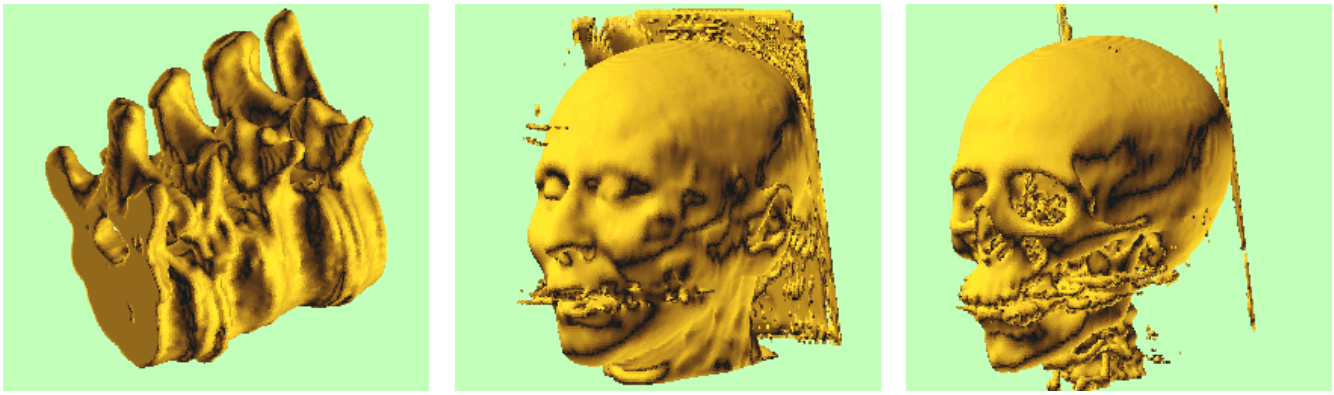
BLIC : Bi-Level Isosurface Compression

Gabriel Taubin*

IBM T. J. Watson Research Center



A B C

Figure 1: Compressed Cuberille isosurfaces as rendered by our simple Java decoder. A: spine set, $91\times512\times512$ voxels, level 1500, 381,278 faces, 381,667 vertices, compressed to **0.6182 bits per face**. B: UNC CThead data set, $113\times256\times256$ voxels , level 600, 294,524 faces, 294,018 vertices, compressed to **0.7437 bits per face**. C: UNC CThead data set, level 1160, 312,488 faces, 312,287 vertices, compressed to **0.8081 bits per face**.

Abstract

In this paper we introduce a new and simple algorithm to compress isosurface data. This is the data extracted by isosurface algorithms from scalar functions defined on volume grids, and used to generate polygon meshes or alternative representations. In this algorithm the mesh connectivity and a substantial proportion of the geometric information are encoded to a fraction of a bit per Marching Cubes vertex with a context based arithmetic coder closely related to the JBIG binary image compression standard. The remaining optional geometric information that specifies the location of each Marching Cubes vertex more precisely along its supporting intersecting grid edge, is efficiently encoded in scan-order with the same mechanism. Vertex normals can optionally be computed as normalized gradient vectors by the encoder and included in the bitstream after quantization and entropy encoding, or computed by the decoder in a postprocessing smoothing step. These choices are determined by trade-offs associated with an in-core vs. out-of-core decoder structure. The main features of our algorithm are its extreme simplicity and high compression rates .

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modelling—surface, solid, and object representations

Keywords: 3D Geometry Compression, Algorithms, Graphics.

*IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598 taubin@us.ibm.com

1 INTRODUCTION

Isosurface extraction algorithms construct polygon mesh approximations to level sets of scalar functions specified at the vertices of a 3D regular grid. The most popular isosurface algorithms [9] are *Cuberille* [1] and *Marching Cubes* [14]. In this paper we refer to the polygon meshes produced by these and related algorithms as *isosurface meshes*. Despite the widespread use of these meshes in scientific visualization and medical applications, and their very large size, special purpose algorithms to compress them for efficient storage and fast download have not been proposed until very recently [24, 18, 38, 20, 37]. We compare our new algorithm with these recent approaches in section 8, after the relevant concepts are introduced.

Polygon Mesh Coding A number of general purpose polygon mesh compression algorithms have been proposed in recent years. Deering [3] developed a mesh compression scheme for hardware acceleration. Taubin and Rossignac [30], Touma and Gotsman [31], Rossignac [21], Gumhold and Strasser [5], and others, introduced methods to encode the connectivity of triangle meshes with no loss of information. King et. al. [12] developed a method to compress quadrilateral meshes. Methods to encode the connectivity of polygon meshes were introduced by Isenburg and Snoeyink [6], Konrod and Gotsman [13], and Khodakovsky et.al. [10]. These algorithms focus on compressing the connectivity information very efficiently, and are all based on a traversal of the primal or dual graph of the mesh. Some of them compress connectivity of very regular meshes to a small fraction of a bit per vertex, and all to 2-4 bits per vertex in the worst case. When the geometry information (vertex coordinates, and optionally normals, colors, and texture coordinates) is also taken into account, the cost per vertex increases considerably. For example, adding only vertex coordinates quantized to 10 bits per vertex lifts the cost to typically 8-16 bits per vertex. In addition,

all of these approaches are incompatible with the out-of-core nature of isosurface extraction algorithms that visit the voxels in scan order.

Resampling and Subdivision Khodakovsky et.al. [11] follow a different approach to compress large connected and uniformly sampled meshes of low topological complexity, based on resampling, subdivision and wavelets. They obtain up to one order of magnitude better compression rates than with the connectivity preserving schemes, by approximating the mesh geometry with a subdivision mesh, and compressing this mesh instead. Wood et.al. [35] introduced a method based on surface wave propagation to extract isosurfaces from distance volumes that produces semi-regular multi-resolution meshes. These meshes can be compressed with Khodakovsky's wavelet-based scheme.

Compressed Isosurfaces Isosurface algorithms take as input very large volume data files, and produce polygon meshes with very large number of vertices and faces. For remote visualization, we can transmit either the volume data and run the isosurface algorithm in the client, or compute the isosurface in the server and transmit the resulting polygon mesh. In both cases the transmission time constitutes a major bottleneck because of the file sizes involved, even using general purpose mesh compression schemes in the second case. And this is true without even considering the computational resources of the client.

