Triangle Mesh Compression

Costa Touma
Department of Computer Science
Technion - Israel Institute of Technology
Haifa 32000, Israel

costa@cs.technion.ac.il

Craig Gotsman[1]
Virtue Ltd.
P.O.Box 199
Tirat Carmel 30200, Israel

gotsman@virtue.co.il

Abstract

A novel algorithm for the encoding of orientable manifold triangle mesh geometry is presented. Mesh connectivity is encoded in a lossless manner. Vertex coordinate data is uniformly quantized and then losslessly encoded. The compression ratios achieved by the algorithm are shown to be significantly better than those of currently available algorithms, for both connectivity and coordinate data. Use of our algorithm may lead to significant reduction of bandwidth required for the transmission of VRML files over the Internet.

Keywords: Triangle mesh, Compression, Coding, VRML.

1. Introduction

The advent of the World Wide Web, and its usage for remote access to data servers through low bandwidth communication lines, has dramatically increased the need for efficient compression schemes for common types of media found on the Web. For media such as audio and video, it is possible to adapt compression schemes devised for other purposes (e.g. storage) to the Internet scenario. However, the Web has introduced new data types, which are not traditional multimedia, and for which compression schemes did not previously exist. One of these is three-dimensional geometry, the main content of VRML97 files. The current VRML97 format [7] is ASCII-based, containing a large amount of redundancy, resulting in large download times. The VRML community has realized that this is a major obstacle to real-world VRML applications and has proposed a new compact binary format [8]. Beyond binary tokenization of the regular ASCII content, the proposal uses a polygonal mesh compression algorithm due to Taubin and Rossignac. [11], which seems to yield more compact datasets than the few other algorithms devised for this purpose [4, 3, 2]. We call the algorithm of Taubin and Rossignac the "IBM algorithm". The IBM algorithm encodes the mesh connectivity in a lossless manner, quantizes the real vertex coordinate information and encodes the result in a lossless manner.

The encoding of an arbitrary polygonal manifold in three-dimensional space is a generalization of the encoding of a planar geometric graph. By "geometric", we mean a graph for which coordinates are associated with each vertex, as opposed to an abstract planar graph, for which only the edge information is meaningful. Encoding geometric graphs involves encoding also the vertex coordinates, in addition to the edge information encoded for abstract graphs. Turan [12] has shown that a planar graph may be encoded in at most $12n$ bits, where n is the number of vertices in the graph. Keeler and Westbrook [9] improved Turan's results, showing that a triangulated planar graph may be encoded in at most 4.6 n bits. The IBM algorithm, which bears some similarity to that of Turan, encodes the edge information in an average of 4 bits per vertex. The algorithm we describe in this paper will be shown to encode the same edge information in less than 1.5 bits per vertex on the average.

The IBM algorithm proposes to encode the coordinate information of the mesh vertices by quantizing each of the three vertex coordinates to a fixed number of bits (8, 10 or 12 bits is typical), and then losslessly encoding the quantized values using a simple linear predic-

[1] On sabbatical leave from the Department of Computer Science, Technion - Israel Institute of Technology.

tion method. The integer prediction errors are then entropy-encoded, resulting in approximately 13 bits per vertex. We use a more sophisticated prediction scheme based on local surface properties, requiring only approximately 9 bits per vertex for typical inputs.

2. Lossless Connectivity Encoding

The key to our encoding scheme is the following (simple) observation: In a polygonal mesh which is an orientable manifold, the vertices incident on any mesh vertex may be *ordered*, i.e. sorted in clockwise order in a consistent manner. This contrasts with general graphs, for which this property does not hold. A consequence of this is the *separation* property of genus-0 meshes, namely, that the removal of any vertex cycle, and all associated edges, separates the mesh into two disjoint meshes - those inside the cycle, and those outside it (in the degenerate case, one of these meshes may be empty).

We shall see that this implies that the mesh connectivity may be encoded as a list of vertex *degrees* in a special order. A few other bits must be dedicated to describe special cases which arise very infrequently in typical cases.

2.1 The Formal Algorithm

In this section, we make some definitions relevant to our coding algorithms, and describe them briefly. Pseudo-code for the encoding and decoding algorithms may be found in Figs. 2-3. A simple example demonstrating the operation of our algorithm appears in Fig 4. The input to our algorithm is an orientable manifold triangle mesh, and the output is the code for the mesh connectivity.

