CEIG - Spanish Computer Graphics Conference (2017)
F. J. Melero and N. Pelechano (Editors)

Downsampling and Storage of Pre-Computed Gradients for Volume Rendering

J. Díaz-García[1] , P. Brunet[1] , I. Navazo[1] and P. Vázquez[1]

[1]Universitat Politècnica de Catalunya

Abstract

The way in which gradients are computed in volume datasets influences both the quality of the shading and the performance obtained in rendering algorithms. In particular, the visualization of coarse datasets in multi-resolution representations is affected when gradients are evaluated on-the-fly in the shader code by accessing neighbouring positions. This is not only a costly computation that compromises the performance of the visualization process, but also one that provides gradients of low quality that do not resemble the originals as much as desired because of the new topology of downsampled datasets. An obvious solution is to pre-compute the gradients and store them. Unfortunately, this originates two problems: First, the downsampling process, that is also prone to generate artifacts. Second, the limited bit size of storage itself causes the gradients to loss precision. In order to solve these issues, we propose a downsampling filter for pre-computed gradients that provides improved gradients that better match the originals such that the aforementioned artifacts disappear. Secondly, to address the storage problem, we present a method for the efficient storage of gradient directions that is able to minimize the minimum angle achieved among all representable vectors in a space of 3 bytes. We also provide several examples that show the advantages of the proposed approaches.

Categories and Subject Descriptors (according to ACM CCS): H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems—Artificial, augmented, and virtual realities I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing Algorithms I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

1. Introduction

The visualization of volume datasets by means of volume rendering algorithms is a demanding task that requires a high degree of computational power and resources. Nowadays it is common to manage medium to high resolution datasets with modern hardware. However, in some environments such as desktop PCs used by physicians in clinics (with the exception of radiologists), the equipment is much less powerful than needed to run diagnostic and interactive visualization tools. Furthermore, thanks to its ubiquity and its increasing capabilities, handheld devices such as mobile phones and tablets have become a targeted hardware platform for the deployment of these kinds of tools. Unfortunately, the horsepower of this hardware lies far behind the requisites of direct volume rendering of large datasets.

The interactive volume rendering of large volume datasets in commodity hardware is a challenging task. Typically, this problem is addressed by using multi-resolution techniques that consist in downsampling the original dataset to produce coarser representations of lesser resolution. The use of coarser datasets alleviates the pressure on the computational tasks, thus easing the task of interactive visualization. However, there is a tradeoff between interactivity and visualization quality, as there is an important loss of information during the downsampling process of the scalar field that directly affects the quality of the final images.

In this paper, we address the problem of quality loss in the visualization of coarse levels in multi-resolution datasets. One of the main aspects that directly affects the quality of volume visualization is shading. In multi-resolution visualizations, shading coarse levels using gradients that have been directly computed from the downsampled scalar field yields undesired results, as the topology of coarser representations do not reflect the original dataset anymore.

As a solution for that, in order to improve the visualization of coarse datasets, this paper presents a downsampling filter for pre-computed gradients that better preserves the direction of gradients as computed from the original dataset. The idea is pre-computing a volume of gradients G_0 from the original dataset V_0, and then iteratively downsampling the volume of pre-computed gradients, starting from G_0, until completing the generation of the multi-resolution pyramid.

The second problem we are addressing is the loss of fidelity of pre-computed gradients due to the precision of the storage format. We need to store the downsampled pre-computed gradients in a

DOI: 10.2312/ceig.20171208

3D array that will ultimately be passed into the GPUs using a 3D texture. Although floating point values are a wiser, more precise choice to store gradients preserving their directions as much as possible, for space and performance efficiency purposes it is interesting to limit the number of bits used for the storage to that end. However, imposing that restriction can also limit the fidelity of the representable gradients.

To alleviate this limitation, we propose an encoding and decoding scheme for pre-computed gradients that is able to maximize the representable space of gradient directions by performing a transformation on the points of the discrete 3D space resulting from all possible combinations of three coordinates of limited precision.

Summarizing, our contributions are:

- A downsampling filter specific for pre-computed gradient datasets that preserves surface orientations from the original scalar field and avoids staircase looking artifacts that come from the downsampling process.
- An encoding/decoding scheme for gradients that maximizes the representable space of gradient directions using an storage of 3 bytes per gradient.

The rest of the paper is organized as follows: Section 2 gives an overview of the most important related work. Section 3 presents a filter for the effective storage of gradient data, and Section 4 explains a proposal for the encoding and decoding method for the efficient storage of gradients that maximizes the fidelity of the stored gradients. In Section 5, the results of the solutions we present are shown and discussed, and the final Section 6 draws some conclusions and points out a few lines of future work.

2. Previous Work

Volume rendering is a costly process that is not always possible to perform interactively. More precisely, visualizing datasets of high resolutions requires approaching this problem by means of multi-resolution and compression techniques [BRGIG*14, BHP14] to maintain interactive frame rates at the expense of image quality.

Many papers speak about multi-resolution structures of scalar field representations [BNS01, CNLE09]. Some of them combine data structures with compression methods to optimize the bandwidth usage [GS04, GIM12]. A few publications concentrate on the preservation of quality in coarse representations by means of storing extra local information at voxels [YMC06, SHKM14] or with a more global approach based on transfer function modifications [DGBN*16]. Regarding the generation of coarse datasets in multi-resolution structures, some authors propose methods that take into account the transfer function to perform importance sampling during downsampling [WWLM11], directly deal with pre-classified color data [KB08], and perform topology preserving downsampling [KE01].

An important aspect in volume rendering is the computation of gradients. In [BLM96], the authors present an analysis of the ideal gradient estimator. There are several methods to evaluate gradients from a scalar field. One of the most used reconstruction filters designed for that purpose is the central differences approach [HKRs*06], which requires 6 extra texture lookups to perform the difference in the scalar field along each direction in the *XYZ* space. There is an even faster version of this filter, at the expense of introducing a small spatial bias, that only uses 3 extra texture lookups by calculating the difference with the central density. There are also methods that achieve gradients of better quality such as the Sobel's operator [DH73] by sacrificing the performance. The classical Sobel's operator, for instance, requires 26 extra texture lookups due to its $3 \times 3 \times 3$ kernel. To alleviate this performance penalty, Sigg and Hadwiger [PR05], use a more efficient version of the Sobel's operator that only needs 8 extra texture lookups at the corners of the voxel containing the sample to shade, and still obtain similar quality results. Furthermore, a few pre-filtering reconstruction schemes to increase the accuracy of the estimated gradients are presented in [CD09].

