Volume Graphics (2005)
E. Gröller, I. Fujishiro (Editors)

GPU-based Object-Order Ray-Casting for Large Datasets

Wei Hong, Feng Qiu, and Arie Kaufman †

Center for Visual Computing (CVC) and Department of Computer Science
Stony Brook University
Stony Brook, NY 11794-4400, USA

Abstract

We propose a GPU-based object-order ray-casting algorithm for the rendering of large volumetric datasets, such as the Visible Human CT datasets. A volumetric dataset is decomposed into small sub-volumes, which are then organized using a min-max octree structure. The small sub-volumes are stored in the leaf nodes of the min-max octree, which are also called cells*. The cells are classified using a transfer function, and the visible cells are then loaded into the video memory or the AGP memory. The cells are sorted and projected onto the image plane front to back. The cell projection is implemented using a volumetric ray-casting algorithm on the GPU. In order to make the cell projection more efficient, we devise a propagation method to sort cells into layers. The cells within the same layer are projected at the same time. We demonstrate the efficiency of our algorithm using the Visible Human datasets and a segmented photographic brain dataset on commodity PCs.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation I.3.7Computer GraphicsThree-Dimensional Graphics and Realism

1. Introduction

High resolution CT data is highly demanded by many current medical applications. The typical size of contemporary clinical 16bit CT data is about 256MBytes (512^3*voxels*). The photographic volumetric datasets have color information, which are usually larger than the CT and MRI datasets of the same resolution. Moreover, the size of datasets will likely keep increasing at a high rate due to the advance of scientific devices. The rendering of large volumetric datasets is a classical problem in visualization.

Algorithms for direct volume rendering generally fall into two categories: image-order algorithms (e.g. ray-casting [Lev90]) and object-order algorithms (e.g. splatting [Wes90] or shear-warp [LL94]). The ray-casting algorithm can produce high quality images, and can achieve an interactive rendering speed using graphics hardware. Unfortunately, it imposes limits on the size of volumetric datasets that we can render at adequate update rates, because most state-of-the-art graphics cards only have 256 MBytes video memory. Real-time rendering of large datasets larger than 256 MBytes using the image-order algorithm is currently infeasible unless super-computers [PSL*98, KMM*01] or PC clusters [MOM*01, SMW*04] are used.

Volumetric datasets used in a variety of fields usually contain many regions that are classified as transparent or empty. The object-order approaches are well-suited for skipping empty regions, but usually the associated filters are too complex to be used for interactive rendering. And the hidden volume removal is also inefficient compared with the ray-casting method. Mora et al. [MJC02] proposed a CPU-based object-order ray-casting algorithm to take the advantages of both image-order and object-order approaches for interactive high-quality volume rendering. However, the cell projection implemented in this method can be efficiently performed only in orthogonal projection.

This paper presents a GPU-based approach for rendering large volumetric datasets on the commodity computers. A volumetric dataset is decomposed into small sub-volumes called *cells*, which are organized using a min-max octree structure. The cells are classified as empty cells or non-empty cells. The non-empty cells are loaded into the video

† Email: {weihong|qfeng|ari}@cs.sunysb.edu

memory or the AGP memory as more as possible. Then, the cells are projected from front to back and composited using the GPU. In order to make the cell projection more efficient, we devise a propagation method to sort the cells into layers such that all cells in one layer can be projected simultaneously.

The reminder of the paper is structured as follows: in the next section, we will briefly review related work. Then, we will describe the overview of our object-order ray-casting algorithm in Section 3. In Section 4 and 5, we will explain the detail of the cell projection and sorting algorithm respectively. The implementation and results are discussed in Section 6, and the paper concludes in Section 7 with some ideas for future work.

2. Related Work

Parker et al. [PSL*98] showed that it is feasible to perform interactive iso-surface rendering of the full resolution Visible Woman dataset with brute-force ray tracing on an SGI Reality Monster. They achieved up to 20 fps when utilizing 128 processors. Kniss et al. [KMM*01] presented a hybrid volume renderer, which can render full resolution time-varying dataset, such as the Raleigh-Taylor fluid flow dataset, at nearly 5 fps on a 128-CPU, 16-pipe SGI Origin 2000 with IR-2 graphics hardware. Muraki et al. [MOM*01] presented a scalable PC cluster system designed specially for simultaneous volumetric computation and visualization, using compositing hardware devices and the latest PC graphics accelerators. Strengert et al. [SMW*04] described a system for the texture-based direct volume rendering of large datasets on a PC cluster equipped with GPUs. Hierarchical wavelet compression is applied to increase the effective size of volumes that can be handled. However, these large scale solutions do not fit in the needs and capacities of an ordinary medical environment. Hence, many approaches have been devised on PCs.

