.com/meshes.htm:* more than 100 VRML1 models downloaded from the 3D CAFE 3D Model library;
- *http://www.ocnus.com/models/models.html:* more than 300 VRML1 models from the Ocnus Rope Company VRML repository.

The geometry content in these files ranges from a couple to more than 10 000 faces. Before compressing these files, the models were converted, if necessary, from VRML1 to VRML97. Also, we transformed any singular `IFS` into nonsingular `IFS`.

Fig. 18 uses a graph with logarithmic scales to illustrate the efficiency of the topology encoding. The plot shows the ratio of the size of the binary compressed topology (without geometric or property information) divided by the number of faces. For large models, this ratio approaches

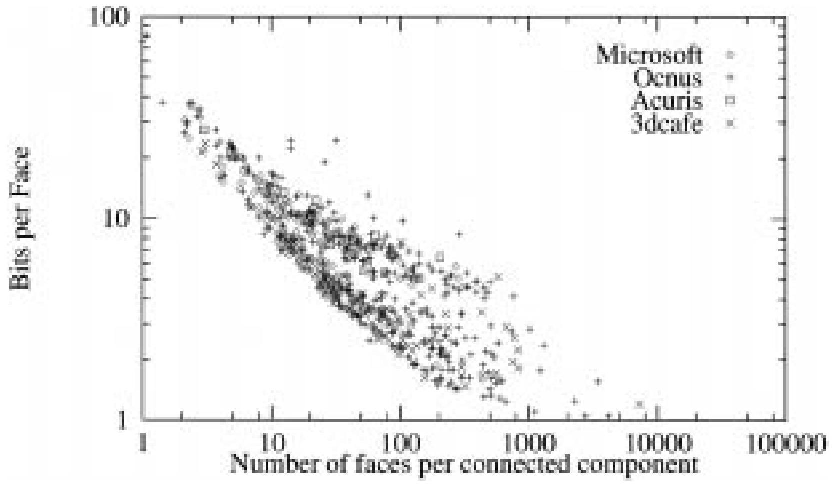


Fig. 18. Logarithmic plot of topological bits per face for 650 VRML97 files.

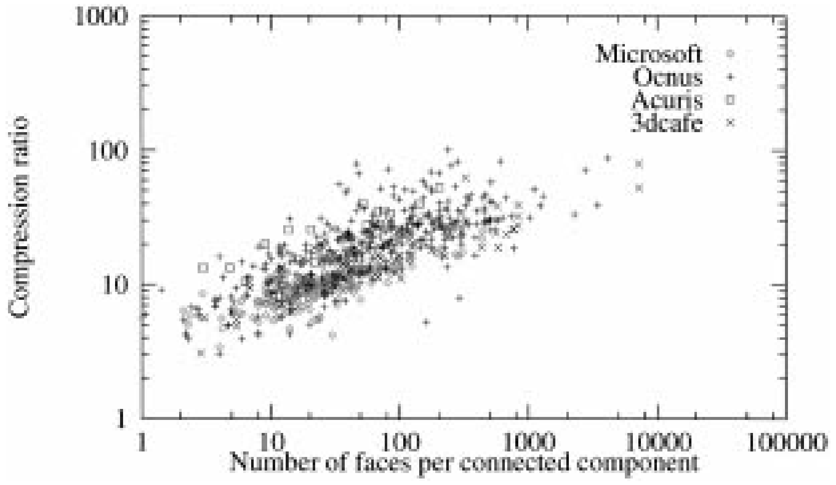


Fig. 19. Logarithmic plot of 8-bit compression ratios for 650 VRML97 files.

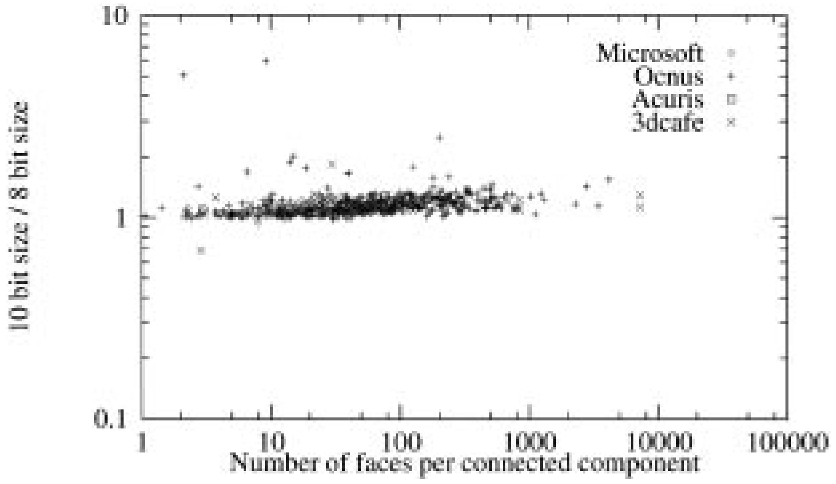


Fig. 20. Logarithmic plot comparing the cost of 10 bits of coordinate precision to the cost of 8 bits of coordinate precision for 650 VRML97 files.