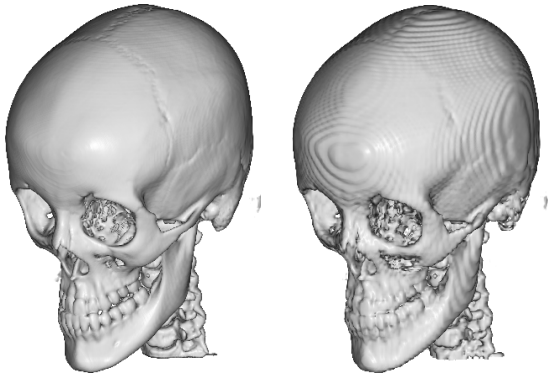
Working with pre-computed gradients [HKRs*06] allows to use slower but precise computations for estimating gradients before rendering (as during pre-process speed is usually not crucial) and speeds up the visualization algorithms taking place in the GPU by moving this rather expensive computation to previous stages. Most existing algorithms for encoding normal vectors and/or gradients cannot be used in the context of volume rendering if gradients are stored in a 3D texture. Trilinear interpolations performed to query gradients from the 3D texture are always convex when gradients are simply encoded by quantizing their cartesian components (G_x, G_y, G_z), but this desirable property is not fulfilled by many other well-known proposals like [DDSD03] (spherical coordinates), [OB06] (recursive subdivision of a Platonic solid), [Dee95] (indexing spherical triangles) or [CNC13] (encoding a point in the surface of a cube).

In this paper we propose a solution based on pre-computed gradients. Our approach consists of a downsampling filter to generate multi-resolution representations of gradient data, and an encoding scheme for gradient directions based on a monotonic transformation that guarantees the above-mentioned property of convex interpolation, thus ensuring that the final algorithm is GPU-friendly.

3. Downsampling of Gradient Data

In modern volume rendering algorithms such as ray casting, shading is performed after estimating the surface orientation at each sample position by means of the scalar field gradient, which is typically computed on-the-fly in shader code by evaluating the surrounding densities. In multi-resolution visualizations, the shading of coarse representations by means of this technique, implies computing gradients from a scalar field that differs from the original dataset. This fact that can lead to an inconsistent shading among different levels of resolution. Figure 1 shows the difference between shading a dataset visualized at its original resolution with respect to a low resolution representation, using gradients computed on-the-fly in both cases.

The quality of the shading in multi-resolution datasets is greatly affected by the way in which gradients are computed in coarse levels. The downsampling process drastically reduces the resolution of the datasets, thus provoking an inevitable information loss and a modification of the topology of the original scalar field. In Figure 2, the effect of downsampling is depicted in a 2D space. It is easily noticeable how the topology of the represented surface gets



(a) (b)

Figure 1: *Ray casting images of the Head model visualized with a transfer function designed to show bone surfaces. The left image (a) corresponds to the original dataset (512^3) whereas the image at the right (b) corresponds to a coarser dataset (128^3). In (b), staircase artifacts are visible due to the shading performed using gradients computed on-the-fly from the downsampled scalar field.*

drastically changed as the resolution decreases. The staircase shape exhibited by the surface in Figure 2-b also affects the direction of the computed gradients, not matching the gradients computed from the original dataset anymore. In Figure 1-b, the staircase artifact is mainly visible because the shading is using inaccurate gradients computed on-the-fly from the coarse dataset.

In order to solve this problem, the solution we propose in this paper is using gradients pre-computed from the original scalar field. In order to keep gradient directions consistent, we pre-compute a dataset of gradients G_0 from the original scalar field V_0, and we iteratively downsample G_0 to generate coarser representations G_k of the gradients vector field that match the resolution of the coarse models V_k. Thus, the visualization pipeline for multi-resolution datasets used in this paper uses an *RGB* 3D texture of pre-computed gradients for the visualization of each coarse dataset.

However, a naive downsampling of pre-computed gradients without having previously applied an appropriate filter achieves unexpected results (see Figure 3). This is due to the fact that the topology of the downsampled dataset has been modified with respect to the original. For that reason, in some cases, regions containing boundaries between materials in the coarse resolution dataset could correspond to regions of an homogeneous material in the high resolution one. Whenever that happens, the locations of the sampled gradients in the low resolution dataset correspond to gradients that are not properly defined in the high resolution dataset, and thus, using gradients that have been downsampled without any further consideration would lead to erroneous visualizations such as in Figure 3.

As a solution for this issue, our proposal consists in a downsampling filter that takes into account the magnitude of the gradients within the support of the filter kernel. The proposed filter performs a convolution over a certain dataset of pre-computed gradients G_k with the following kernel:

$$G_k^f(x) = \frac{1}{\beta} \sum_{i \in B_r} G_k(x+i) \cdot m(x+i) \cdot g(i) \qquad (1)$$

where $m(x+i)$ is the magnitude of the gradient at the neighboring position $x+i$, $g(i)$ is a Gaussian function that gives more importance to those samples nearer to center of the kernel support B_r of radius r, and $\beta = \sum_{i \in B_r} m(x+i) \cdot g(i)$ is the sum of all weights to ensure a normalized contribution of the gradients. For the sake of clarity we have expressed the equation in 1D, although the same definition applies for the 3D case.

Notice the similarity of Equation 1 with a bilateral filter [TM98]. Bilateral filters act essentially as a standard domain filter, averaging values that are similar to the value at the kernel center. The main difference between the bilateral filter and ours is that we are not giving importance to the value at the kernel center, assuming that the gradient at that point might be poorly defined, but giving importance to gradients in the kernel support that inform about a well defined material boundary. After our filter is applied, the filtered output G_k^f can be safely subsampled to obtain the next coarser dataset G_{k+1} in the multi-resolution hierarchy.