Definitions

Vertex cycle: A cyclic sequence of vertices along triangle edges in the mesh.

Active List: A vertex cycle in the mesh. The active list partitions the mesh into an "outer" part containing edges not yet encoded, and an "inner" part containing edges already encoded. Each vertex in the active list has encoded and unencoded incident edges separated by the edges to the two vertices which are its predecessor and successor in the active list.

Focus: One vertex in the active list is designated as the focus vertex. All coding operations are done on the focus vertex. When this is complete, the focus is moved to the next vertex in the active list and the previous focus vertex is removed from the active list to become one of the "inner" vertices. This causes the active list to "expand".

Free Vertex: A vertex not yet encoded.

Free Edge: An edge not yet encoded.

Full Vertex: A vertex with no free edges.

Offset: The number of vertices separating two given vertices along the active list (clockwise).

The encoding algorithm starts off with an arbitrary triangle in the mesh, defining an active list of three edges. An arbitrary vertex of this triangle is designated as the focus. The algorithm proceeds by trying to expand the active list by "conquering" edges in counter-clockwise order around the focus. Each such edge generates an "add" command. When all these edges are exhausted (the vertex is "full") the focus is moved to the next vertex in counter-clockwise order around the active list. The conquering procedure repeats for the new focus. If the active list, during expansion, intersects itself, it is split to two active lists, each of which is traversed recursively. This event also generates a "split" command in the code. The procedure terminates when all vertices are full (all edges traversed).

The decoding procedure is straightforward. Each "add" command encountered in the code causes a new vertex and triangle to be generated, by connecting the new vertex to the focus and its predecessor on the active list.

2.2 Special Cases

Split and Merge

If, during encoding, the first free edge of the focus of an active list does not lead to an unencoded vertex, there are two possibilities: either it leads to a vertex in the same active list or to a vertex in another active list (on the stack). Note that all other vertices are inner vertices and all their edges have already been traversed. In the first case, the active list is *split* into two separate active lists. One is pushed onto the stack for future treatment, and the encoding procedure proceeds with the second. In the latter case, the two active lists are *merged* to form one active list on which the encoding continues. Note that merge will never happen if the object has sphere topology (genus 0), and can only occur in a torus-like topology (genus 1).

Handling Boundaries

If the mesh is not closed, i.e. has boundaries, this is treated by adding a dummy vertex for each boundary and connecting it to the vertices of that boundary, creating a closed topology. These dummy vertices should be removed after the decoding and reconstruction of the mesh. They are tagged when encoding the mesh, located and removed when decoding.

2.3 Entropy Coding of the Command Sequence

The three commands appearing in the connectivity code are "add <degree>", "split <offset>" and "merge <index><offset>". In practice, a typical code contains mostly "add" commands, a few "split" commands, and almost no "merge" commands.

In typical meshes, the average vertex degree is 6, and there is a spread of degrees around this value. This invites entropy coding, e.g. Huffman. The entropy coding results sometimes in long runs of the same bit patterns, which is susceptible to run-length encoding. Combining run-length encoding with entropy encoding of the command sequence yields compression rates of less than 1 bit per vertex. For meshes with regular topologies, where most of the vertices have the same degree, spectacular compression rates, such as 0.2 bits per vertex, result.

3. Vertex Coordinate Compression

The IBM algorithm [11] encodes the mesh vertex coordinates v_i by first quantizing the three vertex coordinates to a finite number of bits (8 bits is typical) by bounding the interval in which the coordinates lie. The algorithm then uses a linear scheme to predict the integer coordinates of the vertex from a small number (k) of vertices immediately proceeding in the code:

$$v_i^p = round\left[\sum_{j=1}^{k} a_j v_{i-j}\right]$$

The coefficients are real numbers, which, for lack of anything better, are usually taken to be $a_1 = 1, a_2 = a_3 = ... = a_k = 0.$ The integer prediction errors are: $\Delta_i = v_i - v_i^p$ and are encoded using entropy coding.

It is possible to use a more sophisticated prediction scheme, albeit still linear, to predict the vertex from those surrounding it on the mesh surface. This results in a more accurate prediction, therefore smaller and a more concentrated distribution of prediction errors, which may be compressed better with entropy coding. Due to our connectivity encoding scheme, using an active list, whenever a new vertex r is to be decoded, we already have a triangle (u,v,w) neighboring on the triangle the new vertex forms with vertices (u,v) in the active list. This enables us to predict the new vertex using the following rule: $r^p = v + u - w$.

