ELSEVIER

Computer-Aided Design 37 (2005) 869–877

COMPUTER-AIDED DESIGN

www.elsevier.com/locate/cad

Lossless compression of predicted floating-point geometry

Martin Isenburg[a,*], Peter Lindstrom[b], Jack Snoeyink[a]

[a]*Department of Computer Science, College of Arts and Sciences, University of North Carolina at Chapel Hill, Campus Box 3175, Sitterson Hall, Chapel Hill, NC 27599-3175, USA*

[b]*Lawrence Livermore National Laboratory, 7000 East Avenue, Livermore, CA 94550, USA*

Accepted 20 September 2004

Abstract

The size of geometric data sets in scientific and industrial applications is constantly increasing. Storing surface or volume meshes in standard uncompressed formats results in large files that are expensive to store and slow to load and transmit. Scientists and engineers often refrain from using mesh compression because currently available schemes 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 to enable 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 a method for compressing floating-point coordinates with 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 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. We report compression results using the popular parallelogram predictor, but our approach will work with any prediction scheme. The achieved bit-rates for lossless floating-point compression nicely complement those resulting from uniformly quantizing with different precisions.
© 2004 Elsevier Ltd. All rights reserved.

Keywords: Mesh compression; Geometry coding; Lossless; Floating-point

1. Introduction

Irregular surface or volume meshes are widely used for representing 3D geometric models. These meshes consists of mesh *geometry* and mesh *connectivity*, the first describing the positions in 3D space and the latter describing how to connect these positions into the polygons/polyhedra that the surface/volume mesh is composed of. Typically there are also mesh *properties* such as colors, pressure or heat values, or material attributes.

The standard representation for such meshes uses an array of floats to specify the positions and an array of integers containing indices into the position array to specify the polygons/polyhedra. A similar scheme is used to specify the various properties and how they attach 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 [6–8,10,14,19,24], of geometry [6,9,10,23,24], and of properties [2,15,16,22]. The most popular compression scheme for triangulated surface meshes was proposed by Touma and Gotsman [24]. It was later generalized to both polygonal surface and hexahedral volume meshes [8–10]. It tends to give very competitive bit-rates and continues to be the accepted benchmark coder for mesh compression [11]. Furthermore, this coding scheme allows single-pass compression and decompression for out-of-core operation on gigantic meshes [12].

While connectivity is typically encoded in a lossless manner, geometry compression tends to be lossy. Current schemes quantize floating-point coordinates and other properties associated with the vertices onto a uniform

* Corresponding author.
E-mail address: isenburg@cs.unc.edu (M. Isenburg).

0010-4485//$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cad.2004.09.015

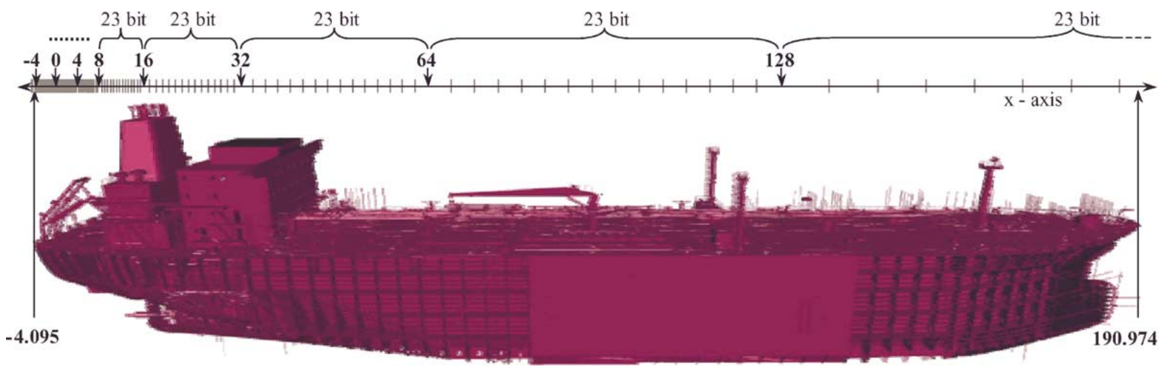


Fig. 1. The x-coordinates of this 75 million vertex Double Eagle tanker range from −4.095 to 190.974. The coordinates above 128 have the least precision with 23 mantissa bits covering a range of 128. There is 16 times more precision between 8 and 16, where the same number of mantissa bits only have to cover a range of 8.

