Lossless Compression of Floating-Point Geometry

Martin Isenburg[1], Peter Lindstrom[2] and Jack Snoeyink[3]

[1]University of North Carolina at Chapel Hill, isenburg@cs.unc.edu
[2]Lawrence Livermore National Laboratory, pl@llnl.gov
[3]University of North Carolina at Chapel Hill, snoeyink@cs.unc.edu

ABSTRACT

The geometric data sets found in scientific and industrial applications are often very detailed. Storing them using standard uncompressed formats results in large files that are expensive to store and slow to load and transmit. Many efficient mesh compression techniques have been proposed, but scientists and engineers often refrain from using them because they modify the mesh data. While connectivity is encoded in a lossless manner, the floating-point coordinates associated with the vertices are quantized onto a uniform integer grid for efficient predictive compression. Although a fine enough grid can usually represent the data with sufficient precision, the original floating-point values will change, regardless of grid resolution.

In this paper we describe how to compress floating-point coordinates using predictive coding in a completely lossless manner. The initial quantization step is omitted and predictions are calculated in floating-point. The predicted and the actual floating-point values are then broken up into sign, exponent, and mantissa and their corrections are compressed separately with context-based arithmetic coding. As the quality of the predictions varies with the exponent, we use the exponent to switch between different arithmetic contexts. Although we report compression results using the popular parallelogram predictor, our approach works with any prediction scheme. The achieved bit-rates for lossless floating-point compression nicely complement those resulting from uniformly quantizing with different precisions.

Keywords: mesh compression, geometry coding, lossless, floating-point.

1. INTRODUCTION

The polygon mesh is the most widely used primitive for representing three-dimensional geometric models. Polygon meshes consist of geometry and connectivity, the first describing positions in 3D space and the latter describing how to connect these positions together into polygons that describe a surface. Typically there are also mesh properties such as texture coordinates, material attributes, etc. that, for example, describe the visual appearance of the mesh at rendering time.

The standard representation of a polygon mesh uses an array of floats to specify the positions and an array of integers containing indices into the position array to specify the polygons. A similar scheme is used to specify the various properties and how they are attached to the mesh. For large and detailed models this representation results in files of substantial size, which makes their storage expensive and their transmission slow.

The need for more compact mesh representations has motivated researchers to develop techniques for compression of connectivity [20,21,6,18,11,7,15], of geometry [3,20,21,14,8], and of properties [19,1,12,13]. The most popular compression scheme was proposed by Touma and Gotsman [21] and generalized to the polygonal case by Isenburg and Alliez [7,8]. It tends to give very competitive bit-rates that continue to be the accepted benchmark for mesh compression [9]. Furthermore, this coding scheme allows compression and decompression to operate out-of-core for compressing gigantic meshes [10].

While connectivity is typically encoded in a lossless manner, geometry compression tends to be lossy. Current schemes require the floating-point coordinates associated with the vertices to be quantized onto a uniform integer grid prior to predictive compression. Usually one can choose a sufficiently fine grid to capture the entire precision that exists in the data. However, the original floating point values will change slightly. Many scientists and engineers dislike the idea of having their data modified by a process outside of their control and therefore often refrain from using mesh compression altogether.

A more scientific reason for avoiding the initial quantization step is a non-uniform precision in the mesh data. Standard 32-bit IEEE floating point numbers have 23 bits of precision within the range of