We call this rule the "parallelogram" rule, because its geometric interpretation is to predict r as the fourth vertex of a parallelogram whose three other vertices are v,u and w. This assumes, incorrectly, that all four vertices are co-planar. If it were possible to estimate the "crease angle" between the two triangles along the edge (u,v), a more accurate prediction of r would result. This angle is a discrete "curvature" value. We estimate this crease angle as the average of the crease angles between two sets of adjacent triangles, whose crease is the closest to parallel to the crease line in question. At least one such set must already have been decoded, so this information is available. See Fig. 1.

The resulting integer prediction errors tend to cluster around zero. We found it most effective to store an explicit codebook for the 32 error values surrounding zero, and to assume a fixed rate code for all remaining error values. This reduces the size of the codebook which must be stored with the codewords without damaging the compression ratio significantly.

4. Complexity

The most complex operation in the mesh connectivity encoding procedure is searching for a given vertex in some active list on the stack. This is needed only to support "merge" operations, which are extremely rare. Apart from that, both the space and time requirements of our encoding and decoding algorithms are linear in the number of mesh edges. For the examples brought in Section 5, both IBM's and our encoding/decoding algorithms ran for a few seconds on a 166MHz Pentium PC computer.

5. Experimental Results

We have run our compression algorithm on a variety of triangle meshes found in some common VRML97 files (i.e. IndexFaceSets), and compared them with the results obtained by simple gzip of the ASCII content (after coordinate quantization), and the IBM algorithm. Fig. 5 shows some of the meshes, and Table 1 summarizes the compression results. As expected, gzipped ASCII is highly inefficient, even after quantized to 8 bits per vertex. The IBM algorithm reduces the dataset

size by an additional factor of 0.16 on the average. On top of that, we reduce the overall dataset size by an average additional factor of 0.61 relative to the IBM algorithm. Our connectivity encoding algorithm is far superior, reducing the dataset size relative to the IBM algorithm by an average of 0.34. Our vertex coordinate compression reduces the dataset size by 0.69. Since this is the dominant component of the mesh geometry, it sets the tone for the overall compression ratio.

Increasing the number of quantization bits per coordinate for the vertices from 8 to 10 increases the size of the code by 30%-40% on the average, both for the IBM algorithm, and ours.

Running the standard gzip compression utility on the compressed data does not reduce its size any further. On the contrary, it only increases it. This is a positive indication that our compression algorithm is doing a good job, and no additional "general purpose" compression techniques are applicable. Our compression algorithm is idempotent, namely, applying it again to a decoded dataset yields an encoded dataset identical to the original encoded version.

6. Discussion

We do not think it will be possible to achieve compression ratios significantly better than ours for mesh connectivity. For regular meshes, such as those obtained from CAD systems by tesellating free-form surfaces, or those obtained by subdivision techniques (e.g. the "eight" and "shape" meshes of Table 1), we are able to compress the connectivity data to almost nothing (0.2 bits/vertex). On the other hand, we are confident that significant improvements are still possible for the vertex coordinate compression, possibly at the cost of longer decoding time. This is the subject of our current research. We are exploring some improved non-linear prediction methods involving local surface curvature ("crease angle"), and preliminary results are encouraging. We are also investigating two-pass encoding/decoding methods, in which the connectivity of the entire mesh is first decoded, and only then the coordinate decoding is started. This contrasts with the method described in Section 3, where both the mesh connectivity and vertex coordinates are decoded in a single pass. Two-pass methods have the advantage that more complete connectivity information is available at the time of the coordinate decoding.

Our treatment of non-manifold meshes is identical to that of the IBM algorithm: A single non-manifold mesh is decomposed into a number of manifold meshes, each of which is encoded separately. General VRML geometric content contains information other than mesh connectivity and geometry, e.g. normal vector data, colors and other properties. We are currently working on efficient coding of the normal data, which seems will be more compact than the coordinate data. Other vertex properties will be addressed too.

