

# Fast Rendering of Large Encoded Isosurfaces from Uniform Grid Datasets

Angel del Río, Jan Fischer, Dirk Bartz, Wolfgang Straßer

WSI/GRIS - VCM. University of Tübingen  
Sand 14, 72076 Tübingen, Germany

Email: {*angel.delrio, fischer, bartz, strasser*}@gris.uni-tuebingen.de

## Abstract

Standard algorithms for the extraction of isosurfaces from volume data (e.g., the Marching Cubes algorithm) are notorious for producing a large amount of small triangles. Unfortunately, simplification is not a possible avenue in many application fields, thus a large number of triangles are transferred to the GPU. The resulting massive data load of transferring polygonal data from main memory down to the GPU is a major bottleneck in traditional polygonal isosurface rendering.

In this paper we propose a novel solution for rendering large isosurface models. Our approach is based on the combination of an encoding scheme for the isosurface and a hardware-accelerated decoding strategy that takes place entirely on the GPU. The encoding scheme takes into account the underlying regular grid structure inherent to this type of surfaces, hence it results in a more compact representation. By using this (lossless) compressed version of the polygonal model, we reduce both the communication between CPU and GPU and the storage requirements without any appreciable loss of quality. Furthermore, no further (slow) bus transfers are necessary, since decoding is performed on the GPU and the decoded geometry is written directly into on-board graphics memory. Therefore a significant speeding up of the overall rendering is achieved. An application example is presented in order to illustrate the benefits introduced by our approach.

## 1 Introduction

Isosurface extraction is a powerful tool for examining scalar fields within a volumetric dataset. In medical imaging, scanning devices, such as computerized tomography (CT) or magnetic resonance tomography (MRI) scanners, provide a measure-

ment of the patient's anatomy sampled on a semi-regular 3D grid. In this case, isosurfaces can be used to visualize different tissues and organs present in the dataset. In scientific visualization, isosurfaces are employed to gain better insight into simulation results. With the increase in resolution of medical scanners and the necessity of more accurate simulation results, the size of volume data expands rapidly. This also has a direct impact on the size and complexity of the isosurfaces necessary to visualize these data. Typical isosurface extraction algorithms, like Marching Cubes [10], produce a large amount of (small) triangles. As the size of the volume increases, so does the number of triangles forming the isosurface. This can make difficult or even impede a real-time visualization of the extracted isosurface. Mesh simplification techniques try to overcome these limitations by reducing the number of vertices (triangles) in the surface. Such approaches, though effective, introduce a loss in terms of quality which, depending on the application (e.g., medical imaging), might not be acceptable. Another possibility to facilitate the visualization of large isosurfaces is the application of compression algorithms in order to reduce their memory size. Depending on the chosen algorithm, whether it applies lossless or lossy compression, the quality of the isosurface can be altered or not. However, a common problem in both cases is the difficulty to efficiently render the compressed surface. Typically, the compressed isosurface is loaded from disk and decompressed in main memory before being sent to the graphics card for rendering. In this paper, we propose a novel approach where a lossless compression scheme is combined with a GPU-based decoder. By performing the decoding step on the graphics card, an encoded version of the isosurface can be copied to video memory, thus reducing the load on the bus between CPU and graphics system. The use of the newly available OpenGL frame

buffer objects, allows to save the result of the decoding process in graphics memory and then to directly access these data to render the isosurface without any read back to main memory, minimizing again CPU-GPU bus transfers.

The remainder of this paper is structured as follows: In the following section, we provide a brief overview of related work. In Section 3 we introduce the encoding scheme used to compress the isosurface. Following in Section 4, the GPU-based decoding process and the subsequent rendering strategy are described in detail. Results are presented in Section 5. Finally, we summarize the most relevant aspects of our work in Section 6.