We follow a third approach based on an observation made by Saupe and Kuska [24]. The only information from the volume data the isosurface algorithm uses to construct the polygon mesh is: which grid edges cross the desired level set, and where these intersection points are located within the edges. As a result the isosurface algorithm can be decomposed into two processes: the *server* or *encoder* process, which scans the volume data, determines intersecting edges, and computes locations of intersection points; and the *client* or *decoder* process, which reconstructs the polygon mesh from the data transmitted by the server process. Our contribution is a very simple scheme to efficiently encode these data. In addition, we consider the tradeoffs associated with optionally computing normal vectors (used mainly for shading) in the server or the client.

2 ISOSURFACE ALGORITHMS

An isosurface algorithm constructs a polygon mesh approximation of a level set of a scalar function defined in a finite 3D volume. The function $f(p)$ is usually specified by its values $f_\alpha = f(p_\alpha)$ on a regular grid of three dimensional points

$$G = \{p_\alpha : \alpha = (\alpha_0, \alpha_1, \alpha_2) \in [\![n_0]\!] \times [\![n_1]\!] \times [\![n_2]\!]\} \ ,$$

where $[\![n_j]\!] = \{0, \ldots, n_j - 1\}$, and by a method to interpolate in between these values. The surface is usually represented as a polygon mesh, and is specified by its *isovalue* f_0. Furthermore, the interpolation scheme is assumed to be linear along the edges of the grid, so that the isosurface cuts each edge in no more than one point. If p_α and p_β are grid points connected by an edge, and $f_\alpha > f_0 > f_\beta$, the location of the point $p_{\alpha\beta}$ where the isosurface intersects the edge is

$$p_{\alpha\beta} = \frac{f_\alpha - f_0}{f_\alpha - f_\beta} \, p_\beta + \frac{f_\beta - f_0}{f_\beta - f_\alpha} \, p_\alpha \ . \tag{1}$$

Marching Cubes One of the most popular isosurface extraction algorithm is *Marching Cubes* [14]. In this algorithm the points defined by the intersection of the isosurface with the edges of the grid are the vertices of the polygon mesh. These vertices are connected forming polygon faces according to the following procedure. Each set of eight neighboring grid points define a small cube called a *cell*

$$C_\alpha = \{p_{\alpha+\beta} : \beta \in \{0,1\}^3\}.$$

Since the function value associated with each of the eight corners of a cell may be either above or below the isovalue (isovalues equal to grid function values are called singular and should be avoided), there are $2^8 = 256$ possible configurations. A polygonization of the vertices within each cell for each one of these configurations is stored in a static look-up table. When symmetries are taken into account, the size of the table can be reduced quite significantly.

Topological Inconsistencies Since some of the cases admit multiple polygonizations, care should be taken during the construction of the table to avoid the generation of polygon meshes with topological inconsistencies [32]. One such approach is based on estimating the value at the cell center using tri-linear interpolation within each cell. In the continuum, the surface defined by a level set of a smooth function without singularities is an orientable manifold without boundary which separates space into two disconnected sets, the *inside* where the function is negative, and the *outside* where the function is positive (or vice-versa). For most applications it is desirable that the isosurface algorithm generates a mesh with the same characteristics. This property is guaranteed when the mesh is the boundary of a regular solid (without self intersections).

Cuberille Kalvin [8] proposed one way to resolve this inconsistency problem by observing that the polygon mesh generated by Marching Cubes is the dual mesh of the quadrilateral mesh generated by the *Cuberille* algorithm [1]. Each vertex of the grid where the scalar function is specified (the primal grid) is the centroid of a *dual grid* cell, or *voxel*. Every edge of the primal grid intersects the common face of the two voxels corresponding to the ends of the edge. The mesh generated by the Cuberille algorithm is the regularized (converted to manifold) boundary surface of the solid defined by the set of voxels corresponding to grid vertices with scalar value above the isovalue. Without regularization, in general this mesh is highly singular (non-manifold). The conversion to manifold requires duplication of vertices and edges, so that in the resulting mesh every edge has exactly two incident faces. Which vertices to duplicate and how to connect the faces can be determined by virtually *shrinking* the solid, moving the faces in the direction of the inside. The multiplicity of each dual grid vertex in the regularized mesh only depends on the local connectivity of the eight incident voxels. Again, the regularization can be done by table look-up while the volume data is being scanned, with a table of size $2^8 = 256$.

Multiple Reconstruction Schemes What is important is to note that the Cuberille algorithm can construct the isosurface mesh from the same information as the Marching Cubes algorithm. The edge intersections in the primal mesh specify the location of the face centroids of the Cuberille mesh. The location of the cuberille vertices can then be computed by local averaging, or by using more accurate schemes [4, 27]. In addition, the client can apply a number of subsequent smoothing algorithms to improve the mesh appearance [26, 28].

The situation is similar for normals. If computed in the server as the gradient of the scalar function at the edge intersection points [36, 16, 19], and included in the compressed data, the Marching Cubes decoder will treat them as vertex normals, and the Cuberille decoder as face normals. If the normals are not included in the compressed data, then it is up to the client to decide how to estimate them from the vertex coordinates and the connectivity information.

```
IsosurfaceEncoder (F, f_0)
  for α_0 = 0,...,n_0−1
    for α_1 = 0,...,n_1−1
      for α_2 = 0,...,n_2−1
        # occupancy bit
        b_α = (f_α > f_0) ? 1 : 0
        encode (b_α)
        for  j = 0,1,2
          if  b_α ≠ b_{α−δ_j}
            # encode intersection point
            encode (p_{α,j})
            # encode normal (optional)
            encode (n_{α,j})
          end if
        end for
      end for
    end for
  end for
return
```

Figure 2: High level description of encoder algorithm.

The implication of these observations is that there is considerable freedom in the implementation of the decoder, making absolutely no changes to the encoder or the compressed bitstream. It is not even necessary for the decoder to produce a polygon mesh as output. For visualization purposes, and in particular if normals are included in the compressed data, a point-based approach [22] could be very effective.

