RC-0(#0) 07/XX/97
Computer Sciences 29 pages

Research Report

GEOMETRY CODING AND VRML

Gabriel Taubin, William P. Horn,
and *Francis Lazarus*
IBM T.J. Watson Research Center
P.O.Box 704
Yorktown Heights, NY 10598
taubin@watson.ibm.com

Jarek Rossignac
GVU, Georgia Institute of Technology
801 Atlantic Drive
Atlanta, GA, 30332-0280
jarek.rossignac@cc.gatech.edu

GEOMETRY CODING AND VRML

Gabriel Taubin, William P. Horn,
and *Francis Lazarus*
IBM T.J. Watson Research Center
P.O.Box 704
Yorktown Heights, NY 10598
`taubin@watson.ibm.com`

Jarek Rossignac

GVU, Georgia Institute of Technology
801 Atlantic Drive
Atlanta, GA, 30332-0280
`jarek.rossignac@cc.gatech.edu`

ABSTRACT:

The Virtual Reality Modeling Language (VRML) is rapidly becoming the standard file format for transmitting 3D virtual worlds across the Internet. Static and dynamic descriptions of 3D 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.

Visually interesting VRML files tend to be large. In this paper we describe our proposal for the VRML Compressed Binary Format. Our proposal significantly reduces file sizes and, subsequently, the time required for transmission across the Internet. Compression ratios of up to 50:1 or more are achieved for large models. The format combines a binary encoding with topologically-assisted compression algorithms to create compact, rapidly-parsable VRML files. The format is currently being evaluated by the Compressed Binary Format Working Group of the VRML Consortium for possible inclusion into the VRML Standard.

The compression scheme represents a polyhedron 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. an average of roughly two bits 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.

CR Categories and Subject Descriptors:
I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling - *curve, surface, solid, and object representations*;

General Terms:
Compression, Algorithms, Graphics.

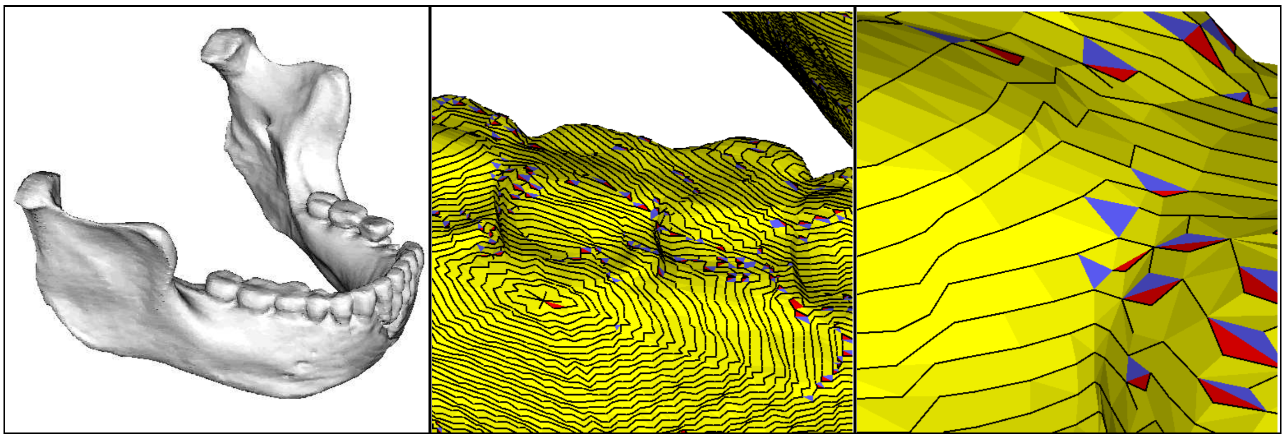


Figure 1: This model contains 86,939 vertices, 173,578 triangles, 1 connected component, and no properties. In the standard ascii VRML format it requires 8,946,259 bytes of storage. Using 11 bits per coordinate, the file in compressed binary format occupies 214,148 bytes for a compression ratio of 41.72. 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.

1. Introduction

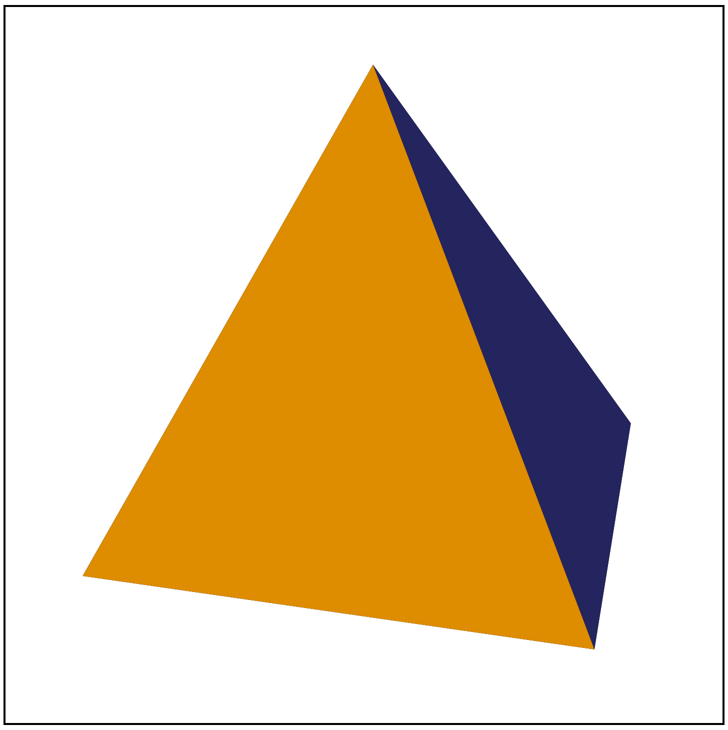
In recent years there has been a rapid growth in the exploitation of the Internet to serve text (HTML) and image (JPG,GIF,PNG) content to client browsers. With faster communication links and improving PC graphics there has been an increasing interest in delivering 3D content. VRML is the emerging standard for the delivery of 3D content in a networked environment. However, the bandwidth requirements for 3D content are significant and have served as a detriment to the wide acceptance of network-delivered 3D graphics. In this article we address the reduction of the bandwidth requirements through the use of topologically-assisted compression.

Figure 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 virtually no loss in visual quality.

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]. 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 over 600 VRML models and examine the results. Finally, we draw some conclusions and address future work.

2. The Virtual Reality Modeling Language

The Virtual Reality Modeling Language (VRML) [3], sometimes pronounced verml, is a format for describing and transmitting 3D objects and worlds composed of geometry and multimedia in a



```
#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 ]
 }
}
```

Figure 2: A simple VRML file.

network environment. VRML targets web applications such as: computer-aided design; engineering and scientific visualization, multimedia products, entertainment and educational offerings, and shared virtual worlds. A small VRML file is shown in figure 2. VRML has several features which make it particularly attractive for authoring virtual worlds. These features include:

- Scene graph
- Event processing
- Behaviors
- Encapsulation and re-use
- Distributed content
- Extensibility
- Interactivity
- Animation

Scene graph. The basic building block for VRML is the node. Nodes have fields, fields 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 also contain children nodes. 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 child nodes relative to its parent coordinate system. The parent relationship provides a sequence of transformations which, 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.

Event processing. Nodes can both receive 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 to 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. However, like nodes, they are defined in the VRML file.

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 URL (Uniform Resource Locator) 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.

Encapsulation and re-use. 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 provide a powerful mechanism to encapsulate content and behavior in a re-usable entity.