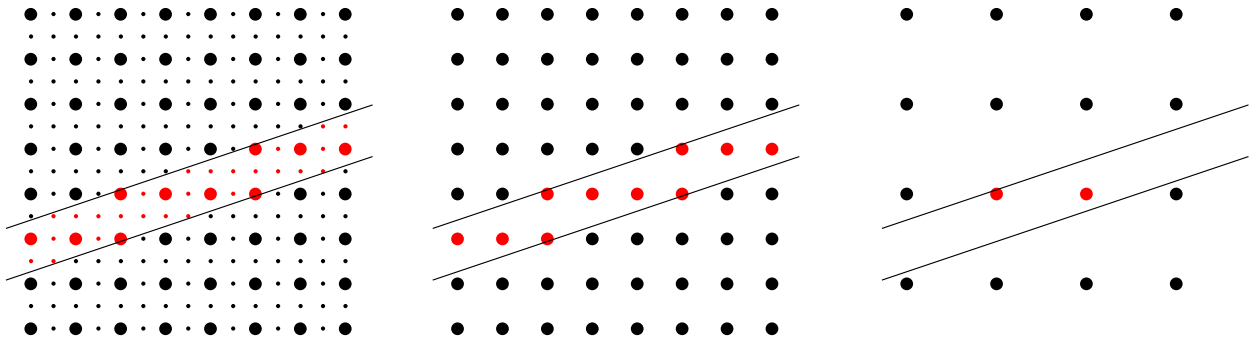
In order to pre-compute the whole multi-resolution pyramid of gradient datasets, the first step consists in pre-computing the gradients from the original dataset, thus obtaining a new volume dataset of gradient data G_0 with the resolution of the original scalar dataset V_0. By applying the proposed filter and then subsampling, the next level of the multi-resolution hierarchy G_1 is obtained from G_0. Following the same procedure, G_2 is obtained from G_1 and so on until the whole pyramid is generated.

Section 5 shows some examples of volume images generated using pre-computed gradients that have been extracted from the original dataset and downsampled with the proposed filter. The presented images demonstrate how using quality gradients obtains better results and makes the aforementioned staircase artifacts disappear.

4. Storage of Gradient Data

Before being used in the GPU by the visualization algorithm, gradients are pre-computed and downsampled in the CPU, stored in a floating point representation to conserve as much precision as possible. However, in order to use these gradients from shaders in the GPU, we make use of an 8 bit component *RGB* 3D texture (for the *XYZ* components of vectors). Although using this byte precision 3D texture is an standard practice, this is an important step where the precision of the stored gradients is decreased. We don't make use of floating point textures in the GPU because of its demanding storage space and its performance penalty to perform texture lookups. Quantizing gradients directly into 3D textures is not optimal, because the reduced bit size of the 3D texture limits the space of gradient directions that a voxel can represent (see Figure 4).

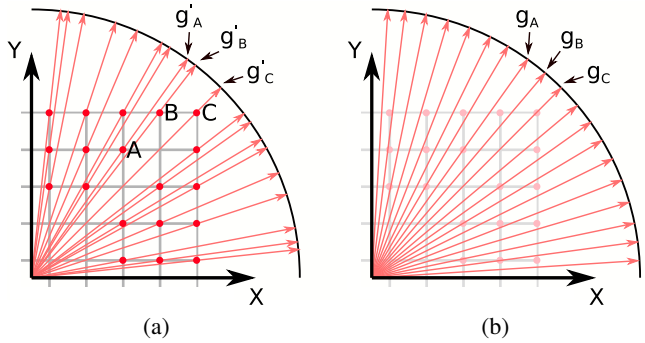
Observe that gradient directions are important (e.g. to perform shading operations), but gradient magnitudes are not usually considered in many rendering algorithms. In what follows, therefore, we will be only interested in encoding and retrieving gradient directions in the Gauss sphere.



(a) (b) (c)

Figure 2: *Images (a), (b) and (c) depict the effect of iteratively downsampling data (images are shown in 2D for simplicity, although the same concept applies for the 3D case). In (a), a feature/surface is well represented by the boundary between different scalar values. After an iteration of downsampling, image (b) shows how the same feature is not so suitably represented by the current scalar field anymore, which actually yields a more staircase-looking topology. In the last image (c), after another downsampling step, the feature even starts disappearing.*

Figure 3: *This ray casting image is shaded using downsampled pre-computed gradients. The topology of the downsampled scalar field V_k has changed with respect to the original dataset V_0. Therefore, using a naively filtered downsampled dataset G_k of the original pre-computed gradients G_0 that does not take into account any changes in the topology produces these annoying hole-like artifacts.*



(a) (b)

Figure 4: *Gradients stored into a given texture (drawings are in 2D for clarity) are quantized to fit the bit size of the components. In the case of using 8 bit components, this quantization (a) limits the representable space of gradient directions. Furthermore, the distribution of gradient directions does not fill the representable space uniformly. The method described here applies a transformation on the pre-computed gradients recovered from the texture so that the final distribution of directions becomes more uniformly distributed (b). The result is that quantized values A, B and C encode the uniform directions g_A, g_B i g_C instead of the uneven directions g'_A, g'_B i g'_C that directly correspond to A, B, and C.*

In this section, we propose an encoding scheme that is able to maximize the representable space of gradient directions when storing them into an *RGB* 3D texture of byte precision components. For that purpose, as a pre-process, pre-computed gradients are encoded with a transformation T and quantized before being stored into the GPU *RGB* 3D texture. Then, the visualization algorithm is able to perform texture lookups to recover the encoded gradients, and perform a fast decoding transformation T^{-1} in the shader code to obtain the final gradients that will be used for shading, which better match the original ones. Figure 5 shows a graphical overview of the proposed encoding/decoding approach.

4.1. Monotonic Gradient Encoding

In order to represent a vector in 3D space, the most common approach is to use 3 values to specify its X, Y and Z coordinates. The three-dimensional vector space described by these values has the shape of a cube if represented as a point cloud, as seen in Figure 6-a. Once these vectors are given a common origin and nor-

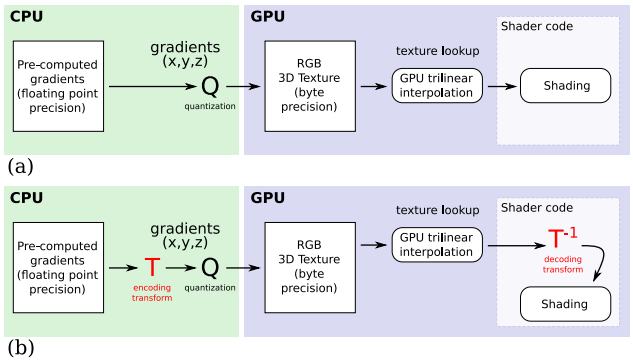
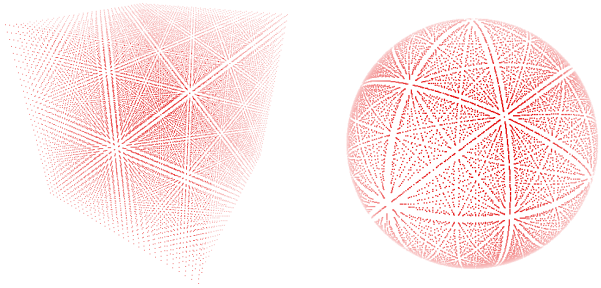