3 ENCODER

The encoder algorithm scans the volume data, determines which grid edges intersect the isosurface, computes the location of the intersection points along the corresponding edges, optionally generates normal vectors for these points as scalar function gradient estimates, and entropy encodes all of this data after quantization. In this section we describe what data is encoded, and the order of the encoded data elements in the bitstream. Figure 2 is a high level pseudo-code description of the encoder algorithm. In the next section we describe the methods we use to entropy encode these data.

Occupancy Image Since whether an edge intersects the isosurface or not depends on the values of the scalar function at the edge ends, we encode the *occupancy image*. This is a 3D binary image defined by one bit per grid vertex

$$b_\alpha = \begin{cases} 1 & \text{if} \quad f_\alpha < f_0 \\ 0 & \text{otherwise} \end{cases}$$

specifying whether the scalar function attains a value above or below the isovalue on that vertex. We also define $b_\alpha = 0$ if $\alpha_j < 0$ or $\alpha_j \geq n_j$ for some $j \in \{0, 1, 2\}$. This ensures that the isosurface generated is closed (water-tight). We encode these bits in scan order. Since there are more edges than vertices, encoding one bit per edge would be wasteful, and may lead to inconsistencies.

Intersection Points The location of the intersection points and the optional normals are associated with the intersecting edges, which can be determined from the occupancy image. Except for boundary vertices, each grid vertex has six incident edges

$$\{p_\alpha, p_{\alpha+\beta}\} : \beta = \pm\, \delta_j \ ,$$

with $\delta_0 = (100)$, $\delta_1 = (010)$, and $\delta_2 = (001)$. To simplify the description, we add the missing edges as virtual edges to the boundary vertices, but we do not include them in the compressed data. Of these six incident edges, three connect the vertex with preceding vertices, and the other three with subsequent vertices in the scan order. Regarding the edges as oriented according to the vertex scan order, each edge has a beginning and end vertex. We order the edges, first by the scan order of the end vertex, and then by the direction of the displacement (0,1,2)

$$e_{\alpha,j} = \{p_\alpha, p_{\alpha-\delta_j}\} : j = 0, 1, 2 \ .$$

The position of the intersection point $p_{\alpha,j}$ along the edge $e_{\alpha,j}$ is determined by equation 1. This data has to be encoded in the compressed data only if the occupancy image has different values at the ends of the edge, i.e., if $b_{\alpha-\delta_j} \neq b_\alpha$. We specify the location of the intersection point along the edge with a number between zero and one $0 \leq t_{\alpha,j} \leq 1$ such that

$$p_{\alpha,j} = (1 - t_{\alpha,j})\ p_{\alpha-\delta_j} + t_{\alpha,j}\ p_\alpha \ .$$

Normal Vectors Since the gradient vector of a function is normal to its level sets, normals used for shading can optionally be computed during the volume traversal as finite difference approximations to the gradient vectors normalized to unit length [36].

Order of Transmission The encoder and decoder must have a hard-coded convention for the order of transmission of all these data. Since the occupancy image is encoded in scan order, after decoding each bit the decoder has all the information necessary to determine which of the three edges ending at the corresponding vertex are intersecting or not. We encode the optional data (positions and normals) corresponding to these edges in edge order, right after the occupancy bit corresponding to the end vertex. With this data organization, in case of loss of data due to interrupted transmission, a partial reconstruction using all the transmitted data can be obtained.

4 ENTROPY ENCODING

Entropy encoding is the problem of how to represent with a minimum number of bits a sequence of independent symbols $X = (x_1, \ldots, x_N)$ that belong to a finite *alphabet* $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ [23]. Symbols that appear more often in the sequence should be represented with fewer bits than those that appear more infrequently. The absolute lower bound for the total number of bits necessary to represent the sequence X with no loss of information is given by the *entropy* [23]. In practice the *arithmetic coder* [34] asymptotically achieves the entropy. Arithmetic coding is used as the basis of many image and data compression schemes and applications [15], very good public domain software implementations [33], and even hardware implementations [25] are available.

To deal with the lack of stationary distribution of symbols in the sequence, *adaptive* models are used. In arithmetic coding with an adaptive model the encoder updates the alphabet probabilities after encoding each symbol. Since encoder and decoder must use the same model to encode and decode each symbol, the model update procedure must be based on data previously encoded, and agreed upon information. Among these data are the initial probabilities, which may be hard-coded or included in the compressed data. A common practice is to start with uniform probabilities and keep track of the relative symbol frequencies as probability estimates.

For binary data, where the alphabet is composed of two symbols $\Sigma = \{0, 1\}$, keeping track of global symbol frequencies is usually not good enough as a model update procedure, and a *context-based* procedure is used. This is a state machine model with separate sets

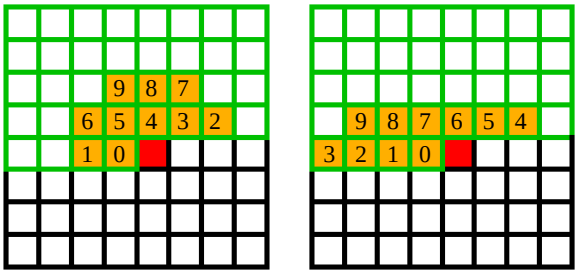


Figure 3: The three and two line templates used in JBIG to determine the arithmetic coding context. The red pixel is about to be encoded. The orange pixels define a 10 bit context. The pixels surrounded by green edges have already been encoded.

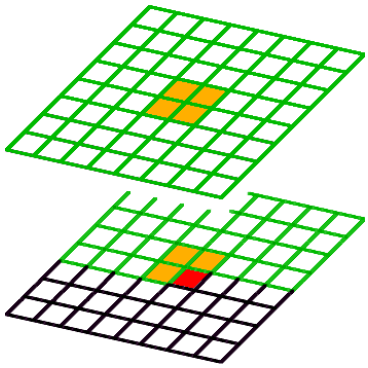


Figure 4: The seven bit template used to determine the context to encode each bit of the occupancy image. The red pixel is about to be encoded. The orange pixels define the 7 bit context. The pixels surrounded by green edges have already been encoded.

of probability estimates associated with each state or *context*. The update procedure determines the context from previously encoded data (such as values of previously encoded neighboring pixels in an image), and after the symbol is encoded with the probabilities associated with a context, the set of probabilities corresponding to that context is updated, but not the other. Context-based arithmetic coding is a very efficient adaptive compression scheme. It is used in the JBIG lossless image compression standard, and is the main reason for the high efficiency of our isosurface compression scheme.