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. Scientists and engineers typically 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 distribution of precision in the data. Standard 32-bit IEEE floating-point numbers have 23 bits of precision within the range of each exponent (see Fig. 1) so that the least precise (i.e. the widest spaced) numbers are those with the highest exponent. If we can assume that all samples are equally accurate, then the entire *uniform* precision present in the floating-point samples can be represented with 25 bits once the bounding box (i.e. the highest exponent) is known. But 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 it was envisioned by Isenburg and Gumhold [12]. 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 investigate how to compress 32-bit IEEE floating-point coordinates with predictive coding in a completely lossless manner. The initial quantization step is omitted and predictions are calculated in floating-point arithmetic. The predicted and the actual floating-point values are broken up into *sign*, *exponent*, and *mantissa* and their corrections are compressed separately with context-based arithmetic coding [25]. As the quality of predictions varies with the exponent, we use the exponent to switch between different arithmetic contexts. We report compression results for single-precision floating-point coordinates predicted with linear predictions. However, our coding technique can also be used for other types of floating-point data or in combination with other prediction schemes. 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.

Compared to the preliminary results of this work that were reported in [13] we achieve improved bit-rates, faster compression and decompression, and lower memory requirements. Furthermore we include a detailed comparison between the proposed compression scheme, simpler predictive approaches, and non-predictive gzip compression. This comparison shows that current predictive techniques are not always the best choice. They are outperformed by gzip on data sets that contain frequently reoccuring floating-point numbers.

The remainder of this paper is organized as follows. In Section 2 we give a brief overview of mesh compression. In Section 3 we describe how current predictive geometry coding schemes operate. In Section 4 we show how these schemes can be adapted to work directly on floating-point numbers. In Section 5 we report compression results and timings. Section 6 summarizes our contributions and discusses current and future work.

2. Mesh compression

The 3D surfaces and volumes that are used in scientific simulations or engineering computations are often represented as irregular meshes. Limited transmission bandwidth and storage capacity have motivated researchers to find compact representations for such meshes and a number of compression schemes have been developed. Compression of connectivity and geometry are usually done by clearly separated, but often interwoven techniques.

The connectivity coder [6–8,10,14,19,23,24] is usually the core component of a compression engine and drives the compression of geometry [6,9,10,23,24] and properties [15, 16,22]. Connectivity compression is lossless due to the combinatorial nature of the data. Compression of geometry and properties, however, is lossy due to the initial quantization of the floating-point values.

All state-of-the-art connectivity compression schemes *grow* a region by encoding adjacent mesh elements one after the other until the entire mesh has been conquered. Most compression engines use the traversal orders this induces on the vertices to compress their (pre-quantized) positions with a predictive coding scheme. Instead of specifying positions individually, previously decoded positions 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 [3,23,24].

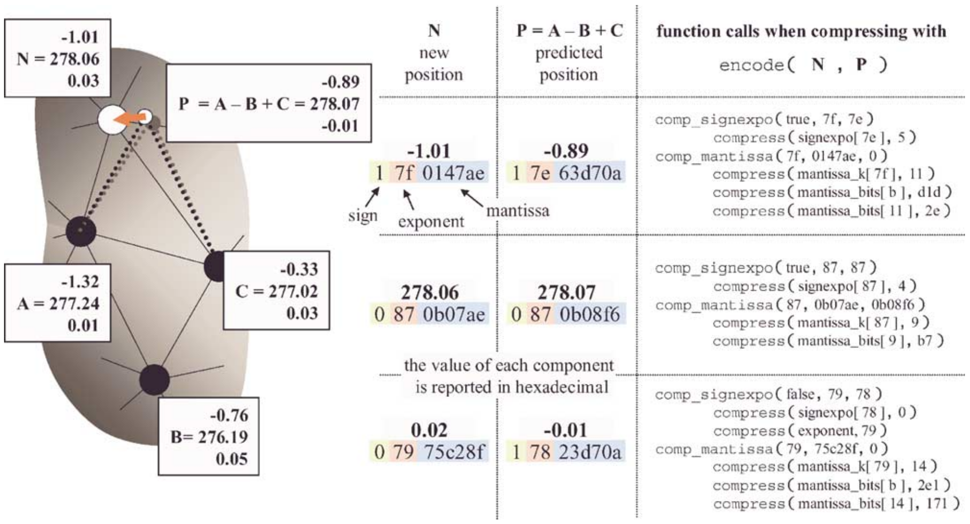
Recently we have seen a number of innovative, yet much more involved approaches to geometry compression. There are spectral methods [17] that perform a global frequency decomposition based on the connectivity, there are space-dividing methods [4] that compress connectivity-less positions using a k–d tree, there are remeshing methods [5,18] that compress a regularly parameterized version instead of the original mesh, and there are high-pass methods [21] that quantize coordinates after a basis transformation with the Laplacian matrix. We do not attempt to improve on these 'lossy' schemes. Instead we show how predictive geometry compression schemes [6,9, 10,23,24] can be adapted to compress floating-point coordinates in a lossless manner.