Ghosh et al. [GPKM03] presented a multi-board schemes for rendering large volumetric datasets interactively. They implemented an image-partitioned rendering method by loading the entire volume on all available boards, but restricting the range of the image and depth buffers to be filled by each board. They achieved 24 fps for rendering the Visible Male on one PC with four VolumePro 1000 boards [PHK*99]. The limitation of this method is that the size of the volumetric dataset that can be rendered is limited by the memory size of the single board.

Guthe et al. [GWGS02] presented a method for rendering large datasets at interactive frame rates on standard PC hardware. The volumetric dataset is converted into a compressed hierarchical wavelet representation in a preprocessing step. During rendering, the wavelet representation is decompressed on-the-fly and rendered using texture mapping hardware. The level of detail used for rendering is adapted to the local frequency spectrum of the dataset and its position relative to the viewer. However, the wavelet compression method degrades the performance and quality of the rendering results. We use an object-order processing method to skip the empty region and graphics hardware to remove the hidden volume, which is efficient without degrading the quality of the rendering.

Grimm et al. [GBKG04] presented a CPU-based volume ray-casting approach based on image-ordered ray-casting with object-ordered processing. They introduced a memory efficient acceleration technique for on-the-fly gradient estimation and a memory efficient hybrid removal and skipping technique of transparent regions. Their method is also limited to orthogonal projection. In contrast to this method, our approach is a hybrid solution, which also supports perspective projection. Moreover, we devised a sorting algorithm to make our object-order ray-casting algorithm more efficient.

3. Algorithm Overview

In our object-order ray-casting approach, we define a cell as a cubical region which corresponds to a sub-volume containing $N \times N \times N$ voxels. A cell is classified as empty, if all voxels of the cell are invisible based on the transfer function. Otherwise, it is classified as non-empty. The min-max ocree [WG92] is used to organize the cells for efficient classification. Each leaf node of the min-max octree contains a cell, as well as the minimum and maximum density values of the cell. Each interior node only contains the minimum and maximum density values found in that node's subtree.

In stead of projecting a reconstruction kernel for each voxel onto the image plane as in the splatting technique, we project the whole cell onto the image plane. Moreover, we use a fragment program to do ray integration for the projected cell on-the-fly, in which a volumetric ray-casting algorithm is performed. For each cell, we need to store its corresponding voxels in a 3D texture. Since the volumetric ray-casting algorithm requires a neighborhood of voxels for proper interpolations and gradient calculations, the neighboring voxels of the cell need to be stored in the 3D texture. Thus, for each cell the resolution of the corresponding 3D texture is $(N+2) \times (N+2) \times (N+2)$.

In order to obtain correct compositing result, we must first determine the visibility order of the cells so that the cells can be projected from front to back. Although the cells can be hierarchically sorted using the min-max octree structure, we devised a more efficient propagation algorithm to sort cells. As a result, the cells are front-to-back sorted and grouped into layers. The cells within the same layer can be projected simultaneously, which dramatically improves the performance of our cell projection algorithm on the GPU. Our cell sorting algorithm and cell projection algorithm can take the advantage of the parallelism between the CPU and the GPU. Thus, when a layer of cells are determined, they

can be projected immediately to trigger fragment programs to be executed on the GPU. The CPU then can be used to generate the next layer of cells.

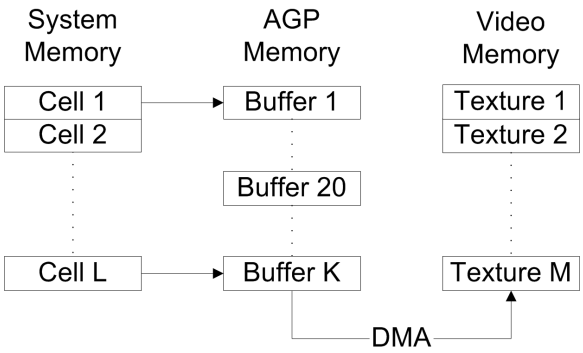


Figure 1: *The three-layer structure used to store the cell data.*

Although a large number of cells are classified as empty cells, which do not need to be uploaded to the GPU, the 3D textures corresponding to the non-empty cells are still too large to be fitted in the video memory. We need to transfer some non-empty cells to the video memory on-the-fly. The OpenGL extension pixel_buffer_object (PBO) defines an interface to using buffer objects for pixel data, which dramatically improves the texture uploading performance. By using this extension, the GPU can asynchronously pull the data from the AGP memory using DMA (Direct Memory Access). Thus, we use a three-level structure to store the cell data in the video memory, the AGP memory, and the system memory as shown in Figure 1. Suppose that we can allocate M 3D textures in the video memory and N buffers of the same size in the AGP memory, and the first 20 buffers are used as a memory pool for transferring data on-the-fly. We first randomly choose N non-empty cells and upload them into the video memory. We then copy the other M-20 non-empty cells into the AGP buffers. The rest of non-empty cells are still resident in the system memory. For each cell, we use a flag to indicate whether its corresponding data is resident in the video memory, the AGP memory, or the system memory. Thus, the size of the dataset that can be rendered by our algorithm is limited by the size of the system memory.