## 2 Related Work

An extensive amount of work has been done in the field of volume visualization based on isosurfaces. With the *Marching Cubes* algorithm [10] as a starting point much research effort has focused on the isosurface extraction process. Extracting an isosurface from a volume dataset consists in two major steps: the identification of the *active cells* corresponding to a given isovalue and the polygonization of this surface. The latter is normally performed by identifying the respective case from a look-up table. The former is usually more computationally costly, and many optimizations have been suggested in order to accelerate the search process that locates those cells intersected by the isosurface. Livnat et al. [9] used a span-space representation to avoid exhaustive scanning of the volume. Cignoni et al. [3] based their approach in a search through an interval tree. Octrees can also be used to minimize the range of the search [15]. Additionally, a collection of seed cells can be identified to perform contour propagation from those seed cells [1, 14]. Recently, a GPU-based implementation of isosurface extraction using the marching tetrahedra algorithm [4], has been presented by Klein et al. [7]. This solution, even though efficient in terms of performance, relies on the utilization of ATI's SuperBuffers functionality, from the proposed `GLATI_super_buffers` OpenGL extension [11]. This cumbersome extension has not been officially accepted by the OpenGL Architectural Review Board (ARB), which makes its functionality difficult to handle and restricts its availability.

In this paper, however, we concentrate in op-

timizing the display of already generated isosurfaces. Standard algorithms for isosurface extraction, such as *Marching Cubes*, can produce an extensive amount of triangles when applied to large volume datasets. High resolution datasets usually generate large meshes with many small triangles. Even though surface simplification techniques [5, 2, 6] can reduce the complexity of these isosurfaces by removing and replacing vertices and triangles of the mesh, in some applications this alteration of the original data is not acceptable. This is the case, for instance, in medical applications, where the accuracy of the original data must be kept also in isosurface representations. An alternative approach to surface simplification is geometry compression. Given the copious amount of literature about compression techniques, here we restrict ourselves to those algorithms oriented to the compression of isosurfaces, which are of relevance for our work.

Yang and Wu [16] described a method to compress triangle meshes generated by the *Marching Cubes* algorithm. Based on the fact that in isosurfaces generated by the *Marching Cubes* algorithm all vertices are placed along one edge of an active cell, they represent each vertex by a cell index, the index of the supporting edge and its position along the supporting edge. The connectivity of the vertices is reconstructed by the decoder, in a rather complex process based on the 3D-chessboard structure first proposed by Cignoni et al. [3].

Saupe and Kuska [12] presented an algorithm to compress isosurfaces that is also based on a similar representation. In this case, the active cells set, often also called *occupancy image*, is encoded with an octree-based scheme in order to efficiently prune large homogeneous regions of empty space. Based on a similar isosurface representation, Taubin [13] developed a different approach that compresses the representation of the isosurface using a context based arithmetic coding scheme known as JBIG, typically dedicated to lossless compression of binary images.

We employ an isosurface encoded representation analogous to those described by Saupe and Kuska [12] and also utilized by Taubin [13]. However, even though these approaches achieve considerable reductions in the memory size of the isosurface, they do not address the problem of rendering these encoded versions of the extracted geometry.

Typically, the compressed isosurface must be decoded by the CPU and the set of polygons (generally triangles) are then stored in main memory before being transferred to the graphics pipeline in order to be rendered, hence making bus bandwidth a major bottleneck. In this paper, we propose a novel alternative for decoding and rendering an isosurface fully on the graphics card. Our method makes use of the newly available functionality on recent graphics cards (`GL_EXT_framebuffer_object`), which allows to *generate* geometry primitives with shader programs running on the GPU, without any read back to application memory. A somehow similar solution has been recently presented by Krüger et al. [8] for rendering of large point scans with point rendering techniques.

### 3 Encoding Scheme

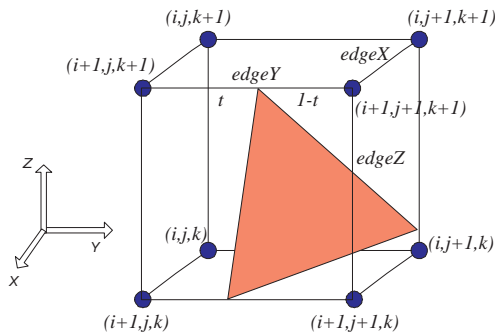


Figure 1: Volume cell representation. Each cell  $Ci, j, k$  contains eight voxels.