3. Predictive geometry coding

The reasons for the popularity of linear prediction schemes are that they are easy to implement robustly, that compression and decompression are fast, and that they deliver good compression rates. For several years already, the simple parallelogram predictor [9,24] (see Fig. 2) has been the accepted benchmark that many recent approaches are compared against. 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 densly 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 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 cluster around zero. This reduces the variation and thereby the entropy of the sequence of numbers, which means they can be efficiently compressed with, for example, an arithmetic coder [25].

The simplest prediction method predicts the next position as the last position, and was suggested by Deering [3].



```
comp_signexpo( true, 7f, 7e )
        compress( signexpo[ 7e ], 5 )
comp_mantissa( 7f, 0147ae, 0 )
        compress( mantissa_k[ 7f ], 11 )
        compress( mantissa_bits[ b ], d1d )
        compress( mantissa_bits[ 11 ], 2e )

comp_signexpo( true, 87, 87 )
        compress( signexpo[ 87 ], 4 )
comp_mantissa( 87, 0b07ae, 0b08f6 )
        compress( mantissa_k[ 87 ], 9 )
        compress( mantissa_bits[ 9 ], b7 )

comp_signexpo( false, 79, 78 )
        compress( signexpo[ 78 ], 0 )
        compress( exponent, 79 )
comp_mantissa( 79, 75c28f, 0 )
        compress( mantissa_k[ 79 ], 14 )
        compress( mantissa_bits[ b ], 2e1 )
        compress( mantissa_bits[ 14 ], 171 )
```

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. The coordinates are broken up into *sign*, *exponent*, and *mantissa* components and differences between actual and predicted value are compressed separately using context-based arithmetic coding. The three components for actual and predicted *x*, *y*- and *z*-coordinates are reported in hexadecimal. The function calls refer to the pseudo code from Fig. 4. Compressing the difference between a vertex coordinate and its prediction requires between three and five calls to the arithmetic coder.

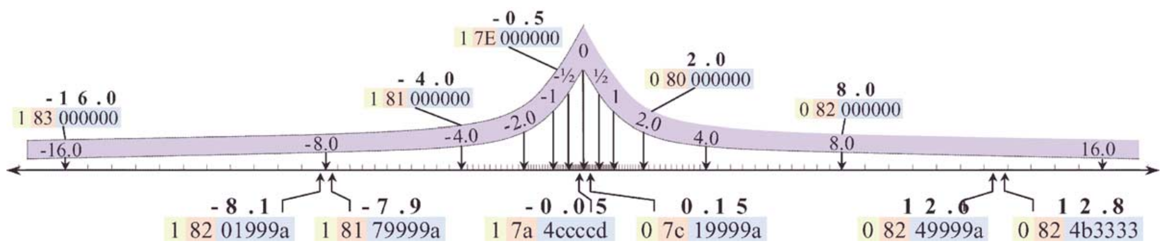


Fig. 3. The non-uniform distribution of floating-point numbers implies that the same absolute prediction error of, for example, 0.2 results in differences that vary drastically with the magnitude (i.e. the exponent) of the predicted numbers.

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 [23]. A weighted linear combination of two, three, or more parents in a vertex spanning tree is used for prediction. By far the most popular scheme is the parallelogram rule introduced by Touma and Gotsman [24]. A position is predicted to complete the parallelogram that is spanned by the three previously processed vertices of a neighboring triangle.

The first vertex of a mesh component has no obvious predictor. We predict its position using the position that was processed last or—if this is the first vertex of the entire mesh—as zero. There will be only one such *null* prediction per mesh component. Also the second and the third vertex of a mesh component cannot be predicted with the parallelogram rule. We predict their position as that of a previously processed vertex to which they connect by an edge. There will be only two such *delta* predictions per mesh component. For all following vertices of a mesh component we use the parallelogram predictor. To maximize compression it is beneficial to compress correctors of the less promising null and delta predictions with different arithmetic contexts [9]. For meshes with few components this hardly makes a difference, but the Power Plant and the Double Eagle tanker each consist of millions of components.

