Texture Partitioning and Packing for Accelerating Texture-based Volume Rendering

Wei Li Arie Kaufman

Center for Visual Computing (CVC) and Department of Computer Science
Stony Brook University, Stony Brook, NY 11790-4400, USA
{liwei,ari}@cs.sunysb.edu

Abstract

To apply empty space skipping in texture-based volume rendering, we partition the texture space with a box-growing algorithm. Each sub-texture comprises of neighboring voxels with similar densities and gradient magnitudes. Sub-textures with similar range of density and gradient magnitude are then packed into larger ones to reduce the number of textures. The partitioning and packing is independent on the transfer function. During rendering, the visibility of the boxes are determined by whether any of the enclosed voxel is assigned a non-zero opacity by the current transfer function. Only the sub-textures from the visible boxes are blended and only the packed textures containing visible sub-textures reside in the texture memory. We arrange the densities and the gradients into separate textures to avoid storing the empty regions in the gradient texture, which is transfer function independent. The partitioning and packing can be considered as a lossless texture compression with an average compression rate of 3.1:1 for the gradient textures. Running on the same hardware and generating identical images, the proposed method however renders 3 to 6 times faster on average than traditional approaches for various datasets in different rendering modes.

Key words: Texture-based volume rendering, empty space skipping, graphics hardware, box growing, lossless texture compression, texture partitioning, texture packing.

1 Introduction

A volumetric dataset typically contains a large amount of voxels with zero values, and some parts of the volume are assigned a fully transparent (invisible) opacity, depending on the transfer function. All these areas do not contribute to the rendering and can be ignored. In traditional texture-based volume rendering, the whole volume is represented as a 3D texture or stacks of 2D textures. All the textures are loaded into the texture memory and blended for rendering.

In this paper, we partition the volume space into sub-volumes. The basic idea is to group voxels with similar properties in both the volume domain (position) and the transfer function domain (densities and gradient magnitudes) into the same sub-volume. Due to the coherence in the transfer function domain, voxels having similar properties are probably assigned similar opacities. A reasonable transfer function generally maps certain densities (or gradient magnitudes) clustered in a neighborhood of the transfer function domain to fully transparent. Consequently, some of the sub-volumes comprise of only invisible voxels, which need to be neither stored nor rendered. Since volumes are treated as textures, we use the words volume (voxel) and texture (texel) interchangeably.

Note that the textures has to be axis-aligned rectangles. Intuitively, the partitioning can be done by region growing, followed by dividing the connected regions so that the region borders are approximated by their bounding boxes. Instead, we propose the so-called *box growing* method that combines the two steps and provides more control over the partitioning. Figure 1a shows the projection of the boxes on a slice of the foot dataset, while Figure 1d is a 3D view of the boxes. The color of the boxes represent the type of the boxes. Red means uniform density, in that the voxel densities vary in a small range, while blue boxes contain voxels with high gradient magnitude. The box growing works on unclassified data, hence is transfer-function-independent.

To achieve real time or interactive rendering, all or most of the textures needed for generating the current view are required to reside in the texture memory. When the size of a dataset exceeds the memory capacity, swapping between the texture memory and the main memory degrades the performance, due to the limited bandwidth of the AGP bus. This scenario is more likely to happen when the gradients are also stored to achieve various illumination effects, whose storage is usually three times larger than that of the densities. Fortunately, a gradient volume generally has more zero values than the corresponding densities, since both zero and uniform density regions correspond to gradients with zero magnitude. Fortunately, the empty regions in the gradient volume

are transfer-function-independent. Therefore, we choose to store the densities and the gradients in separate textures. Both the density textures and the gradient textures are partitioned by the same set of boxes, but the gradient sub-textures with only zero gradient magnitude are simply skipped for storage and rendering.

To reduce the number of textures as well as the overhead of texture setup and switching, we propose a greedy algorithm that packs multiple variable-sized smaller textures into larger ones. The smaller textures have similar densities or gradient magnitudes, but are not necessarily neighbors in the volume space. Figure 1b displays only the boxes with a non-zero gradient magnitude, while Figure 1c shows the packed gradient texture. It can be seen that texture partitioning and packing also serves as a lossless texture compression with no explicit decoding for the gradient textures that account for most of the memory requirement. An alternative is to exploit texture compression available in standard graphics APIs, such as OpenGL and D3D. However, to date, neither of the two supports lossless compression, while with the lossy compression, the errors in the textures, especially those in the gradients, significantly degrade image quality [9]. Furthermore, our texture partitioning and packing is a combination of compression and acceleration in that it reduces not only the memory requirement, but also the number of texels rendered. Even if there is a hardware support for lossless texture compression and the decompression has no impact on performance, it only saves memory.