JBIG JBIG is short for *Joint Bi-level Image experts Group*. This is both the name of a standards committee, and of a particular scheme for the lossless compression of binary images [7]. It can also be used for coding gray scale and color images with limited numbers of bits per pixel. JBIG is one of the best available schemes for lossless image compression.

The JBIG algorithm is based on *context based arithmetic coding*. For each pixel in an image a *context* is derived from a specific fixed pattern of surrounding pixels preceding the current pixel in the scan order. The standard defines several such neighborhoods. The 10 pixels included in two of these neighborhoods are illustrated in figure 3. These binary pixels values are used to construct a 10-bit context number, used to index into a list of context probability estimates.

Encoding the Occupancy Image We can look at the occupancy image as a stack of binary images, one for each value of $\alpha_0 = 0, \ldots, n_0 - 1$. One possibility is to use JBIG to encode these binary images independently of each other. The results are good, but the high correlation normally existing among spatially close voxels in neighboring layers is not taken into account. Instead, we use values from neighboring voxels not only in the current layer, but also in the previous layer, to build the context used to encode each voxel value. This simple idea allows us to increase the encoding efficiency quite significantly. In average we reduce the size of the compressed data by 50% compared to compressing the layers individually. There is a trade-off to be made in deciding how to build the context from neighboring voxels. Using more voxels increases the number of contexts, and so, the amount of memory needed to maintain the probability estimates, but can potentially lead to more efficient encoding. We have found that the simplest possible neighborhood performs very well. Of all the voxels that share a vertex, edge, or face with a given voxel, we use seven that precede it in the scan order to build a seven bit context c_α by concatenating the bits of these voxels in scan order. We use this context to encode the voxel bit b_α. The configuration is illustrated in figure 4. If we denote $c_{\alpha|j}$ the j-th bit of c_α from least to most significant, we have

$$\begin{cases} c_{\alpha|0} &= b_{\alpha-(001)} \\ c_{\alpha|1} &= b_{\alpha-(010)} \\ c_{\alpha|2} &= b_{\alpha-(011)} \\ c_{\alpha|3} &= b_{\alpha-(100)} \\ c_{\alpha|4} &= b_{\alpha-(101)} \\ c_{\alpha|5} &= b_{\alpha-(110)} \\ c_{\alpha|6} &= b_{\alpha-(111)} \end{cases}$$

Encoding the Intersection Points The second piece of information that needs to be encoded is the position of each intersection point $p_{\alpha,j}$, i.e., the number $0 \le t_{\alpha,j} \le 1$. This number is uniformly quantized to B bits, and the quantized value, which corresponds to an integer number between 0 and $2^B - 1$, is entropy encoded with no further loss. The decoder reconstructs the quantized value $\hat{t}_{\alpha,j}$ as the centroid of the segment defined by the corresponding integer. For example, if $B = 0$, which is sufficient in many cases, the reconstructed value is $\hat{t}_{\alpha,j} = 0.5$ independent of α and δ_j. In this case, no intersection point data is actually included in the compressed data, and all the geometry information is derived from the occupancy image. If $B = 1$, i.e., one bit is encoded per intersection point, there are two possible reconstructed values: 0.25 and 0.75. In general, if the B-bits integer to be encoded is h, the reconstructed value is

$$\hat{t}_{\alpha,j} = \frac{h + 0.5}{2^B} .$$

We look at each bitplane of these encoded integers as a new 3D binary image of the same dimensions as the occupancy image. By encoder-decoder convention we set the values corresponding to non-intersecting edges to zero, and we encode these $3B$ three-dimensional binary images (one each for $j = 0, 1, 2$ and each bitplane) with the same context-based method as we encode the occupancy image, except that we maintain separate context probabilities for each bitplane and each axis. The only difference is that none of the zero values agreed upon by the encoder and decoder are included in the bitstream. The encoder just skips them, and the decoder can set the corresponding bits to zero because it can determine which are these bits from the occupancy image bits. Both the encoder and the decoder have to reconstruct these bits, though, because they are needed to build the contexts used to encode and decode the corresponding bits.

Encoding Normal Vectors The last piece of information to be encoded is the normal vectors at the intersection points. These vectors were optionally computed during the volume traversal as finite

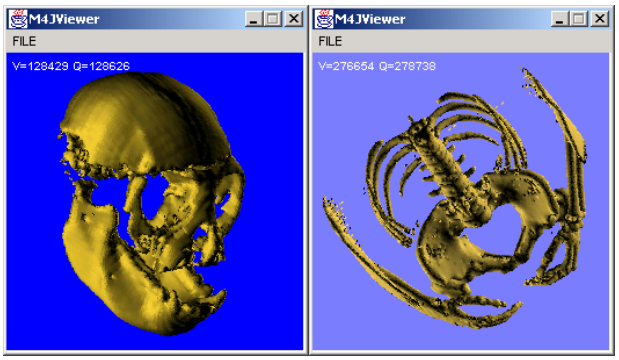


Figure 5: Decoder implemented in Java and integrated with a simple interactive viewer.

difference approximations to the gradient vectors, and normalized to unit length.

In the MPEG-4 standard [17] polygon mesh normals are quantized to $3 + 2S$ bits as follows. The first 3 bits determine the octant the normal vector belongs to. Each of these octants corresponds to a triangular face of a regular octahedron with vertices on the coordinate axes. Each of these triangular faces is recursively quadrisected (Loop subdivided) S times. The centroids of the 2^{2S} resulting triangles define vectors, which after normalization to unit length, define the quantized values. The angle distribution of quantized normals is not uniform here, but the lack of uniformity is not severe. On the positive side, the scheme does not require evaluation of transcendental functions. We quantize the normals with a variation of this method to $B = 2S$ bits. As in the case of intersection points, if $B = 0$ nothing is encoded, and the quantized normals are defined by the edges containing the intersection points. Otherwise, the first *two* bits determine the octant, and the remaining bits the level of triangle subdivision. We only need two bits to determine the octant because each normal cannot deviate by more than 90° from the quantized value corresponding to zero bits. Alternatively, the quantization scheme proposed by Deering [3] can be used.