Distributed content. One of the key features of VRML is its support of the World Wide Web. VRML has several nodes which use URLs to connect the scenegraph to the network. These nodes include:

- fetch on demand of additional VRML content (Inline node)
- hyperlinks to other URLs (Anchor node)
- fetch on demand of audio content in uncompressed PCM (wavefile) format or MIDI file type 1 sound format (AudioClip node)
- fetch on demand of texture content in JPEG or PNG, with optional support of CGM and GIF (ImageTexture node)
- fetch on demand of video content in MPEG1-Systems (audio and video) and MPEG1-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, lets 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. Figure 3 sketches the flow of information in a network environment while processing a VRML file.

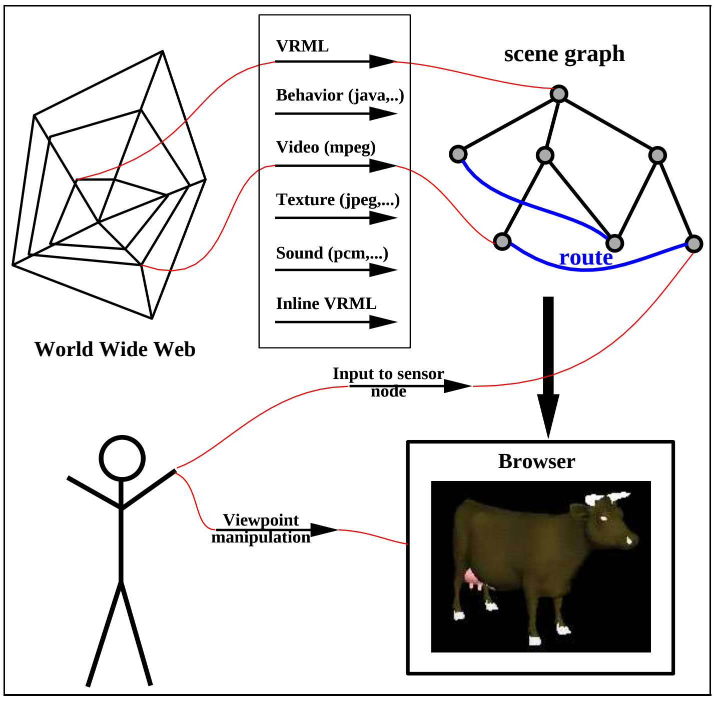


Figure 3: Flow of information while processing a VRML file.

Extensibility. In addition to enabling encapsulation and re-use, 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's call it NurbSurface, to model 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.

Interactivity. VRML supports a variety of sensor nodes including environment sensors, pointing device select sensors, and point 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 scenegraph 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.

Animation. VRML has a variety of interpolator nodes for use in creating linear keyframed animation supporting interpolation in: colors, coordinates (arrays of 3-tuple floats), normals, orientations, positions (3-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.

3. An Overview of the VRML Compressed Binary Format

The VRML format was designed to be minimal yet complete. It does not suffer from the bloat of earlier scene graph technologies [11] but it is still powerful enough to describe complex, animated worlds. Unfortunately, even small VRML files tend to be quite large (>100 KB). Our proposed VRML Compressed Binary Format [8] addresses this problem by representing the 3D 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] 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 the following fields at the file scope:

- Any `SFColor` or `MFColor` field
- `MFVec3f` when used as a normal for a built-in node, specifically:
  - `Normal::vector`
  - `NormalInterpolator::keyValue`

Another, more interesting, mechanism, referred to as *topologically-assisted compression*, enables the compression of the following VRML nodes:

- `CoordinateInterpolator`
- `ElevationGrid`
- `IndexedFaceSet`
- `PointSet`
- `NormalInterpolator`

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, 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 number 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 work horse 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 kilobytes. The description of a large virtual world can easily reach 10 megabytes. As will be shown in subsequent sections, the most dramatic reductions in storage requirements result from the topologically-assisted compression of the `IFS`.

4. 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 is 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.

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 figure 4, a triangle strip is a chain of triangles where each new vertex reference implicitly defines a new triangle. The trailing

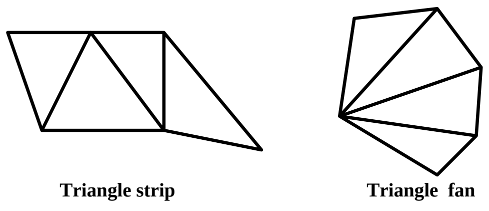


Figure 4: Triangle strip an fan.

edge of the previous triangle is used with the incremental vertex index to form the next triangle in an alternating, “zig-zag” 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 figure 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

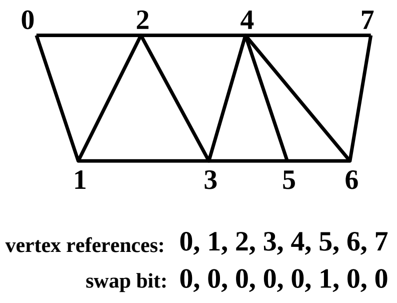

0
2
4
7
1
3
5
6
vertex references: 0, 1, 2, 3, 4, 5, 6, 7
swap bit: 0, 0, 0, 0, 0, 1, 0, 0

Figure 5: A generalized triangle strip.

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 four bit address buffer along with a couple of new operations. The four 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.

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, start with a vertex spanning tree. Turán [10] shows that a planar graph and, consequently, faces and edges of a closed surface with genus 0, can be encoded using $12n$ bits. As will be shown in section 7, the method used in the current work encodes the topology for large polygonal models with storage requirements approaching 1 bit per face.

5. Topologically-assisted Compression

In this section we briefly describe the method of Taubin and Rossignac for representing triangular meshes in compressed form [9]. A triangular mesh is defined by the location of its vertices (*positions*) and by the association between each triangle and its sustaining vertices (*connectivity*). Optionally, properties such as colors, normals, and texture information which do not affect the 3D geometry, but do influence the way it is rendered, may be attached to the mesh.

To develop the basic concepts, we will first concentrate on *simple* triangular meshes, that is, triangulated connected oriented manifolds without boundary, of Euler characteristic 2, and without properties (normals, colors, or texture mapping coordinates). Examples of simple and non-simple meshes are shown in figure 7.

5.1. Simple Mesh

Let us assume 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 $(2V-2)-(E+V-1)+T = (V-E+T)-1 = 1$;
5. Every connected oriented manifold triangular mesh of Euler characteristic equal to 1 is homeomorphic to a topological disk [7].

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 the run is the last run sharing the *current branching node* or not, 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.