Figure 5: *Typically, visualization methods that use pre-computed gradients apply a direct quantization of the normalized floating point vectors (a) in order to store them into a byte-precision RGB 3D texture. This is an important step where gradients lose precision. We propose a transformation T that is able to maximize the representable space of gradients obtained from a byte-precision RGB 3D texture. We encode the original gradients and quantize them in a smart way to optimize the usage of the encoded space. Decoding is performed in the GPU using the inverse transformation T^{-1}.*



(a) (b)

Figure 6: *Point cloud represented by three values XYZ. (5 bits per value are used in this case to avoid cluttering and see the empty patterns on the surface). In (a) the points are evenly distributed in the 3D space. On the other hand, (b) shows the corresponding* sphere-dots *after projecting the point cloud onto the surface of a unit sphere.*

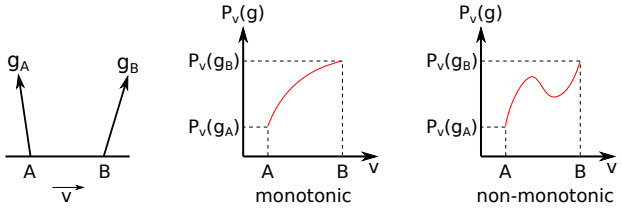


Figure 7: *Condition of monotonicity. $P_v(g)$ is the projection of g in a given direction v ($P_v(g) = g \cdot v$). To ensure that the transformation of a dot (Figure 6) is monotonic, this assertion must be true for all possible tangent directions on the surface of the Gauss sphere.*

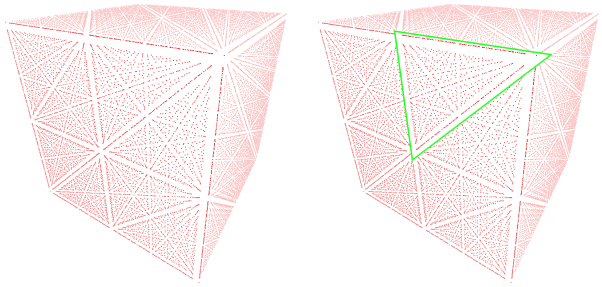


Figure 8: *In (a), the point cloud represented by three values XYZ projected onto* cube-dots *on the surface of a unit cube (5 bits per value are used in this case to avoid cluttering and see the empty patterns on the surface). In (b) one of these 48 triangular regions that we have identified on the surface of the cube is highlighted.*

malized, the set of points that represent gradient directions can be shown in the Gauss sphere, Figure 6-b. In what follows, gradient points projected onto the Gauss sphere will be named *sphere-dots*. We can notice that there are several patterns of empty regions (that is, regions without *sphere-dots*) onto the surface of the sphere (see Figure 6-b). Those empty regions correspond to gradient directions that cannot be directly represented with the tree coordinates X, Y and Z due to their bit size precision.

To alleviate this issue and to optimize the representable space of gradient directions, we propose to perform a transformation of the *sphere-dots* to achieve a more uniform distribution over the surface of the Gauss sphere. The solution we propose reduces the *biggest hole* on the surface of the sphere (that is, the region with the biggest separation among dots), minimizing the maximum angle achieved between two neighbouring gradient directions.

As the pre-computed datasets of gradients are stored into a 3D texture in the GPU, and the shader code of the ray casting algorithm queries this texture to evaluate gradients at any position using hardware-enabled tri-linear interpolation, it is important that the transformation applied to each dot is *monotonic*, so that dots do not get mixed on the surface of the unit sphere (Figure 6-b). In other words, the relative position among dots on the surface of the sphere before applying the transformation should not change after applying the transformation. If this condition were not satisfied during this operation, interpolated vectors could be wrong at the moment the transformation takes place in the GPU. The condition of monotonicity (see Figure 7) ensures that if we sample the texture using tri-linear interpolation, the decoded results will provide gradient directions that remain within the decoded directions of the surrounding voxel centers.

If we project the point cloud in Figure 6-a onto the surface of a cube (instead of a sphere), we obtain the distribution shown in Figure 8. The points projected on the surface of the unit cube will be named *cube-dots* from now on to distinguish them from the points on the sphere. With *cube-dots*, the patterns of empty regions on the Gauss sphere are noticeable easily. With this projection, we can observe that the overall *cube-dots* follow a pattern of a triangular region that repeats itself 8 times on each face of the cube. That adds up to a total number of 48 triangular regions on the entire

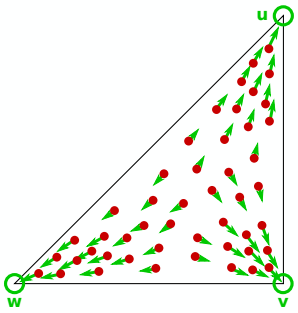


Figure 9: *The effect of the decoding transformation T^{-1} shown in Equation 2 can be seen here. The transformation is applied on the* cube-dots *once they are expressed in barycentric coordinates, and it generates the movement of these dots towards the directions shown by the green arrows, filling the empty areas near the boundaries of the triangle.*

cube. Figure 8-b shows one of the 48 regions. The outer part of each triangle has an empty margin, not filled with *cube-dots*. This means that these gradient directions cannot be properly encoded. In order to improve the distribution of normals, we need a set of dots that does not exhibit those holes and whose distribution becomes more uniform.

Thus, our purpose is to treat each of those triangular regions independently, performing a monotonic transformation on each *cube-dot* within the space of its triangle, thus making the empty bands near the edges shrink.