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) increases. This is demonstrated in Table 3, which reports bit-rates for parallelogram predicted geometry at different quantization levels: the achieved compression ratios increase 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 straight-forward. 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 can in general not be represented by a 32-bit floating-point number computed using floating-point arithmetic 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 only be a difference of a few bits in the mantissa, but for smaller numbers this difference will increase. Especially when the sign or the exponents were miss-predicted we cannot expect any correlation 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. Predictive floating-point compression

In order to compress a floating-point coordinate using a floating-point prediction without loss we split both numbers into sign, exponent, and mantissa and then treat these components separately. For a single-precision 32-bit IEEE floating-point number [1], 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 23 bit number that is used to represent 2^{23} uniformly spaced numbers within the range associated with a particular exponent.

We compress the differences in sign, exponent, and mantissa between a floating-point number and its prediction component by component with a context-based arithmetic coder. Especially for the mantissa, the success of the prediction is tied to the magnitude (i.e. the exponent) of the number (see Fig. 3). The same absolute 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. This is because the spacing between consecutive floating-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 exponent. This prevents the high-entropy correctors

```
ArithmeticCoder* arithmetic_coder;
ArithmeticContext* signexpo[256];
ArithmeticContext* exponent;
ArithmeticContext* mantissa_k[256];
ArithmeticContext* mantissa_bits[23];

void encode(float actual, float predicted)
{
  int a_sign = get_sign(actual);
  int a_expo = get_exponent(actual);
  int a_mant = get_mantissa(actual);
  int p_sign = get_sign(predicted);
  int p_expo = get_exponent(predicted);
  int p_mant = get_mantissa(predicted);

  bool same_sign = (a_sign == p_sign);
  bool same_expo = (a_expo == p_expo);

  comp_signexpo(same_sign, a_expo, p_expo);

  if (same_sign && same_expo) {
    comp_mantissa(a_expo, a_mant, p_mant);
  } else {
    comp_mantissa(a_expo, a_mant, 0x0);
  }
}

#define DIFFERENT_SIGN 0
#define SAME_EXPONENT 4
#define OTHER_EXPONENT 8

void comp_signexpo(bool same, int a, int p)
{
  if (same) {
    int d = a - p;
    if (-3 <= d && d <= 3) {
      compress(signexpo[p], SAME_EXPONENT+d);
    } else {
      compress(signexpo[p], OTHER_EXPONENT);
      compress(exponent, a);
    }
  } else {
    compress(signexpo[p], DIFFERENT_SIGN);
    compress(exponent, a);
  }
}
```

```
void comp_mantissa(int expo, int a, int p)
{
  // c will be within [1-2^23 ... 2^23-1]
  int c = a - p;
  // wrap c into [1-2^22 ... 2^22]
  if (c <= -(1<<22)) c += 1<<23;
  else if (c > (1<<22)) c -= 1<<23;
  // find tightest [1-2^k ... 2^k] containing c
  int k = 0;
  // loop could be replaced with faster code
  int c1 = (c < 0 ? -c : c);
  while (c1) { c1 = c1 >> 1; k++ }
  // adjust k for case that c is exactly 2^k
  if (k && (c == 1<<(k-1))) k--;
  // compress k that is now between 0 and 22
  compress(mantissa_k[expo], k);
  // compress lowest k+1 bits of corrector c
  if (k < 12) {
    // translate c into [0 ... 2^(k+1)-1]
    c += ((1<<k) - 1);
    // compress c with appropriate context
    compress(mantissa_bits[k], c);
  } else {
    // break the k+1 bits into two smaller chunks
    int k1 = k - 11;
    // store lower k1 bits in c1
    // store higher 12 bits in c
    if (c < 0) {
      c1 = (-c) & ((1<<k1) - 1); c = -((-c) >> k1);
    } else {
      c1 = c & ((1<<k1) - 1); c = c >> k1;
    }
    // translate c into [0 ... 2^12-1]
    c += ((1<<11) - 1);
    // compress c and c1 with appropriate context
    compress(mantissa_bits[11], c);
    compress(mantissa_bits[k], c1);
  }
}

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

Fig. 4. Pseudo code illustrating the proposed scheme for lossless compression of predicted floating-point numbers. We first compress the common case of a correctly predicted sign and a (nearly) correct predicted exponent while switching contexts based on the *predicted* exponent. Occasionally we need to compress the exponent explicitly. Then we correct the mantissa. If sign or exponent were not predicted correctly we adjust the mantissa's prediction to zero. Next we compress the number of significant corrector bits while switching contexts based on the *actual* exponent. Finally these bits are compressed in one or two chunks depending on their number.

from predictions for numbers with small exponents from spoiling the potentially lower entropy of correctors from predictions for numbers with higher exponents.