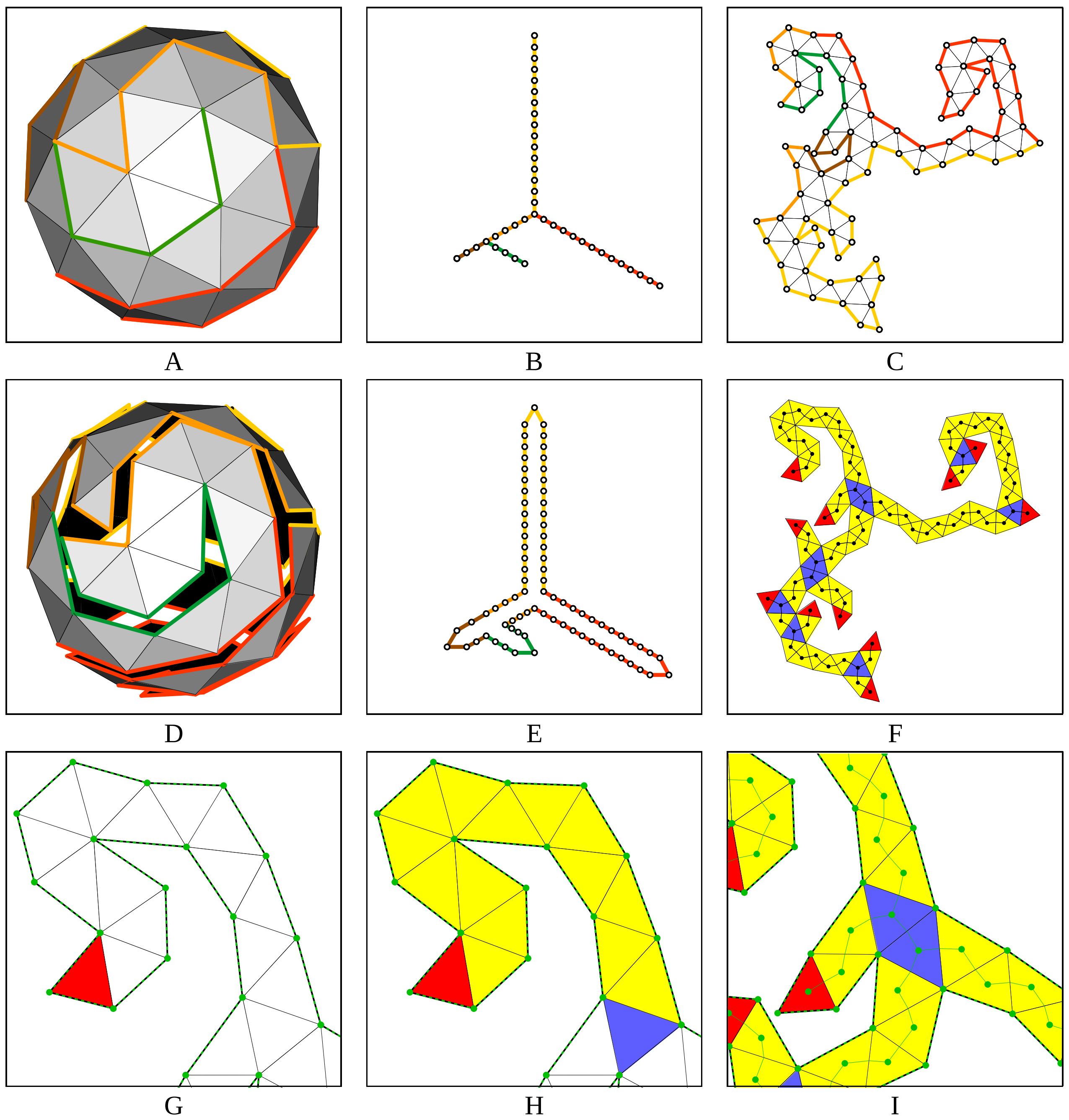


Figure 6: Representation of a simple mesh in compressed form. The vertex spanning tree (A,B) is composed of vertex runs. Cutting through the vertex tree edges produces a topological simply connected polygon(C,D). The bounding loop (E) is the boundary of the polygon. The dual graph of the polygon is the triangle spanning tree (F). Triangle runs end in leaf or branching triangles. Leaf triangles are red, regular triangles are yellow, and branching triangles are blue. 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).

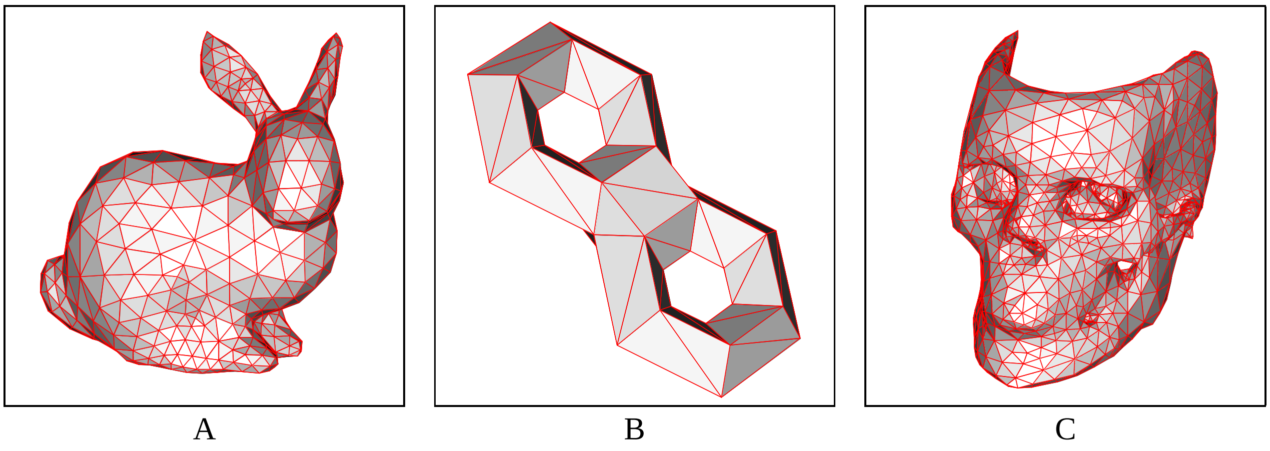


Figure 7: Examples of simple meshes (A), and non-simple meshes (B,C).

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 indices. 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*) which 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 look-up table is not included in the compressed representation of the mesh. However, both compression and decompression algorithms must construct the look-up table.

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 is encoded. However, because the triangle spanning tree is binary, 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 which 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.

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.

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 indices 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 look-up 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 look-up table*.

For each branching triangle, the distance along the loop from either the left or right vertices to the Y-vertex, the *left branch boundary length* and *right branch boundary length*, 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 1 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.

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 fixed precision format. 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:

$$v_n = \epsilon(v_n) + P(\lambda, v_{n-1}, \ldots, v_{n-K}) \; , \tag{5.1}$$

where $\epsilon(v_n)$ is the *vertex position correction* associated with that vertex, P is a vertex positions predictor function, λ and 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 pre-order, and, as mentioned above, further entropy encoded.

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, and
5. computing and encoding the marching pattern.

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 (look-up table pointing to the vertex position table),
4. computing the relative indices for Y-vertices in the order in which they will be used, and
5. reconstructing and linking of triangle runs.

Several methods for constructing the spanning trees are described by Taubin and Rossignac [9]. 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 figures 8-A, 8-C, and 8-E. A vertex is chosen as the root of the vertex tree. The singleton consisting of the root vertex is the first boundary. The n-th triangle layer is the set of triangles which are incident to one or more vertices of the n-th boundary but do not belong to a previous triangle layer. The $(n+1)$-st boundary consists of all the edges of triangles of the n-th layer with neither one of the two end vertices belonging to the n-th 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's assume that a vertex tree has been constructed with all the vertices included in the first n boundaries, and 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 n-th 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. Figure 9 illustrates this construction.