Most scanning devices, as well as many 3D simulation results produce volume datasets sampling a continuous magnitude at discrete points on a regular grid. Let  $\mathbf{r}_{i,j,k} \in \mathbb{R}^3$  with  $i = 0, \dots, i_{max}$ ,  $j = 0, \dots, j_{max}$  and  $k = 0, \dots, k_{max}$ , be the position of the grid points composing the volume (voxels). Since we work with scalar datasets, a scalar value  $I_{i,j,k} \in \mathbb{R}$  is associated to each grid position. In this representation, a cell  $Ci, j, k$  in the 3D regular grid has eight voxels at its corners forming a cube (see Figure 1). An isosurface corresponding to an isovalue  $c \in \mathbb{R}$ , can be defined as the solution of the equation

$$\phi(\mathbf{r}) - c = 0$$

where  $\phi(\mathbf{r})$  is a continuous interpolation function  $\phi : \mathbb{R}^3 \mapsto \mathbb{R}$ , such that  $\phi(\mathbf{r}_{i,j,k}) = I_{i,j,k}$  for all voxels  $\mathbf{r}_{i,j,k}$ .

During the isosurface extraction process, the first step consists in finding those cells  $Ci, j, k$ , whose voxel intensities  $I_{i,j,k}$  have values both above and below the sought after isovalue  $c$ . Such cells are usually denominated *active cells* or *intersecting cells* and are the only ones in the volume that contribute to the generation of an isosurface. Furthermore, in this scenario, all vertices of the generated mesh representing the isosurface are on edges of an active cell. In the standard case of Marching Cubes, vertex positions are linearly interpolated along one edge of an active cell according to the intensities of both voxels at the extremes of the edge, and the given isovalue. Taking this into account, we can build an alternative representation for isosurfaces, where instead of storing the 3D coordinates of each vertex (three floating point numbers, 32 bits each) and each normal vector (three floating point numbers, 32 bits each) of the mesh, indices identifying the vertex position within an active cell can be used. More specifically, a vertex position can be determined as

$$\mathbf{r} = \mathbf{r}_{i,j,k} + t_{\alpha} \mathbf{e}_{\alpha}$$

with  $\alpha = x, y, z$ , where  $\mathbf{e}_{\alpha}$  is a unit vector along one of the three main directions of the regular grid  $X, Y, Z$ , and  $t_{\alpha} \in [0, 1]$  acts as linear interpolation factor from a voxel  $\mathbf{r}_{i,j,k}$  to one of its first order neighbors along the edge selected by  $\mathbf{e}_{\alpha}$ . The value of  $t_{\alpha}$  can be then expressed as

$$t_{\alpha} = \frac{c - \phi(\mathbf{r}_{i,j,k})}{\phi(\mathbf{r}_{i,j,k} + \mathbf{e}_{\alpha}) - \phi(\mathbf{r}_{i,j,k})}$$

At this point it is important to note that with this representation, each voxel position can identify up to six vertices along one of its six incident edges. However, the half of these positions are redundant and can be assigned to one of its direct neighbors, by defining a scan direction. In this case, we associate to each voxel the three edges along the negative direction of the main axis,  $\{e_{-x}, e_{-y}, e_{-z}\}$ . This way, a vertex lying on one edge will be assigned to the *ceiling* voxel. This choice is arbitrary and for the sake of clarity, in the remainder of this paper we will refer to the directions and edges as  $X, Y, Z$  instead of  $-X, -Y, -Z$ .

By using this representation, an encoded version of the isosurface can be created. Each vertex is now

encoded as a 5-tuple of values  $(i, j, k, t_\alpha, e_\alpha)$ . Even though we have replaced the three coordinates of the vertex by five values, the memory requirements associated to the encoded version are considerably lower. The voxel indices  $(i, j, k)$  are integer values. In our implementation, these indices are stored using one byte per index, which allows to address volumes of size up to  $256 \times 256 \times 256$ . Larger volumes could be either split in blocks of this size, or encoded using 16 bits per index. The unit vector identifying the edge on which the vertex lies could be ideally encoded using only two bits, since only three values are possible. However, given that the encoded isosurface is to be transferred to the GPU as the content of a set of textures, the edge identifier must be stored using one byte, the smallest depth value of the supported texture format. Finally, the interpolation factor  $t_\alpha \in [0, 1]$  is a real number that can be quantized for encoding. In our system,  $t_\alpha$  is quantized with 8 bits and mapped to the range  $[0, 255]$ . This choice is sufficient for datasets acquired with a depth of 8 bits/voxel and does not introduce any extra uncertainty in the isosurface (see [12] for a complete discussion). Consequently, the position of each vertex is encoded using 5 bytes (40 bits) instead of the standard 12 bytes (96 bits), meaning that the encoded isosurface needs around 58% less memory than a standard OpenGL representation.