5 DECODER

The decoder algorithm can be decomposed into two parts. The first part is a loop similar in structure to the encoder algorithm shown in figure 2. In this loop the occupancy image is decoded in scan order, and the optional values of intersection points and normals are reconstructed, if present in the compressed data. The second part of the algorithm, which is performed simultaneously within the loop, is the reconstruction of the data structure used for subsequent processing. This could be the polygon mesh produced by Marching Cubes, the quadrilateral mesh produced by Cuberille, or just a set of oriented points organized in a hierarchical data structure used by Qsplat [22]. As we mentioned before, we have considerable freedom in the implementation of this second part. Our current implementation is based on the Cuberille method.

IN-CORE vs. OUT-OF-CORE The main decision in the design of the decoder is determined by whether the reconstructed mesh can be kept in memory (*in-core*) or not (*out-of-core*). The highest compression rates are obtained with an in-core implementation, where intersection points and normal vectors are quantized to zero bits. In this case the encoder is simplified because the computation of intersection points and estimation of normal vectors is avoided. Instead, to improve the appearance of the reconstructed cuberille mesh, smoothing schemes are used to displace the vertices and normals from their reconstructed positions aligned with the grid [4, 26, 28].

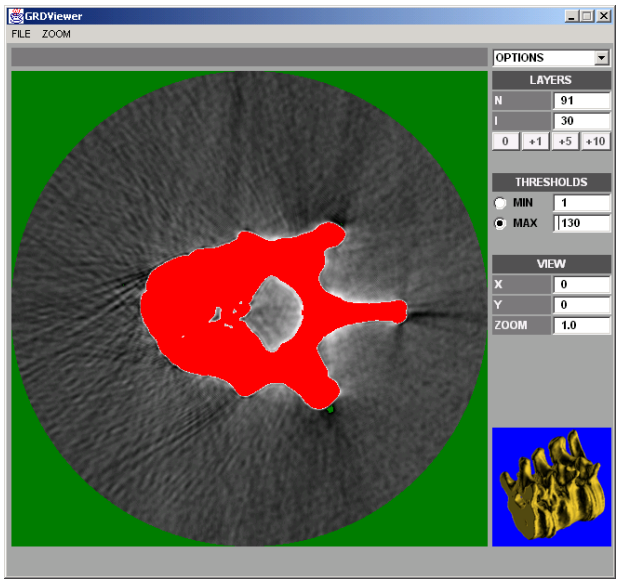


Figure 6: Interactive isosurface selection and volume data visualizer.

6 IMPLEMENTATION

We have implemented one encoder in C++, and two decoders; one in C++, and the other in Java. The C++ encoder and decoder are command line applications. For the arithmetic coder, we used the public domain software implementation by Wheeler [33], with minor modifications. The encoder reads a header file and a binary data file, and produces a compressed data file. The compressed data file includes information such as number of samples along each axis, as well as a linear transformation to scale, rotate, and translate the reconstructed mesh to user coordinates. This is important for example when processing medical data (CT and MRI) in which the sampling rate along one axis is different from the others. The C++ decoder reads the compressed data, constructs a Cuberille mesh in-core, optionally applies smoothing operators to vertex coordinates and normals, and saves the result as a VRML file. The Java decoder implements the same algorithms, but instead of saving the result as a file, it is integrated with a simple 3D rendering engine which allows the user to navigate around the reconstructed mesh and change the orientation of the light source. All the isosurfaces shown in the paper have been generated with this decoder. More advanced rendering techniques produce much better images. Figure 5 shows screen-dumps of the application. In addition, we implemented a simple Java user interface to visualize volume data, and to choose appropriate thresholds. Figure 6 shows a screen dump of this application, which also works as a front-end for the command line encoder.

In our current in-core decoder implementation, the encoding of intersection points is supported, but normal vectors are estimated by smoothing the normals to the faces of the cuberille. We look at these normals as a vector field defined on the dual graph of the Cuberille mesh. We iteratively transfer normals from the dual to the primal graph, and then back from the primal to the dual. We compute a primal vector field (defined on the graph of the Cuberille mesh) from the dual vector field as follows. For each vertex of the Cuberille mesh we average the values associated with incident

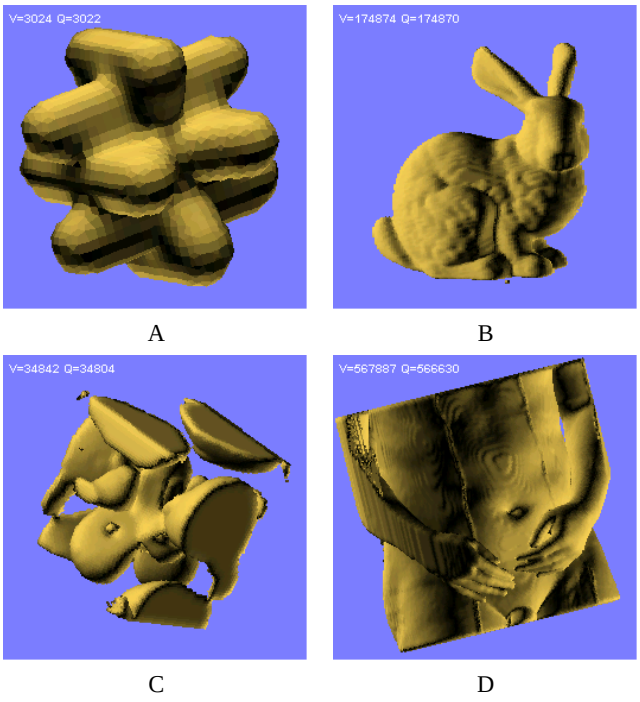


Figure 7: Some isosurface meshes used in our experiments, as rendered by our simple Java decoder. A: data set generated by evaluation of smooth analytic function on grid vertices. B: Stanford bunny CT data set. C: numerical simulation of electrostatic potential of iron molecule. D: section 6 of Visible man fresh CT data set.

