Volume 25 (2006), number 4 pp. 709–716 COMPUTER GRAPHICS *forum*

Compression of Dense and Regular Point Clouds

Bruce Merry, Patrick Marais and James Gain

Collaborative Visual Computing Laboratory, University of Cape Town, South Africa
{bmerry|patrick|jgain}@cs.uct.ac.za

Abstract

We present a simple technique for single-rate compression of point clouds sampled from a surface, based on a spanning tree of the points. Unlike previous methods, we predict future vertices using both a linear predictor, which uses the previous edge as a predictor for the current edge, and lateral predictors that rotate the previous edge 90° *left or right about an estimated normal.*

By careful construction of the spanning tree and choice of prediction rules, our method improves upon existing compression rates when applied to regularly sampled point sets, such as those produced by laser range scanning or uniform tesselation of higher-order surfaces. For less regular sets of points, the compression rate is still generally within 1.5 bits per point of other compression algorithms.

Keywords: compression, point clouds, range scanning, spanning tree

ACM CCS: I.3 [Computer Graphics]: *Picture and image generation* E.4 [Coding and information theory]: *Data compaction and compression*

1. Introduction

Many model acquisition techniques, such as laser range scanning, produce dense sets of 3D points. A mesh may later be created from these points, but this can be a time-consuming process [1]. On the other hand, systems such as QSplat [2] are able to directly render points, and easily support such features as multiresolution rendering. There are thus clear applications for high-quality point-cloud compressors. While mesh compression is a mature field [3], point cloud compression is still relatively new. Furthermore, the bulk of the existing research either focuses on progressive techniques, whereby a model is streamed and displayed at progressive higher detail but at the expense of compression performance, or resamples the point cloud to make it more amenable to a particular form of compression.

Our contribution is a single-rate point-cloud compressor that is optimised for densely and regularly sampled models. The grid-like structure shown in figure 1 is a characteristic of such models, and can arise from range scanning [4], stereopsis [5], isosurface extraction [6] or resampling [7].

Section 2 reviews previous work on point cloud compression. Our technique, based on a spanning tree of the vertices, is presented in Section 3. We present results in Section 4 and conclude in Section 5.

2. Related Work

Here we will review only point-cloud compression; for a discussion of mesh compression, refer to the survey by Alliez and Gotsman [3].

The first class of point-cloud compressors are progressive coders. These begin by encoding a coarse representation of the point cloud, followed by a stream of refinements. Users who download the stream over a network can begin viewing the coarse representation almost immediately, and see refinements as they arrive. Gandoin and Devillers [8] base their approach on a kd-tree: the space is recursively split in half, and the number of points in one half is encoded (the number in the other half being implicit). Empty cells are not further subdivided. They also show how this can be used to encode

Submitted September 2005
Revised April 2006
Accepted September 2006

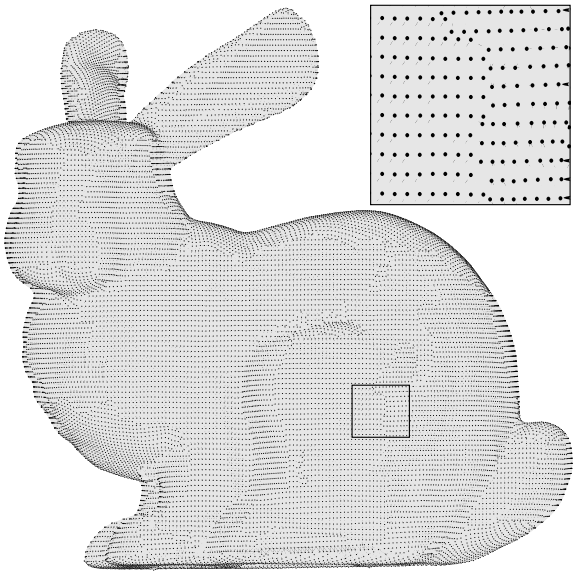


Figure 1: *The bunny model, showing the regular grid patterns. The inset shows a close-up of part of the surface where two scans have been joined.*

a mesh, by encoding extra refinements for the connectivity. Peng and Kuo [9] adapt this approach to an octree, and indicate only whether each subcell is occupied. They obtain better results by using connectivity information to guide the encoding, but do not report the cost of encoding the connectivity information. Botsch *et al.* [10] use a similar octree encoding, but apply it to the intersection of the surface with a voxel grid (i.e., the sampling resolution is equal to the voxel resolution).