So far we have dealt with the encoding of the vertex positions. If necessary, it is possible to encode the normal vectors too. Since a normal vector is usually normalized to be a unit vector, its magnitude can be ignored, focussing only on encoding its orientation. This can be easily represented using the azimuth and zenith spherical coordinates  $(\theta, \phi)$ , with  $\theta \in [0, 2\pi)$ ,  $\phi \in [0, \pi]$ . Here again,  $(\theta, \phi)$  are real numbers that must be quantized in order to be efficiently transferred to the GPU in a texture. Different resolutions can be selected for the quantization of both components. In our implementation, we scale each value to be in the interval  $[0, 255]$  and use 8 bits to encode each  $\theta$  and  $\phi$  value. Even though this introduces a certain error in the normal orientation, our tests show that this is negligible and does not result in visible artifacts. With such encryption, a normal vector can be encoded using 2 bytes (16 bits) instead of the 12 bytes (96 bits) of a standard representation, a reduction of more than 83%.

As briefly mentioned before, we have selected this representation, not only due to the reduction of memory requirements associated to the isosurface, but specially because of being well suited to be directly uploaded to graphics memory as the content of a set of textures. Specifically, two textures are employed to transfer the geometry information (vertex positions) and one texture is utilized to upload the encoded normal vectors, if required. The first geometry texture is a `GL_RGBA8` texture (8 bits/component), where the RGB components contain the  $(i, j, k)$  voxel indices, respectively, and the alpha component comprises the quantized and scaled interpolation factor  $t_\alpha$ . The second geometry texture is even more simple and contains the edge identifier  $e_\alpha$  in a `GL_ALPHA8` (8 bits/component). Finally, the normal components, if needed, are saved as the content of a `GL_LUMINANCE8_ALPHA8` (8 bits/component) texture. This distribution of the data in three textures obeys not only to a logical separation of the encoded components, but also to the requirement that the chosen texture formats are directly supported by the graphics card, in order to avoid undesired transformations.

In order to obtain textures of the form  $2^m \times 2^n$ , with  $m, n \in \mathbb{Z}$ , the size of the textures is computed automatically depending on the number of vertices in the isosurface as

$$\begin{aligned} texture_{width} &= 2^{\lceil \log_2 \lceil \sqrt{n_{verts}} \rceil \rceil} \\ texture_{height} &= 2^{\lceil \log_2 \lceil \frac{n_{verts}}{texture_{width}} \rceil \rceil} \end{aligned}$$

This way, the textures containing the encoded isosurface can be generated and uploaded to graphics memory, from where the GPU-based decoder will access them to produce the vertices of the mesh for being rendered.

## 4 Decoding and Rendering

The fast development suffered by graphics cards in recent years, together with their increasing level of programmability has brought to the point where GPU-based computations can clearly outperform CPU-based ones in scenarios where the highly parallel structure of the former can be fully utilized. We make use of this to accelerate the rendering of the encoded version of an isosurface presented in the previous section. Our solution is based

on the newly available framebuffer objects extension (`GL_EXT_framebuffer_object`), that allows to render to an off-screen memory buffer in the graphics card and to reuse this memory buffer as the source of a vertex array (an alternative to implement the so called *render\_to\_vertex\_array* functionality). This way a costly read back from graphics memory to main memory can be avoided, thus speeding up the rendering process.

The functional pipeline of our decoding and rendering strategy are the following:

- Upload data (geometry + normals) as textures (see Section 3).
- *Render* geometry (and normals) texture/-s to off-screen buffer/-s.
- Decode vertex position and normal vector using a fragment program and write results to a framebuffer object.
- Bind content of framebuffer object as source of a vertex array.
- Render vertex array.

## Upload Data

The textures described in Section 3, are uploaded to graphics memory, using OpenGL functions (`glTexImage2D`). This way, the geometry and the normals can be efficiently transferred to the graphics card as a whole. Note that with the given encoding scheme, only the information corresponding to the vertices must be sent to the graphics pipeline, in contrast to GPU-accelerated isosurface extraction algorithms, where the whole volume must be copied.