grid size	B	bits	bits/face
$361\times128\times128$	0	116,377	0.5005
$361\times128\times128$	1	339,379	1.4595
$361\times128\times128$	2	541,244	2.3276
$361\times256\times256$	0	306,077	0.7582
$361\times256\times256$	1	707,895	1.7536
$361\times256\times256$	2	1,109,041	2.7473
$361\times512\times512$	0	763,598	0.7440
$361\times512\times512$	1	1,788,794	1.7428
$361\times512\times512$	2	2,813,713	2.7413

Figure 8: Compression results for the Stanford bunny CT scanned data set, isolevel 1500. The grid of size $361\times512\times512$is the original data set. The other two were generated by down-sampling within each layer.

grid size	B	bits	bits/face
$160\times128\times128$	0	123,142	0.6967
$160\times128\times128$	1	281,829	1.5944
$160\times128\times128$	2	439,966	2.4891
$160\times256\times256$	0	404,792	0.7144
$160\times256\times256$	1	908,257	1.6029
$160\times256\times256$	2	1,411,703	2.4914
$160\times512\times512$	0	1,201,172	0.6475
$160\times512\times512$	1	2,829,295	1.5251
$160\times512\times512$	2	4,452,208	2.3999

Figure 9: Compression results for the Visible Man fresh CT data set, section 6, isolevel 600. The grid of size $160\times512\times512$ is the original data set. The other two were generated by down-sampling within each layer. Number of faces are 1855144, 566630, and 176758, respectively.

faces, and then we normalize the value to unit length. This simple procedure produces satisfactory results. More complex procedures can be used as well [16, 28]. For medical applications, where guarantees of conservation of geometric or differential properties are often required, the smoothing process can be constrained so that each mesh vertex stays within a voxel center at the initial Cuberille vertex position. However, in our current implementation these constraints are not taken into account.

The reconstructed intersection points are the locations of the vertices of the Marching Cubes mesh (without triangulation of faces). Since the connectivity of the Cuberille mesh that our algorithm constructs is dual of Marching Cubes, we determine the location of its vertices as the centroids of the Marching Cubes faces. That is, in our implementation each Cuberille vertex position is computed as the average of the face centroids of incident faces. As a more complex alternative we can determine the location of the vertices by dual mesh resampling [27]. It is important to implement an algorithm that produces meshes without topological inconsistencies to prevent artifacts.

7 RESULTS

In this section we present some preliminary results. We have experimented extracting and compressing isosurfaces corresponding to different isolevels on the same volume data set, on different volume data sets, and on volume data sets generated by down-sampling. Figure 7 shows renderings of some Cuberille isosurface meshes produced by our Java decoder. The tables in figures 8 and 9 show numerical results for some of these data sets.

For the data sets we used in our experiments, a quantization level B of zero bits, coupled with smoothing of vertex positions and normals as a decoder postprocessing step, produced excellent results, with a typical bitrate of less than one bit per face. The bitrate increases somehow when the data is noisy, and the ratio of isosurface faces to voxels increases. Roughly speaking, if the intersection points are uniformly quantized to B bits, we observe compression bitrates of about $0.60-0.95$ bits per face for $B=0$, $1.20-1.80$ bits per face for $B=1$, and $2.10-2.90$ bits per face for $B=3$. An these results seem to be fairly independent of the grid dimensions. Remember that this is without including normals in the compressed data, which are computed by smoothing. Note that the best compression bitrates are obtained for $B=0$, which corresponds to the very efficient encoding of the occupancy image, and every increase of one bit in the quantization parameter B results in roughly one bit increase in the compression bitrate. In our experience, this high entropy in the quantized intersection points is due to the lack of predictors. For example, in [29] a smoothing operator is used as a predictor to efficiently encode vertex displacements from a coarse mesh to a refined mesh. We could do the same here to predict each bitplane of the quantized intersection points, and then encode the correction bits which hopefully will have lower entropy. We intend to explore this issue in the near future.

Figure 10 shows the same isosurface at different quantization levels, and the table in figure 11 shows the sizes and compression rates for these meshes. The last column of this table shows our distortion measurements. We compare the position of the vertices of a reference and a distorted mesh, both with the same connectivity. As the reference mesh we take what we would obtain without quantization. The distorted mesh corresponds to finite values of B. In both cases we apply the same smoothing step before making the measurements. As distortion measure we consider the ratio of average vertex displacement divided by average edge length in

$B = 0$ $B = 2$ $B = 4$

Figure 10: No apparent visual dependence on number of bits of quantization for intersection points. Data set CThead, volume size $113 \times 256 \times 256$, level 600. Rate-Distortion information is in the table shown in figure 11.

name	level	B	bits	bits/face	distortion
CThead	600	0	219,039	0.7437	0.2845
CThead	600	1	468,209	1.5897	0.1466
CThead	600	2	740,414	2.5139	0.0736
CThead	600	3	1,020,380	3.4645	0.0367
CThead	600	4	1,304,247	4.4283	0.0186
CThead	1160	0	252,527	0.8081	0.3295
CThead	1160	1	530,025	1.6961	0.1560
CThead	1160	2	830,961	2.6592	0.0769
CThead	1160	3	1,139,971	3.6480	0.0389
CThead	1160	4	1,451,176	4.6439	0.0194

Figure 11: Compression results as a function of quantization parameter B, corresponding to the volume data set *CThead* meshes isolevel 600 (CThead-600-0, CThead-600-2, and CThead-600-4 shown in figure 10), and isolevel 1160. The mesh of level 600 has 294524 faces, and the mesh of level 1160 has 312488 faces. The last two columns can be used to plot rate-distortion curves.

name	level	grid	bits	bits/face
CTbunny	1500	$361 \times 128 \times 128$	185,320	1.0598
CTbunny	1500	$361 \times 256 \times 256$	468,840	1.1615
CTbunny	1500	$361 \times 512 \times 512$	1,218,888	1.1876
CTman6	600	$160 \times 128 \times 128$	252,216	1.4269
CTman6	600	$160 \times 256 \times 256$	818,736	1.4450
CThead	600	$113 \times 256 \times 256$	423,952	1.4394
CThead	1160	$113 \times 256 \times 256$	517,720	1.6567

Figure 12: Connectivity-only compression results (no geometry) for the polygon meshes of figures 8, 9, and 11, with the polygon-mesh coder of Khodakovsky et.al. [10].

the reference mesh. This is not the most common way of measuring distortion. The Metro tool [2], which measures the Haussdorf distance between two meshes, has become a de-facto standard, but incompatible file formats and time constraints have prevented us from reporting results based on this tool. We plan to perform more exhaustive rate-distortion testing in the near future.