In our method, every box is associated with summarized properties of the enclosed voxels, such as the ranges of the densities and the gradient magnitudes. Before rendering an image, the visibility of each box is determined according to whether any of the enclosed voxel is assigned a non-zero opacity by the current transfer function. Only the sub-textures from the visible boxes are rendered and only the packed textures containing visible sub-textures are required to reside in the texture memory. Figure 1e shows a mixed rendering of boxes and textures. Note that it renders only a portion of all the boxes displayed in Figure 1d. We also apply gradient magnitude modulation and thus only boxes with non-zero gradient magnitude (blue) are visible in Figure 1e.

In the rest of the paper, after a brief review of related work, we present texture partitioning by box growing. Specifically, we discuss how voxels with different densities and gradient magnitudes are partitioned with the aim of accelerating the rendering. We then present a greedy algorithm for texture packing. For convenience, we coined the phrase *PPed texture*, short for partitioned and packed texture. Next, we discuss volume rendering with *PPed* textures. Finally, we present experimental results showing the effectiveness of our techniques.

2 Previous Work

Empty space skipping has been extensively exploited to accelerate volume rendering, mainly for software-based methods. It avoids processing empty voxels with the help of various pre-computed data structures, such as proximity cloud [3] and bounding convex polyhedra [1]. On the other hand, contiguous empty regions and other redundancy inside volumetric datasets have been utilized for compression to reduce memory requirement as well as to accelerate rendering. Examples of such compression are transform-based compression (e.g., [5]) and 3D adjacency data structure [10], to name a few.

Due to the recent advances of commodity graphics hardware, texture-based volume rendering [4, 11] has achieved frame rate better than software-based methods with satisfying image quality. Applying empty space skipping in texture-based volume rendering has the potential to further accelerate the rendering or makes it possible to handle larger datasets. Since the shape of a texture has to be a rectangle or a box, it is not straightforward for many of the empty-space-skipping and compression techniques, which are designed for software renderers, to be applied to hardware accelerated rendering.

Both Boada et al. [2] and LaMar et al. [7] subdivide the texture space into an octree. They skip nodes of empty regions and use low-resolution textures for regions far from the view point or of lower interest. In this paper, our growing box is not restricted to any regular grid or octree and BSP tree node, hence better approximates the boundaries of different regions with fewer partitions.

Our previous work [8] computes the texture hulls of all the connected non-empty regions. Only the sub-textures defined by the bounding rectangles are stored and the texels inside the hulls are rendered. However, the texture hulls have to be recomputed as the transfer function changes and is less efficient in reducing storage for empty regions enclosed by non-empty regions, which unfortunately is common in a gradient volume. Besides, texture hulls are limited to 2D textures. The *PPed* texture is transfer function independent and can be exploited for either 2D or 3D texture-based methods.

Kraus and Ertl [6] integrate decoders for texture data into the programable texture hardware. They pack regularly divided sub-textures into uniform-sized larger textures. The packing is similar in purpose to our work. However, their texture blocks are uniformly shaped with power-of-two sizes along each axis, while in our work, sub-textures are of arbitrary size, and the decoding is implicit by transforming the texture coordinates. Beside, our focus is on acceleration rather than compression. Ac-

tually, we achieve significant speedup even without the compression by simply mapping the unpartitioned textures on the partitioning boxes.

3 Texture Partitioning

3.1 Partitioning with Boxes

Texture partitioning divides the volume (texture) into a set of sub-textures. The partitioning results in a set of boxes, each defining a sub-volume. The boxes don't overlap and each box is visited at most once, which guarantees that no voxel is blended multiple times. Recall that during rendering, we determine the visibility of a box based on whether the current transfer function maps any of its enclosed voxel opacity to a non-zero value. Obviously, just rendering the sub-textures defined by those visible boxes generates exactly the same image as that of rendering all the textures. With the aim of accelerating volume rendering, the visible box subset of a good partitioning should encloses as few invisible voxels as possible for various reasonable transfer functions. Naively, it can be approached by decreasing the size, or equivalently, increasing the number of the boxes. However, the more boxes we have, the more textures we need to render, and the more overhead in setting-up, binding and switching the textures. Besides, to ensure proper linear interpolation, we need to replicate voxels at the border of the boxes [6]. Hence, more boxes, more duplicated storage.