Waschbüsch *et al.* [11] use a bottom-up approach. To create a lower-resolution model, the points are grouped into pairs and each pair is replaced by a single point at the midpoint. The offset from the midpoint to the original points is encoded and used during decompression to reconstruct the original points. The offset is encoded in a local coordinate system to take advantage of the fact that the point cloud represents a surface.

Fleishman *et al.* [12] base their approach on an implicit surface defined by the points—the moving least-squares (MLS) surface. They sample an initial (low-resolution) point set from the MLS surface, then encode a sequence of refinements of this point set. Ochotta and Saupe [13] resample the MLS surface at full resolution, then use wavelet techniques to encode a set of height-fields describing the surface. Since both schemes resample the surface, the compression is inherently lossy.

Progressive geometry encoders suffer from a common problem: at the coarser levels of the hierarchy, the distance between points is larger and thus relationships between points are more expensive to encode. Waschbüsch *et al.* [11] partially address this problem by limiting the number of levels in the hierarchy, but the hierarchy cannot become too shallow as the coarsest level is essentially uncompressed.

An alternative to progressive encoding is single-rate encoding, where the entire compressed file must be available before the model can be properly viewed. Taubin and Rossignac [14] compress the geometry of a mesh using an approach similar to our own, in which a spanning tree is built over the edges of a mesh. The spanning tree is constructed in a spiralling pattern from a root vertex. The goal of this algorithm is to construct a tree with long runs of valence-2 vertices, that can be efficiently run-length encoded. Each vertex of the tree is predicted using a linear combination of the ancestor vertices, and a correction to the prediction is encoded. This technique was originally developed for triangle meshes and cannot trivially be adapted to point clouds (because the spiral construction is based on the triangles of the mesh), but the spanning tree idea is nevertheless important.

Gumhold *et al.* [15] again use a similar approach based on spanning trees. Points are added to the tree in a predetermined order, with each point being attached to the parent that best predicts the new point. One of two simple predictors is used: either the child point is predicted at the location of the parent, or the difference between parent and child is predicted to be the same as the difference between the grandparent and the parent.

3. Spanning Tree Compression

Our method is based on similar ideas to the encoding of Taubin and Rossignac [14] and Gumhold *et al.* [15]; however, by using different heuristics and multiple predictors we obtain better compression ratios.

Initially, all coordinates are quantised to some fixed number of bits, as is done for most compression schemes. Although Lee *et al.* [16] have shown that better results can be obtained by quantising in a local coordinate system that is aligned to the surface, global quantisation has the advantage that a point cloud may be decompressed, edited, and recompressed without introducing additional loss. This makes our scheme "lossless" in the same sense that image formats such as PNG [21] are "lossless", even though they quantise colour values to 8 bits per channel.

A rooted spanning tree is then constructed over the points of the model. Each point is predicted from its ancestors in the spanning tree, and corrections to these predictions are encoded. We have several predictors, so the choice of predictor is also encoded. Finally, the connectivity of the tree must be encoded to allow decompression.

The construction of the spanning tree has two conflicting goals. Since the structure of the tree must be encoded, the tree should have an easily encoded structure. The simplest tree

would be just a path with no branches. However, geometry data constitutes the bulk of the code, so it is important to use a tree that generates good geometry predictions. This is most easily ensured by using short edges, which in turn may require more branches in the tree.

In order to eliminate very long edges and also to reduce running time, we initially identify all edges of some maximum length. The maximum we have chosen is the longest edge of the minimum spanning tree, which we denote L. From Kruskal's minimum spanning tree algorithm [17, p. 458], it is clear that this is the smallest global bound that will yield a connected graph.

The compression stage proceeds as follows:

1. Build a minimum spanning tree, and identify the longest edge. This spanning tree is used only to determine L, and is immediately discarded.
2. Build a graph containing all edges that are no longer than L.
3. Construct a spanning tree of this graph. This is the spanning tree used for compression.
4. Assign a predictor and correction to every point other than the root.
5. Encode the choice of predictors and the structure of the spanning tree.
6. Encode the corrections to the predicted positions.

Some of these steps are implemented in parallel, but logically they can be treated as separate passes. We now describe the individual steps in more detail, in the order 4, 3, 5, 6 (step 3 is described after step 4 because it depends on some of the concepts from step 4).