Beyond its obvious use for reducing storage and transmission costs, mesh encoding is relevant to the efficient rendering of the meshes on graphics engines. Most modern graphics engines employ a graphics pipeline, through which the mesh polygons travel during their rendering. The pipeline first performs the geometric projection transformation on the vertices, and then scan-converts the projection interior, with appropriate shading and depth (z-buffer) calculations. Naive rendering of a triangle mesh involves sending each individual triangle down the pipeline. This is also the naive way of specifying (encoding) the edge information of the mesh: The vertices are assigned indices, and a list of index triplets describe the triangle mesh. This means that each vertex index will be specified three times on the average (to describe all edges, we need to specify only *half* the triangles). In terms of rendering, each vertex will undergo the projective transformation three times on the average. Encoding the mesh more efficiently can help reduce this cost, possibly at the price of some extra storage space. One way of doing this is to partition the mesh to triangle *strips* [5], each of which may be rendered efficiently while projecting each vertex only once. Even if a mesh may be described as a single strip, each vertex must still be specified twice on the average. In the typical case, since a mesh consists of many such strips, more vertices are needed in the "code". In order to decode and make use of the triangle strip information, the rendering engine must use a vertex store for three vertices. Deering [4] proposed to extend the vertex store to more than three vertices, enabling use of a more compact encoding scheme. Deering also pointed out the connection to mesh compression in general.

Chow [3] has recently described some heuristic algorithms, using a store of 16 vertices, that generate codes where each vertex of a typical mesh appears only 1.3 times on the average. Sun Microsystems' Java3D API [10] makes use of these techniques. Independently, Bar-Yehuda and Gotsman [1] proposed a similar scheme, described algorithms for mesh decomposition and provided theoretical bounds on the tradeoff between the size of the vertex store and the rendering efficiency. One of their results is that a n-vertex mesh

may be specified (encoded), such that each vertex appears only once in the code, only if a vertex store of size $\Omega(\sqrt{n})$ is available. This is a lower bound on the space complexity of our decoding algorithm.

Progressive meshes [6] have gained popularity over recent years as a mechanism for streaming geometry over a network, such that the decoder may immediately begin decoding the bit stream, and progressively refine it as more bits are received. However, not only is this type of data hard to compress, but might even enlarge the amount of data which has to be ultimately transmitted, relative to the original mesh size. Future work will address this problem.

References

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

[2] F. Bossen. Geometry compression. *Contribution No. MPEG96/M1236*, October 1996 Chicago MPEG meeting of ISO/IEC JTC1/SC29/WG11.

[3] M. Chow. Optimized geometry compression for real-time rendering. *Proceedings of Visualization '97*, pp. 347-354, IEEE Computer Society Press, 1997.

[4] M. Deering. Geometry compression. *Proceedings of SIGGRAPH* , pp. 13-20, ACM Press, 1995.

[5] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. *Proceedings of Visualization '96*. IEEE Computer Science Press, 1996.

[6] H. Hoppe. Progressive meshes. *Proceedings of SIGGRAPH*, pp. 99-108, ACM Press, 1996.

[7] http://www.vrml.org/Specifications/ VRML2.0

[8] http://www.research.ibm.com/vrml/binary/ specification

[9] K. Keeler and J. Westbrook. Short encoding of planar graphs and maps. *Discrete Applied Maths*, 58:239-252, 1995.

[10] H. Sowizral. *The Java3D API Specification*. Academic Press, 1998.

[11] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *Research Report RC-20340*, IBM Research Division, 1996.

[12] G. Turan. On the succinct representation of graphs. *Discrete Applied Maths*, 8:289-294, 1984.