## Render Textures to Off-Screen Buffers

Once the textures have already been defined and their content has been uploaded, we need to compute the appropriate texture coordinates to access the correct texels for each vertex. This can be easily achieved by rendering a multi-textured quad of the same size as the textures. If a 2D orthogonal projection is defined and the viewport is also set to the size of the textures, every texel corresponds to a pixel on the screen, or as in our case, to a fragment in the off-screen buffer. This is important because, this way, a fragment program can be utilized to decode both the vertex position and the normal vector in a single pass. In order to do this, we must create a framebuffer object with two different

draw buffers (`GL_COLOR_ATTACHMENT0_EXT`, `GL_COLOR_ATTACHMENT1_EXT`). These act as two independent rendering targets similar to a regular framebuffer, but without the precision restrictions associated to the latter. This is a crucial aspect, since floating point rendering targets can be defined, thus making it possible to *render* (write) the results of the decoding process directly to a floating point buffer.

## Decode Geometry

For decoding the isosurface on the GPU, a fragment program in the OpenGL Shading Language (GLSL) is used. Even though the rendering of the textures to off-screen buffers and decoding the geometry are two different logical steps, both are performed in one single pass. Therefore, the same shader program is run once to carry out both tasks. The shader program is formed by a vertex shader and a fragment shader. Given the adequate disposition of the projection matrix and the viewport, the vertex shader is extremely simple. It only must compute one texture coordinate (`glTexCoord[0] = glMultiTexCoord0`) and transform and orthogonally project the four corners of the multi-textured quad. The actual decoding process is executed by the fragment shader.

The fragment shader receives from the application the volume parameters (volume offset and voxel's spacings), as well as the identifiers of the textures involved (two for geometry, three if normals are encoded). At the same time, the texel coordinate corresponding to the current fragment—vertex—texel is passed automatically from the vertex shader. With this information, the decoding process consists in first identifying the position of the two voxels on both extremes of the respective edge, and then interpolate between both positions using the linear factor  $t_\alpha$ . The linear interpolation factor can be directly read from the texture since its value, as all other fixed point precision texture values, is automatically scaled to be within  $[0, 1]$ . The rest of values read from the textures must therefore be scaled back to their original values with a multiplication by 255. The position of the first voxel  $\mathbf{r}^1$  can be computed as

$$r_i^1 = o_i + s_i v_i$$

where  $i = x, y, z$ , with  $o_i$  being the  $i$ -th component of the volume offset,  $s_i$ , the  $i$ -th component of

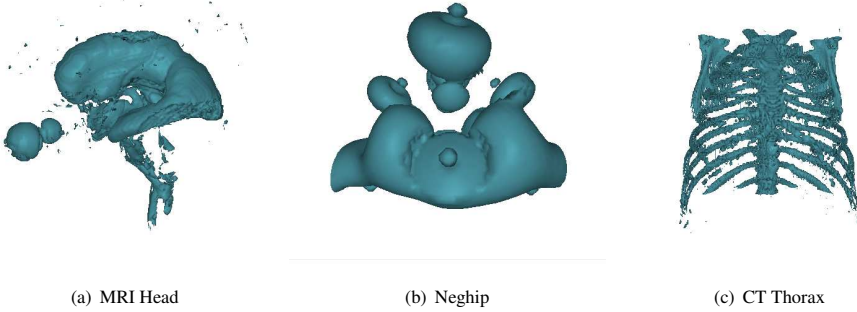


Figure 2: Isosurfaces utilized to evaluate the proposed method.

the voxel spacing and  $v_i$ , the voxel index in the  $i$ -th direction read from the texture. Once computed the position of the first voxel and known the corresponding edge  $e_\alpha$  by reading the respective texel value, the 3D coordinates of the second voxel  $\mathbf{r}^2$  can be easily obtained as

$$\mathbf{r}^2 = \mathbf{r}^1 + \mathbf{se}_\alpha$$

For the linear interpolation,

$$\mathbf{r} = t_\alpha \mathbf{r}^1 + (1 - t_\alpha) \mathbf{r}^2$$

the OpenGL Shading Language function `vec3 mix(vec3 x, vec3 y, float a)` is used. The obtained vertex position is written as an RGB value in the first draw buffer bound to the framebuffer object, where  $R = r_x, G = r_y, B = r_z$ .

On the other hand, the normal vector can also be decoded and the three cartesian components of the vector can be written to the second draw buffer bound to the framebuffer object. The coordinates of the normal vector are computed from the encoded direction vector  $(\theta, \phi)$  as

$$\begin{aligned} n_x &= -\cos(\theta)\sin(\phi) \\ n_y &= -\sin(\theta)\sin(\phi) \\ n_z &= \cos(\phi) \end{aligned}$$

Here again, these values are saved as the RGB content of a second draw buffer (off-screen buffer) with  $R = n_x, G = n_y, B = n_z$ .