3.1 Geometry encoding

The naïve approach to geometry coding would be to directly encode the difference between the position of each point and that of its parent in the spanning tree. However, while the edges in the spanning tree may be short, they are still too long and too variable to be efficiently encoded.

For a given point v, let v' be the parent in the spanning tree, with corresponding positions $\mathbf{v}$ and $\mathbf{v}'$ in $\mathbb{R}^3$. We define δ_v to be the position of v relative to v' i.e., $\delta_v = \mathbf{v} - \mathbf{v}'$. At each point we also maintain an estimate of the unit normal to the underlying surface, $\mathbf{n}_v$, and the implied "left" vector $\mathbf{l}_v := \mathbf{n}_v \times \delta_v$. For a surface consisting of a rectilinear grid of evenly spaced points, we can expect that δ_v will be either $\delta_{v'}$, $\mathbf{l}_{v'}$ or $-\mathbf{l}_{v'}$ (forward, left or right—see figure 2). To encode $\mathbf{v}$, we indicate which predictor is used (i.e., which is closest), and a correction to the predictor. In practice we also use $\mathbf{0}$ as a predictor, as it yields better predictions where there are

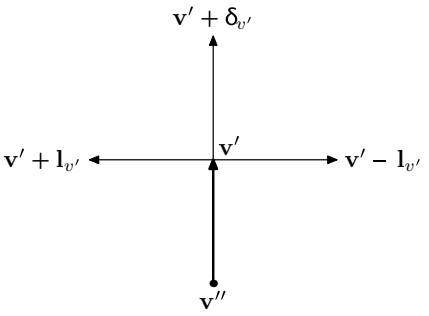


Figure 2: *The possible predictors. The bold arrow indicates the known edge, and the estimated normal $\mathbf{n}_v$ points out of the page. The other arrows indicate the predicted left, forward and right edges. The fourth predictor is at $\mathbf{v}'$.*

sharp changes in the surface; we refer to this as the *base* predictor.

We have used two approaches to encode the corrections. In the *axial* scheme, each coordinate of the correction is separately encoded using progressive arithmetic coding [18]. Although we expect the length of the correction to have some non-uniform distribution (small corrections will hopefully be the most common), the angle of the corrections is more uniform. If the normal prediction was perfect then the portion of the error parallel to the normal could be expected to be smaller than the in-plane error, but in practice we have not been able to exploit this. In the *radial* scheme we encode the length (rounded to the nearest integer) using progressive arithmetic coding. All K possible corrections with the same quantised length are enumerated and the index of the actual correction is encoded using $\log_2 K$ bits.

We also need to propagate the normal estimate over the point set. Since the decompressor requires the normals to interpret the left and right predictors, we must estimate the normal at each point using already available information. We heuristically determine the normal at $\mathbf{v}$ from δ_v, $\delta_{v'}$ and $\mathbf{n}_{v'}$: if the angle between $\delta_{v'}$ and δ_v is at least 30° then we treat $\delta_{v'}$ and δ_v as a basis for the tangent plane, and set

$$\mathbf{n}_v = \delta_{v'} \times \delta_v. \tag{1}$$

If, however, the angle between the vectors is small, this approach leads to instabilities. In this case, we project $\mathbf{n}_{v'}$ onto the plane orthogonal to δ_v:

$$\mathbf{n}_v = \mathbf{n}_{v'} - \frac{(\mathbf{n}_{v'} \cdot \delta_v)\delta_v}{||\delta_v||^2}. \tag{2}$$

After either computation we of course normalise $\mathbf{n}_v$.

We also considered determining the normal by fitting a curve to a set of ancestor edges. Unfortunately, our policy of

favouring forward edges means that the immediate ancestors are often collinear, and the curvature of any fitted curve is a poor indicator of surface normal.

3.2 Spanning tree construction

We construct the spanning tree over the graph of short edges in a priority-first search manner, similar to Prim's algorithm [17, p. 457] for minimum spanning trees. Initially, an arbitrary vertex is designated as the root of the spanning tree. As each vertex is added to the tree, its neighbours are considered as potential children of this vertex and they are added to or updated in a priority queue. In each step, the vertex with lowest cost is added to the tree and removed from the priority queue (a global optimisation may produce better results, but would of course also be prohibitively expensive).