The overview of the proposed algorithm is shown in Figure 2. The min-max octree construction, classification, and texture loading are performed in the pre-processing step, which are view independent. Our cell sorting algorithm organizes cells into layers. When a layer of cells are generated, we first check whether all non-empty cells are resident in the video memory. If any non-empty cell within the layer is not resident in the video memory, we need to upload it on-the-fly. Before we can transfer the data, we must determine which 3D texture object is used to receive the data. In

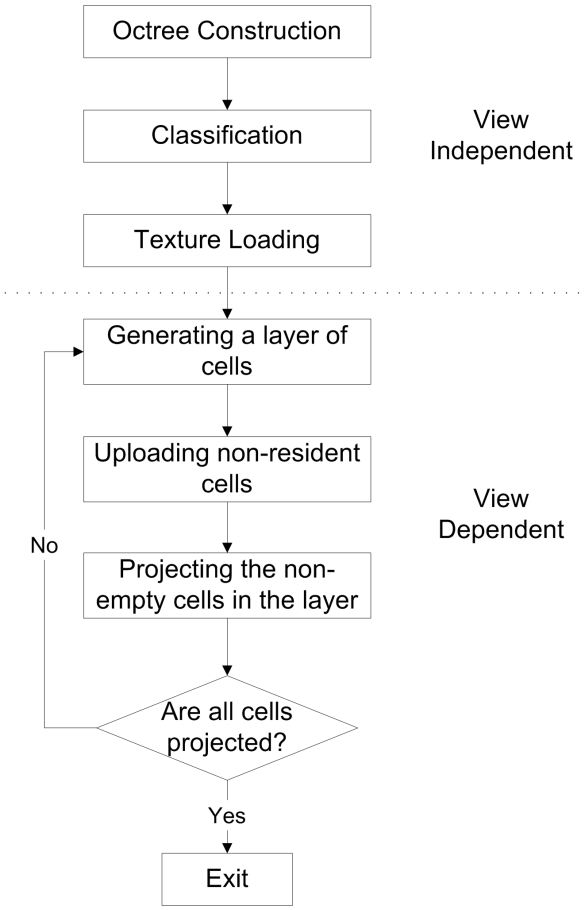


Figure 2: *The overview of our GPU-based object-order ray-casting algorithm.*

other words, the current data stored in that 3D texture is replaced by the new data. We use a replacement queue to hold the cells that are already projected and can be switched out. When a layer of cells are sent to the GPU, we can not put them into the replacement queue immediately. Because we do not know whether the corresponding fragment programs executed on the GPU are finished or not. We use a Nvidia OpenGL extension NV_fence to determine whether the cell projection of a layer of cells is finished on the GPU. This extension introduces the concept of a "fence" to the OpenGL command stream. Once the fence is inserted into the command stream, it can be queried whether it is finished. After all OpenGL commands for cell projection of the layer of cells are issued, we insert a fence into the commands. Then, we query the fence's state after every layer of cells are projected. If the fence is completed, the cells before the fence are inserted into the replacement queue, and a new fence is inserted into the OpenGL commands stream again. In case the replacement queue is empty, we randomly choose a 3D

texture whose corresponding cell has not been projected to receive the data. We will discuss the detail of the cell projection algorithm and cell sorting algorithm in the following sections.

4. Cell Projection

When the orthogonal projection is used, every cell projection on the image plane is given by the same hexagon shape per viewing direction. This projection can be computed once, and then used as a template for all cells, which can be obtained by translation. The rays intersecting with the cell are then determined by the cell projection efficiently. However, when the perspective projection is applied, the situation becomes more complicated. The cell projections on the image plane are different, and the pre-computed template can not be used any more, which make the CPU-based object-order ray-casting algorithm infeasible. The good thing is that the cell projection can be efficiently implemented on the recent graphics card even when a perspective projection is used, which makes it possible to implement an object-order ray-casting algorithm on the GPU.