This way, both the vertices of the mesh representing the isosurface and their respective normal vectors are written to graphics memory and ready to be used for rendering.

## Render Isosurface

In order to render the isosurface, we reuse the content of the two draw buffers where the geometry has been written to during the decoding process. This can be done making use of the `GL_EXT_pixel_buffer_object` OpenGL extension, together with the already mentioned `GL_EXT_framebuffer_object` extension. This allows us to bind each of the draw buffers of the framebuffer object as the source for two vertex attributes of a vertex array. The first vertex attribute (identified by *index 0*) is mandatory and represents the vertex position, while the second is optional and can be used to define the normal vector of the corresponding vertex. Once these vertex attributes are bound by the application, the vertex array can be rendered using a second shader program so that each normal vector can be assigned to the built-in attribute `gl_Normal` in a vertex shader.

## 5 Results

We have evaluated a prototypical implementation of the proposed method with a variety of datasets, some of which are depicted in Figure 2. Our test system is a PC with an Intel R XeonTM processor running at 2.66 GHz and a graphics card based on an NVidia R GeForceTMFX 6800 chipset. Our implementation is cross-platform, having been tested both in a Windows and in a Linux operating system. The timing data presented here correspond to the Linux tests. Due to space restrictions, we analyze here an illustrative example of the results obtained with our method. These results correspond to an isosurface extracted from an magnetic

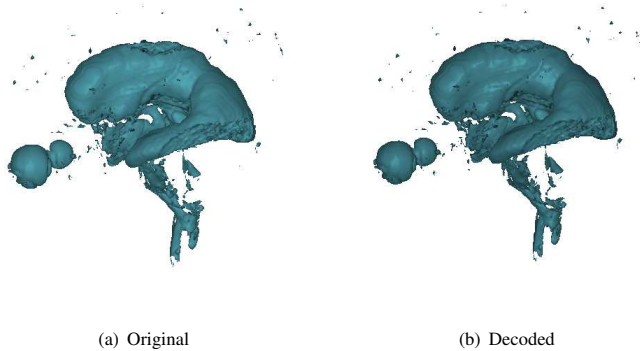


Figure 3: Image quality comparison.

resonance (MRI) scan of a human head (see Figure 2(a)), with a resolution of  $90 \times 90 \times 109$  mm and a size of  $256 \times 256 \times 114$  voxels. For this volume, we have extracted the isosurface corresponding to the isovalue 190 using Marching Cubes. The obtained isosurface has been reconverted into a sequence of triangle strips, in order to minimize the amount of connectivity information, resulting in a mesh of 69924 vertices forming 26401 triangle strips. Even though this mesh cannot be considered really *large*, its size is enough to obtain representative values. We compare the performance obtained with our method in contrast with an approach using the standard OpenGL pipeline for rendering the mesh. The average rendering time obtained when OpenGL is used to display the standard version of the isosurface was 42 ms ( $\sim 24$  FPS). With our strategy for decoding and rendering the isosurface on the GPU, we have measured an average rendering time of 22 ms ( $\sim 45$  FPS). From this time,  $\sim 3$  ms are employed to upload the textures to graphics memory and the rest for decoding and rendering the isosurface. This means an increase of around 100% in the rendering performance, and has been confirmed by our experiences with other datasets (see Table 1. This is due mostly to the fact that all the geometry is stored directly in graphics memory and the reduced transfer time necessary to pass this geometry as textures from system memory.

Figure 3 presents both rendering results corresponding to rendering the isosurface with standard OpenGL and to apply our decoding and rendering approach running fully on the GPU. As can be seen, both images are undistinguishable apart from slight

differences in the illumination conditions.

Table 1: Timing results obtained for different isosurfaces.  $T_A$ : rendering time using standard OpenGL.  $T_B$ : time for decoding and rendering using our method.  $T_C$ : extract from  $T_B$  employed to upload the encoded isosurface.

| <i>Dataset</i>   | #Strips<br>(#Verts.) | $T_A$ | $T_B$ | $T_C$ |
|------------------|----------------------|-------|-------|-------|
| <i>MRI Head</i>  | 26401 (69924)        | 42    | 22    | 3     |
| <i>Neghip</i>    | 6284 (17183)         | 11    | 5     | 1     |
| <i>CT Thorax</i> | 61566 (144933)       | 88    | 41    | 4     |