a value of one. This limit is due the fact that, in our current implementation, there will always be at least one marching bit per face. Further improvement might be obtained by run-length encoding these bitstreams. Fig. 19 illustrates the relationship between the average number of faces per connected component and the compression ratio obtained by dividing the size of the original ASCII file by the size of the compressed binary file. The following parameters were used to compress the model: 8 bits per vertex coordinate, six subdivision levels per normal, 6 bits per color component, and 8 bits per texture coordinate. In this plot compression, ratios vary from three to over 100,

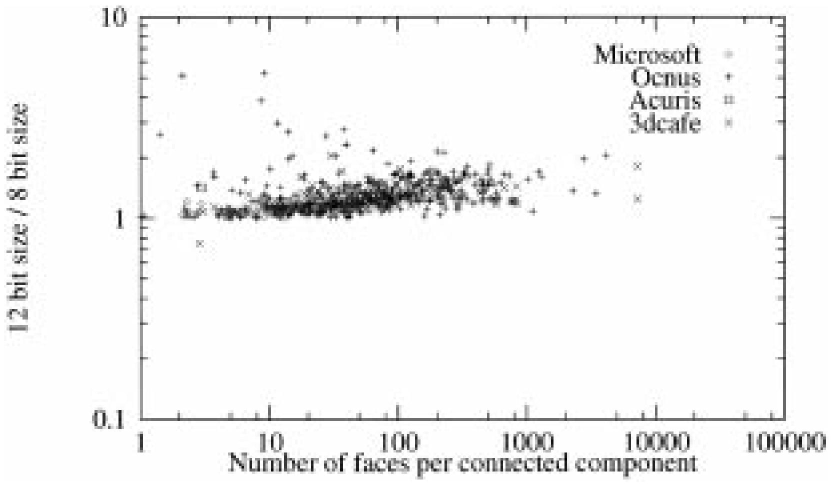


Fig. 21. Logarithmic plot comparing the cost of 12 bits of coordinate precision to the cost of 8 bits of coordinate precision for 650 VRML97 files.

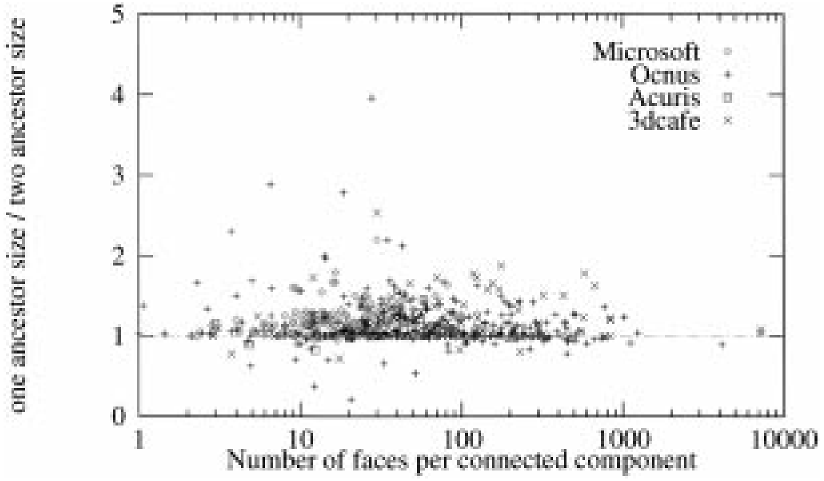


Fig. 22. Logarithmic plot comparing the cost of one predictor ancestor to the cost of two ancestors for 650 VRML97 files.

with the majority of the models possessing ratios greater than ten. Since an IFS with a large number of faces per connected component typically have larger quantities of spatially coherent data, they will, in general, compress better than an IFS possessing a smaller numbers of faces per connected component. This trend shows up in the above plot as a general increase in compression ratios as the average number of faces per connected component per file increases from left to right.

Sometimes, 8 bits of precision is not adequate for geometric coordinates. Fig. 20 examines the cost of increasing the coordinate precision from 8 to 10 bits. On average, an increase from 8 to 10 bits results in less than a 15% increase in the compressed binary file size. For just about all graphic applications, 12 bits of coordinate precision should suffice. Fig. 21 examines the cost of increasing the coordinate precision from 8 to 12 bits. On average, an increase from 8 to 12 bits results in less than a 23% increase in the compressed binary file size.

Fig. 22 demonstrates the influence of the number of coefficients used in the predictor model. In this figure, we compare the size of the compressed binary file resulting from one predictor against the size resulting from two predictors. In general, but not always, using two coefficients is more efficient than using one.