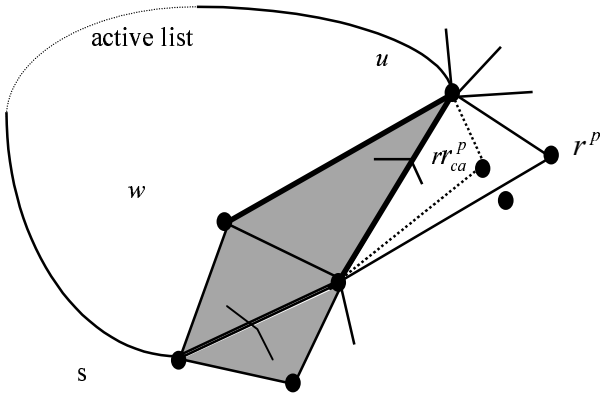


Figure 1: The "parallelogram" rule without and with crease angle prediction. Filled triangles have already been decoded. Vertex r^p is the prediction of true vertex r using the parallelogram rule, and r^r_{ca} the prediction adopting the crease angle between triangles (s,v,w) and (s,v,t).

```
Procedure EncodeConnectivity (Tmesh M)
{
    Stack S;  // Stack of active lists
    ActiveList AL, AL1;

    while not all triangles of M visited {          // catch all connected components
        pick an unvisited triangle (v1,v2,v3) of M;
        AL.Add(v1,v2,v3);
        output ("add %d",v1.degree);
        output ("add %d",v2.degree);
        output ("add %d",v3.degree);
        AL.focus := v1;
        S.Push(AL);

        while not S.Empty() {
            AL := S.Pop();
            while not AL.Empty() {
                e := AL.focus.FreeEdge();  // next free edge in clockwise order
                u := AL.focus.Neighbor(e); // neighboring vertex along edge e
                if u.Free() {
                        AL.Add(u);              // this is always possible
                        output ("add %d", u.degree);
                } else
                if AL.Isin(u) {
                        (AL,AL1) := AL.Split(e);  //split AL to AL and AL1 at edge e, inherit focus
                        S.Push(AL1);
                        output ("split %d", offset from AL.focus to u)
                } else {
                AL1 := S.Isin(u);     // u is already in some list on the stack - find it
                        k := S.Pop(AL1);    // k is index of AL1 on stack
                        AL.Merge(AL1,u);  // merge AL1 with AL at u
                        output ("merge %d %d", k, offset from AL.focus to u)
                }
                AL.RemoveFullVertices();
                        if AL.focus.Full() AL.focus := AL.focus.NextNeighbor();
            }
        }
    }
}
```

Figure 2: Pseudo-code of the connectivity encoding algorithm.

```
Procedure DecodeConnectivity (TMeshPtr pM)
{
    ActiveList AL,AL1;
    Stack S;

   while not EOF {
        read degrees of vertices (v1,v2,v3);
        M.add(v1,v2,v3);   // triangle
        AL.add(v1,v2,v3);
        AL.focus := v1;
        S.Push(AL);

        while not S.Empty() {
            AL := S.Pop();
            while not AL.Empty() {
                e := AL.focus. FreeEdge();
                cmd  := ReadCommand();
                if cmd = "add <deg>" {
                        Vertex u(deg);    // create new vertex u, with the given degree
                        AL.Add(u);              // insert u between focus and its predecessor
                        pM->Add(u);             // update the mesh to have two more edges
                } else
                if cmd = "split <offset>" {
                        (AL,AL1) := AL.Split(e,offset);   // split AL into two at the given offset
                        S.Push(AL1);
                } else
                if cmd = "merge <i> <offset>" {
                        AL1 := S.Pop(i);   // pop the i'th active list from the stack. It must be there.
                        AL.Merge(AL1,offset);  // merge the two active lists together at the given offset
                }
                AL.RemoveFullVertices();
                        if AL.focus.Full() AL.focus := AL.focus.NextNeighbor();
            }
        }
    }
}
```

Figure 3: Pseudo-code of the connectivity decoding algorithm.