Prim's algorithm uses the edge weight as the cost function to produce a minimum spanning tree. We modify this cost function in two ways:

1. We wish to favour edges that are similar to their parent edges, because these edges will be well predicted. However, using only this metric leads to poor results overall, since edge lengths are not constrained. We have found that the metric
$$E_v = \log(||\delta_v - \delta_{v'}|| + 1) \cdot ||\delta_v|| \tag{3}$$
produces good results over a range of meshes: the first factor favours edges that are similar to their parent edges, while the second penalises long edges. Finding the best metric is still an area of future work.
2. In order to favour long runs, edges emanating from the most recently added vertex are given higher priority than any other. In practice, this means that as soon as any edge is added to the tree, a walk is started from this edge and allowed to continue (always using the best outgoing edge) until there are no unvisited vertices within a distance of L. Here the bound L on edge length plays a role: without a bound, a walk would make a large jump (which cannot be succinctly encoded) to another part of the mesh rather than terminate.

3.3 Spanning tree encoding

We encode the choice of predictors and the connectivity of the spanning tree in a single stream. We first encode the valence of the root with the predictors used for the children of the root. The children are then recursively encoded (depth-first) in the order they are listed in the parent.

We use five symbols for the encoding: B, L, R, F and T. The first four indicate predictors used for child nodes (base, left, right and forward). The T code terminates the list of child predictors, and implicitly specifies the valence. For example, the sequence LFT indicates a node with two children, one predicted left and one forward.

During compression, we concatenate the symbol sequences for all the vertices in a depth-first walk. We expect the model to be dominated by vertices with one forward-predicted child, corresponding to the sequence FT. Hence no one symbol dominates, but from each symbol it is possible to obtain a good prediction of the next symbol. We thus use context-dependent progressive arithmetic coding [18] to encode the sequence. We order the children in the canonical order shown above (B, L, R and F) to improve the performance of this coding scheme.

3.4 Upper bound

We can obtain an asymtotic upper bound for the compression rate in terms of the length of the longest edge of the minimum spanning tree, L (we assume that the point set is scaled such that a quantisation cell has side-length 1). Since we restrict ourselves to edges no longer than L, every δ_v has a correction of length at most L (if necessary, using the base predictor). For reasonably large L, there are about $\frac{4}{3}\pi L^3$ possible corrections that satisfy this property and hence each correction can be encoded in $3 \log_2 L + 2.07$ bits.

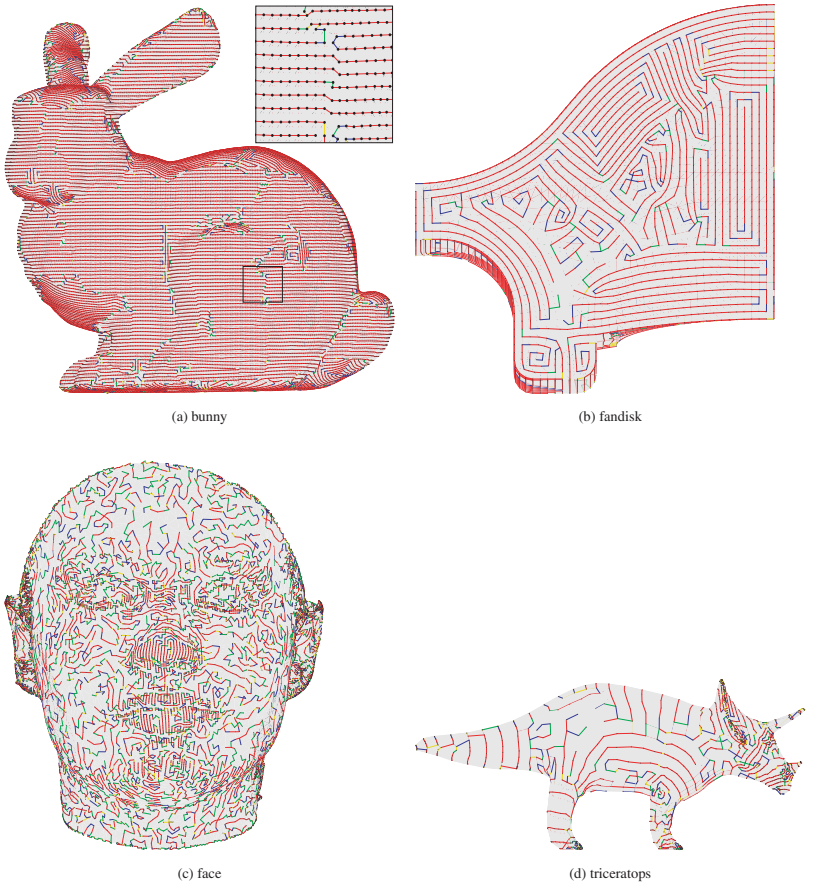
There are five codes used to represent the connectivity of the spanning tree: four predictors and a terminator. The sequence will contain one T code per point (indicating the valences) and one predictor per point (as part of its parent's list of children). In the worst case, all four predictors are used equally often and the entropy is 4 bits per point (this can be achieved by using a 1-bit code for T and a 3-bit code for each predictor). If instead we only use the base predictor, we can reduce the upper bound to 2 bits per point, although in practice this generally increases the number of bits required for the geometry coding.