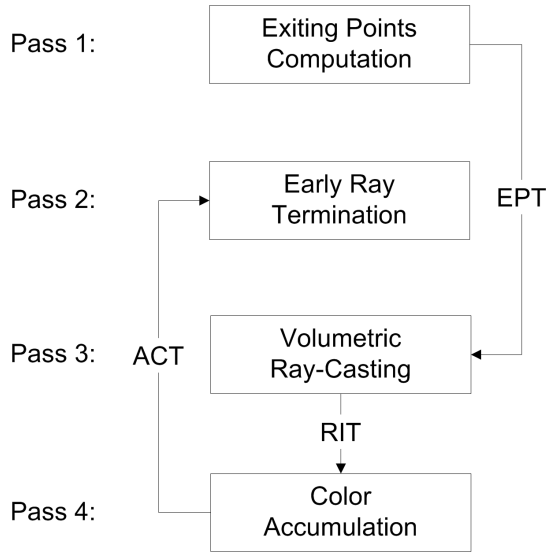


Figure 3: *The pipeline of the cell projection.*

Our cell projection algorithm is implemented using fragment programs running on the GPU. When a cell is rendered, a number of fragments are generated, which are correspondent to the rays intersecting with that cell. For every non-empty cell that has to be projected, the rendering pipeline is shown in Figure 3. The proposed algorithm consists of four rendering passes for each cell. The modelview matrix and projection matrix remain unchanged for all four rendering passes. Hence, the fragments generated at the same window position in the four rendering passes correspond to the same ray intersecting with the cell.

OpenGL provides pixel buffers (pbuffer for short) for off-screen rendering. Combined with the render_texture extension, it allows the color buffer of the pbuffer to be used for both rendering and texturing. Three pbuffers are used as rendering targets for different render passes in our algorithm. The first pbuffer, the rendering target of the first rendering pass, is used to store the exiting points of the rays that intersect with the projected cell. It is also bounded to a 2D RGB floating point texture, named *exiting points texture* (EPT), which is accessed in the third rendering pass to compute the length of each ray segment and normalized ray direction. The second pbuffer is made up of a depth buffer and a color buffer, which are the rendering targets of the second and third rendering pass, respectively. The depth buffer is used to implement early ray termination with the early-z test technique described in [KW03]. This optimization happens only if the fragment program is not going to modify the fragment's depth. However, we need to modify the depth values based on the opacity values. We thereby use a separate rendering pass to modify the depth values. The color buffer is used to store the result of the ray integration, which is bound to a 2D RGBA floating point texture, named *ray integration texture* (RIT) and accessed in the last rendering pass. The third pbuffer is the rendering target of the last rendering pass, which is used to accumulate the color values. It is bound to a 2D RGBA floating point texture in the second rendering pass, which is named *color accumulation texture* (CAT). Its opacity values are accessed in the second rendering pass to modify the depth values accordingly for culling the fragments whose corresponding rays have already saturated their opacity values. The cell projection algorithm is described as follows:

- **Pass 1** (Exiting Points Computation): In the first rendering pass, the exiting points for the rays intersecting with the projected cell are computed by only rendering the back faces of the cell. For each vertex of the cell, we assign its texture coordinates in the corresponding 3D texture space as its primary color. The fragment program is straightforward, which just passes the fragment's primary color as output. As a result, we obtain a texture coordinate for each fragment, which is the coordinate for the exiting point of the ray in the texture space.
- **Pass 2** (Early Ray Termination): The opacity value of the fragment is accessed through the color accumulation texture (CAT). For any fragment whose opacity value exceeds 0.99, the depth value is set to one. As a consequence, if the depth test is set to GREATER, the corresponding fragment in the third rendering pass is discard.
- **Pass 3** (Volumetric Ray-Casting): The front faces of the cell are rendered to compute the entry points for the rays using the same method as Pass 1. In the fragment program, the exiting point is obtained through accessing the exiting point texture (EPT). The normalized ray direction and length of ray segment are computed in the 3D texture space. The ray is then evenly sampled with a sampling