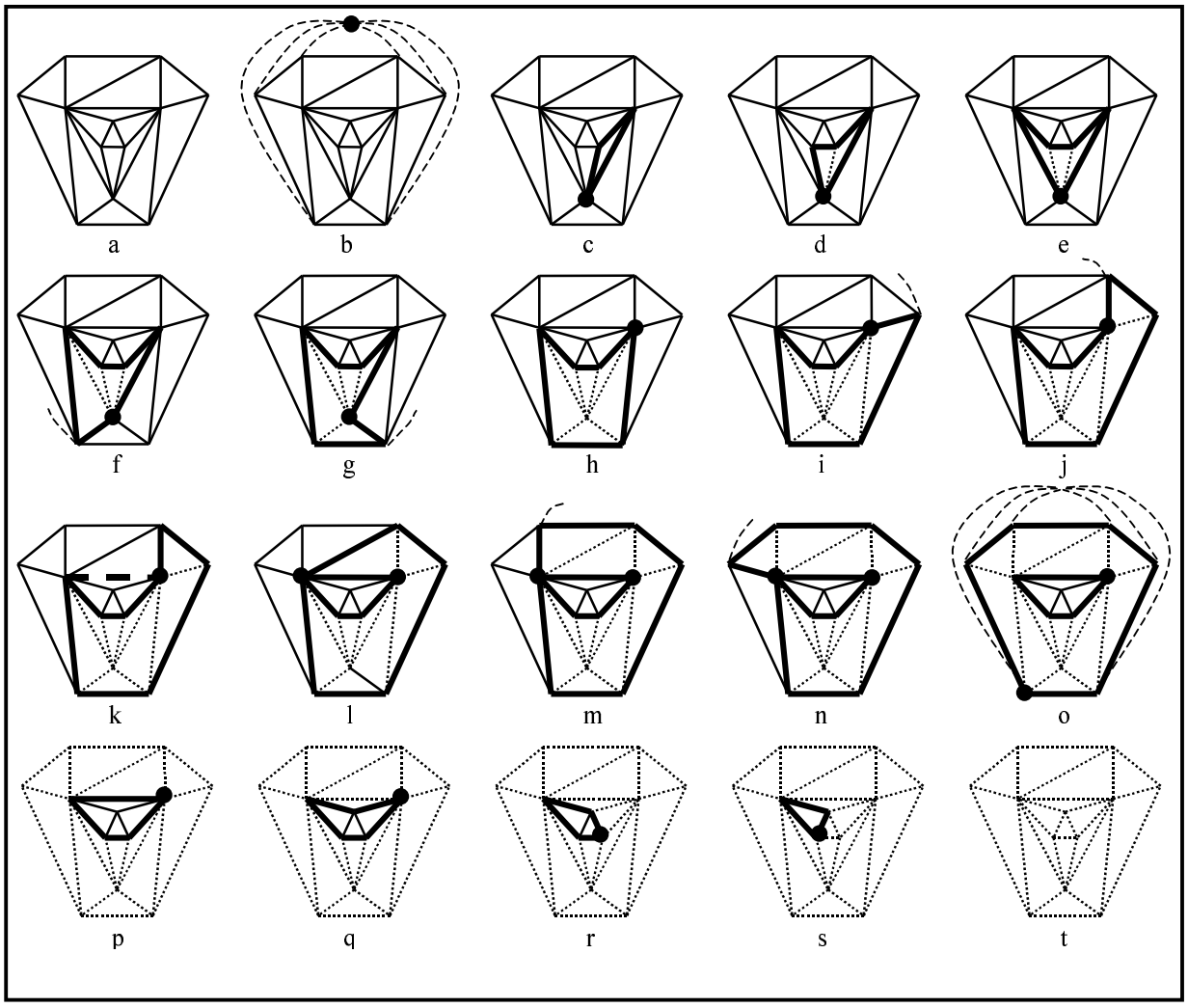
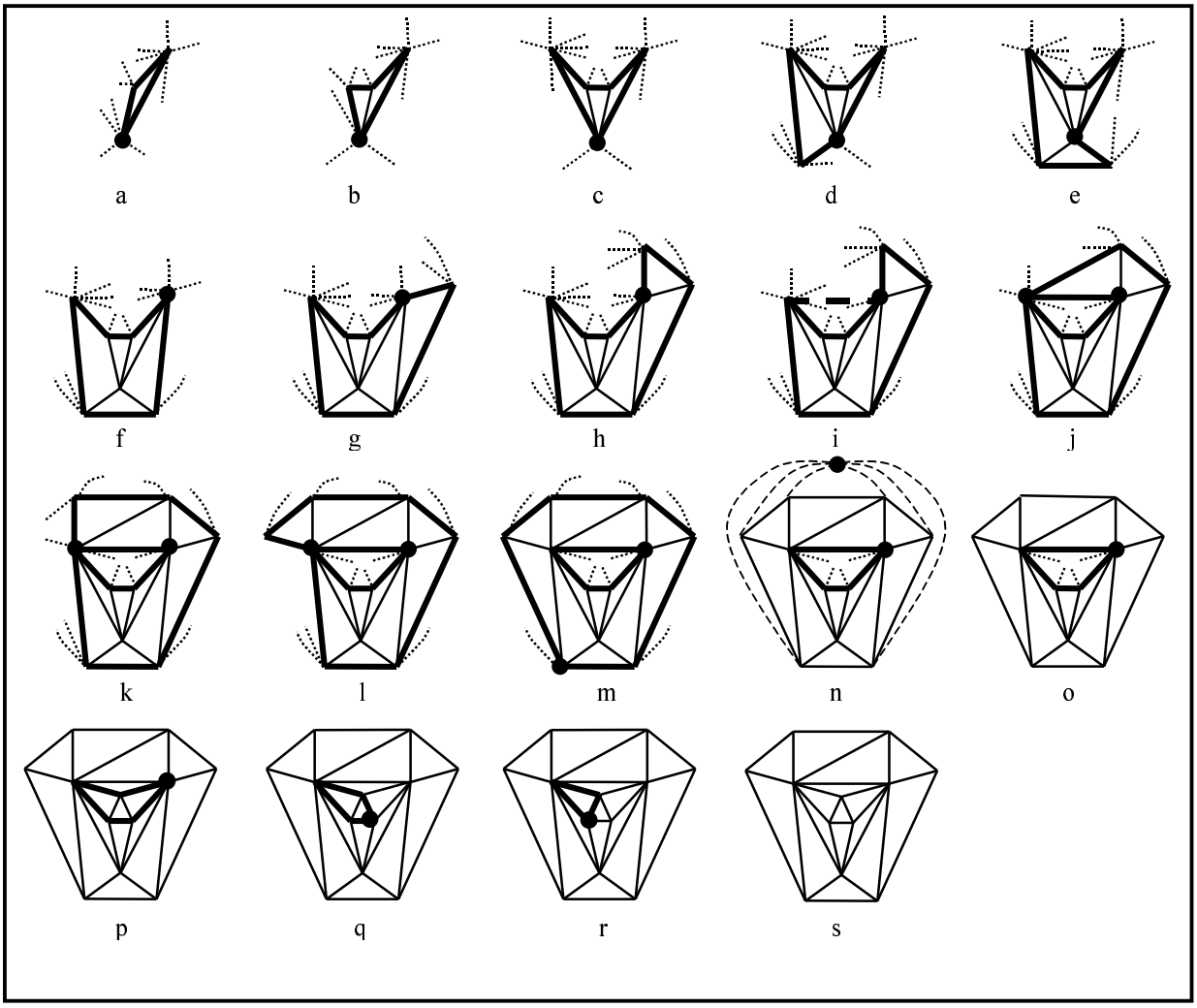