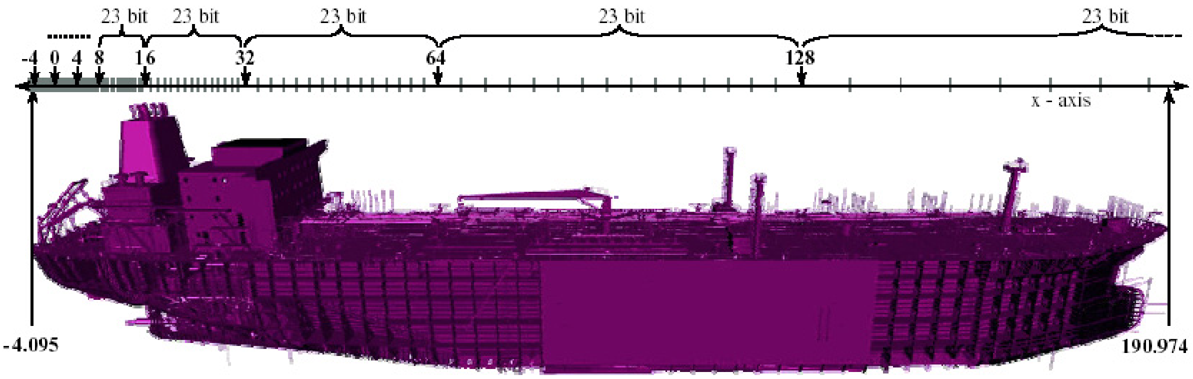


Fig. 1. The x-coordinates of the 81 million triangle Double Eagle tanker range from –4.095 to 190.974. The x-coordinates above 128 have the least precision with 23 mantissa bits covering a range of 128 (i.e. between 128 and 256). There is sixteen times more precision between 8 and 16, where the same number of mantissa bits only have to cover a range of 8.

each exponent (see Fig. 1) so that the least precise (i.e. the widest spaced) numbers are those with the highest exponent. Assuming a uniform sampling, the entire uniform accuracy present in the floating-point samples could be represented with 25 bits once the bounding box (i.e. the highest exponent) is known. If this assumption does not hold because, for example, the mesh was specifically aligned with the origin to provide higher precision in some areas, then uniform quantization is not an option.

Finally, if neither the precision nor bounding-box of the floating-point samples is known in advance it may be impractical to quantize the data prior to compression. Such a situation may arise in streaming compression, as envisioned by Isenburg and Gumhold [10]. In order to compress the output of a mesh-generating application on-the-fly, one may have to operate without a-priori knowledge about the precision or the bounding-box of the mesh.

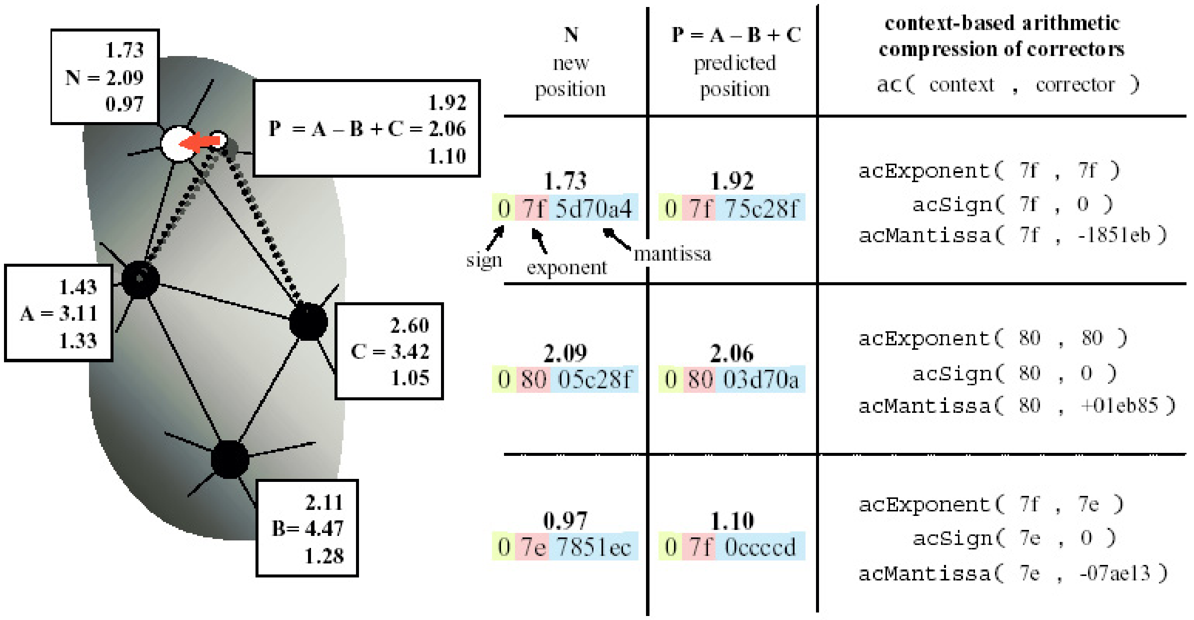
In this paper we describe how to compress floating-point coordinates with predictive coding in a completely lossless manner. The initial quantization step is omitted and the predictions are calculated in floating-point arithmetic. The predicted and the actual floating-point values are then broken up into sign, exponent, and mantissa and their corrections are compressed separately with context-based arithmetic coding. As the quality of the predictions varies with the exponent, we use the exponent to switch between different arithmetic contexts. Although we report compression results using the popular parallelogram predictor, our approach works with any prediction scheme. The achieved bit-rates for lossless floating-point compression nicely complement those resulting from uniformly quantizing with different precisions. Hence, our approach is a completing rather than a competing technology that can be used whenever uniform quantization of the floating-point values is—for whatever reason—not an option.