For the sake of clarity, let us start by discussing the decoding of gradient values obtained from the 3D texture of encoded gradients. To convert the retrieved, uneven gradient directions g'_A onto the corresponding uniform directions g_A (Figure 4), we must perform the decoding (inverse) transformation T^{-1} of cube-dots in the space of a triangle. This is a simple and GPU-friendly operation. For that reason, each point in the cloud belonging to a certain triangle (one of the 48 on the cube), has its Euclidean coordinates (XYZ) converted into Barycentric coordinates (UVW). We propose the following equation system using barycentric coordinates to transform the dots:

$$
\begin{aligned}
&\textbf{\textit{// Attract to vertices}}\\
&u1 = \lambda u^2 + (1-\lambda)u\\
&v1 = \lambda v^2 + (1-\lambda)v\\
&w1 = \lambda w^2 + (1-\lambda)w\\
&\\
&\textbf{\textit{// Normalization}}\\
&sum = u1 + v1 + w1\\
&u1 = u1/sum\\
&v1 = v1/sum\\
&w1 = w1/sum
\end{aligned}
\tag{2}
$$

where (u_1, v_1, w_1) are the transformed barycentric coordinates. The graphical effect of this transformation T^{-1} on the *cube-dots* is shown in Figure 9. Note that, in those equations, different values of λ cause different final distributions of dots. We need to find the value of λ that achieves the best distribution, that is, the optimal

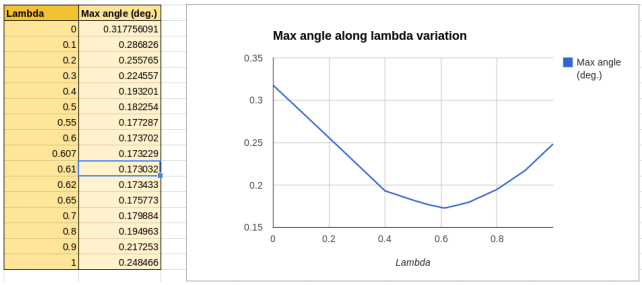


Figure 10: *Maximum angle (hole) in the sphere of transformed projected points achieved by varying λ from 0 to 1. We can see that the optimal value for λ is 0.61. With this value, Equation 2 achieves its best distribution of dots, maximizing the representability of gradient directions.*

Method	Angle (deg.)
No treatment	0.3177
48 regions decoding	0.1730
Theoretical min.	0.0615

Table 1: *Angle denoting the biggest hole in the distribution of dots on the surface of a sphere. The measures here presented have been achieved by generating a triangulation of the* sphere-dots *and taking the diameter of the biggest circumscribed circle among all triangles.*

value that achieves the best minimization of the empty regions at the boundaries of the triangles. This can be done by measuring the biggest angle between the directions of all pairs of neighbouring *sphere-dots* over the whole sphere.

Optimization of the Monotonic Gradient Transformation

Figure 10 shows that the maximum angle among all possible pairs of neighbouring vectors varies as λ goes from 0 to 1. We have found that the best value for λ is 0.61, value for which we achieve a maximum angle of 0.1730 *deg*. Table 1 shows the maximum angle between two neighbouring gradients when not treating the projected dots, using the proposed transformation and the theoretical minimum. The transformation proposed here achieves a resulting dot distribution that approaches to the theoretical optimum (see the paragraphs below), which might not be achievable, given the total number of points representable by the three XYZ byte-components.

Figure 11 shows the comparison between the *sphere-dots* in both cases: without any treatment and with the proposed decoding transformation T^{-1}. It is clear that empty bands patterns are more evident in the former case.

Demonstration of the maximum angle

Let's suppose that we have 2^{24} points uniformly distributed over the surface of a unit sphere (consider an almost-isotropic distribution of points over the sphere given by an iterative subdivision of a tetrahedron). As the surface of a certain sphere is given by the formula $4\pi R^2$, and $R = 1$ in this case, our sphere will have a surface of 4π. If points are uniformly distributed, as it is the case, a

Figure 11: *Comparison of both cases, untransformed and transformed gradients XYZ (5 bits per value are used in this case to avoid dot cluttering and see the empty patterns on the surface). Untransformed gradients (a) present empty bands, which supposes directions that cannot be encoded. Transforming gradients with the presented method reduces those empty bands, and thus, optimizes the usage of the 3D space to represent more gradient directions.*

triangulation of this point cloud on the surface will only include triangles that are practically equilateral. In closed triangle meshes, the number of triangles is twice the number of vertices ($T = 2V$). Each triangle surface will be then $4\pi/(2V) = 4\pi/(2 \times 2^{24})$. The maximum angle (in radians) in this case will be the diameter of the circumscribed circle of any of these triangles, and that is because in this distribution of points, we consider all *holes* between points to be equal. We have that the surface S of a triangle circumscribed within a circle of diameter D is $S = \frac{3}{16}\sqrt{3}D^2$. Then:

$$D = 1.756\sqrt{S} = 1.756\sqrt{2\pi}/2^{12} = 4.4/4096\ rad = 0.0615\ deg$$

4.2. Decoding

The aforementioned procedure is able to maximize the representable space of gradient directions using 3 values XYZ of 8 bits. That step actually corresponds to the decoding stage of the pipeline that will take place in the GPU after evaluating the gradient from the 3D texture (see Figure 12). The gradient to use in shading operations is in fact the one obtained after the transformation T^{-1} takes place. Summarizing, the steps to follow in the decoding stage after obtaining the encoded gradient from the 3D texture are the following:

1. Identify the corresponding triangular region.
2. Obtain the barycentric coordinates of the *cube-dot*.
3. Perform the transformation T^{-1} on the *cube-dot*.
4. Convert the transformed *cube-dot* back to euclidean coordinates and normalize to obtain the interpolated gradient direction.