4. Results

Figure 3 shows the spanning trees generated by our algorithm for several models. The colour of each edge indicates which predictor was used (red for forward predictors, green and blue for left and right predictors, and yellow for base predictors). In some cases it appears that left and right are switched; this occurs because we make no attempt to preserve the sign of the normal.

The top models are regularly sampled, and as a result the models are dominated by the forward predictor and by long runs, which reduces the entropy of the spanning tree encoding. The bottom models are less regularly sampled, and the tree is correspondingly less structured.

Table 1 shows our compression ratios on a range of models. The numbers are bits per point (bpp) for the total encoding (geometry, spanning tree connectivity and predictors).



(a) bunny (b) fandisk

(c) face (d) triceratops

Figure 3: *Spanning trees for some of the compressed models. The colours indicate which predictors are used. The upper models are regularly sampled, and the trees consist mostly of runs of forward predictions which are succinctly encoded. The lower models are less regularly sampled and the resulting trees are less structured.*

We show results with both the axial and radial encoders described in section 3.1. We also compare our results to those of Gumhold *et al.* [15], Gandoin and Devillers [8] and Waschbüsch *et al.* [11], where available (we have listed the results reported by Gumhold *et al.* [15] for the method of Waschbüsch *et al.* [11]). For reference, we list results for the Touma and Gotsman [19] mesh compressor as well. Here we have listed the combined compression rate (i.e., geometry plus connectivity), because the geometry portion of the code cannot be used to reconstruct the point cloud without the connectivity code. The right-most columns show the improvement (in bpp) that we make relative to the best shown result for point-cloud compressors and the Touma–Gotsman mesh compressor.

The upper half of the table shows models that have been sampled along regular grids, for which our algorithm was designed. For these models, our algorithm gives the best compression ratios. Note that for several models (such as the bunny), the other point-cloud compressors are unable to outperform the Touma-Gotsman mesh compressor, despite not having to encode mesh connectivity. For these models, the axial and radial coders give similar results.

The lower half of the table shows models for which the sampling pattern is less regular. This may occur as the result of model simplification, as in the case of *horse-lres* (which is a simplified version of *horse*—note that what most compression authors refer to as *horse* is in fact *horse-lres*). Here the kd-tree compressor [8] produces better results, as it does not depend on the sampling pattern. Nevertheless, our algorithm produces results that are within 1.5 bits per point of the kd-tree compressor and very similar to the Touma-Gotsman mesh compressor. For these models the radial coder gives better compression ratios than the axial coder in all but one case.

It should be noted that the Touma-Gotsman compressor is no longer the state of the art for mesh compression; we have used it because it is freely available online, making it possible to generate a complete set of results. For comparison,

Table 1: *Compression rates of our compression, using the axial and radial encoders (best result highlighted). For comparison, we also show results for other point cloud compressors (PPCC [15], KD [8], PCPM [11]; best result highlighted), as well as the Touma and Gotsman [19] mesh compressor (TG). The upper half shows regularly sampled models, while the lower half shows irregularly sampled models. All models were quantised to 12 bits per coordinate.*