The pseudo code in Fig. 4 illustrates how we compress the differences in sign, exponent and mantissa. First, we compress a number between 0 and 8 that specifies: 0 the signs are different, 1–7 the exponents are within plus/minus three, and 8 the exponents differ by more than three. When compressing this number we switch contexts based on the *predicted* exponent. For cases 0 and 8 we have to explicitly compress the exponent, which is done with a separate context. Then we compress how to correct the mantissa. If both signs and exponents were in agreement we use the predicted mantissa as the actual mantissa's prediction or 0 otherwise. We compute the signed corrector that is the shortest modulo 2^{23} and the number k that specifies the tightest interval $(1–2^k, 2^k)$ into which this corrector falls. Next, this number k, which ranges between 0 and 22 is compressed while switching contexts based on the *actual* exponent. Finally the $k+1$ significant bits of the corrector are compressed. This is done in one chunk of $k+1$ bits given that k is smaller than a threshold t and in two chunks of t bits and $k-t+1$ bits otherwise.

The particular choice for the threshold t that splits the $k+1$ significant bits of the corrector into two chunks is mainly a trade-off between keeping the size of the arithmetic tables small and the number of chunks to compress low. In the worst case k is 22 so that there are 23 significant corrector bits to compress. Using a large t means that we often need to compress only one chunk, but results in higher memory requirements and slower updates for the arithmetic tables. We found that the best trade-off is achieved for a t of either 12 or 13. Using $t=12$ results in the smallest tables, but requires more often to compress two chunks than $t=13$. For the results in the paper we used $t=12$.

In order to illustrate that the approach of switching contexts based on the exponent is indeed reasonable, we show in Fig. 5 the distribution of exponents for some of our test models and in Fig. 6 the average number of significant corrector bits $k+1$

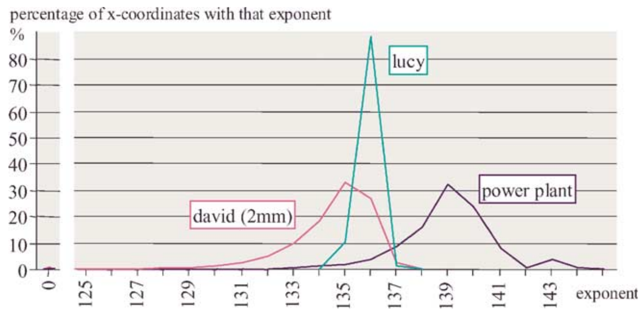


Fig. 5. The distribution of exponents among all x-coordinates for the david (2 mm), the lucy, and the power plant model as percentages of the total. The power plant's exponents of 143 belong to a building situated far from the main complex.

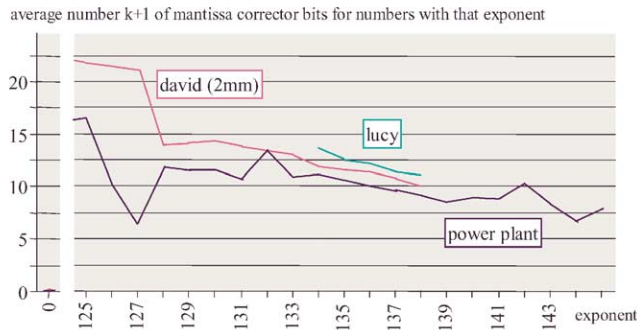


Fig. 6. The average number of significant corrector bits $k+1$ that need to be compressed during predictive coding of the mantissas of all x-coordinates with the same exponent. Predictions for numbers with high exponents are evidently better since they result in correctors with fewer significant bits.

that need to be compressed during predictive coding of the mantissa for numbers with that exponent. The first set of plots shows that only a few exponents are used frequently in typical models. The second set of plots confirms that mantissa predictions are better for the more frequent numbers with large exponents since they result in correctors that have fewer significant bits. These plots also confirm that thresholds t of 12 or 13 assure that a large number of correctors are compressed with a single chunk.

4.1. A simpler prediction scheme

We have also implemented a simpler predictive scheme for lossless floating-point compression for which we give pseudo code in Fig. 7. This scheme simply breaks the actual and the predicted floating-point number into several chunks c_i that are b_i bits long, computes for each chunk c_i a corrector modulo 2^{b_i}, and then compresses those with different arithmetic tables. One objective for having this simpler scheme is to validate if and when the additional correlations that our scheme tries to exploit are worthwhile this effort.

5. Results

In Table 1 we compare our predictive scheme with simpler predictive schemes and standard gzip in terms of compression performance. Our scheme outperforms the simpler schemes by a difference of up to 10 bits per vertex. However, this gain shrinks to just above one bit per vertex as the meshes become large. The simpler schemes '9–12–11' and '10–11–11' are nearly identical in compression performance, but '10–11–11' is slightly faster and uses less memory.

The biggest surprise is the superb performance of standard gzip on the power plant model. This can be explained with the high reoccurance of floating-point numbers by the vertex coordinates. This results in repeating byte pattern that suit gzip compression, but cannot be exploited by traditional predictive coders. On the contrary, the application of prediction rules may *increase* the entropy of the mesh geometry if the corrective values have a distribution with a wider spread than the original positions.

If the compression engine is allowed to produce separate streams or files for each of connectivity, x, y, z-coordinates, and additional properties, then it would be easy to substitute the predictive geometry compressor with gzip or bzip2 compressor whenever suitable. However, in order to create