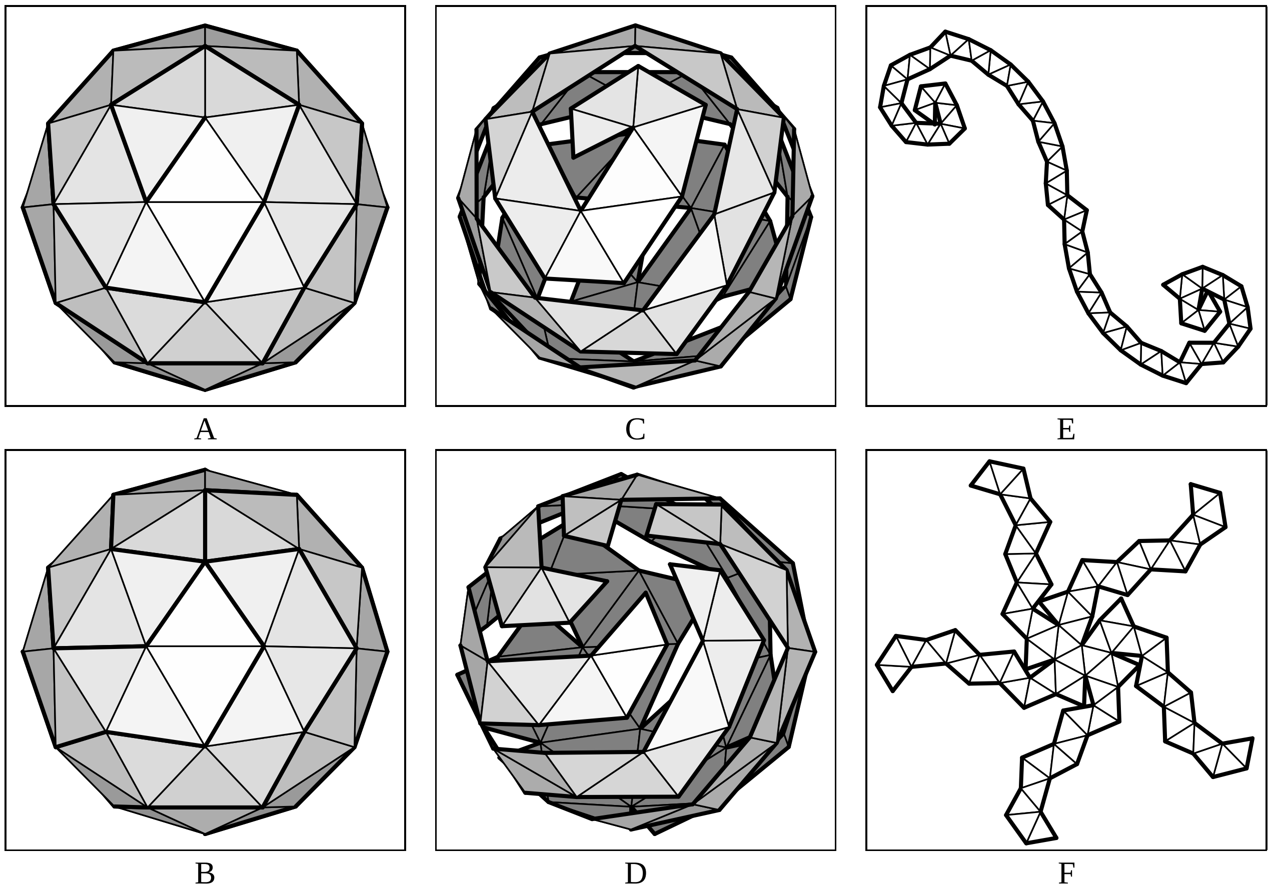


Figure 8: Two ways of peeling an orange, A,B: The thick edges are the edges of the vertex tree constructed on the mesh. C,D: The mesh is cut through the vertex tree edges (the vertex positions have been modified here only to illustrate the creation of the cut). E,F: The result is a topological simply connected polygon. The dual graph of this polygon is the triangle tree.

5.2. More General Meshes

Triangular manifold meshes of Euler characteristic other than 2, non-orientable, 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).

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

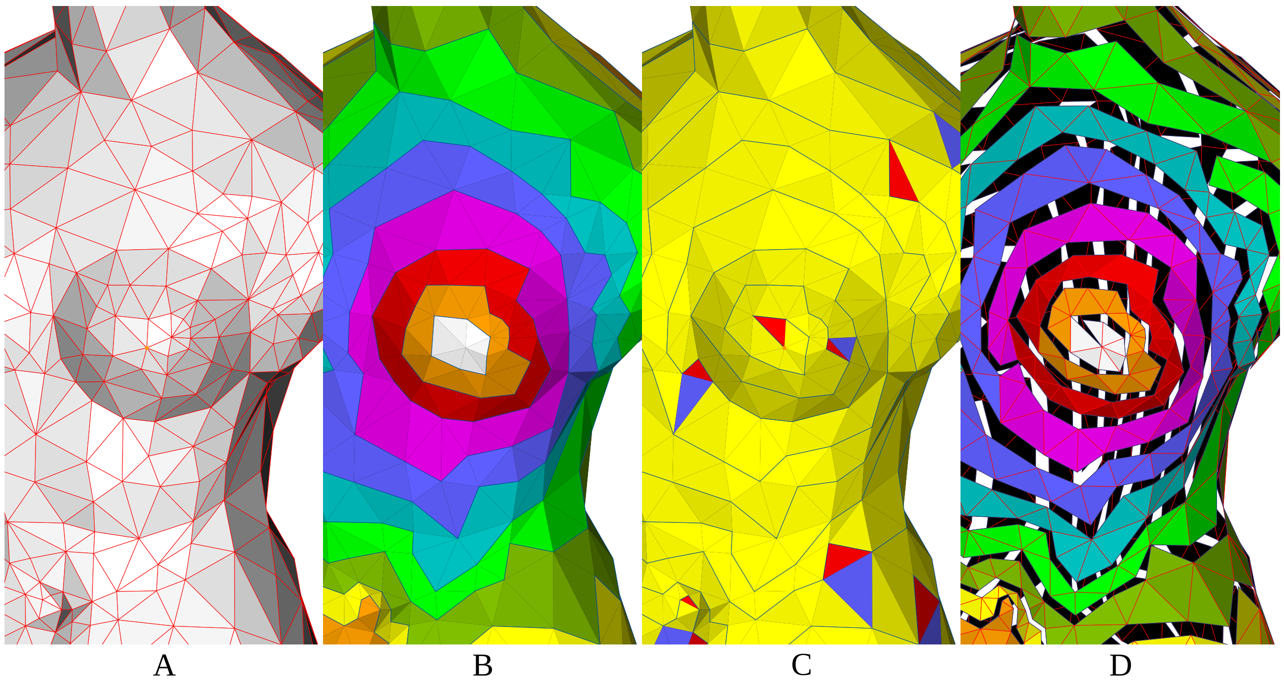


Figure 9: Compression algorithm A: Triangular mesh. B: The topological distance from a chosen vertex defines the layers. C: Vertex tree and triangle tree are constructed by traversing the layers in order. D: Polygon resulting of cutting along cut edges with artificial gap introduced. Triangles are color-coded according to their corresponding layer.

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 (non-branching and non-leaf) 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.

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.

Non-orientable 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 non-orientable 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.

6. 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. Finally, we introduce a new scheme to compactly encode corner properties for corners sharing both a common vertex and property value.

6.1. From Triangles to Polygons

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

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 *non-polygonal edges*. The other edges are called *polygonal edges*.

In order 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 non-polygonal 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 bit stream with as many bits as the number of interior edges to differentiate between polygonal and non-polygonal edges. This surface can be compressed using the scheme discussed in the previous section. The compressed triangulated surface together with this bit stream provides sufficient information to recover the polygonal surface.

Polygons are recovered by first reconstructing the triangular surface and subsequently removing the extra non-polygonal 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 non-polygonal interior edge. Each polygon-connected component corresponds to a polygon to be recovered.

As shown in figure 10, a polygon may spread over several triangle runs. As explained in Section 5 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.

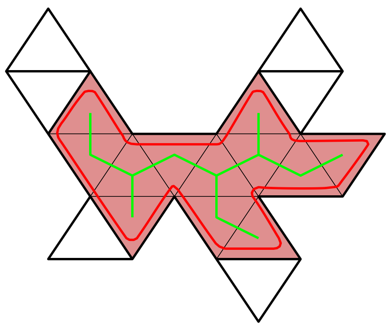


Figure 10: One can think of a polygon as an amoeba constrained to stay in the interior of the triangle tree.