Model	Points	PPCC	KD	PCPM	TG	Axial	Radial	Gain (PC)	Gain (TG)
male	148,138	**7.29**		13.59	7.76	**6.45**	6.98	0.8	1.31
igea	134,345	**10.83**		14.28	11.56	**9.20**	9.22	1.6	2.36
rabbit	67,039				12.46	9.86	**9.76**		2.70
horse	48,485				12.63	11.09	**11.03**		1.60
santa	75,781	**12.23**		18.28	11.93	11.57	**11.50**	0.7	0.43
bunny	34,834	**14.31**	14.8	18.22	13.62	11.68	**11.53**	2.8	2.09
armadillo	172,974				12.25	**11.84**	12.01		0.41
fandisk	6475	**12.94**		20.69	14.84	**12.79**	12.87	0.1	2.05
buddha	543,652				13.67	**10.60**	10.79		3.07
feline	49,864				17.63	17.02	**16.89**		0.74
venus	50,002				18.36	17.41	**17.31**		1.05
horse-lowres	19,851	17.41	**16.4**	21.22	17.50	17.86	**17.65**	−1.3	−0.15
dinosaur	14,070				19.80	18.75	**18.38**		1.42
face	12,530				19.73	19.11	**18.85**		0.88
triceratops	2832		**19.2**		22.18	21.80	**20.50**	−1.3	1.68
blob	8036		**20.1**		21.29	21.92	**21.58**	−1.5	−0.29

Table 2: *Statistics of the compression of the models used. L is the longest edge in the spanning tree. Correction length is the L_2 mean length of the correction vectors. The geometry results are for the radial encoding method.*

				Predictors			Correction	Bits per point		
Model	Points	L	Base %	Left %	Right %	Forward %	length	Tree	Geometry	Total
male	148,138	17.5	2	2	4	92	2.4	0.65	6.33	6.98
igea	134,345	29.3	3	4	6	87	4.4	0.96	8.26	9.22
rabbit	67,039	28.5	5	2	2	91	4.4	0.70	9.06	9.76
horse	48,485	83.8	7	6	6	81	8.4	1.13	9.90	11.03
santa	75,781	27.0	4	5	10	80	5.4	1.35	10.15	11.50
bunny	34,834	58.7	2	3	4	91	8.4	0.74	10.80	11.53
armadillo	172,974	18.5	3	11	17	68	4.9	1.97	10.04	12.01
fandisk	6475	128.8	2	3	9	86	16.9	0.97	11.91	12.87
buddha	543,652	29.1	35	15	12	38	4.6	2.00	8.79	10.79
feline	49,864	61.6	12	19	19	51	13.9	2.12	14.78	16.89
venus	50,002	54.2	8	19	19	54	14.5	2.25	15.07	17.31
horse-lowres	19,851	74.6	6	16	16	62	17.1	1.92	15.73	17.65
dinosaur	14,070	79.9	17	23	19	41	18.7	2.26	16.12	18.38
face	12,530	112.4	16	20	18	46	23.6	2.20	16.65	18.85
triceratops	2832	171.6	18	10	16	56	33.4	1.91	18.59	20.50
blob	8036	90.3	12	15	17	56	35.3	2.44	19.13	21.58

FreeLence [20] obtains bitrates of 11.16 bpv and 11.31 bpv for *feline* and *horse-lowres* respectively, but bitrates for the other models above are not reported (also note that these results are for mean square error equivalent to 12-bit quantization, but that no error bound is provided).

Table 2 shows a number of statistics regarding the compression of the models. For the regularly sampled models (top half), the forward predictor dominates and the correction lengths are much less than L, which causes compression ratios to be well below the upper bound.

On a 2 GHz PC, compression and decompression speeds are around 10 and 75k points per second respectively. However, for sparse point-sets (which have a larger value for L), the radial coder is less efficient as it requires the construction of an $O(L^3)$ lookup table. Where compression and decompression of sparse models may be required, the axial

coder eliminates this overhead, generally at the expense of less than 0.5 bpp. Gumhold *et al.* [15] report compression and decompression speeds of 5 and 500k points per second respectively for their scheme, which is also based on spanning trees but is less sophisticated.

The compression time is dominated by the construction of the initial graph (computation of the minimum spanning tree, and determination of all edges of length at most L). Although we use a kd-tree to accelerate the process, our minimum spanning tree implementation is quite crude and could potentially be optimised.

Decompression time is dominated by the progressive arithmetic coder. If decompression speed is a concern, improvements can be made by using a non-progressive coder and storing frequency tables in the header. This would require a two-pass compression, but eliminate the need to dynamically update frequency tables during decompression.