The polygon-mesh coder introduced by Khodakovsky et.al. [10] is considered as one of the best general polygon mesh connectivity coders. For comparison purposes, we show in figure 12 the results obtained compressing the connectivity information of the meshes of figures 8, 9, and 11 with this coder. Note that no geometry information is included in the bitrates because the current implementation does not compress the geometry data. With our in-core implementation we could not reconstruct the full resolution visible man isosurface due to lack of memory.

8 RELATED WORK

Despite their widespread use, only during the last year researchers started to address the problem of compressing isosurfaces.

Saupe and Kuska [24] presented an algorithm to compress isosurfaces closely related to ours. They extract and encode the occupancy image and intersection points. Normals are computed from the reconstructed Marching Cubes polygon mesh. The occupancy image is encoded with an octree-based scheme to deal more efficiently with large homogeneous regions of empty space. The intersection points are encoded with a multi-symbol context-based arithmetic coder. They compare the compression rates of their method with a number of existing schemes. They report results on isosurfaces corresponding to five different isovalues, extracted from a CT scan with a grid of size $250 \times 192 \times 168$, and used $B = 4$ to quantize intersection point to obtain a global vertex quantization of 12 bits per coordinate. They do this to be able to compare with irregular mesh compression schemes. They report compression rates between 11.56 and 11.77 bits per polygon. We do not have access to the same data, and in our experience $B = 0$ is almost always good enough for this kind of data, if a postprocessing smoothing step is added in the decoder. Look at the example shown in figure 10. But for $B = 4$ and similar medical data, such as figure1 B and C, we obtain compression rates of 4.42 and 4.64. This is about 2.5 times better. But again, with $B = 2$, $B = 1$, and sometimes even $B = 0$, plus smoothing of vertices and normals, the results are almost indistinguishable. They also report results corresponding to four isosurface extracted from smooth analytic functions evaluated on grid vertices. The grid size is $192 \times 192 \times 192$. The compression rates here range from 9.68 to 10.69 bits per polygon. Still more than twice what our simpler scheme produces.

Zhang et.al. [38] have a short discussion about isosurface compression with the larger context of massively parallel isosurface rendering. They propose a scheme similar in nature to ours, where the occupancy image and intersection points are encoded, but very superficial details are provided given the limited space. They propose to entropy encode the occupancy image using run-length encoding or arithmetic coding, but they do not seem to take advantage of correlation between consecutive layers. They report total byte size of compressed files for five different data sets, but it is difficult to compare due to lack of details about the meshe sizes.

Mroz and Hauser [18] encode the occupancy image using a more complex scheme based on chain coding, where the voxels that contain isosurface intersections are linked in long chains and represented as a sequence of symbols, each one specifying in which direction to go to visit the next cell. This is potentially more efficient, because the number of symbols to be encoded is proportional to the number of vertices or faces of the reconstructed mesh, as opposed to the number of voxels in our case. This representation also rules out an out-of core implementation, because typically chains traverse the volume in random fashion. On the positive side, this data can be rendered directly from the compressed data, i.e., decoded on-the-fly. This representation is ideal for a decoder based on oriented particles or volume rendering, in which case one additional normal per cell must be encoded in a separate channel. This method is also significantly less efficient than ours, even if normals are not included in the compressed data. They report typical rates between 2.0 and 2.5 bits per chained voxel. Since they do not include additional information to specify the location of the voxel more precisely, these results are $3 - 4$ time worse than our scheme without normals and $B = 0$ bits of quantization for intersection points.

Yang and Wu [37] describe a rather complex method to compress triangle meshes generated by the Marching Cubes algorithm. Each mesh vertex is represented by the index of the containing cube, the index of the supporting edge, and the position of the vertex along

the supporting edge (our $t_{\alpha,j}$). The decoder interconnects these vertices forming triangles using the occupancy image, as in the original Marching Cubes paper [14]. But the occupancy image is not encoded in the bitstream. Instead, it is reconstructed from the cube and edge indices in the encoding of mesh vertices by a complex procedure that in fact determines the connected components of the grid graph after removing the edges where mesh vertices are supported. Normal vectors are not compressed. Compression bitrates are several times worse than with our scheme. And it is not possible to do an out-of-core implementation.

9 CONCLUSIONS

In this paper we introduced a simple algorithm to compress isosurface data based on the overall structure of the JBIG binary image compression standard, and exploiting the correlation between consecutive layers of the volume scalar data to increase the compression rates quite significantly. The algorithm can be implemented in out-of-core or in-core fashion. The highest compression is achieved in the in-core version with smoothing of vertex positions and normal vectors. Despite its simplicity, this algorithm beats all the other methods proposed so far to deal with the same problem by a factor of at least $2-3$, and more typically 10, and existing general purpose mesh compression algorithms by higher factors.

As for future work, we envision several areas to improve compression efficiency. We plan to concentrate on reducing the compression bitrates of intersection points and normals by using better prediction schemes, and simultaneously compressing multiple isosurfaces corresponding to different isolevels on the same volume data. This can be used for example in medical data to simultaneously show different tissues rendered as semi-transparent surfaces. We believe that even further compression gains can be achieved when several levels are compressed jointly by exploiting the relations between the order of the isolevels.

Finally, the main limitation of our method is its computational complexity, because although the length of the compressed bitstream is proportional to the number of faces in the output mesh, the time complexity of the decoder algorithm is proportional to the number of voxels in the grid. To remove this obstacle we plan to investigate ways to combine the ideas presented in this paper with the alternative approaches described above based on hierarchical space partition data structures.

10 ACKNOWLEDGEMENTS

Thanks to Andrei Khodakovsky et.al. for providing an executable version of their polygon mesh connectivity coder [10] for comparison purposes. Thanks to Alan Kalvin for useful discussions about isosurface algorithms. Thanks to Alan Kalvin, Chris Morris, Stanford University, and UNC for providing access to volume data sets.