We refer to a connected subset of the boundary of a polygon as a *polygon segment*. A polygon partially covering 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 indices. 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. Figure 11.a through figure 11.k illustrates the recovery of polygons contained in the the triangle tree of figure 11-L. The triangle tree is composed of three runs and its root triangle is marked “Root”. In figure 11, arrows are marked **Li** or **Ri** where **L** and **R** indicate an extremity of the left or right edge for the **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 figure 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 (**L2**,**R2**) the polygon segment in the right stack is popped onto the left stack. This event is represented by the arrow marked “1” in figure 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 figure 11-C. Figure 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.

Figure 11-E shows the initial processing of edge edge (**L4**,**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 (**L4**,**R4**) is a polygon edge, the right top segment (a single vertex) is popped onto the left stack. This event is shown in figure 11-F.

Next, in figure 11-G, we start the pink polygon by processing edge (**L5**,**R5**). Since the run ends in a leaf triangle we can connect the current left and right segments. As shown in figure 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 (**L6**,**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 figure 11-I. Figures 11-J and 11-K show 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. Figure 12-A shows the corner ordering in the output coordIndex for the example used in figure 11.

6.2. 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 option is to specify an index into a palette of values. Finally 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 (3 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 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,
- for faces is the same ordering as used in the output coordIndex field, during recovery, and,
- for corners is the same ordering as used in the output coordIndex field.

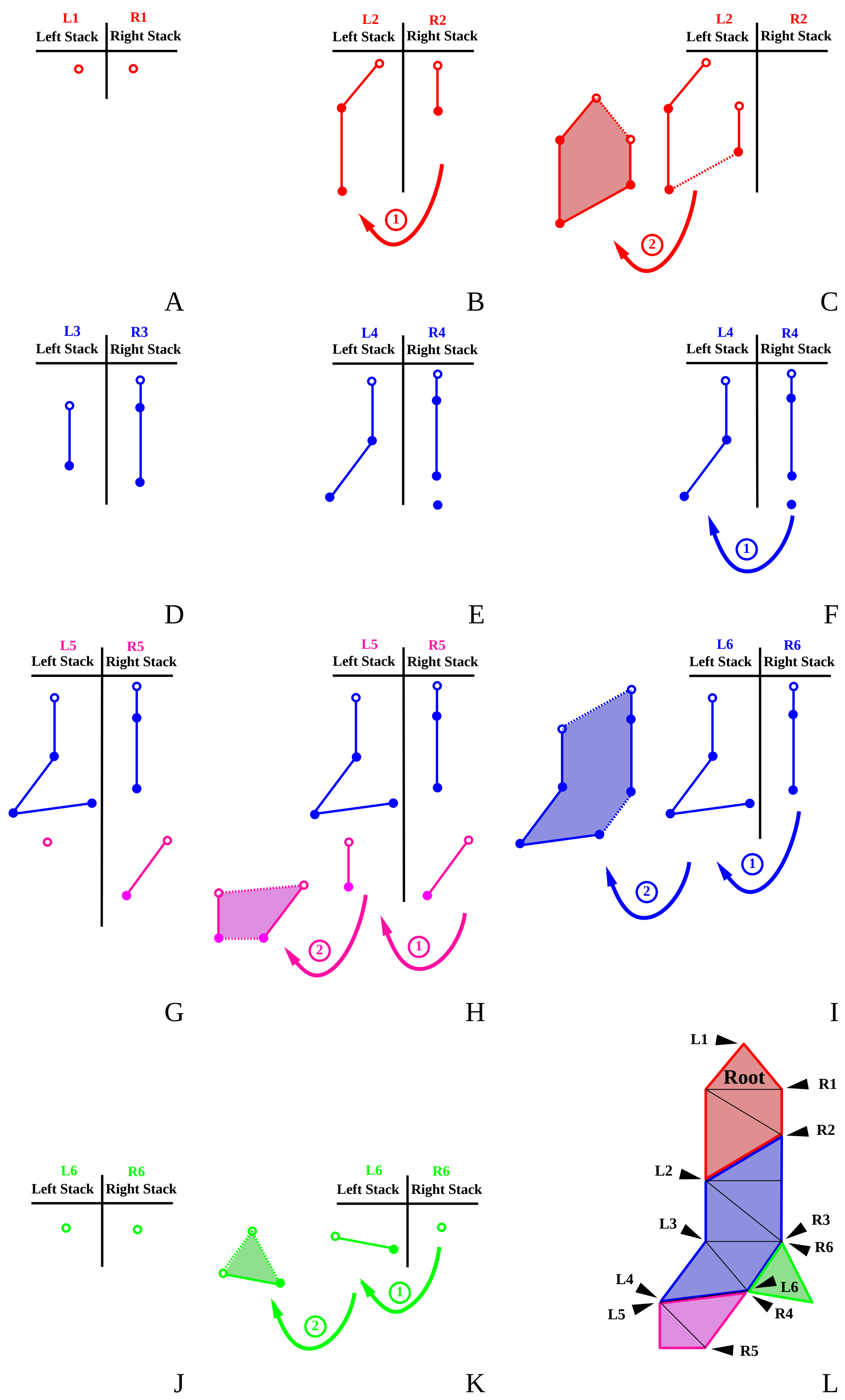


Figure 11: Schematic view of the polygon reconstruction.

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 5.1, 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 (3 for the coordinates). For coordinate data, our experience has been that a precision of 8 to 12 bits is sufficient. An 8 (12) bit precision corresponds to 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 non polygon edges as shown Figure 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:

- 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.

When the mesh contains non-triangular 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 Figure 12-C.

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.

Color encoding. Colors are integerized and stored with a user-specified fixed number of bits.

Normal encoding. Normals are quantized with a subdivision scheme that produces an optionally lossy compression of 3 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 3 canonical vectors. An octant is discretized by recursively subdividing the base triangle n times as

[1]This encoding scheme was developed with Fabio Pettinati

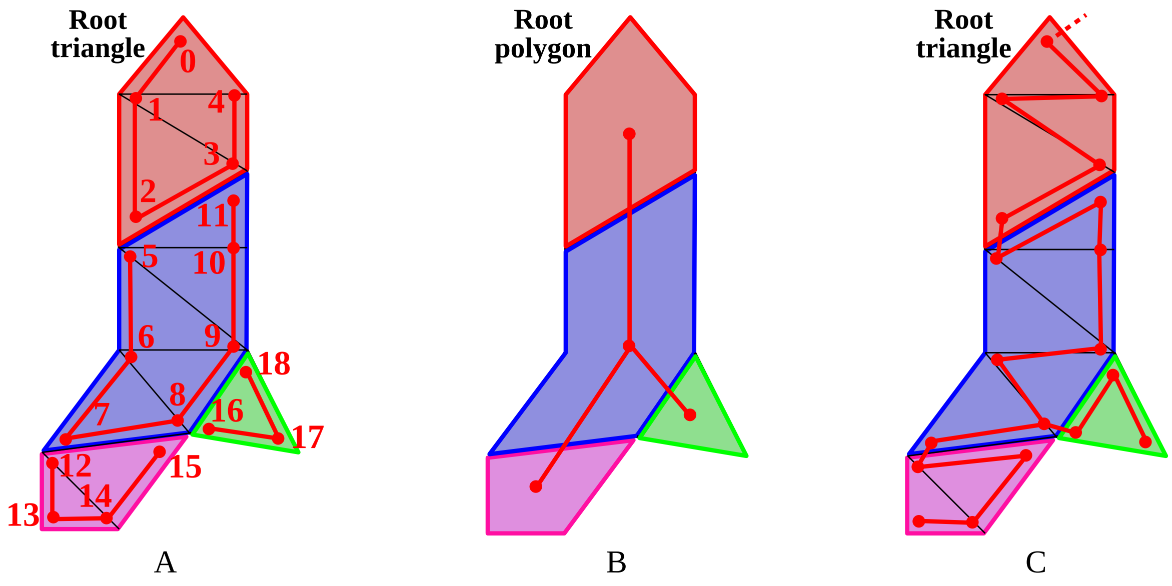


Figure 12: Implicit ordering and parent relationship from the triangle tree. A: The corner ordering. B: The face tree. C: The corner tree.

shown figure 13-A - 13-C. A normal is encoded using the triangle's number, where the triangles are enumerated as shown in Figure 13-C.