4.3. Encoding

Figure 12 shows an overview of the gradient encoding/decoding pipeline. The process of encoding gradients T is exactly the inverse from the decoding transformation explained in the previous section. Before performing the encoding process, the gradients we are working with are stored with floating point precision coordinates to avoid precision loss as much as possible. The steps to follow in order to obtain an encoded vector $\vec{v_e}$ given an original vector $\vec{v_o}$ are quite similar to the decoding operation:

1. Identify the corresponding triangular region.
2. Obtain the barycentric coordinates of the *cube-dot*.

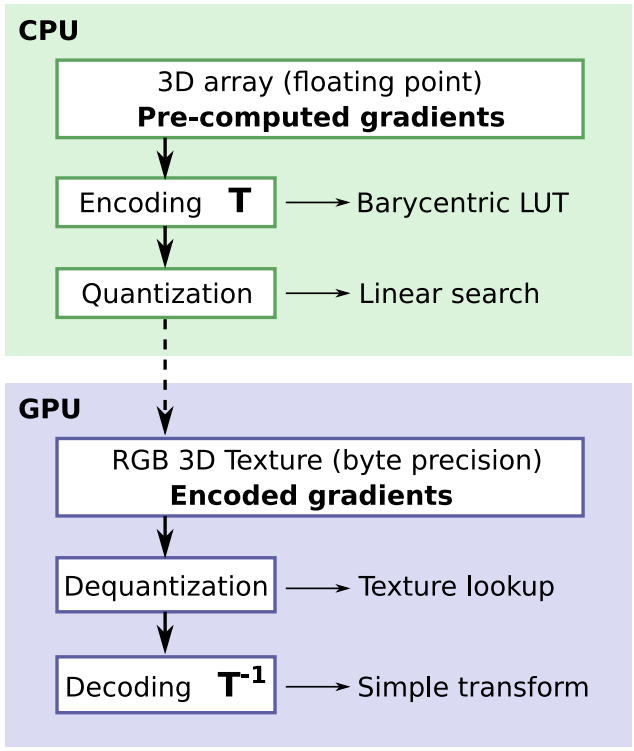


Figure 12: *Diagram of the tasks in the different stages of the encoding/decoding pipeline. The tasks carried out by the CPU happen in pre-processing time. Encoding gradients requires retrieving the transformed values from the pre-calculated LUT, and quantization involves performing a linear search to find the best matching quantized value. In GPU, however, all tasks are fast: dequantization is done at the time of retrieving the encoded value from the 3D texture, and the decoding transformation T^{-1} is a fast calculation involving a few product calculations.*

3. Perform the inverse transformation T on the *cube-dot*.
4. Convert the transformed *cube-dot* back to euclidean coordinates.
5. Quantize the transformed gradient.

Transformation of barycentric coordinates During the encoding stage, the transformation T of barycentric coordinates in the triangular region must be the inverse as explained in the formulas for decoding in Equation 2 (see the effect of this transformation in Figure 9). These equations are a system of three dependent quadratic equations. The easiest way to find the inversion of this system is to proceed with numerical methods. External mathematical packages provide us with tools to solve this problem easily. In our case, we have used *R*, a language designed for statistical analysis. However, this kind of computation is an expensive operation that is better to avoid during the encoding process of many gradients.

Look up table We have numerically solved the inversion of this system for relatively big set of barycentric coordinates in order to construct a look up table that maps barycentric coordinates to

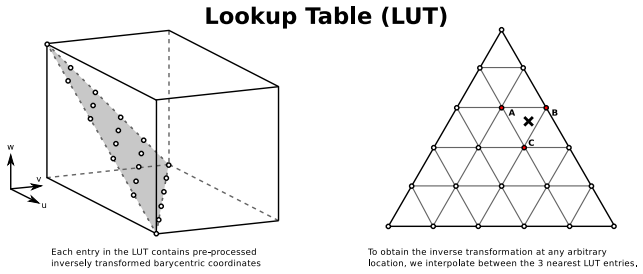


Figure 13: *Barycentric transformation T look up table (LUT). Each point in the figure corresponds to an entry of the LUT. Entries contain the transformed barycentric coordinates needed for the encoding process. Notice how, despite the fact that barycentric coordinates have 3 components, only the subspace depicted by the triangular plane contained in the cube is used (barycentric coordinates not belonging in this plane are not normalized and hence they are not useful for us). We can obtain transformed barycentric coordinates at any arbitrary location in the triangle by interpolating the contents of the 3 nearest entries.*

their transformed correspondences (T). Figure 13 shows the geometrical representation of the look up table. Points representing valid barycentric coordinates lie on the triangular diagonal plane in the cube. This structure stores transformed barycentric coordinates at several points on that plane. In order to know the inverse barycentric transformation of a certain coordinate, we find its location on that plane and compute the interpolation among the tree pre-computed values stored in the three nearest vertices. By converting the resulting *cube-dot* back to euclidean coordinates again, we obtain a new high-precision gradient v_l that once decoded is practically equal to the original gradient v_o.

Quantization In this step the high precision transformed gradient v_l is downcasted into an 8 bit component quantized gradient v_q. This operation finds the point in the original point cloud (Figure 6) that once transformed by the decoding operation, best matches the original floating point precision gradient to be encoded. We use the gradient v_l obtained from the look up table explained above to perform a linear search over the point cloud, starting at the center of the point cloud and following the direction of v_l. Figure 14 shows how the transformed gradient direction (the one achieved by means of the transformation LUT) can be used to greatly limit the search space over the whole possible quantized values, only considering the quantized values surrounding the direction of the transformed gradient. By performing this fast linear search, we select the quantized value which, once decoded, best represents the original gradient direction.

5. Results

Figure 15 shows the difference between shading several coarse datasets using gradients computed on-the-fly and using pre-computed, filtered gradients, with respect to the ray casting image of their original datasets. It is easy to see how the downsampled datasets rendered with pre-computed filtered gradients obtain much

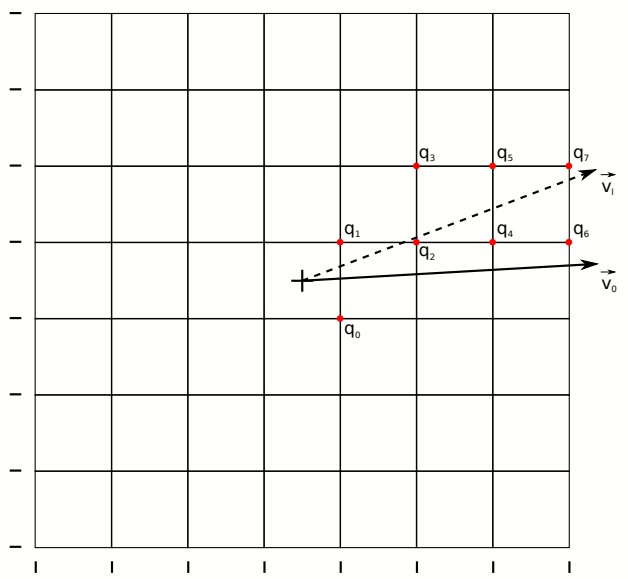


Figure 14: *Linear search in the quantization process. The transformed gradient v_l is obtained from the original vector v_0 with the help of the LUT. v_l is then used to perform a linear search over a limited subspace of the point cloud. The final quantized point q_i is the one that, after applying the decoding transformation T^{-1}, is more similar to the original vector v_0.*