```
ArithmeticCoder* arithmetic_coder;
ArithmeticContext* chunks[32];

int num = 3;
int bit[]= {10, 11, 11};

// int num = 3;
// int bit[]= {9, 12, 11};
// int num = 4;
// int bit[]= {8, 8, 8, 8};
// int num = 8;
// int bit[]= {4, 4, 4, 4, 4, 4, 4, 4};

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

```
void comp_chunk(int i, int a, int p)
{
  int c = a - p;
  if (c < 0) c += (1 << b[i]);
  compress(chunks[i], c);
}

void encode(float actual, float predicted)
{
  unsigned int a_cast = (unsigned int&)actual;
  unsigned int p_cast = (unsigned int&)predicted;
  for (int i = num-1; i >= 0; i--) {
    int mask = ((1 << bit[i]) - 1);
    comp_chunk(i, a_cast & mask, p_cast & mask);
    a_cast = a_cast >> bit[i];
    p_cast = p_cast >> bit[i];
  }
}
```

Fig. 7. Pseudo code illustrating a simple scheme for lossless compression of predicted floating-point numbers. The number and its predictor are broken into several chunks and corrected chunk-wise using a different arithmetic context for each chunk. We refer to variations of this scheme in Tables 1 and 2 by indicating into which set of chunks the floating-point numbers were broken (i.e. 8–8–8–8 or 9–12–11).

Table 1
This table illustrates the compression performance of our predictive scheme in comparison to simpler schemes and standard gzip compression

Mesh name	Number of vertices	Predictive compression					gzip-9		Unique coordinates		
		4–4–…–4	8–8–8–8	9–12–11	10–11–11	New	*xyzx…*	*x…y…z…*	*x* (%)	*y* (%)	*z* (%)
Golf club	209,779	47.3	43.1	38.4	38.4	29.0	62.0	50.3	8.2	12	8.0
Hip	530,168	53.3	51.6	48.3	48.3	37.9	73.0	67.0	11	10	12
Happy buddha	543,652	55.6	53.2	52.8	52.9	47.4	55.8	50.0	40	29	42
David (2 mm)	4,129,614	38.2	36.0	36.1	36.1	33.7	56.1	47.1	53	38	22
Power plant	11,070,509	36.9	31.8	28.8	28.9	27.2	23.7	8.7	4.4	1.5	2.0
Lucy	14,027,872	45.4	44.8	44.9	44.9	43.7	78.7	73.8	49	82	77

For '*xyzx…*' the vertex coordinates are stored alternating into a single file. For '*x…y…z…*' the vertex coordinates are multiplexed into three different files. The last three columns list for each coordinate the percentage of floating-point coordinates that are actually different.

single-stream encodings it will be necessary to incorporate capabilities into a predictive coder that allow it to deal with data sets that have a high reoccurance rate of sparsely scattered floating-point numbers.

In Table 2 we compare four predictive scheme in terms of compression rates and speeds. While the simple scheme '10–11–11' is the winner in compression speed with nearly 2 million vertices per second, the proposed scheme achieves higher compression rates and has the fastest decompression speeds of up to 1 million vertices per second. The new scheme outperforms our older scheme from [13] in every respect while using fewer and smaller arithmetic tables.

It should be noted that neither scheme's implementation is particularly hand-optimized. The computation speeds are mainly dictated by the efficiency of the entropy coder, which in our case is a range coder implementation adapted from [20]. In particular the simple scheme '8–8–8–8' could achieve significant speed-ups by using an entropy coder optimized for byte-sized symbols. For large models this could make it therefore an overall better choice.

In Table 3 we list the bit-rates of our lossless floating-point geometry compressor side by side with the results of [12] 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 to between 28 and 49% of the original 96 bits per vertex (bpv) required for uncompressed storage.