The remainder of this paper is organized as follows: We give a brief overview of mesh compression in the next section. Following, we describe predictive geometry coding and the initial quantization step. In Section 4 we show how these techniques can operate on floating-point numbers in a completely lossless manner and report compression results. The last section summarizes our contributions and discusses current work.

2. MESH COMPRESSION

The three-dimensional surfaces that are used in interactive visualization or for scientific computations are often represented as polygonal meshes. To accurately represent a detailed model a large number of polygons may be required. Limited transmission bandwidth and storage capacity have motivated researchers to find compact representations for such data, and a number of mesh compression schemes have been proposed. Traditionally, the compression of connectivity, that is the incidence relation among the vertices, and the compression of geometry, that is the actual 3D location of each individual vertex, are done by clearly separated (but often interwoven) techniques. The connectivity coder [20,21,17,6,18,11, 7,15] is usually the core component of a compression engine and drives the compression of geometry [20,21,8] and properties [19,12,13]. Connectivity compression is lossless due to the combinatorial nature of the data. Compression of geometry and properties, however, is lossy due to the mandatory quantization step for the floating-point values.

Recent connectivity compression schemes use the concept of region growing [7] and process adjacent faces one after the other until the entire mesh has been conquered. Most geometry compression schemes use the traversal order this induces on the vertices to compress their (pre-quantized) positions using a predictive coding scheme. Instead of specifying each



N new position	P = A – B + C predicted position	context-based arithmetic compression of correctors ac(context , corrector)

1.73
0 7f 5d70a4
	1.92
	0 7f 75c28f
		acExponent(7f , 7f)
		acSign(7f , 0)
		acMantissa(7f , -1851eb)

2.09
0 80 05c28f
	2.06
	0 80 03d70a
		acExponent(80 , 80)
		acSign(80 , 0)
		acMantissa(80 , +01eb85)

0.97
0 7e 7851ec
	1.10
	0 7f 0ccccd
		acExponent(7f , 7e)
		acSign(7e , 0)
		acMantissa(7e , -07ae13)

Fig. 2. The parallelogram predictor uses the vertices of a neighboring triangle to predict the next vertex. Only a small correction (here: the red arrow) needs to be encoded. We compress the corrections for sign, exponent, and mantissa separately using context-based arithmetic coding [22]. Above, each of the three components is shown in hexadecimal for both the predicted and the actual floating-point number. First we compress the actual exponent while switching contexts based on the predicted exponent. Then we compress whether predicted and actual sign are identical or not. For this we can already switch contexts based on the (previously encoded) actual exponent. Finally we compress the difference between predicted and actual mantissa. Only if sign and exponent were predicted correctly do we actually use the predicted mantissa as the prediction. If the sign was different or if the predicted exponent was smaller we use 0 as the prediction. If the predicted component was larger we use $2^{23} – 1$ as the prediction. This happens above for the z-coordinate: because the exponent 0x7E was miss-predicted as 0x7F we predict the mantissa with 0x7FFFFF and compress the correction 0x7851EC - 0x7FFFFF = –0x07AE13. The function calls on the right refer to the pseudo code from Fig. 6.