References

[1] L.S. Chen, G.T. Herman, R.A. Reynolds, and J.K. Udupa. Surface shading in the cuberille environment. *IEEE Computer Graphics and Applications*, 5(12):33–42, 1985.

[2] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, June 1998.

[3] M. Deering. Geometric compression. In *Siggraph'95 Conference Proceedings*, pages 13–20, August 1995.

[4] S. Gibson. Constrained elastic surface nets: generating smooth surfaces from binary segmented data. In *Medical Image Computation and Computer Assisted Interventions, Conference Proceedings*, pages 888–898, 1998.

[5] S. Gumhold and W. Strasser. Real time compression of triangle mesh connectivity. In *Siggraph'98 Conference Proceedings*, 1998.

[6] M. Isenburg and J. Snoeyink. Face fixer: Compressing polygon meshes with properties. In *Siggraph'2000 Conference Proceedings*, pages 263–270, July 2000.

[7] ITU-T T.82 Information technology - Coded representation of picture and audio information - Progressive bi-level image compression, March 93. http://www.itu.int.

[8] A.D. Kalvin. *Segmentation and Surface-Based Modeling of Objects in Three-Dimensional Biomedical Images*. PhD thesis, New York University, New York, March 1991.

[9] A.D. Kalvin. A survey of algorithms for constructing surfaces from 3d volume data. Technical Report RC 17600, IBM Research Division, January 1992.

[10] A. Khodakovsky, P. Alliez, M. Desbrun, and P. Schröder. Near-optimal connectivity encoding of 2-manifold polygon meshes. *Geometric Models*, 2002. Special Issue on Processing of Large Polygonal Meshes (to appear).

[11] A. Khodakovsky, P. Schröder, and W. Sweldens. Progressive geometry compression. In *Siggraph'2000 Conference Proceedings*, pages 271–278, July 2000.

[12] A. King, D. Szymczak and J. Rossignac. Connectivity compression for irregular quadrilateral meshes. Technical Report GIT-GVU-99-36, Georgia Tech GVU, 1999.

[13] B. Konrod and C. Gotsman. Efficient coding of non-triangular meshes. In *Proceedings of Pacific Graphics*, Hong-Kong, 2000.

[14] W.E. Lorensen and A.V. Cline. Marching cubes: a high resolution 3d surface construction algorithm. *ACM Computer Graphics (Siggraph Conference Proceedings)*, 21(4):163–196, 1987.

[15] K. M. Marks. A JBIG-ABIC compression engine for digital document processing. *IBM Journal of Research and Development*, 42(6), 1998.

[16] T. Möller, R. Machiraju, K. Müller, and R. Yagel. A comparison of normal estimation schemes. In *IEEE Visualization'97, Conference Proceedings*, pages 19–26, 1997.

[17] ISO/IEC 14496-1 Information technology - Coding of audio-visual objects, Part 2: Visual / PDAM1 (MPEG-4 v.2), mar 1999.

[18] L. Mroz and H. Hauser. Space-Efficient Boundary Representation of Volumetric Objects. In *Proceedings of the Joint Eurographics-IEEE TCVG Symposium on Visualization (VisSym01)*, Ascona, Switzerland, May 2001.

[19] L. Neumann, B. Csébfalvi, A. König, and E. Gröller. Gradient estimation in volume data using 4d linear regression. In *Eurographics 2000, Conference Proceedings*, pages 351–358, 2000.

[20] K. G. Nguyen and D. Saupe. Rapid high quality compression of volume data for visualization. In *Eurographics'2001, Conference Proceedings*, 2001.

[21] J. Rossignac. Edgebreaker: Connectivity compression for triangular meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, January-March 1999.

[22] S. Rusinkiewicz and M. Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Siggraph'2000, Conference Proceedings*, 2000.

[23] D. Salomon. *Data Compression – The Complete Reference*. Springer-Verlag, 1997. ISBN0-387-98280-9.

[24] D. Saupe and J.-P. Kuska. Compression of isosurfaces. In *Proceedings of IEEE Vision, Modelling and Visualization (VMV 2001)*, Stuttgart, Germany, November 2001.

[25] M. J. Slattery and J. L. Mitchell. The Qx-coder. *IBM Journal of Research and Development*, 42(6), 1998.

[26] G. Taubin. A signal processing approach to fair surface design. In *Siggraph'95 Conference Proceedings*, pages 351–358, August 1995.

[27] G. Taubin. Dual mesh resampling. In *Pacific Graphics 2001, Conference Proceedings*, Tokyo, Japan, October 2001.

[28] G. Taubin. Linear Anisotropic Mesh Filtering. Technical Report RC-22213, IBM Research, October 2001.

[29] G. Taubin, A. Guéziec, W. Horn, and F. Lazarus. Progressive forest split compression. In *Siggraph'98 Conference Proceedings*, pages 123–132, July 1998.

[30] G. Taubin and J. Rossignac. Geometry Compression through Topological Surgery. *ACM Transactions on Graphics*, 17(2):84–115, April 1998.

[31] C. Touma and C. Gotsman. Triangle mesh compression. In *Graphics Interface Conference Proceedings*, Vancouver, June 1998.

[32] A. Van Gelder and J. Wilhelms. Topological considerations in isosurface generation. *ACM Transactions on Graphics*, 13(4):337–375, 1994.

[33] F. Wheeler. Arithmetic coding package. http://www.cipr.rpi.edu/wheeler/ac, February 1996.

[34] I.H. Witten, R.M. Neal, and J.G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6), June 1987.

[35] Z.J. Wood, M. Desbrun, P. Schr"oder, and D. Breen. Semi-regular mesh extraction from volumes. In *IEEE Visualization 2000, Conference Proceedings*, pages 275–282, October 2000.

[36] R. Yagel, D. Cohen, and A. Kaufman. Normal estimation in 3d discrete space. *The Visual Computer*, pages 278–291, 1992.

[37] S.N. Yang and T.S. Wu. Compressing isosurfaces generated with marching cubes. *The Visual Computer*, 18(1):54–67, 2002.

[38] X. Zhang, C. Bajaj, and W. Blanke. Scalable Isosurface Visualization of Massive Datasets on COTS-Cluster. In *Proceedings of IEEE Symposium on Parallel Visualization and Graphics*, San Diego, CA, October 2001.