6. Summary and current work

In this paper we have described how to efficiently compress floating-point coordinates of irregular meshes using predictive coding. For this we omit the traditional

Table 2
This table lists compression rates (rate) in bits per vertex, compression and decompression times (t) in seconds and compression and decompression speed (s) in thousand vertices per second for four predictive floating-point compression schemes: predicting in 4 chunks of 8 bits, predicting in three chunks of 10, 11, and 11 bits, using the old scheme of [13], and using the new scheme proposed here

Mesh name	8–8–8–8			10–11–11			Old [13]			New		
	Rate	$t_{\text{dec}}^{\text{enc}}$	$s_{\text{dec}}^{\text{enc}}$	Rate	$t_{\text{dec}}^{\text{enc}}$	$s_{\text{dec}}^{\text{enc}}$	Rate	$t_{\text{dec}}^{\text{enc}}$	$s_{\text{dec}}^{\text{enc}}$	Rate	$t_{\text{dec}}^{\text{enc}}$	$s_{\text{dec}}^{\text{enc}}$
Golf club	43.1	0.16	1355	38.4	0.12	1741	33.7	0.18	1162	29.0	0.14	1551
		0.26	821		0.23	910		0.34	625		0.22	974
Hip	51.6	0.36	1492	48.3	0.28	1893	43.5	0.46	1151	37.9	0.38	1391
		0.78	679		0.71	746		0.85	622		0.66	802
Happy buddha	53.2	0.34	1596	52.9	0.30	1837	49.9	0.46	1180	47.4	0.38	1448
		0.69	792		0.65	835		0.88	620		0.63	862
David (2 mm)	36.0	2.44	1693	36.1	2.07	1997	34.7	3.29	1257	33.7	2.48	1663
		4.82	856		4.59	897		6.28	657		4.02	1028
Power plant	31.8	6.69	1656	28.9	5.59	1979	29.1	9.41	1175	27.2	6.93	1596
		14.46	765		13.31	831		17.45	634		11.32	978
Lucy	44.8	8.67	1618	44.9	7.38	1901	44.4	11.86	1183	43.7	9.11	1539
		17.91	783		17.60	796		21.70	646		15.77	889
	36 tables		47 KB	27 tables		188 KB	363 tables		1350 KB	217 tables		210 KB

Time measurements are taken on a computer with a 3 GHz Pentium 4 and 1 GB of RAM running Windows XP. The bottom most row reports for each compression scheme the number of range tables used and the total amount of memory needed for storing them.

Table 3
This table lists results for lossless geometry compression in bits per vertex (bpv) side-by-side with the bit-rates that are obtained by [12] after 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

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
Golf club	15.67	20.77	21.56	22.34	25.50	29.05	33	38	36	34	35	30
Hip	19.37	25.37	27.14	27.96	33.60	37.90	40	47	45	42	47	39
Happy buddha	21.79	26.44	32.15	36.92	43.95	47.42	45	49	54	56	61	49
David (2 mm)	12.54	17.81	23.22	28.37	34.13	33.69	26	33	39	43	47	35
Power plant	11.57	15.26	18.54	21.48	24.23	27.15	24	28	31	33	34	28
Lucy	14.60	20.41	26.51	32.87	39.08	43.74	30	38	44	50	54	46

These are calculated as three times the precision for the pre-quantized geometry and as three times 32 bits for the full precision floating-point geometry.

quantization step, compute a prediction in floating-point, and separately compress the differences 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 the order of dependency. 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 sign or 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 uniform quantization of the floating-point values is for some reason not possible. Our scheme 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.

We identified a serious shortcoming of predictive coding schemes that arises whenever a data set contains many sparse samples of high precision that reoccur frequently. As evidenced by comparing the gzip compression results for the power plant from Table 1 with the compression results after quantizing from Table 3 this is also the case when performing predictive coding on pre-quantized data. It seems that it has not been noticed before that current predictive schemes perform so poorly on such data. This will have to be investigated more closely in the future.

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

The Golf Club and the Hip model are courtesy of Cyberware. 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. This work was—in part—performed under the auspices of the U.S. DOE by LLNL under contract no. W-7405-Eng-48.