5. Conclusions

Our compression algorithm yields impressive compression rates for models that have been regularly sampled. For less regular samplings, and in particular for decimated models, the compression rates are not as competitive but are nevertheless reasonable. Furthermore, no triangulation is required, which makes our algorithm suitable for real-time compression of point clouds whereas a triangulation pass followed by a (possibly more efficient) mesh compression would be too computationally expensive.

As presented, our method will perform very poorly on models with disjoint components (due to the long minimum spanning tree edge required to connect them), but it could easily be modified use a spanning forest rather than a spanning tree.

While developing our compressor, we experimented with a number of cost functions for constructing the spanning tree, improving compression rates by up to 2 bpp. While our heuristic produces good results, this is clearly still a fertile area for future work. For real-world applications, improvements to the spanning tree construction are particularly attractive because the file-format and decompression algorithm are unaffected.

We have considered only the compression of geometry, but points may have other associated attributes such as colours or normals. We expect that our normal estimation method could be used for effective compression of normals. Where the normal prediction fails, the cost of encoding a large correction would be amortised by the improvements to the lateral predictions due to having a correct normal.

References

1. G. Turk and M. Levoy. Zippered polygon meshes from range images. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, New York, NY, USA, ACM Press, pp. 311–318, 1994.

2. S. Rusinkiewicz and M. Levoy. QSplat, a multiresolution point rendering system for large meshes. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, ACM Press/Addison-Wesley Publishing Co., pp. 343–352, 2000.

3. P. Alliez and C. Gotsman. Recent advances in compression of 3D meshes. In *Advances in Multiresolution for Geometric Modelling*, Springer, 2004.

4. M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade and D. Fulk. The digital Michelangelo project: 3D scanning of large statues. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, ACM Press/Addison-Wesley Publishing Co., pp. 131–144, 2000.

5. O. Faugeras. *Three-Dimensional Computer Vision: A Geometric Viewpoint*, MIT Press, Cambridge, MA, 1993.

6. W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, ACM Press, pp. 163–169, 1987.

7. M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, New York, NY, USA, ACM Press, pp. 173–182, 1995.

8. P.-M. Gandoin and O. Devillers. Progressive lossless compression of arbitrary simplicial complexes. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, ACM Press, pp. 372–379, 2002.

9. J. Peng and C. C. J. Kuo. Octree-based progressive geometry encoder. In *Internet Multimedia Management Systems IV*. Edited by Smith, John R.; Panchanathan, Sethuraman; Zhang, Tong. Proceedings of the SPIE, Vol. 5242, pp. 301–311, 2003.

10. M. Botsch, A. Wiratanaya and L. Kobbelt. Efficient high quality rendering of point sampled geometry. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, Aire-la-Ville, Switzerland, Eurographics Association, pp. 53–64, 2002.

11. M. Waschbüsch, M. Gross, F. Eberhard, E. Lamboray and S. Würmlin. Progressive compression of point-sampled models. In *Eurographics Symposium on Point-Based Graphics*, pp. 95–102, 2004.

12. S. Fleishman, D. Cohen-Or, M. Alexa and C. T. Silva. Progressive point set surfaces. *ACM Trans. Graph.* 22(4): 997–1011, 2003.

13. T. Ochotta and D. Saupe. Compression of point-based 3d models by shape-adaptive wavelet coding of multi-height fields. In *Proceedings Symposium on Point-Based Graphics*, Zürich, Switzerland, June 2004.

14. G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, 1998.

15. S. Gumhold, Z. Karni, M. Isenburg and H.-P. Seidel. Predictive point-cloud compression. In *ACM SIGGRAPH Conference Abstracts and Applications*, 2004.

16. H. Lee, P. Alliez and M. Desbrun. Angle-analyzer: A triangle-quad mesh codec. In *Eurographics Conference Proceedings*, pp. 383–392, 2002.

17. R. Sedgewick. *Algorithms in C*, 2nd ed. Addison-Wesley, 1990.

18. I. H. Witten, R. M. Neal and J. G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, 1987.

19. C. Touma and C. Gotsman. Triangle mesh compression. In *Proc Graphics Interface*, pp. 26–34, 1998.

20. M. Wardetzky, F. Kaelberer, K. Polthier and U. Reitebuch. Freelence—coding with free valences. *Eurographics*, 2005.

21. G. Randers-Pehrson. *Portable Network Graphics (PNG) Specification*, 2nd ed. W3C, October 2003. http://www.w3.org/TR/PNG/.