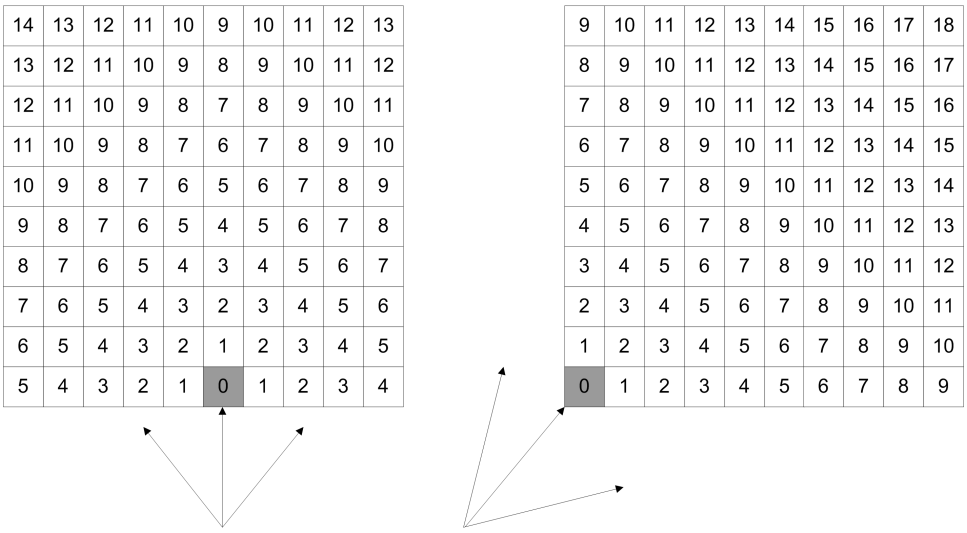


Figure 4: *A layer of cells have the same Manhattan distance and can be projected together.*

distance 0.5 to do ray integration. It is impossible to precompute the gradient information and store them on the GPU for large datasets. Thus, we estimate the gradient on each sampling point on the fly. We use texture lookup to obtain the density at six neighboring positions, then estimate the gradient using central difference.

- **Pass 4** (Color Accumulation): The front faces of the cell are rendered again to generate corresponding fragments. In the fragment program, the color value and opacity value of the projected cell are accessed through ray integration texture (RIT), and returned as color output directly. OpenGL blending is enabled in this rendering pass for accumulating the color and opacity values.

When all non-empty cells are projected, the color accumulation texture (CAT) holds the final image. In fact, the rendering Pass 2 is not need to be executed for every layer. In our implementation, we enable the rendering Pass 2 every other two layers.

Because the rendering context are switched three times during the cell projection, this may cause a significant loss in performance on current GPUs. In order to decrease the number of rendering context switching, we need to project more cells in each rendering pass to improve the performance of the cell projection. Thus, we devise a cell sorting algorithm, which allows us to project a layer of cells each pass.

5. Cell Sorting

For a given viewing direction vector in the octree coordinate system, the signs of the coordinates determine the order in which the eight children are visited when parallel projection is used. When perspective projection is used, visibility order of the eight children can still be determined by the location of the camera to the octree. However, the octree structure only allows us to project at most four cells in one pass for some viewing directions. We need to project cells as more as possible to decrease the number of rendering context switching.

The main idea of our algorithm is to divide the cells into layers. We only need to determine the visibility order of layers. The cells within the same layer can be projected at the same time. It is observed that the cells that have the same distance to the camera can be projected together. However, using the Euclidean distance from the cells to the camera to do the cell sorting is inefficient. In order to improve the performance, we use the Manhattan distance instead of the Euclidean distance. Moreover, we use the Manhattan distance between a source cell and the other cells to group the cells into layers. A source cell is determined first for a given view point, which is the closet cell to the camera. We then use a propagation method to compute the Manhattan distance for the other cells. The cells that have the same Manhattan distance to the source cell are put into the same layer. We first describe our cell sorting algorithm in the 2D case, and then extend it to 3D.

In the 2D case, the whole object can be represented with a square, and the camera can be set up around the square. We first find the closest cell to the camera based on the camera's location with respect to the square. If the camera is located at the corner region as shown in the right image of Figure 4, the closest cell is the corresponding corner cell shown in grey. Otherwise, the closest cell is on the edge of the square that is opposite to the camera as shown in the left image of Figure 4. The closest cell can be obtained by shooting a ray perpendicular to the edge. The intersected cell is the closest cell. If the ray intersects two cells, the two cells are both

used as source cell. In Figure 4, we shows the Manhattan distance of each cell. It is observed that the cells show a very clear layer structure. It is also noted that each layer consists of more cells by using the Manhattan distance than using the Euclidean distance. In fact, we do not need to explicitly compute the Manhattan distance. From the Figure 4, we can see that the source cell is made up of the first layer. And, the second layer consists of the edge neighboring cells of the source cell. Thus, we can use a propagation method to group the cells into layers from the source cell C_0. The propagation algorithm is described as follows:

1. Let $C_0.visited = 1$ and put C_0 into a list L_0. Set the other cells to be un-visited.
2. For each cell C_i in the list L_0
   a. Obtain the four edge neighboring cells $C_{ij}(j = 0,1,2,and3)$ of C_i. If $C_{ij}.visited$ is 0, let $C_{ij}.visited = 1$ and put C_{ij} into the list L_1.
3. Project the non-empty cells of L_0. If all non-empty cells are projected, the algorithm is terminated.
4. Copy L_1 to L_0, and goto 2.