The most commonly used transfer function is a 1D lookup table mapping density to color and opacity. Non-zero opacity in the transfer function domain usually corresponds to one or more density ranges. It is natural to partition neighboring texels whose densities vary in a small range (a typical value is 32 for 8-bit volume) into a sub-texture. We refer to a box enclosing such a sub-texture as a *uniform* box. To determine visibility, we only need to compare the density range of the sub-texture with the non-zero density range of the transfer function. This strategy encourages the border of the boxes to separate neighboring voxels with large density difference where there is an edge or a surface.

However, this only works well for unilluminated rendering. For volume rendering with lighting, only the voxels near edges or surfaces contribute to the lighting. In most cases, the uniform boxes on both sides of the edge/surface are visible which involves many voxels with zero gradient magnitude for lighting computation. To avoid such inefficiency, we first separate the voxels with non-zero gradient magnitude from the rest by a set of *gradient* boxes. Then, we cover the uniform regions with *uniform* boxes. Because we place a restriction on the minimal size of *uniform* boxes, there may still be some spaces left, which are filled by *other* boxes. Figure 1a shows the projection of the box set onto a slice, while Figure 1d shows the boxes in a 3D view. The type of the boxes is represented by color, red for *uniform*, blue for *gradient*, and green for *other*.

As discussed before, we separate the storage of the gradient textures from the density textures. The gradient and the density textures are partitioned by the same set of boxes. Density sub-textures are created for each box while only *gradient* boxes are associated with gradient textures. During rendering, the visibility of the density textures and the gradient textures, if any, of the same box is determined independently. We use *density-visibility* and *gradient-visibility* to differentiate them. Obviously, only a *gradient* box can be *gradient-visible*.

We define a cost function for each box. For a *uniform* box, its cost function is the density range of its voxels, while for a *gradient* box, it is the percentage of its voxels with zero gradient magnitude. The cost function for an *other* box is the negative value of its area or volume. Since we search for different types of boxes at different stages, it is unnecessary to compare the cost functions of different box types. The partitioning should balance between the number of boxes and the total cost of the boxes.

3.2 Box Growing

An intuitive way of computing the boxes by region growing is to find all the connected regions each observing a certain criteria, such as low density variance, followed by dividing the connected regions so that they can be approximated by a set of boxes. Instead, we propose box growing that allows boxes to grow by themselves. The general rule is to let the boxes grow as large as possible while keeping the accumulated cost function smaller than a predetermined threshold. We illustrate below box growing on 2D textures, although it extends naturally to 3D.

Each box is started from a seed texel. The criteria of the seed texel depends on the type of the box to be grown. For any box type, a seed texel should not be enclosed in any of an existing box. We refer to a texel with nonzero gradient magnitude as a *gradient texel*. For a *gradient* box, a seed has to be a *gradient texel*. For a *uniform box* or an *other box*, any untaken texel suffices. Every step of growing merges one row or column of texels onto its side (one slice for 3D box growing). Each potential region to merge is a box as well, which we refer to as a side-box. Naturally, we pick the side-box with the minimal cost function. If after merging the selected side-box, the accumulated cost function of the box being grown is less than a given threshold, the growing step is executed. Otherwise, the growth for this box is completed.

For convenience, we number the sides with an index starting from zero. The sides are then visited in ascend-

ing order of the index. It is possible that a cost function of a side-box reaches the smallest possible value, for example, 0 for a *uniform box*. In such a case, we don't need to test the remaining sides. To prevent a box from growing on the same side, we pick the first side to compute the cost function at each step in a round-robin fashion, which is the next side of the side-box being merged in the previous step. If the side of the largest index has been reached and there is still an unvisited side for the current growing step, the index wraps around to 0. While growing, the ranges of the densities and the gradients of the voxels are recorded. For a *uniform* box, we maintain a single *range* structure that stores the minimum and maximum of the densities. For a *gradient* box or an *other* box, we maintain a list of such *ranges*, since their maximal and minimal densities usually differ greatly, whereas all the densities cluster into two or more small ranges. A range of gradient magnitude is associated with each density range.