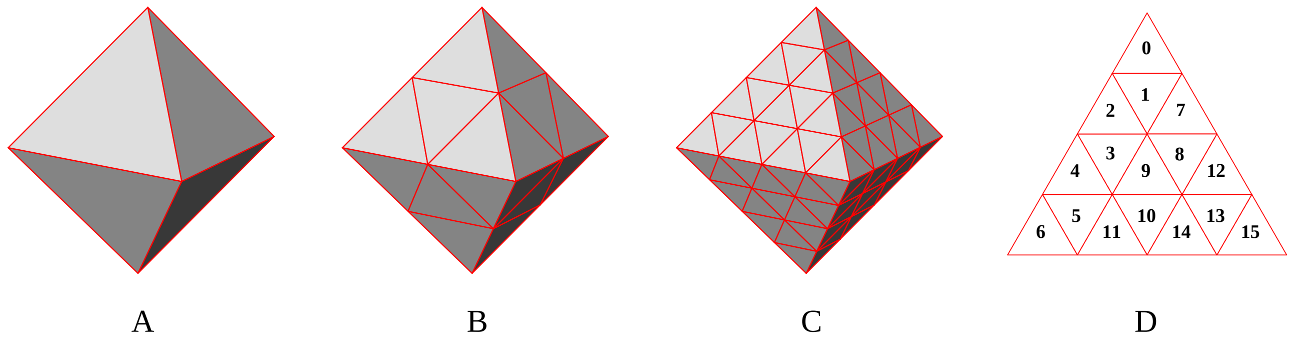


Figure 13: Three subdivision levels of the base octahedron. A: subdivision level 0. B: subdivision level 1. C: subdivision level 2. D: the triangle enumeration for level 2.

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.

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 bit stream 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 corner whenever the prior corner had a different property. Otherwise we assign a "0". Figure 14 shows a vertex shared by 6 faces with different properties. The "0"s and "1"s in the figure are values from the discontinuity bit stream. 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.

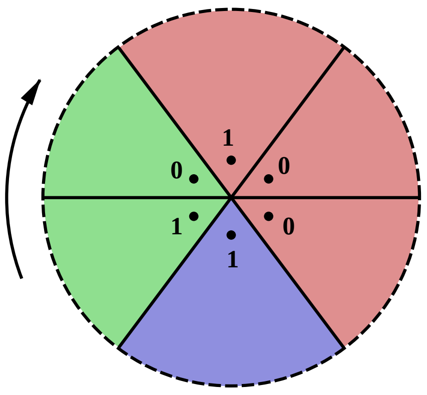


Figure 14: The property star of a vertex and the discontinuity bits associated to the corners.

7. 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 affect the visual quality of the final model, it is difficult to automatically choose optimal parameters. In the absence of good heuristics, it appears that the best solution is to interactively 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.

Coordinate quantization. The sequence of images on figure 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.

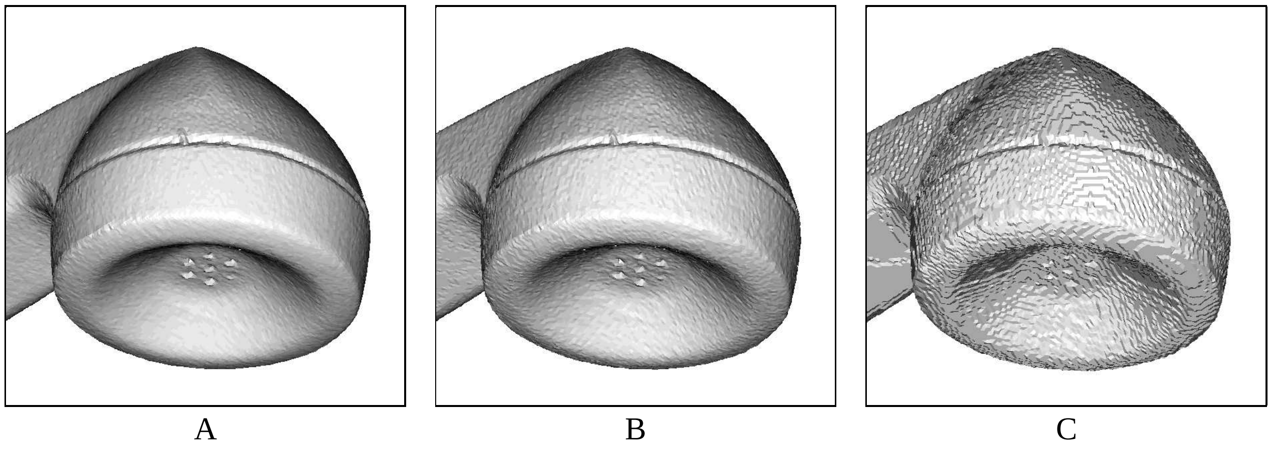


Figure 15: A: The original model. B: The same model quantized to 11 bits per coordinate. C: The same model quantized to 9 bits per coordinate.

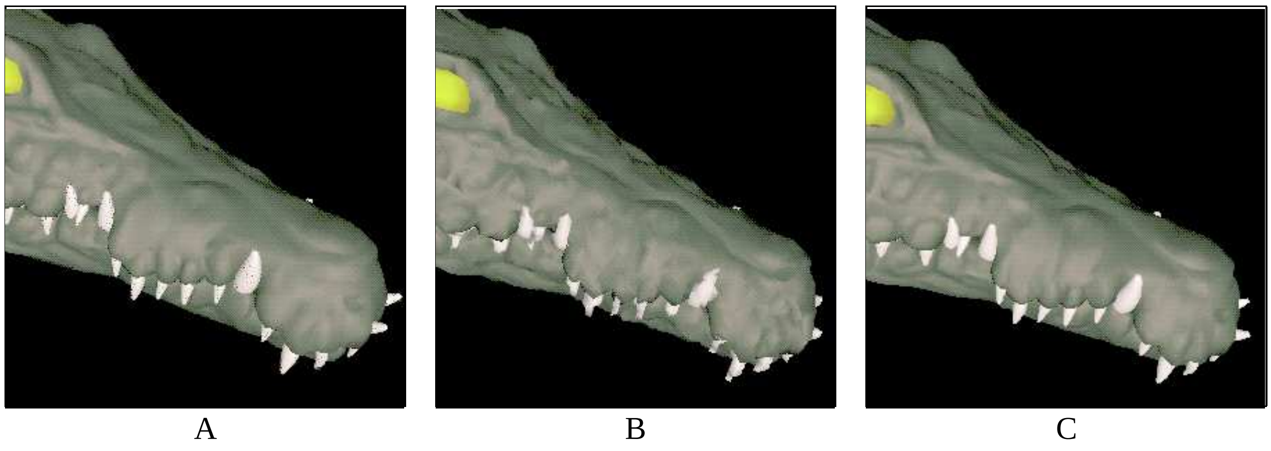


Figure 16: A: The original model. B: The same model quantized to 9 bits per coordinate using one common bounding box for the head and the teeth. C: The same model quantized to 9 bits per coordinate using a separate bounding box for the head and the teeth.