## 6 Conclusions and Future Work

In this paper we have presented a novel method for accelerating the rendering of large isosurfaces. Our strategy works on an encoded version of the isosurface that reduces its memory size, while other encoding and compression schemes are compatible with the method and might be incorporated in the future. The main contribution of our work focusses on decoding and rendering the isosurface. By making use of the programmability of recent graphics cards and the newly available framebuffer objects OpenGL extension, the whole decoding and rendering process can be performed directly on the GPU. Since no slow read back to main memory through the CPU is required, the rendering performance is increased. One major bottleneck for the rendering of large isosurfaces, the bandwidth of the bus

connecting CPU and GPU, is also alleviated due to the reduced size of the encoded isosurface. This can greatly benefit applications where the isosurface must be updated very often, like the visualization of time sequences of animated volumes. The obtained results are encouraging and illustrate the benefits of our method: reduced bandwidth requirements and faster rendering.

As future lines of work, we plan to investigate the feasibility of implementing a GPU-accelerated entropy decoder, which would be a perfect complement to the strategy proposed in this paper. This would help to further reduce the communication requirements associated to the upload of geometry data from system memory to graphics memory, hence providing a new rendering performance boost.

## References

- [1] Chandrajit L. Bajaj, Valerio Pascucci, and Daniel R. Schikore. Fast Isocontouring for Improved Interactivity. In *Proceedings of the Symposium on Volume Visualization*, pages 39–ff., Piscataway, NJ, USA, 1996. IEEE Press.
- [2] Fausto Bernardini, Holly Rushmeier, Ioana M. Martin, Joshua Mittleman, and Gabriel Taubin. Building a Digital Model of Michelangelo’s Florentine Pietà. *IEEE Computer Graphics and Applications*, 22(1):59–67, 2002.
- [3] Paolo Cignoni, Paola Marino, Claudio Montani, Enrico Puppo, and Roberto Scopigno. Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.
- [4] A. Doi and A. Koide. An Efficient Method of Triangulating Equi-valued Surfaces by Using Tetrahedral Cells. *IEICE Transactions on Communications, Electronics, Information and Systems*, E-74(1):214–224, 1991.
- [5] Hugues Hoppe. Smooth View-Dependent Level-Of-Detail Control and its Application to Terrain Rendering. In *Proceedings of IEEE Visualization*, pages 35–42, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.
- [6] Martin Isenburg, Peter Lindstrom, Stefan Gumhold, and Jack Snoeyink. Large Mesh Simplification using Processing Sequences. In *Proceedings of IEEE Visualization*, pages 465–472, 2003.
- [7] T. Klein, S. Stegmaier, and T. Ertl. Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids. In *Proceedings of Pacific Graphics ’04*, pages 186–195, 2004.
- [8] Jens Krüger, Jens Schneider, and Rüdiger Westermann. DuoDecim - A Structure for Point Scan Compression and Rendering. In *Proceedings of Eurographics Symposium on Point-Based Graphics*, 2005.
- [9] Yarden Livnat, Han-Wei Shen, and Christopher R. Johnson. A Near Optimal Isosurface Extraction Algorithm Using the Span Space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, 1996.
- [10] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proceedings of ACM SIGGRAPH*, pages 163–169, New York, NY, USA, 1987. ACM Press.
- [11] R. Mace. OpenGL ARB Superbuffers. Available on: <http://www.ati.com/developer/gdc/SuperBuffers.pdf>, 2004.
- [12] Dietmar Saupe and Jens-Peer Kuska. Compression of Isosurfaces for Structured Volumes. In *Proceedings of Vision, Modeling and Visualization (VMV)*, 2001.
- [13] Gabriel Taubin. BLIC: Bi-Level Isosurface Compression. In *Proceedings of IEEE Visualization*, 2002.
- [14] Marc van Kreveld, René van Oostrum, Chandrajit Bajaj, Valerio Pascucci, and Dan Schikore. Contour Trees and Small Seed Sets for Isosurface Traversal. In *Proceedings of ACM Symposium on Computational Geometry*, pages 212–220, New York, NY, USA, 1997. ACM Press.
- [15] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics (TOG)*, 11(3):201–227, 1992.
- [16] Shi-Nine Yang and Tian-Sheng Wu. Compressing Isosurfaces Generated with Marching Cubes. *The Visual Computer*, 18:54–67, 2002.