The following is the box growing algorithm: (1) Take a remaining seed texel, start a new box containing only that texel. (2) Find the side-box having the least cost. The cost is set to infinity, if the side-box encloses any *taken* texel or the side-box is outside the volume, which prevents the box from overlapping with other boxes or getting out of the dataset. (3) Compute the accumulated cost by assuming the side-box is merged. If the cost is smaller than a given threshold, merge and mark all the newly added texels as *taken*. Otherwise, the growth of the current box ends; go to step (1).

To compute the cost function, we need to traverse all the texels inside the side-boxes. This is done incrementally. For example, if a box grew left in the previous step, then the cost function for the right remains unchanged, while those for the top and the bottom are the previous values combined with the statistics (e.g., range of the density and the gradient magnitude) of the corner texels at the top-left and the bottom-left, respectively.

3.3 Mapping the Partitioned Textures

For rendering, the sub-textures are mapped on geometric primitives derived from the corresponding boxes. For 2D, the primitive is simply the box itself; for 3D, it is the intersection of the box with a plane at proper position and orientation. Like Kraus and Ertl [6], we consider that the texel values are sampled at the lower corner of the domain that the texel occupies, rather than the center of the domain as specified in OpenGL. When implementing our algorithm in OpenGL, all the texture coordinates are shifted by the size of one half texel divided by the corresponding size of the texture. (For Nvidia's texture rectangle extension, it is simply the size of one half texel if using .)

In Figure 2, the center of each large dot represents the sample point of a texel and the rectangle is the border of a box. All the texels enclosed or crossed by the box are included in a sub-texture associated with the box and are mapped onto the box. Note that the texels on the border of the box are shared by the neighboring boxes (not drawn). Because of the sharing, after a box is grown, we replace the *taken* mark of any texel crossed by the box edges to *border*, so long as the texel has at least one *untaken* neighbor. For example, in Figure 2, texel *T1* is left as *taken*, while texel *T2* is changed to *border* so that it can be included in the neighboring box on the right.

With linear interpolation, each non-empty texel spreads the "non-emptiness" to its neighbors in a circle of radius one. If a border between empty and non-empty texels lies inside a box, the linear interpolation between them is implicitly done by texture mapping. Since our rendering algorithm discards textures from invisible boxes completely, we need to consider the case when a section of the border coincides with the boundary of boxes. For gradients, we dilate the regions of the *gradient texels* by one (texel) before partitioning. Therefore, no *gradient texel* lies on a border that is not shared by two *gradient* boxes. Since all the *gradient* boxes are *gradient-visible*, it is guaranteed that any *gradient texel* on one side of the border has a *gradient* neighbor on the other side in a neighboring *gradient* box. Both of the *gradient texels* contribute to the lighting computation. The density range of a box considers the texels that are shared with other boxes. Therefore, all the boxes sharing the border texels are *density-visible* and contribute to the interpolation.

3.4 Controlling the Number and Shape of Boxes

Increasing the number of boxes is likely to increase the approximation accuracy of the region boundaries. However, the number of textures increases as well, so does the number of the replicated texels and texture overhead. Besides, too many boxes can possibly turn the geometric transformation stage into the bottleneck of the rendering pipeline and make the rendering even slower than that with the unpartitioned textures. Therefore, we need a scheme to keep the number of boxes at a reasonable level. During box growing, the thresholds for the cost function affect the size of the boxes, or equivalently, the number of boxes. However, it is not easy to control the number of boxes simply by adjusting these thresholds.

Generally, the number of boxes is not an issue for small datasets, which inspires us to apply box growing on down-sampled volumes for large datasets. The down-sampling is applied by dividing the volume into uniformly sized blocks and each block is then converted into a single voxel. For the gradient volume, the value of each down-sampled voxel is the maximal gradient

magnitude of the corresponding block. For the density volume, the value of a down-sampled voxel is a range, storing the minimal and maximal densities of the block. Note that the dilation of the gradient is applied before the down-sampling. Otherwise, the dilated region will be larger than necessary. The down-sampling is only for box growing. The grown boxes are then scaled back and sub-textures are extracted from the original volume. The thresholds for determining whether a down-sampling is necessary are different for the gradient volume and the density volume, which we chose empirically to be 128^3 and 256^3, respectively.