position individually, previously decoded vertices are used to predict the next position and only a corrective vector is stored. Virtually all predictive coding schemes used in industry-strength compression engines employ simple linear predictors [3,20,21].

Recently we have seen a number of innovative, yet much more involved approaches to mesh compression. There are spectral methods [14] that perform a global frequency decomposition of the surface, there are space-dividing methods [4] that specify the mesh connectivity relative to a geometric triangulation of connectivity-less coded positions, there are remeshing methods [16,5] that compress a regularly parameterized version instead of the original mesh, and there are feature-based methods [2] that try to find repeated geometric features in a model. We do not attempt to improve on these schemes. Instead we show how predictive geometry compression schemes [3,20,21,8] can be adapted to compress floating-point coordinates in a lossless manner.

3. PREDICTIVE GEOMETRY CODING

The reasons for the popularity of simple prediction schemes are that they are easy to implement robustly, that compression or at least decompression is fast, and that they deliver good compression rates. For several years already, the simple parallelogram predictor [21,8] (see Fig. 2) has become the accepted benchmark that many recent approaches compare themselves with. Although better compression rates have been reported, in practice it is often questionable whether these gains are justified given the sometimes immense increase in algorithmic and asymptotic complexity of the coding scheme. Furthermore these improvements are often specific to a certain type of mesh. Some methods achieve significant gains only on models with sharp features, while others are only applicable to smooth and sufficiently dense sampled meshes. Predictive geometry compression schemes work as follows: First all floating-point positions are converted to integers by uniform quantization with a user-defined precision of for example 12, 16, or 20 bits per coordinate. This introduces a quantization error, as some of the floating-point precision is lost. Then a

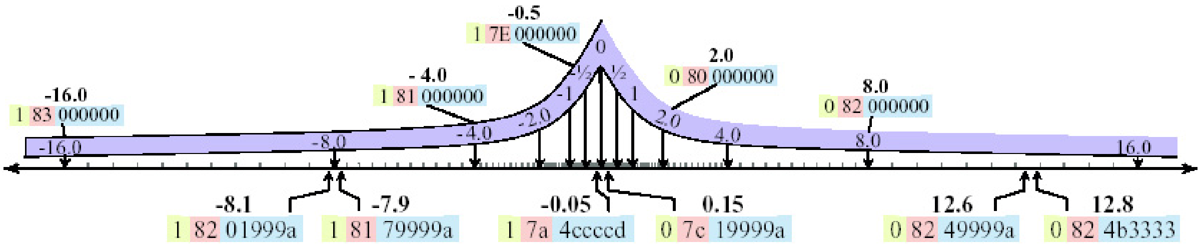


Fig. 3. The non-uniform distribution of floating-point numbers implies that the same absolute prediction error of, for example, 0.2 results in differences that vary drastically with the magnitude (i.e. the exponent) of the predicted numbers.

prediction rule is applied that uses previously decoded integer positions to predict the next position. Finally, an offset vector is stored that corrects the difference between predicted and actual integer position. The values of these corrective vectors tend to spread around zero. This reduces the variation and thereby the entropy of the sequence of numbers, which means that they can be efficiently compressed with, for example, an arithmetic coder [22].