Stage	Vix	Chameleon	Head
Pre-computation	10	20	18
Downsampling	75	143	134
Encoding	62	129	117

Table 2: *Time (in seconds) used to pre-compute, downsample and encode gradient data of various datasets. The algorithms have been executed in a single CPU thread, traversing the whole sample space of each dataset without being parallelized.*

better results than computing gradients on-the-fly, even when the coarse datasets are the same. This demonstrates the importance of gradients in shading. The staircase artifacts that are visible in the downsampled datasets shaded with gradients evaluated on-the-fly in shader code could be smoothened by increasing the size of the kernel used in the downsampling filter of the scalar field. However, as the size of the kernel grows, the scalar field becomes smoother and smoother, and more features are prone to disappear consequently. Using pre-computed and downsampled gradients makes it possible to remove these undesirable staircase artifacts without sacrificing important features.

Table 2 shows the times for the pre-computation, downsampling and encoding of gradient directions using the three datasets in our tests. We have used the central differences approach to pre-compute gradients from the original dataset. The most time consuming, pre-processing stage, is the use of the downsampling filter for the generation of the coarse dataset of gradients, followed by the encoding

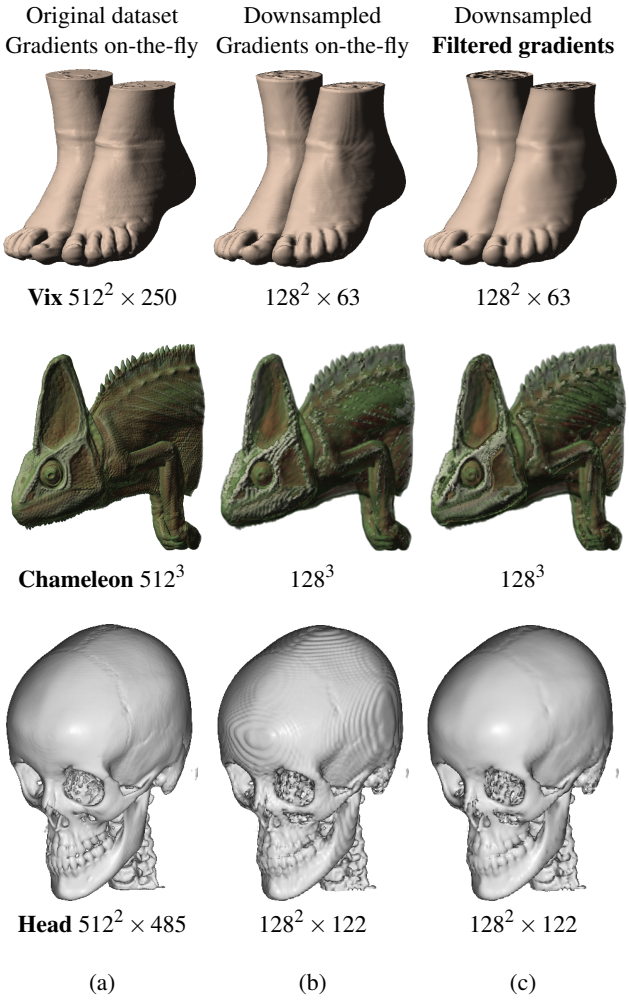


Figure 15: *Comparison of ray casting images rendered with different gradients. Column (a) shows ray casting images of several datasets at its original resolution (the shading was done using gradients computed on-the-fly). Images in column (b) show a coarse version of the datasets shaded with gradients also computed on-the-fly in the shader (notice the staircase shape of the surfaces). In (c) the same coarse datasets are rendered, using pre-computed, filtered gradients that better preserve the orientation of the original surfaces.*

LUT size	32^3	64^3	128^3
Generation time (seconds)	0.55	1.59	6.17

Table 3: *Time needed to generate LUTs of different sizes for the transformation T. Notice that, although these LUTs represent a space in 3D, the time increases approximately by a factor of 4 when the dimension of the LUT increases by a factor of 8. This is due to the fact that the subspace of useful entries in these LUTs is represented by a triangular plane representing only the barycentric coordinates that make sense (see Figure 13).*

Encoding method	**Max. error**
No encoding	0.318 *deg.*
Encoding (LUT 32^3)	0.165 *deg.*
Encoding (LUT 64^3)	0.165 *deg.*
Encoding (LUT 128^3)	0.165 *deg.*

Table 4: *This table shows the maximum error introduced during the storage of gradients in the 3D texture for a big set of randomly generated gradients. Notice that encoding gradients using our transformation plus quantization scheme produces results twice as good as using a plain quantization without any encoding. Given the monotonic shape of the transformation, even small LUTs (32^3) are enough to obtain the best results (the LUTs are encoding a low-frequency transformation, this is why there is no need for bigger LUTs to achieve good representations of the transformation).*

(using the LUT) plus the quantization of the gradients to pass them to the GPU. As shown in Table 2 the whole process takes a few minutes for the bigger dataset we have tested (Chameleon 512^3), which is an acceptable amount of time for a pre-process. These computations (and the following ones) have been done in a machine with an Intel(R) Core(TM) i7 CPU 930 at 2.80GHz and 8GB of RAM memory. Although, the processor has several cores, the calculations done for this paper have not been optimized to make use of multi-threading or SIMD instructions.

In order to perform the encoding of gradients, the LUT for the transformation T must have been previously generated, as solving such difficult operation for all gradients, given the large amount of voxels that supposes a dataset, is too expensive. This process requires solving the complex system of equations that supposes inverting the simple decoding transformation (Equation 2). We have used the software package *R* in combination with its module *rootSolve* to compute the transformation T and storing it into a LUT. We have generated LUTs of different sizes and tested their effectiveness to encode pre-computed gradients in terms of encoding time and error. Table 3 shows the time taken to generate LUTs of three different sizes. Notice that the cost of generating a LUT of 128^3 entries (which is more than needed as explained later) is very small.

We have tested the goodness of the usage of a LUT for the barycentric transformation T by completing the whole process of encoding/quantizing/decoding for a huge set of randomly generated gradients. Table 4 shows the maximum error obtained after performing the test with several storing methods. As we can see, there is no need to use big LUTs (which would be wasting space

in main memory), as the lowest resolution LUT used in our tests (32^3) is able to perform the transformation T without exceeding the *biggest hole* angle mentioned in Table 1. Although the proposed method for gradient encoding does not provide results that are visually much superior, it achieves a significant improvement in the numerical results that may be useful in other scenarios.