After a box is grown, if any of its axis-aligned side is too small, the box is discarded and all the texels enclosed are flagged as *attempted*. The box growing continues from different seed texels. Discarding small and sliver boxes improves the shape and the distribution of the boxes. As shown in Figure 3, without discarding, the partitioning contains many sliver boxes, which increases the surface of the boxes as well as the texel duplication. This is also the reason that we need to have *other* boxes to fill in the uncovered regions. An important trick is that no *attempted* texel is used as a seed for growing a box of the same type, without which, the time for box growing is typically an order of magnitude longer. Most sliver boxes are prevented from further growing by existing boxes. It is very likely that using any of the *attempted* texel as a seed results in the same sliver box. If an *attempted* texel will be in a good shaped box, it can be reached from a texel outside the sliver box.

4 Texture Packing

Texture partitioning increases substantially the number of textures, as well as the overhead of texture set up and switching. The impact is evident when textures are transferred between the graphics card and the host computer. Many subdivided textures take several times longer than a single large texture, even though the aggregate size of the former may be just half of the latter. To solve this problem, we pack the sub-textures into a box as tight as possible with no overlap, and stitch them together to create a larger texture. Like partitioning, sub-textures with similar texel properties that are likely to be rendered or skipped together are packed into the same texture. The sub-textures are not required to be neighbors in the volume domain. The offsets and the orientations of the sub-textures are book-kept in the enclosing boxes which are used to compute the texture coordinates in the packed textures.

Texture packing is similar to the NP-hard bin packing or strip packing problems. Therefore, we don't attempt to find an optimal solution, which is also unnecessary. Instead, we have developed a greedy algorithm. The basic ideas are similar to the box growing, in that, the target box grows as necessary to add in more textures. The algorithm is as follows: (1) Sort the textures by area (volume) in descending order. (2) Initialize the target box to be empty. (3) Add the largest texture to the target. There are four choices: add-right, add-right-transposed, add-top, and add-top-transposed. We choose the one that generates the smallest empty region. The target box then grows accordingly. (4) Find the largest texture in the list that fills into the empty region generated in the previous step, either in the original pose or transposed. Then, subdivide the rest of the empty regions into two smaller empty regions, and repeat this step recursively. (5) Go back to step (3) until the box list is empty.

Figure 1c shows the result of the algorithm by packing the sub-textures in Figure 1b. Note that the packed texture is smaller than half of the unpartitioned texture.

There are three strategies for the packing: (1) Pack 3D sub-textures defined by the box; (2) Pack 2D sub-textures, in which all the sub-textures from the same box are in the same packed texture; (3) Pack 2D sub-textures, with all the sub-textures from the same slice of the original volume are in the same packed texture. Strategy 1 is for 3D textures while the other two are restricted to 2D textures. The proposed greedy algorithm works for all the three. However, for strategy 2, many sub-textures are of the same size since they are from the same box, hence there are other optimizations to improve the packing. Choosing the strategy of packing is correlated to the choice of the compositing order, discussed next.

5 Volume Rendering with *PPed* Textures

Generally, densities and gradients have different empty spaces for skipping. The empty space in the densities depends on the transfer function, while that in the gradients corresponds to regions of uniform densities. As mentioned above, we adopt the strategy of storing densities and gradients in separate textures with the advantage of being stored in different layouts, with different compression methods, and even have different resolutions. The gradient textures are significantly compressed with the proposed approach (see Experimental Results below). Furthermore, we utilize the remaining channel in the gradient texture for gradient magnitude for efficient implementation of gradient magnitude modulation. Another advantage is that it is less expensive if the user just wants unilluminated volume rendering so that the gradient textures are not created and loaded at all. It is especially true for large datasets, when the texture memory can't hold both the density textures and gradient textures anyway.