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 the figure 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 times the precision. Because the overall the 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

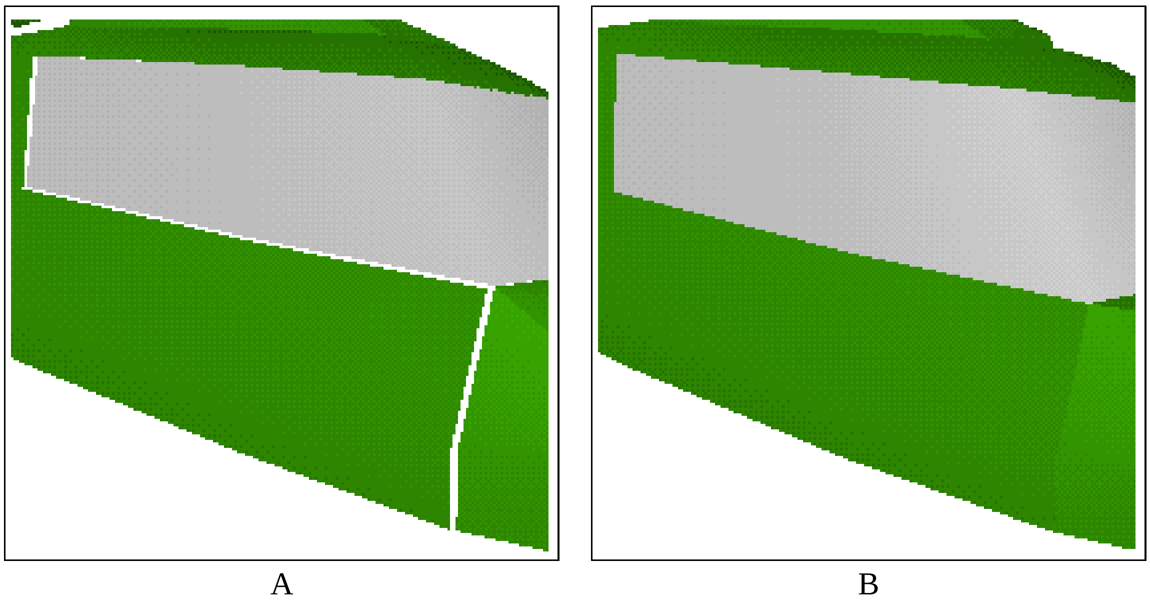


Figure 17: A: Model quantized with one bounding box. B: The same model quantized with multiple bounding boxes.

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 define a rectangular grid in 3D. 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 3D. However, in our current implementation 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. Figure 17 shows cracks caused by using one bounding box per `IFS`. Cracks are created only when two vertices that coincide in 3D in the original uncompressed geometry are quantized using combinations of bounding boxes and numbers of bits per coordinate that produce non-matching grids.

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 produce the minimize file size. Instead it uses a sub-optimal 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.

7.1. Examples

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

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

- `http://www.microsoft.com/vrml/stack`. The suite contains over 200, relatively small, Viewpoint (R) 3D models in four categories: nature, architecture, transportation, and accessories.
- `http://www.acuris.com/free_25.htm`. Some 23 models, both VRML 1 and VRML 2.0, from the Acuris (R) web site.
- `http://www.3dcafe.com/meshes.htm`. More than 100 VRML 1 models downloaded from the 3D CAFE (R) 3D Model library.
- `http://www.ocnus.com/models/models.html`. Over 300 VRML 1 models from the Ocnus Rope Company (R) VRML repository.

The geometry content in these files ranges from a couple faces to more than 10,000 faces. Before compressing these files the models were converted, if necessary, from VRML 1 to VRML97. Also, we transformed any singular `IFS` into non-singular `IFS`.

Figure 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 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 bit streams. Figure 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, 6 subdivision levels per normal, 6 bits per color component, 8 bits per texture coordinate. In this plot compression ratios vary from 3 to over 100, with the majority of the models possessing ratios greater than 10. Since `IFS` with a large number of faces per connected component typically have

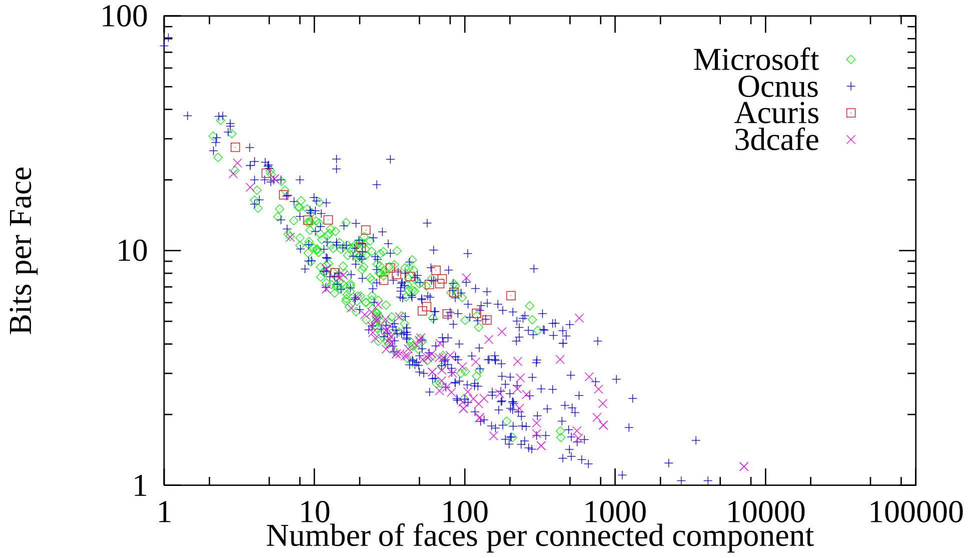


Figure 18: Logarithmic plot of topological bits per face for 650 VRML97 files.

larger quantities of spatially coherent data, they will, in general, compress better than `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. Figure 20 examines the cost of increasing the coordinate precision from 8 bits to 10 bits. On average, an increase from 8 bits to 10 bits results results in less than a 15% increase in the compressed binary file size. For the just about all graphic applications, 12 bits of coordinate precision should suffice. Figure 21 examines the cost of increasing the coordinate precision from 8 bits to 12 bits. On average, an increase from 8 bits to 12 bits results in less than a 23% increase in the compressed binary file size.

Figure 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.

8. 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 3D 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.

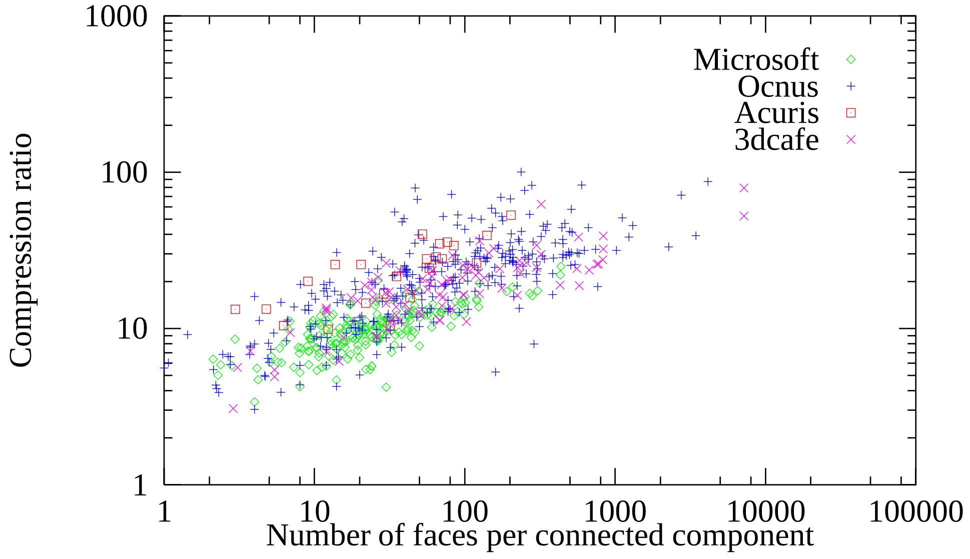


Figure 19: Logarithmic plot of eight bit compression ratios for 650 VRML97 files.