6. Conclusions

In volume rendering, the way in which gradients are computed affects in great measure the quality of the shading of coarse datasets in multi-resolution structures. Commonly, gradients are evaluated on-the-fly by the shader code by accessing neighbouring positions. However, the new topology of downsampled datasets provides gradients of worse quality that do not resemble the originals as much as desired, and thus shading shows non desirable artifacts.

To solve this issue we have presented two contributions:

- A downsampling filter for pre-computed gradients.
- An encoding/decoding scheme for pre-computed gradient directions.

The proposed downsampling filter for pre-computed gradients provides improved gradients that better match the original dataset gradients such that the aforementioned artifacts disappear.

Regarding the second contribution, existing algorithms for encoding normal vectors and/or gradients cannot be used in the context of volume rendering by storing them into a 3D texture. These solutions cited in Section 2 have serious interpolation issues at the time of sampling values. The presented method to encode gradient directions into a byte precision 3D texture, besides not presenting this problem, maximizes the space of representable directions and reduces the maximum error introduced by the storage format.

In the future, we would like to test the goodness of the proposed encoding scheme beyond the scope of volume rendering, for instance, in combination with triangle meshes.

Acknowledgements

The material in this paper is based upon work supported by the Spanish *Ministerio de Economía y Competitividad* and by FEDER (EU) funds under Grant No. TIN2014-52211-C2-1-R. We would also like to thank the anonymous reviewers for their contributions and useful comments to improve the contents of this paper.

References

[BHP14] BEYER J., HADWIGER M., PFISTER H.: A Survey of GPU-Based Large-Scale Volume Visualization. *Proceedings EuroVis 2014* (2014). 2

[BLM96] BENTUM M. J., LICHTENBELT B. B. A., MALZBENDER T.: Frequency analysis of gradient estimators in volume rendering. *IEEE Transactions on Visualization and Computer Graphics 2*, 3 (Sep 1996), 242–254. 2

[BNS01] BOADA I., NAVAZO I., SCOPIGNO R.: Multiresolution Volume Visualization with a Texture-based Octree. *The Visual Computer 17*, 3 (2001), 185–197. 2

[BRGIG*14] BALSA RODRÍGUEZ M., GOBBETTI E., IGLESIAS GUITIÁN J. A., MAKHINYA M., MARTON F., PAJAROLA R., SUTER S.: State-of-the-Art in Compressed GPU-Based Direct Volume Rendering. *Computer Graphics Forum 33*, 6 (2014), 77–100. 2

[CD09] CSÉBFALVI B., DOMONKOS B.: Prefiltered gradient reconstruction for volume rendering. *Journal of WSCG 17*, 1 (Jan 2009), 49–56. 2

[CNC13] CAMPOALEGRE L., NAVAZO I., CROSA P. B.: Gradient octrees: A new scheme for remote interactive exploration of volume models. In *2013 International Conference on Computer-Aided Design and Computer Graphics* (Nov 2013), pp. 306–313. 2

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2009), I3D '09, ACM, pp. 15–22. 2

[DDSD03] DÉCORET X., DURAND F., SILLION F. X., DORSEY J.: Billboard clouds for extreme model simplification. *ACM Trans. Graph. 22*, 3 (July 2003), 689–696. 2

[Dee95] DEERING M.: Geometry compression. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1995), SIGGRAPH '95, ACM, pp. 13–20. 2

[DGBN*16] DÍAZ-GARCÍA J., BRUNET P., NAVAZO I., PEREZ F., VÁZQUEZ ALCOCER P.-P.: Adaptive transfer functions. *Vis. Comput. 32*, 6-8 (June 2016), 835–845. 2

[DH73] DUDA R. O., HART P. E.: *Pattern Classification and Scene Analysis*. John Wiley and Sons, 1973. 2

[GIM12] GOBBETTI E., IGLESIAS GUITIÁN J., MARTON F.: COVRA: A compression-domain output-sensitive volume rendering architecture based on a sparse representation of voxel blocks. *Computer Graphics Forum 31*, 3pt4 (2012), 1315–1324. Proc. EuroVis 2012. 2

[GS04] GUTHE S., STRASSER W.: Advanced techniques for high-quality multi-resolution volume rendering. *Computers & Graphics 28*, 1 (2004), 51–58. 2

[HKRs*06] HADWIGER M., KNISS J. M., REZK-SALAMA C., WEISKOPF D., ENGEL K.: *Real-Time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006. 2

[KB08] KRAUS M., BÜRGER K.: Interpolating and Downsampling RGBA Volume Data. In *Proceedings of Vision, Modeling, and Visualization 2008* (2008). 2

[KE01] KRAUS M., ERTL T.: Topology-Guided Downsampling. In *Volume Graphics* (2001), Mueller K., Kaufman A., (Eds.), The Eurographics Association. 2

[OB06] OLIVEIRA J. A. F., BUXTON B. F.: Pnorms: Platonic derived normals for error bound compression. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (New York, NY, USA, 2006), VRST '06, ACM, pp. 324–333. 2

[PR05] PHARR M., RANDIMA F.: *GPU Gems 2*. Addison-Wesley Professional, 2005. 2

[SHKM14] SICAT R., HADWIGER M., KRÜGER J., MÖLLER T.: Sparse PDF volumes for consistent multi-resolution volume rendering. *IEEE Transactions on Visualization and Computer Graphics (Proc. IEEE Visualization) 20*, 12 (2014), 2417–2426. 2

[TM98] TOMASI C., MANDUCHI R.: Bilateral filtering for gray and color images. In *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)* (Jan 1998), pp. 839–846. 3

[WWLM11] WANG Y. S., WANG C., LEE T. Y., MA K. L.: Feature-Preserving Volume Data Reduction and Focus+Context Visualization. *IEEE Transactions on Visualization and Computer Graphics 17*, 2 (Feb 2011), 171–181. 2

[YMC06] YOUNESY H., MÖLLER T., CARR H.: Improving the quality of multi-resolution volume rendering. In *Proc. Joint Eurographics/IEEE VGTC conference on Visualization* (2006), Eurographics Association, pp. 251–258. 2