By using this sorting algorithm, each layer of the cells have the same Manhattan distance to the source cell. The cells within the same layer does not occlude with each other, which can be projected at the same time. This algorithm can be easily extended to the 3D case. In the 3D case, the closest cell still can be find efficiently based on the region where the camera is located with respect to the volumetric dataset. The propagation process is almost same, except that we need to use the six face neighboring cells for propagation in the 3D case.

If the camera is located inside the dataset, the sorting algorithm becomes even simpler. The source cell is right the cell where the camera is located. Moreover, we only need to propagate the order information along with the viewing direction of the camera. The cell projection of the starting cell is implemented a little different from the of other cells. Only one rendering pass is needed to implement the projection of the starting cell. The rendering target is the third pbuffer used for color accumulation. The back faces of the starting cells are rendered to trigger the fragment program, which also give the exiting points of the corresponding rays. The camera position is passed to the fragment program as an uniform parameter. The ray direction is computed by using the exiting points and camera position. Then, the ray is evenly sampled to do ray integration from the camera position. Thus, our algorithm can be used for the fly-through applications, such as virtual colonoscopy.

6. Implementation and Results

In this section, we present some implementation details and testing results. The presented algorithm is implemented using C/C++, and fragment programs are implemented using Cg [MGAK03]. The experiments have been conducted on a 3.0GHz Intel Pentium IV PC, with 2G RAM and a NVIDIA Quadro FX 3400 graphics card. We list the information of the datasets used in our experiments in Table 1.

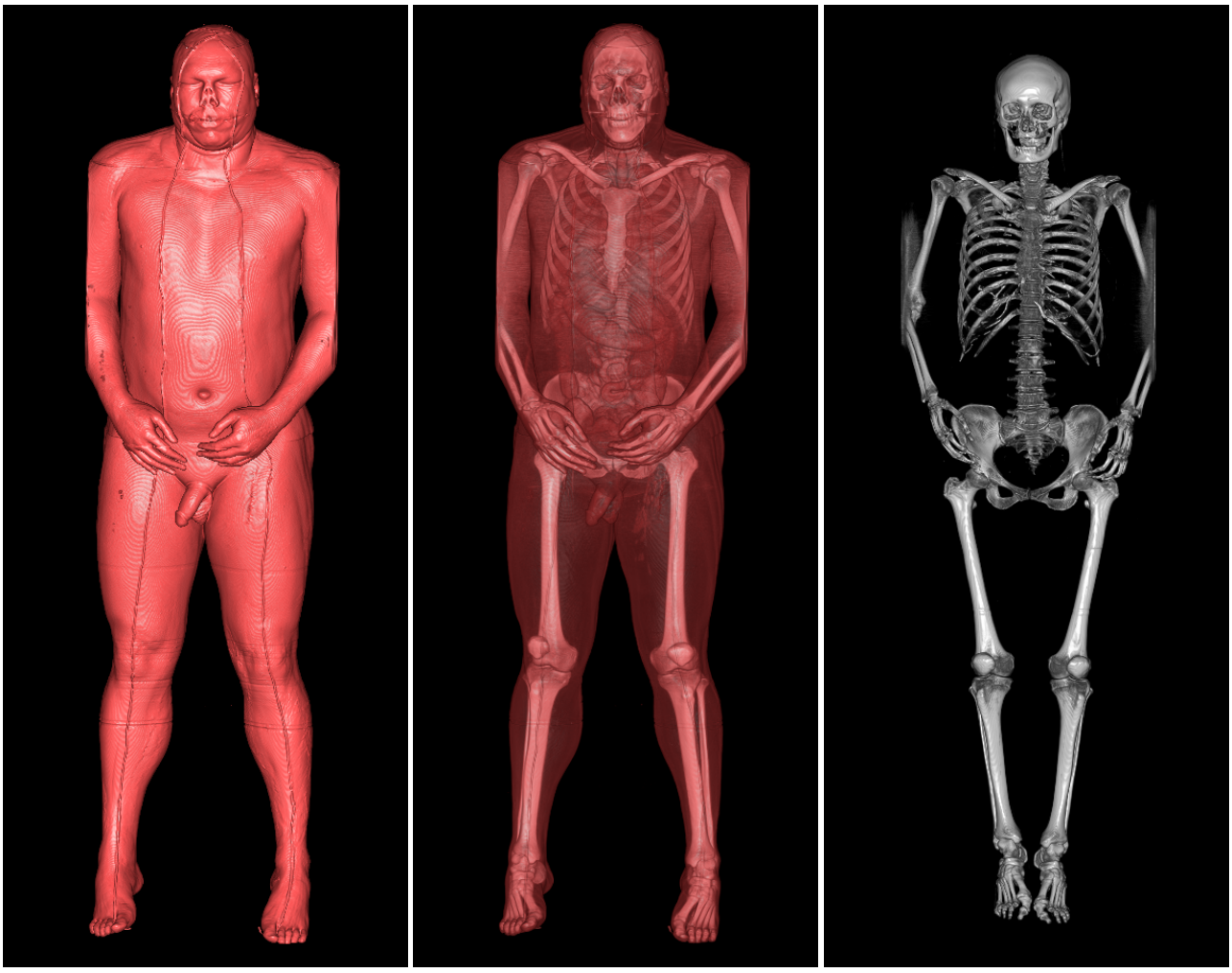
The size N of the cell is crucial to our algorithm. A smaller N is efficient for empty space skipping, but inefficient for the cell projection executed on the GPU. Because using a smaller N will increase the number of rendering context switching, which decreases the performance. And, it also increase the number of texture objects switching because our cells are stored in separate 3D textures. A smaller N will result in the projection of the cell covering less pixels on the image plane, which degrade the efficiency of the volumetric ray casting because of the poor caching. Moreover, for each cell normalized ray direction and length of the ray segment are needed to be computed for the rays intersecting with that cell. A larger N can decrease such computation. We choose $N = 64$ in our implementation for the purpose of the trade-off between the empty space skipping and the cell projection on the GPU.