The simplest prediction method predicts the next position as the last position, and was suggested by Deering [3]. While this technique, also known as delta-coding, makes as a systematic prediction error, it can easily be implemented in hardware. A more sophisticated scheme is the spanning tree predictor by Taubin and Rossignac [20]. A weighted linear combination of two, three, or more parent vertices in a vertex spanning tree is used for prediction. By far the most popular scheme is the parallelogram predictor introduced by Touma and Gotsman [21]. A position is predicted to complete the parallelogram that is spanned by the three previously processed vertices of a neighboring triangle.

Predictive compression does not scale linearly with increased precision. Such techniques mainly "predict away" the higher-order bits. If more precision (i.e. low bits) is added the compression ratio (i.e. the compressed size in proportion to the uncompressed size) decreases. This is clearly demonstrated in Table 1, which reports bit-rates for parallelogram predicted geometry at different quantization levels: the achieved compression ratios decrease with increasing precision.

The initial quantization step that maps each floating-point number to an integer makes predictive coding simple. The differences between predicted and actual numbers are also integers and the same absolute prediction error always results in the same difference. When operating directly in floating-point, predictive coding is less straightforward. The non-uniform distribution of floating-point numbers makes compression of the corrective terms more difficult in two ways: First, the difference between two 32-bit floating-point numbers cannot simply be represented by another 32-bit floating-point number (without loss in precision). Second, the same absolute prediction error results in differences that vary drastically with the magnitude of the predicted number, as illustrated in Fig. 3. For the largest numbers there will often only be a difference of a few bits in the mantissa, but for smaller numbers this difference will increase. Especially if the exponent is miss-predicted we can expect a large difference between the mantissas. Miss-predictions of the exponent become more likely for numbers close to zero. Here also the sign may often be predicted incorrectly.

4. LOSSLESS COMPRESSION

In order to compress floating-point coordinates in a lossless manner using the parallelogram predicition rule we split the floating-point numbers into sign, exponent, and mantissa and then treat these three components separately. For a single-precision 32-bit IEEE floating-point number, the sign s is a single bit that specifies whether the number is positive (s = 0) or negative (s = 1), the exponent e is an eight bit number with an added bias of 127 where 0 and 255 are reserved for un-normalized near-zero and infinite values, and the mantissa m is a twenty-three bit number that is used to represent 2^{23} uniformly-spaced numbers within the range associated with the exponent.

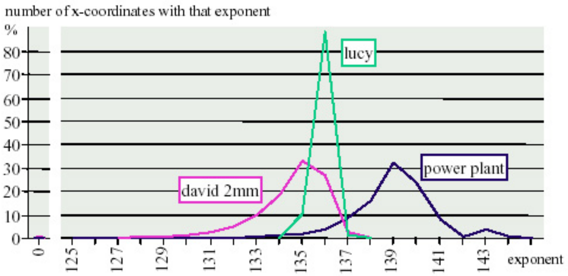


Fig. 4. The distribution of exponents among all x-coordinates for the david (2mm), the lucy, and the powerplant model as percentages of the total. The powerplant's exponents of 143 belong to a building situated far from the main complex.

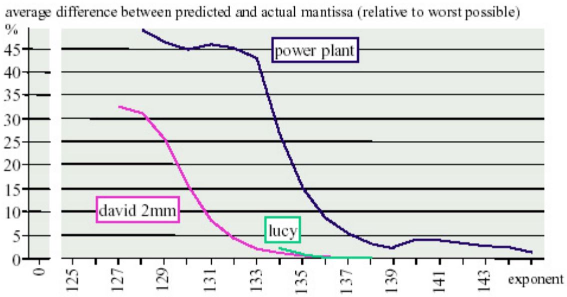


Fig. 5. The average absolute difference between predicted and actual mantissa over all x-coordinates that have the same exponent. The worst possible difference is $+/-2^{22}$.