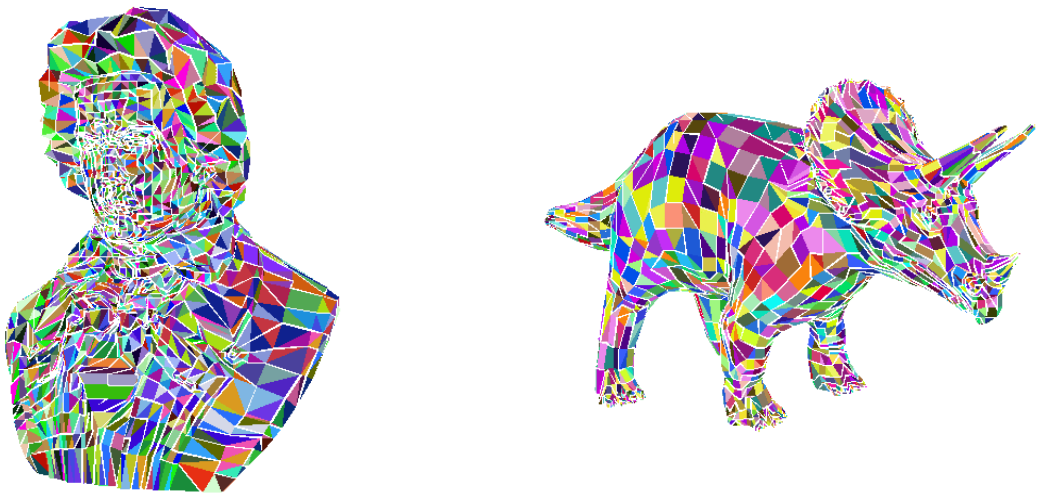
Figure 4: An example of a run of the connectivity encoding algorithm. The active lists are marked by thick lines. Edges already traversed (encoded) are dashed lines. **(a)** Input mesh. **(b)** Dummy vertex added and connected to all boundary vertices. **(c)** Pick initial triangle to start, mark focus vertex, and generate code words "add 6, add 7, add 4". **(d)** Expand the active list and generate code word "add 4". **(e)** "add 8". **(f)** "add 5". **(g)** "add 5". Focus vertex becomes full (all edges encoded). **(h)** Focus vertex removed, and focus moved on along active list. **(i)** "add 4". **(j)** "add 5". Now the next free edge of the focus leads to a vertex already in the active list. **(k)** Active list split into two. Generate code word "split 5" (5 is the offset), and smaller one pushed on stack. **(l)** Focus vertex removed, and focus moved on. **(m)** "add 4". **(n)** "add 4". Focus vertex is full so it is removed. **(o)** The dummy vertex is added "add dummy 6". **(p)** First active list complete. The second active list popped from the stack. **(q)** "add 4". **(r)** Focus vertex removed, and focus moved on. **(s)** Focus vertex removed, and focus moved on. **(t)** Second active list complete. The resulting code is "add 6, add 7, add 4, add 4, add 8, add 5, add 5, add 4, add 5, split 5, add 4, add 4, add dummy 6, add 4".