References

[1] ANSI/IEEE. IEEE standard 754 for binary floating point arithmetic. New York: ANSI/IEEE; 1985.
[2] Bajaj C, Pascucci V, Zhuang G. Single resolution compression of arbitrary triangular meshes with properties. In: Data compression conference'99 conference proceedings; 1999. p. 247–56.
[3] Deering M. Geometry compression for interactive transimmion. In: SIGGRAPH'95 conference proceedings; 1995. p. 13–20.
[4] Devillers O, Gandoin P-M. Geometric compression for interactive transmission. In: Visualization'00 conference proceedings; 2000. p. 319–26.
[5] Gu X, Gortler S, Hoppe H. Geometry images. In: SIGGRAPH'02 conference proceedings; 2002. p. 355–61.
[6] Gumhold S, Guthe S, Strasser W. Tetrahedral mesh compression with the cut-border machine. In: Visualization'99 conference proceedings; 1999. p. 51–8.
[7] Gumhold S, Strasser W. Real time compression of triangle mesh connectivity. In: SIGGRAPH'98 conference proceedings; 1998. p. 133–40.
[8] Isenburg M. Compressing polygon mesh connectivity with degree duality prediction. In: Graphics Interface'02 conference proceedings; 2002. p. 161–70.
[9] Isenburg M, Alliez P. Compressing polygon mesh geometry with parallelogram prediction. In: Visualization'02 conference proceedings; 2002. p. 141–6.

[10] Isenburg M, Alliez P. Compressing hexahedral volume meshes. Graph Models 2003;65(4):239–57.
[11] Isenburg M, Alliez P, Snoeyink J. A benchmark coder for polygon mesh compression. http://www.cs.unc.edu/~isenburg/pmc/
[12] Isenburg M, Gumhold S. Out-of-core compression for gigantic polygon meshes. In: SIGGRAPH'00 conference proceedings; 2003. p. 935–42.
[13] Isenburg M, Lindstrom P, Snoeyink J. Lossless compression of floating-point geometry. Comput Aided Des Appl (CAD'04) 2004; 1(1–4):495–502.
[14] Isenburg M, Snoeyink J. Face fixer: compressing polygon meshes with properties. In: SIGGRAPH'00 conference proceedings; 2000. p. 263–70.
[15] Isenburg M, Snoeyink J. Compressing the property mapping of polygon meshes. Graph Models 2002;64(2):114–27.
[16] Isenburg M, Snoeyink J. Compressing texture coordinates with selective linear predictions. In: Proceedings of computer graphics International'03; 2003. p. 126–31.
[17] Karni Z, Gotsman C. Spectral compression of mesh geometry. In: SIGGRAPH'00 conference proceedings; 2000. p. 279–86.
[18] Khodakovsky A, Schroeder P, Sweldens W. Progressive geometry compression. In: SIGGRAPH'00 conference proceedings; 2000. p. 271–8.
[19] Rossignac J. Edgebreaker: connectivity compression for triangle meshes. IEEE Trans Vis Comput Graph 1999;5(l):47–61.
[20] Schindler M. Rangecoder (v1.3) http://www.compressconsult.com/rangecoder/
[21] Sorkine O, Cohen-Or D, Toledo S. High-pass quantization for mesh encoding. In: Proceedings of symposium on geometry processing'03; 2003. p. 42–51.
[22] Taubin G, Horn WP, Lazarus F, Rossignac J. Geometry coding and VRML. Proc IEEE 1998;86(6):1228–43.
[23] Taubin G, Rossignac J. Geometric compression through topological surgery. ACM Trans Graph 1998;17(2):84–115.
[24] Touma C, Gotsman C. Triangle mesh compression. In: Graphics interface'98 conference proceedings; 1998. p. 26–34.
[25] Witten IH, Neal RM, Cleary JG. Arithmetic coding for data compression. Commun ACM 1987;30(6):520–40.

Martin Isenburg is a PhD student in Computer Graphics at the University of North Carolina at Chapel Hill. He received his MSc in Computer Science from the University of British-Columbia at Vancouver in 1999. His dissertation focuses on compressing and streaming of polygon meshes, both in-core and out-of-core. Recently he has worked on streaming techniques for large data sets at Lawrence Livermore National Laboratory. He enjoys having dinner at Japanese noodle restaurants.

Peter Lindstrom received BS degrees in computer science, mathematics, and physics from Elon College in 1994 and a PhD in computer science from the Georgia Institute of Technology in 2000. He is currently a computer scientist at the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research interests include mesh simplification and compression, multiresolution modeling, scientific visualization, and interactive rendering.

Jack Snoeyink is a professor in the Department of Computer Science at the University of North Carolina at Chapel Hill. He received his PhD in Computer Science from Stanford University in 1990 and spent the rest of the previous millenium at Utrecht as a postdoc and UBC as a faculty member. Jack's research area is computational geometry and its application in GIS, molecular modeling, and computer graphics.