Our encoder—and also our decoder—compute the parallelogram predictions in floating-point arithmetic and then compress the difference between the predicted and the actual numbers component by component using a context-based arithmetic coder. Especially for the mantissa, the success of the prediction rule is tied to the magnitude (i.e. the exponent) of the number (see Fig. 3). The same prediction error results in a smaller difference in mantissa for numbers with larger exponents. In particular, this difference doubles/halves when the exponent is decreased/increased by one. The spacing between consecutive floating-point numbers changes with the exponent so that more/less of these spacings are required to express that difference. We account for this by switching arithmetic contexts based on the exponents. This prevents the correctors from predictions of numbers with smaller exponent from spoiling the potentially lower entropy of correctors from predictions of numbers with higher exponent.

In order to illustrate the viability of out approach we list in Fig. 4 the distribution of exponents in some of our models and in Fig. 5 the average length of the mantissa corrections for different components. The first set of plots show that only a few exponents are used frequently in typical models. The second set of plots confirm that the mantissa predictions are better for the more frequent numbers with larger exponents.

The pseudo code in Fig. 6 illustrates how we compress the differences in sign, exponent, and mantissa. First we compress the exponent using the predicted exponent to switch contexts. Then we compress whether the sign was predicted correctly or not using the previously encoded exponent to switch contexts. Finally, we compress the difference between predicted and actual mantissa. However, if there was a miss-prediction in the sign or in the exponent we first adjust the prediction for the mantissa. If the sign was miss-predicted or if the exponent was miss-predicted as too large we use the smallest possible mantissa (i.e. 0) as prediction. And if the exponent was miss-

```
void encode(float predicted, float real)
{
  int p_sign = get_sign(predicted);
  int p_expo = get_exponent(predicted);
  int p_mant = get_mantissa(predicted);
  int r_sign = get_sign(real);
  int r_expo = get_exponent(real);
  int r_mant = get_mantissa(real);

  acExponent(p_expo, r_expo);

  if (p_sign == r_sign)
  {
    acSign(r_expo, 0);
    if (p_expo == r_expo)
      acMantissa(r_expo, r_mant - p_mant);
    else if (p_expo > r_expo)
      acMantissa(r_expo, r_mant - ((1<<23)-1));
    else
      acMantissa(r_expo, r_mant - 0x0);
  } else {
    acSign(r_expo, 1);
    acMantissa(r_expo, r_mant - 0x0);
  }
}
```

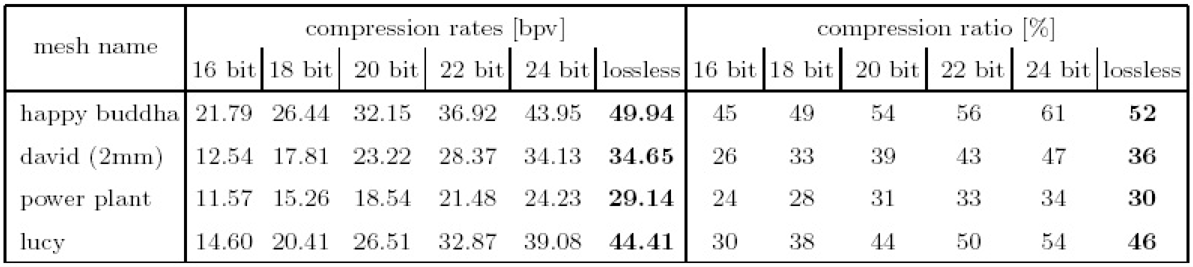
```
void acSign(int context, int corrector)
{
  compress(sign[context], corrector)
}

void acExponent(int context, int exponent)
{
  compress(exponent[context], exponent)
}

void acMantissa(int context, int corrector)
{
  if (corrector < -(1<<22)) corrector += 1<<23;
  if (corrector > (1<<22)) corrector -= 1<<23;
  compress(mantissa[context], corrector)
}

void compress(ArithmeticContext* ac, int sym)
{
  arithmetic_coder->compress(ac, sym);
}
```