Decode the mesh using the code “add 6, add 7, add 4, add 4, add 8, add 5, add 5, add 4, add 5, split 5, add 4, add 4, add dummy 6, add 4”. **(a)** Build the initial triangle from the commands “add 6, add 7, add 4”. Mark the first vertex as focus. **(b)** “add 4”: add a vertex with degree 4 and connect it to the focus and its predecessor in the active list. **(c)** The same with the next “add 8”. **(d)** “add 5”. **(e)** “add 5”. Now the focus is full (all edges accounted for). **(f)** Focus vertex removed and focus moved on along active list. **(g)** “add 4”. **(h)** “add 5” **(i)** “split 5”:Split the active list at an offset of 5 free edges (clockwise) from the focus vertex . **(j)** Focus is full and removed. **(k)** “add 4”. **(l)** “add 4”. Focus vertex is now full. **(m)** Focus vertex removed and focus moved on. **(n)** “add dummy 6”: Add a dummy vertex with degree 6, and connect it. **(o)** The first active list is exhausted, and the second popped from the stack. **(p)** “add 4”: A vertex with degree 4 is added to the second active list and connected. **(q)** The focus vertex is full and removed. **(r)** Focus vertex removed and focus moved on. **(s)** The second active list is exhausted and the decoding complete.

Model	Vertices	Gzipped VRML97	IBM's scheme		Our scheme		Our/IBM ratio		
			Conn.	Coord.	Conn.	Coord.	Conn.	Coord.	Total
blob	8036	117K (119)	3447 (3.4)	10352 (10.3)	1709 (1.7)	7951 (7.9)	0.50	0.77	0.70
tricerotops*	2832	44K (127)	1523 (4.3)	3673 (10.4)	764 (2.2)	2937 (8.3)	0.51	0.80	0.71
eight	766	11K (118)	363 (3.8)	1146 (12.0)	53 (0.6)	683 (7.1)	0.16	0.60	0.49
shape	2562	35K (112)	713 (2.2)	4578 (14.3)	48 (0.2)	2990 (9.3)	0.09	0.65	0.57
beethoven*	2655	36K (111)	1585 (4.8)	4982 (15.0)	781 (2.4)	3576 (10.8)	0.50	0.72	0.66
engine	2164	24K (91)	1041 (3.8)	4703 (17.4)	330 (1.2)	3425 (12.0)	0.32	0.73	0.65
dumptruck*	11738	114K (80)	4929 (3.4)	20351 (13.9)	1210 (0.8)	11162 (7.5)	0.25	0.55	0.49
cow	3066	40K (108)	1766 (4.6)	4878 (12.7)	779 (2.0)	3376 (8.8)	0.44	0.69	0.63
Average		(108)	(3.8)	(13.3)	(1.4)	(9.0)	0.34	0.69	0.61

Table 1: Quantitative comparison of compression results between gzipped VRML97, the IBM algorithm and ours. Dataset sizes are measured (for connectivity and coordinate data separately) in bytes. The numbers in parantheses are in bits per vertex. The models are mostly those used in the IBM benchmarks. Models marked by (*) had to be triangulated (in a very simple manner) prior to compression. Quantization of vertex coordinates was to 8 bits.



(a) (b)

Figure 5: Sample meshes used to test our compression algorithm. The mesh polygons are drawn in random colors. The white line shows the active lists used for the compression. These lists traverse all mesh vertices. **(a)** Beethoven **(b)** Tricerotops.