There are two choices for the compositing order of

the textures. One is slice-by-slice, in which we slice the dataset in either front-to-back or back-to-front order. For each slicing plane, we render the corresponding sub-textures from the boxes intersected by the plane. The other is box-by-box, in which we first determine the visibility order of the boxes. Then, for each box, slices from the sub-texture are rendered in order. In our experiments we found that slice-by-slice outperforms box-by-box noticeably on current graphics hardware. Therefore, we adopt strategy 3, which is slice-by-slice 2D packing, to put all the sub-textures from the same slice into the same texture.

Whenever the transfer function is changed, the visibility of the boxes are refreshed. Recall that the visibility of the gradient textures and density textures are independent. The density values with non-zero opacity of the transfer function are clustered into one or more ranges. For each box, we then test whether these ranges overlap with the density range(s) of the box. If so, the box is visible, otherwise, invisible. The transfer function does not affect *gradient visibility*.

Our system supports three modes of rendering: unilluminated, normal illumination, and gradient modulation. In the last two modes, the colors and opacities of a rendered image come from two sources: the transfer function mapped intensities that can be considered as self-emission, and the lighting computed from the light sources and the gradients using the Phong lighting model. Different rendering modes have different criteria to determine whether the textures associated with a box is rendered. For unilluminated rendering, only the density textures of *density-visible* boxes are rendered. For rendering with gradient magnitude modulation, only boxes that are both *density-visible* and *gradient-visible* are rendered. Their density textures and gradient textures are used to compute self-emission and lighting, respectively, and the opacity is multiplied by the gradient magnitude. The illuminated rendering without gradient modulation is divided into two interchangeable steps for each slice. In one step, the boxes that are both *density-visible* and *gradient-visible* are rendered to contribute both self-emission and lighting. In the other step, boxes that are *density-visible* but not *gradient-visible* are rendered for self-emission only. Note that the projections onto the image plane of the boxes rendered in the two steps do not overlap.

6 Experimental Results

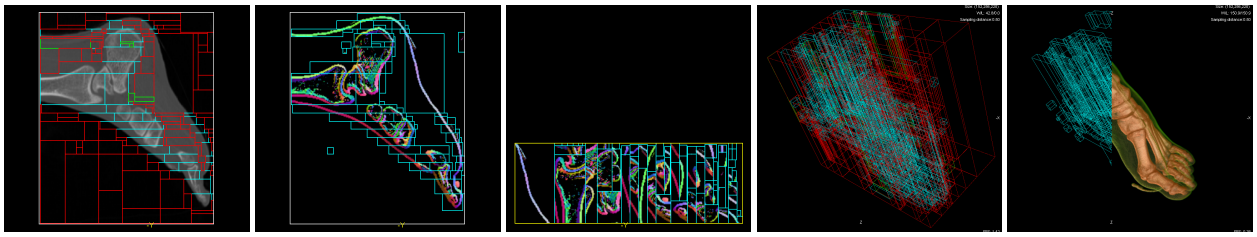
In our implementation, the density volume is stored as paletted textures, while the gradient textures contain the normalized gradients and the gradient magnitudes in RGBA format. Considering that accessing a 3D texture is still much slower than accessing a 2D texture on today's graphics hardware, the current implementation of the rendering and the texture packing only supports 2D textures, although the partitioning is done in 3D. Because all the density voxels have to be held in textures to accommodate all possible transfer functions and we adopt the slice-by-slice compositing order, we actually use the original density-texture stacks, instead of cutting out sub-textures and packing them. However, the density textures are still mapped to the boxes so that empty regions can be skipped for rendering. Whereas for the gradient textures, which account for 80% of the memory requirement, both partitioning and packing are applied. If we choose the box-by-box order, both the density and gradient textures are preferred to be *PPed*. We implement the technique proposed by Rezk-Salama et al. [11] to achieve trilinear interpolation with 2D textures for both the densities and the gradients and use dot-products for per-pixel lighting. Although not implemented yet, it is not difficult for *PPed* textures to support pre-integrated rendering [4] and lighting by environment maps [9].

We have tested our proposed methods on a 128MB GeForce 4 Ti 4600 card. Figures 4, 5, and 1e show the volume-rendered images of four datasets. To save space, only one image is shown for each dataset, although all of them are timed for all the three rendering modes. The size of each dataset is given in Table 1.