Table 1: *The information of the datasets used in the experiment.*

Dataset	Dimension	Size
Visible Male	$512 \times 512 \times 1887$	0.71GB
Visible Female	$512 \times 512 \times 1734$	0.65GB
Brain	$1080 \times 1110 \times 158$	0.93GB

We use the full resolution Visible Human CT datasets to test our algorithm. About a half of cells are skipped after the classification. Thus, most cells are fitted into the video memory and the AGP memory. Only a small number of cells are still resident in the system memory. We can achieve several frames per second for such large datasets on a commodity PC. We show some resulting images in Figure 5, which are all rendered at the resolution of 512×1024. In Figure 5(a), we show the skin of the Visible Male using an opaque transfer function. In Figure 5(b), we show the bone structure and some organs of the Visible Male using a semi-transparent transfer function. In Figure 5(c), the bone of the Visible Female is shown by using an opaque transfer function. It is natural that we achieve higher rendering speed when the opaque transfer functions are applied. Because more cells are skipped in the object-space and less cells need to be projected.

We also use a segmented photographic volumetric dataset to demonstrate the efficiency of our algorithm. Compared with the CT datasets, the volume rendering for photographic datasets requires an opacity transfer function from the non-linear color space, which is more complicated than that for the CT datasets. We use the CIE *Luv* color space to obtain



(a) 3.1 fps (b) 1.8 fps (c) 3.5 fps

Figure 5: *Visible Human CT datasets rendered using our algorithm.*

a perceptually uniform representation of the color volume, and assign an opacity value for each voxel using the method proposed by Ebert et al. [EMRY02]. Thus, each voxel of this dataset contains RGB color, opacity and segmentation information. In order to render segmented datasets, for each cell we store a list of labels that are used for labeling the voxels in the cell. When an organ is chosen for rendering, only the cells containing the corresponding label are loaded into the video memory for projection. These cells usually can be fitted into the video memory without on-the-fly transferring data, which allows us to interactively explore the segmented organs. In Figure 6, we show some resulting images rendered from the segmented brain dataset with a resolution of 512×512. A top view of the full resolution brain dataset is shown in Figure 6 (a). The segmented brain stem and ventricle can be rendered in real-time shown in Figure 6 (b) and (c), because all the related cells can be fitted in the video memory.

7. Conclusions and Future Work

We presented a GPU-based object-order ray-casting algorithm for rendering large volumetric datasets such as the Visible Human CT datasets. The volume dataset is decomposed into small cells, and organized using a min-max octree structure. The empty cells are skipped immediately after the classification. The volumetric ray-casting algorithm is performed on the GPU for each non-empty cell projection, and the resulting integration of the cell are front-to-back composited to generate the final image. We devised a cell sorting algorithm to allow us project a layer of cells at the same time, which improves the performance of the fragment pro-

(a) 1.5 fps (b) 21.8 fps (c) 23.1 fps

Figure 6: *Brain dataset of the Korean Visible Human rendered using our algorithm.*

grams on the GPU. The hidden volume is culled by using the assistance of the graphics hardware.