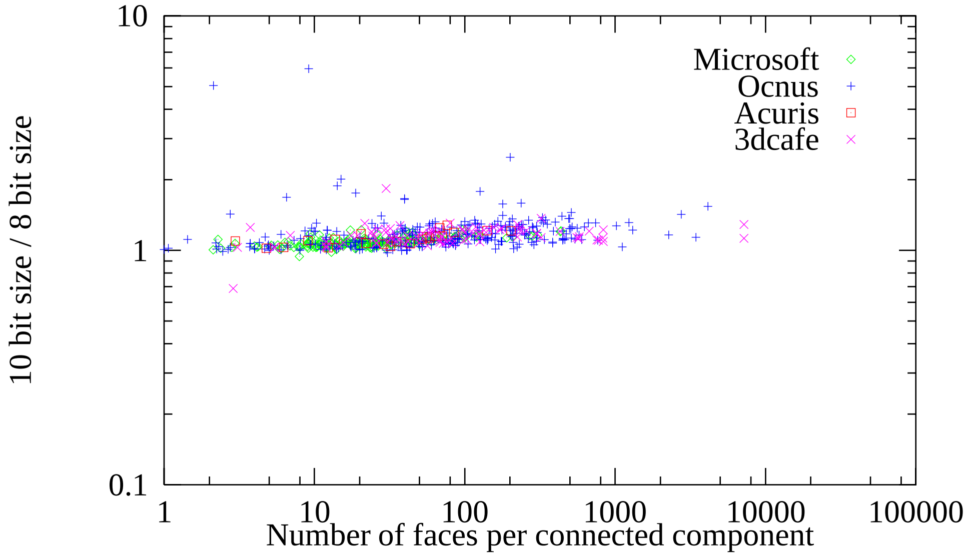


Figure 20: Logarithmic plot comparing the cost of 10 bits of coordinate precision to cost of 8 bits of coordinate precision for 650 VRML97 files.

Future work. As we saw in the previous section coordinates for several `IFS` can be quantized with a common bounding box. Sometimes this is desirable since it prevents quantization cracks between

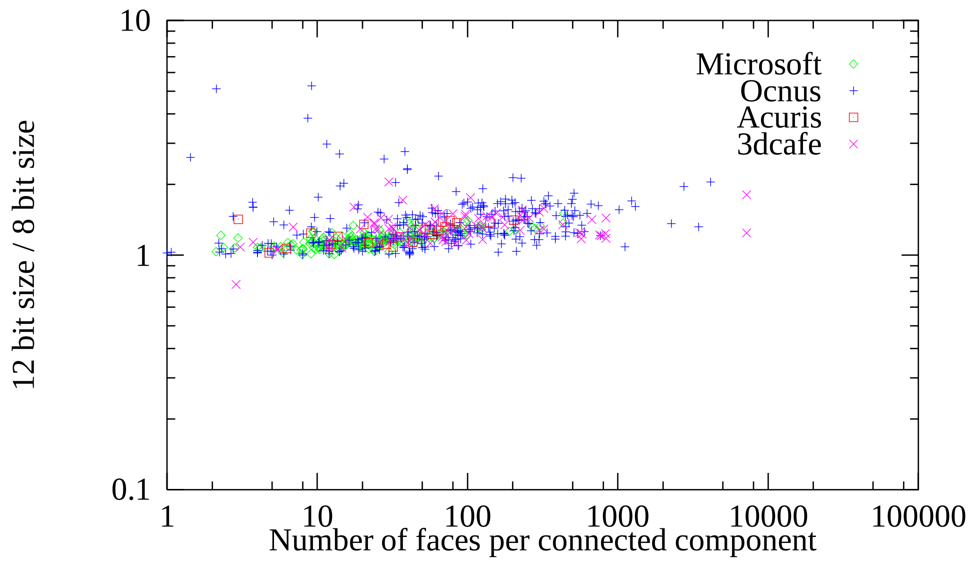


Figure 21: Logarithmic plot comparing the cost of 12 bits of coordinate precision to the cost of 8 bits of coordinate precision for 650 VRML97 files.

coincident edges from different `IFS`. However, as was shown in figure 16, 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 code book or common predictor/corrector coefficients. Finally, 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 *Second Annual European Symposium on Algorithms*, volume 855, pages 36–47. Springer Verlag, September 1994.

[2] R. Bar-Yehuda and C. Gotsman. Time/space tradeoffs for polygon mesh rendering. *ACM Transactions on Graphics*, 15(2):141–152, April 1996.

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

[4] T.M. Cover and J.A.Thomas. *Elements of information theory*. Wiley, New York, 1991.

[5] M. Deering. Geometric Compression. *Computer Graphics (Proc. SIGGRAPH)*, pages 13–20, August 1995.

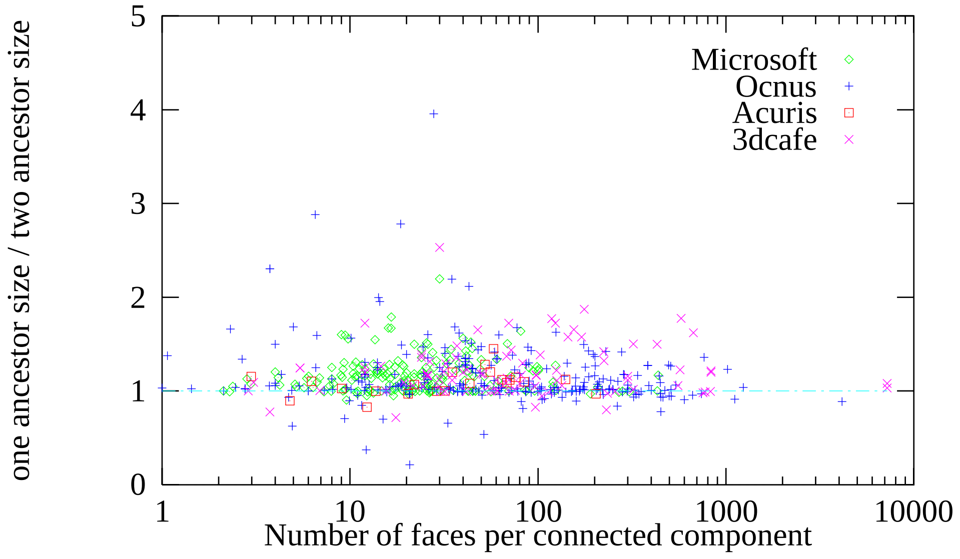


Figure 22: Logarithmic plot comparing the cost of one predictor ancestor to the cost of two ancestors for 650 VRML97 files.

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

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

[8] G. Taubin, W.P. Horn, and F. Lazarus. The VRML Compressed Binary Format, June 1997. `http://www.research.ibm.com/vrml/binary`.

[9] G. Taubin and J. Rossignac. Geometry Compression through Topological Surgery. Technical Report RC-20340, IBM Research Division, January 1996. `http://www.research.watson.ibm.com/vrml/binary/pdfs/ibm20340.pdf`.

[10] G. Turán. On the succint representation of graphs. *Discrete Applied Mathematics, North-Holland*, 8:289–294, 1984.

[11] J. Wernecke. *The Inventor Mentor*. Addison Wesley, New York, 1994.