Figure 4 shows the unilluminated rendering of the neuron dataset. Table 1 compares the frame rates of the proposed approach (PPT) with the traditional method (Basic). The average acceleration rates for the four datasets are 3.8, while the rendered images from *PPed* textures is exactly the same as those from the traditional texture-based volume rendering. Note that the rendering speed depends on several factors, such as the size of the window, the zoom factor, the sampling distance, and when rendering with *PPed* texture, also the transfer function. However, within each row of the tables showing the frame rates, the values are obtained under exactly the same condition, except for employing different rendering methods.

Dataset	Size	Basic	PPT	Speedup
foot	$152 \times 256 \times 220$	10.3	35.6	3.5
neuron	$384 \times 256 \times 200$	12.0	60.0	5.0
head	$256 \times 256 \times 225$	8.7	16.8	1.9
engine	$256 \times 256 \times 110$	18.3	35.1	1.9

Table 1: Rendering speed (frames/sec) of unilluminated rendering with unpartitioned textures (Basic) and PPed *textures (PPT).*



(a) (b) (c) (d) (e)

Figure 1: Algorithm overview: (a) A slice of the foot dataset is partitioned by growing boxes. (b) Gradient sub-textures defined by the boxes enclosing all the voxels of non-zero gradient magnitude. (c) The gradient sub-textures in (b) are packed into a single larger texture, which is significantly smaller than the original slice. (d) A 3D view of all the boxes. (e) Rendering of the foot with mixed boxes and textures. Only the visible boxes are rendered, according to the current transfer function.

Figures 5 and 1e are images lighted with *PPed* gradient textures using the Phong model. Like unilluminated rendering, the image quality is identical to those rendered from the unpartitioned textures. Figure 5a is rendered with normal illumination, while Figures 5b and 1e show images rendered with gradient magnitude modulation.

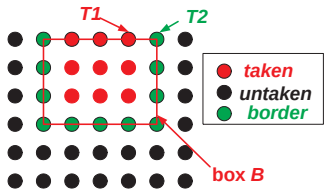
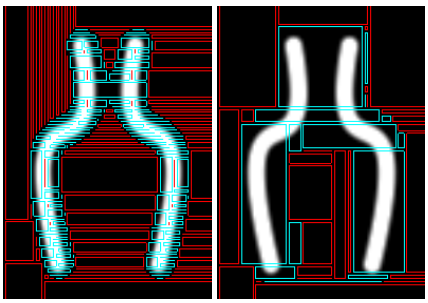


Figure 2: The alignment of a box with its enclosed texels.



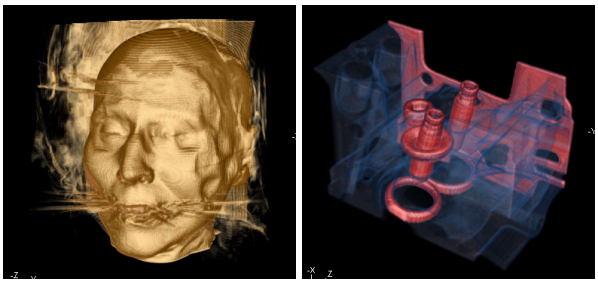
(a) (b)

Figure 3: (a) Without and (b) with discarding small and sliver boxes.

Table 2 shows the compression rates of the gradient textures. The original size (in MB) is the number of voxels times four, since each gradient (and its magnitude) requires four bytes. The columns under "Before packing" and "After packing" are the total size (in MB) of the partitioned sub-textures, including the replicated texels, before and after packing, respectively. Note that packing increases the requirement of texture memory by 10%. The mean value of the compression rate is 3.1:1.

Data Set	Original size	Before packing	After packing	Compre. rate
foot	34.2	12.6	14.9	2.3
neuron	78.6	11.1	12.2	6.4
head	59.0	29.5	33.9	1.7
engine	28.8	12.8	14.6	2.0

Table 2: Compression rates of the gradient textures



(a) *Head.* (b) *Engine.*

Figure 5: Datasets rendered with Phong lighting.

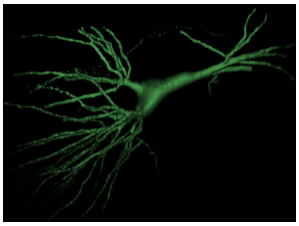


Figure 4: Dataset rendered without illumination.