VIII. Conclusion and Future Work

Perhaps the biggest challenge currently facing VRML is the rapid delivery of compelling content. Compelling content frequently requires the specification of significant quantities of 3-D geometries with attached properties. As a result, the transmission times for VRML files containing compelling content frequently prohibits access in reasonable time. Our proposed binary format address this challenge by reducing the size of representative VRML files by an order of magnitude.

A. Future Work

As we saw in the previous section, coordinates for several IFS's can be quantized with a common bounding box. Sometimes, this is desirable, since it prevents quantization cracks between coincident edges from different IFS's. As was shown in Fig. 16, however, the sharing of a bounding box may also result in a poor quantization. It would be nice to automate the grouping process using a heuristic based on the relative scales, proximity, and coincident boundaries.

A similar grouping issue occurs for Huffman encoding and the predictor/corrector parameters. It is not easy to decide when different objects should share a common codebook or common predictor/corrector coefficients. Last, a mechanism to evaluate the effects of quantization parameters on the visual quality of an `IFS` would allow the automatic specification of these parameters.

References

[1] E. Arkin, M. Held, J. Mitchell, and S. Skiena, "Hamiltonian triangulations for fast rendering," in *Proc. 2nd Ann. Eur. Symp. Algorithms*, Sept. 1994, vol. 855, pp. 36–47.

[2] R. Bar-Yehuda and C. Gotsman, "Time/space tradeoffs for polygon mesh rendering," *ACM Trans. Graph.*, vol. 15, no. 2, pp. 141–152, Apr. 1996.

[3] R. Carey, G. Bell, and C. Marrin. (Apr. 1997). The Virtual Reality Modeling Language, ISO/IEC DIS 14772-1. [Online]. Available WWW: http://www.vrml.org/Specifications/VRML97/DIS.

[4] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. New York: Wiley, 1991.

[5] M. Deering, "Geometric compression," in *Comput. Graph. (Proc. SIGGRAPH)*, Aug. 1995, pp. 13–20.

[6] F. Evans, S. Skiena, and A. Varshney, "Optimizing triangle strips for fast rendering," in *Proc. IEEE Visualization '96*, Oct. 1996.

[7] W. S. Massey, *Algebraic Topology, An Introduction*. Orlando, FL: Harcourt Brace & World, 1967.

[8] G. Taubin, W. P. Horn, and F. Lazarus. (June 1997). The VRML compressed binary format. [Online]. Available WWW: http://www.research.ibm.com/vrml/binary.

[9] G. Taubin and J. Rossignac. (Jan. 1996). Geometry compression through topological surgery. IBM Research Division, Tech. Rep. RC-20340. [Online]. Available WWW: http://www.research.watson.ibm.com/vrml/binary/pdfs/ibm20340r1.pdf.

[10] ——, "Geometry compression through topological surgery," *ACM Trans. Graph.*, to be published

[11] G. Turán, "On the succinct representation of graphs," *Discrete Appl. Math.*, vol. 8, pp. 289–294, 1984.

[12] J. Wernecke, *The Inventor Mentor*. New York: Addison-Wesley, 1994.

Gabriel Taubin (Senior Member, IEEE) received the M.Sc. degree in pure mathematics from the University of Buenos Aires, Argentina, and the Ph.D. degree in electrical engineering from Brown University, Providence, RI, in the area of computer vision.

He is Manager of the Visual and Geometric Computing group at the IBM T. J. Watson Research Center, Yorktown Heights, NY. Previously, he spent five years as a Member of the Exploratory Computer Vision group. His research interests include geometric computation and image-based algorithms for three-dimensional (3-D) modeling, network-based graphics, 3-D scanning, and data visualization. He has received eleven patents and published more than 30 papers.

William P. Horn is a Manager in the Visualization Technologies group at the IBM T. J. Watson Research Center, Yorktown Heights, NY. His research interests include geometric computations for computer-aided design and multimedia applications. He is a coauthor of the VRML 2.0 binary standard proposal.

Francis Lazarus received the Ph.D. degree from the University of Paris VII, France, in 1995.

His doctoral dissertation was on morphing algorithms. He is an Assistant Professor of computer science at the University of Poitiers, France. He was a Postdoctoral Researcher at the IBM T. J. Watson Research Center, Yorktown Heights, NY, from 1996 to 1997. His research interests include geometry compression, geometric modeling, computer animation, and three-dimensional morphing.

Jarek Rossignac received the Ph.D. degree in electrical engineering from the University of Rochester, Rochester, NY, in the area of solid modeling and the engineering degree from the ENSEM, Nancy, France.

He was the Strategist for Visualization and the Senior Manager of the Visualization, Interaction, and Graphics Department at IBM Research, with responsibilities over a broad area of research in design, modeling, graphics, and visualization and over two IBM products: Data Explorer and 3-D Interaction Accelerator. He currently is a Professor in the College of Computing at the Georgia Institute of Technology, Atlanta, and Director of the Graphics, Visualization & Usability Center. His research interests focus on compact representation schemes and on efficient algorithms for geometric and visual computing.