Fig. 6. Pseudo code illustrating our floating-point compressor: First we compress the exponent, then whether the sign was predicted correctly, and finally the difference between predicted and real mantissa. Miss-predictions in sign or exponent are used to adjust the prediction for the mantissa. The calls to the functions on the right are mainly for clarity. They call the arithmetic coder with the appropriate context to do the actual compression of the corrective values.



mesh name	compression rates [bpv]						compression ratio [%]					
	16 bit	18 bit	20 bit	22 bit	24 bit	lossless	16 bit	18 bit	20 bit	22 bit	24 bit	lossless
happy buddha	21.79	26.44	32.15	36.92	43.95	**49.94**	45	49	54	56	61	**52**
david (2mm)	12.54	17.81	23.22	28.37	34.13	**34.65**	26	33	39	43	47	**36**
power plant	11.57	15.26	18.54	21.48	24.23	**29.14**	24	28	31	33	34	**30**
lucy	14.60	20.41	26.51	32.87	39.08	**44.41**	30	38	44	50	54	**46**

Table 1. This table lists results for lossless geometry compression in bits per vertex (bpv) side-by-side with the bit-rates that are obtained when first uniformly quantizing the geometry with 16, 18, 20, 22, and 24 bits of precision. In addition we list the achieved gains as the ratio between the compressed and the corresponding uncompressed bit-rates. These are calculated as three times the precision for the pre-quantized geometry and as three times 32 bits for lossless floating-point geometry.

predicted as too small then we use the largest possible mantissa (i.e. 2^{23}-1) as prediction.

Whereas the corrector for the sign and the exponent can be efficiently compressed in "one piece", the mantissa corrector needs to be broken into several chunks prior to arithmetic coding. Otherwise we would require one gigantic arithmetic context table with 2^{23} entries in order to accommodate all possible correctors between -2^{22}+1 and 2^{22}. We first compress whether the corrector is positive or negative using a binary context. Then we compress the lower and the upper 11 bits of its absolute value 0 to 2^{22}-1 using two context tables with 2^{11} entries. The largest possible absolute value of 2^{22} is clamped to 2^{22}-1. Only if the compressed value happens to be this value of 2^{22}-1 we need a final binary context to specify whether this is really 2^{22}-1 or whether this is the clamped value of 2^{22}.

In Table 1 we list example bit-rates for our lossless floating-point geometry compressor side by side with the results of [10] where the bounding box is first uniformly quantized with 16, 18, 20, 22, and 24 bits. The achieved bit-rates for lossless compression nicely complement those resulting from quantizing at different precisions. On various example models, our encoding scheme compresses the floating-point data down to between 30% and 52% of the 96 bits per vertex (bpv) required for uncompressed storage.

5. SUMMARY AND CURRENT WORK

In this paper we have described how to efficiently compress floating-point coordinates of polygonal meshes in a lossless manner. For this we omit the quantization step, compute a prediction in floating-point, and separately compress the difference between predicted and actual sign, exponent, and mantissa using context-based arithmetic coding. We exploit the correlation among these three components by compressing them in correlation order. In particular, we use the exponent to switch contexts between predictions. This prevents predictions for numbers with smaller exponents, which are expected to be less accurate, from spoiling the entropy of better predictions. Furthermore, we use miss-predictions in the sign or the exponent to adjust the prediction of the mantissa. The presented approach can be seen as a completing rather than competing technology that can be used whenever quantization of the floating-point values is not an option. It may also be used to predictively compress floating-point data in other contexts given that reasonable predictions are available. Without modification our coder also compresses special numbers such as infinity or zero in an efficient way.

One benefit of lossless floating-point compression is that it does not require a-priori knowledge about the precision or bounding-box of the data. However, if the precision in the data is known to be uniform or if it is sufficient to preserve, for example, only 16 uniform precision bits then it would be wasteful to losslessly compress the floating-point values. Currently we are designing a scheme that can quantize and compress a stream of floating-point numbers on-the-fly (i.e. in a single pass) by learning the bounding box while guaranteeing a user-specified number of precision bits.