Table 3 demonstrates the system performance on illuminated rendering. "Basic" denotes the rendering speed using unpartitioned textures for either normal illumination or gradient magnitude modulation, since the two run at the same speed for unpartitioned textures. "Illum." and "Mod." refer to normal illumination and gradient modulation based on the *PPed* textures. The average speedup factors for normal illu-

Data set	Basic FPS	Illum. FPS	Mod. FPS	Illum. speedup	Mod. speedup
foot	12.0	56.5	68.5	4.7	5.1
neuron	10.2	42.6	56.5	4.2	5.5
head	3.1	23.9	24.0	7.7	7.7
engine	12.4	19.0	24.5	1.5	2.0

Table 3: Rendering Speed (frames/sec) with unpartitioned textures (Basic), and PPed *textures in normal illumination (Illum.) and gradient magnitude modulation (Mod.).*

mination and gradient modulation are 4.5 and 5.1 on the GeForce 4 for the four datasets. Remember that with *PPed* textures, gradient modulation never renders more boxes than normal illumination for the same transfer function (see Section 5), and we use the same number of register combiner stages for the two lighting modes. Consequently, the frame rates and the speedup factors of the gradient modulation are greater than or equal to those of the normal illumination mode. In gradient modulation mode, the ratio of the number of texels rendered with *PPed* textures versus unpartitioned textures equals approximately to the compression rate. However, the speedup factors are always greater than the corresponding compression rates, since fewer textures improve cache performance. In some cases, such as for the head dataset, texture compression enables the texture memory to hold all the *PPed* textures while it can't for the unpartitioned texture, hence the acceleration rate is significantly larger.

7 Conclusion

We propose texture partitioning and packing as a lossless texture compression which is suitable for applications based on graphics hardware. We propose box growing to efficiently divide the texture domain into a set of boxes. The sub-textures defined by the boxes are then packed with a greedy algorithm. With our technique, we have achieved average speedup factors ranging from 3 to 6 for various datasets at different rendering mode. The partitioning and packing are independent on the transfer function.

In texture partitioning, allowing sub-textures to rotate or shear produces smaller boxes. Besides, the sub-textures for packing do not need to have the same resolution. In either case, the compression becomes lossy. In the future, we will attempt to improve the *PPed* texture with such lossy compression.

Acknowledgments

This work has been supported by ONR grant N000140110034 and NIH grant CA82402, and CAT Biotechnology grant. The datasets are courtesy of National Library of Medicine Visible Human, Center for Visual Computing of Stony Brook University, and UNC.

References

[1] R. Avila, L. Sobierajski, and A. Kaufman. Towards a Comprehensive Volume Visualization System. *IEEE Visualization*, pages 13–20, 1992.

[2] I. Boada, I. Navazo, and R. Scopigno. Multiresolution Volume Visualization with a Texture-Based Octree. *The Visual Computer*, 17(3):185–197, 2001.

[3] D. Cohen and Z. Sheffer. Proximity clouds, an acceleration technique for 3D grid traversal. *The Visual Computer*, 11(1):27–28, 1994.

[4] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. *Workshop on Graphics Hardware*, pages 9–17, 2001.

[5] S. Guthe, S. Roettger, A. Schieber, W. Strasser, and T. Ertl. High-quality unstructured volume rendering on the pc platform. *Workshop on Graphics Hardware*, pages 53–60, 2002.

[6] M. Kraus and T. Ertl. Adaptive Texture Maps. *Workshop on Graphics Hardware*, pages 7–15, 2002.

[7] E. LaMar, B. Hamann, and K. Joy. Multiresolution techniques for interactive texture-based volume visualization. *IEEE Visualization*, pages 355–362, October 1999.

[8] W. Li and A. Kaufman. Accelerating volume rendering with texture hulls. *Symposium on Volume Visualization and Graphics*, pages 115–122, October 2002.

[9] M. Meißner, S. Guthe, and W. Straßer. Interactive Lighting Models and Pre-Integration for Volume Rendering on PC Graphics Accelerators. *Graphics Interface*, pages 209–218, May 2002.

[10] J. Orchard and T. Möller. Accelerated splatting using a 3D adjacency data structure. *Graphics Interface*, pages 191–200, June 2001.

[11] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. *Workshop on Graphics Hardware*, pages 109–118, August 2000.