The main bottleneck of our algorithm is on-the-fly transferring data from the system memory to the video memory, although we have already used OpenGL pixel buffer object (PBO) extension to accelerate it. Moreover, the 3D textures belonging to the culled cells due to the opacity occlusion are still uploaded to the video memory, if they are not resident the video memory. We would like to study how to avoid this kind of data transferring to improve the performance further in our future work. In our current implementation, the cells are stored in the separate 3D textures. We also would like to pack the cells and store them in a single 3D textures to decrease the number of texture objects switching. The size of the dataset that can be rendered by our method is limited by the size of the system memory. We would like to extend our method to PC clusters to interactively render the datasets such as the full resolution photographic Visible Human datasets.

Acknowledgement

This work has been supported by an NSF grant CCR-0306438. The Visible Human datasets are courtesy of the National Library of Medicine. The Korean Visible Human dataset is courtesy of HuminTec Inc., Korea.

References

[EMRY02] EBERT D. S., MORRIS C. J., RHEINGANS P., YOO T. S.: Designing effective transfer functions for volume rendering from photographic volumes. *IEEE Transactions on Visualization and Computer graphics 8* (Apr. 2002), 183–197. 7

[GBKG04] GRIMM S., BRUCKNER S., KANITSAR A., GROLLER E.: Memory efficient acceleration structures and techniques for cpu-based volume raycasting of large data. *IEEE Symposium on Volume Visualization and Graphics* (Oct. 2004), 1–8. 2

[GPKM03] GHOSH A., PRABHU P., KAUFMAN A., MUELLER K.: Hardware assisted multichannel volume rendering. *Proceedings of the Computer Graphics International Conference* (July 2003), 2–7. 2

[GWGS02] GUTHE S., WAND M., GONSER J., STRABER W.: Interactive rendering of large volume data sets. *Proceedings of IEEE Visualization '02* (Oct. 2002), 53–60. 2

[KMM*01] KNISS J., MCCORMICK P., MCPHERSON A., AHRENS J., PAINTER J., KEAHEY A., HANSEN C.: Interactive texture-based volume rendering for large data sets. *IEEE Computer Graphics and Applications 21* (July 2001), 52–61. 1, 2

[KW03] KRUEGER J., WESTERMANN R.: Acceleration techniques for gpu-based volume rendering. *Proceedings of IEEE Visualization '03* (Oct. 2003), 38–46. 4

[Lev90] LEVOY M.: Efficient ray tracing of volume data. *ACM Transactions on Graphics 9*, 3 (July 1990), 245–261. 1

[LL94] LACROUTE P., LEVOY M.: Fast volume rendering using a shear–warp factorization of the viewing transformation. *Proceedings of SIGGRAPH '94* (July 1994), 451–458. 1

[MGAK03] MARK W. R., GLANVILLE R. S., AKELEY K., KILGARD M. J.: Cg: A system for programming graphics hardware in a c-like language. *Proceedings of SIGGRAPH '03* (2003), 896–907. 6

[MJC02] MORA B., JESSEL J. P., CAUBET R.: A new object-order ray-casting algorithm. *Proceedings of IEEE Visualization '02* (Oct. 2002), 203–210. 1

[MOM*01] MURAKI S., OGATA M., MA K.-L., KOSHIZUKA K., KAJIHARA K., LIU X., NAGANO Y., SHIMOKAWA K.: Next generation supercomputing using pc clusters with volume graphics hardware devices. *IEEE Supercomputing '01* (Nov. 2001), 51–58. 1, 2

[PHK*99] PFISTER H., HARDENBERGH J., KNITTEL J., LAUER H., SEILER L.: The volumepro real-time ray-casting system. *Proceedings of SIGGRAPH '99* (July 1999), 251–260. 2

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.-P.: Interactive ray tracing for isosurface rendering. *Proceedings of IEEE Visualization '98* (Oct. 1998), 233–238. 1, 2

[SMW*04] STRENGERT M., MAGALLON M., WEISKOPF D., GUTHE S., ERTL T.: Hierarchical visualization and compression of large volume datasets using gpu clusters. *Eurographics Symposium on Parallel Graphics and Visualization '04* (2004), 41–48. 1, 2

[Wes90] WESTOVER L.: Footprint evaluation for volume rendering. *Computer Graphics 24*, 4 (Aug. 1990), 367–376. 1

[WG92] WILHELMS J., GELDER A. V.: Octrees for faster isosurface generation. *ACM Transactions on Graphics 11* (July 1992), 201–227. 2