Acknowledgements
This work was performed under the auspices of the U.S. DOE by LLNL under contract no. W-7405-Eng-48. The Happy Buddha and Lucy are courtesy of the Stanford Computer Graphics Laboratory. The Power Plant model was provided by the Walkthru Project at the University of North Carolina at Chapel Hill. The Double Eagle model is courtesy of Newport News Shipbuilding. The David statue is courtesy of the Digital Michelangelo Project at Stanford University.

6. REFERENCES

[1] C. Bajaj, V. Pascucci, and G. Zhuang. Single resolution compression of arbitrary triangular meshes with properties. In Data Compression Conference'99 Conference Proceedings, pages 247–256, 1999.

[2] S. Bhakar D. Shikhare and S.P. Mudur. Compression of 3D engineering models using

discovery of repeating geometric features. In Proceedings of Workshop on Vision, Modeling, and Visualization, pages 233–240, 2001.

[3] M. Deering. Geometry compression. In SIGGRAPH'95 Conference Proceedings, pages 13–20, 1995.

[4] O. Devillers and P.-M. Gandoin. Progressive and lossless compression of arbitrary simplicial complexes. In SIGGRAPH'02 Conference Proceedings, pages 372–379 , 2002.

[5] X. Gu, S. Gortler, and H. Hoppe. Geometry images. In SIGGRAPH'02 Conference Proceedings, pages 355–361, 2002.

[6] S. Gumhold and W. Strasser. Real time compression of triangle mesh connectivity. In SIGGRAPH'98 Conference Proceedings, pages 133–140, 1998.

[7] M. Isenburg. Compressing polygon mesh connectivity with degree duality prediction. In Graphics Interface'02 Conference Proceedings, pages 161–170, 2002.

[8] M. Isenburg and P. Alliez. Compressing polygon mesh geometry with parallelogram prediction. In Visualization'02 Conference Proceedings, pages 141–146, 2002.

[9] M. Isenburg, P. Alliez, and J. Snoeyink. A benchmark coder for polygon mesh compression. In http://www.cs.unc.edu/ ˜ isenburg/pmc/

[10] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. In SIGGRAPH'03 Conference Proceedings, pages 935–942, 2003.

[11] M. Isenburg and J. Snoeyink. Face Fixer: Compressing polygon meshes with properties. In SIGGRAPH'00 Conference Proceedings, pages 263–270, 2000.

[12] M. Isenburg and J. Snoeyink. Compressing the property mapping of polygon meshes. In Pacific Graphics'01 Conference Proceedings, pages 4–11, 2001.

[13] M. Isenburg and J. Snoeyink. Compressing texture coordinates with selective linear predictions. In Proceedings of Computer Graphics International'03, pages 126–131, 2003.

[14] Z. Karni and C. Gotsman. Spectral compression of mesh geometry. In SIGGRAPH'00 Conference Proceedings, pages 279–286, 2000.

[15] A. Khodakovsky, P. Alliez, M. Desbrun, and P. Schroeder. Near-optimal connectivity encoding of 2-manifold polygon meshes. Graphical Models, 64(3-4):147–168, 2002.

[16] A. Khodakovsky, P. Schroeder, and W. Sweldens. Progressive geometry compression. In SIGGRAPH'00 Conference Proceedings, pages 271–278, 2000.

[17] J. Li, C. C. Kuo, and H. Chen. Mesh connectivity coding by dual graph approach. Technical report, March 1998.

[18] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. IEEE Transactions on Visualization and Computer Graphics, 5(1):47–61, 1999.

[19] G. Taubin, W.P. Horn, F. Lazarus, and J. Rossignac. Geometry coding and VRML. Proceedings of the IEEE, 86(6):1228–1243, 1998.

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

[21] C. Touma and C. Gotsman. Triangle mesh compression. In Graphics Interface'98 Conference Proceedings, pages 26–34, 1998.

[22] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. Communications of the ACM, 30(6):520–540, 1987.