Parallel Multi-PC Volume Rendering System

Chandrajit Bajaj Sangmin Park Anthony Gene Thane
University of Texas at Austin

Abstract

We present a parallel multi-PC volume rendering system using off-the-shelf commodity components. We exploit the hardware accelerated 3D texture mapping of the GeForce3 graphics processors to visualize volume datasets. The system consists of two parts. One is client interface running on Windows and the other is the rendering server running on Linux. We also use the GeForce3 processor in the client interface to edit the transfer functions and color maps interactively using decimated datasets, which fit on a single machine. After the client interface requests high quality images from the rendering server, the rendering server uses multiple PCs to render the images within a second. Our implementation addresses the problem of the frame buffers lack of precision, which causes artifacts when sub-volumes are blended individually, then blended together.

1 Introduction

Direct volume rendering is one of the most important methods to visualize volume datasets like tomographic images and computer simulation data. As the performance and memory size of computers increase, the size of the volume data also increases at a comparably high rate. For example, the size of the visible human female dataset is around 40 gigabytes [Vis n. d.] and a computer simulation data size could be more than a couple of hundred gigabytes. The ability to visualize the enormous datasets is beyond the current capabilities of a single-pipe graphics machine. Parallel techniques must be applied to achieve interactive rendering of gigabyte scale datasets. If the data size is bigger than the total main memory of a parallel machine, then it can be visualized using out-of-core algorithms [Zhang et al. 2001; Chiang et al. 2001] at the expense of interactivity. Visualization is always accompanied by transfer functions and viewing parameters. Semi-automatic generation methods [Kindlmann and Durkin 1998; Kniss et al. 2001a] and the contour spectrum [Bajaj et al. 1997] have been developed for transfer function generation. Since there is no automatic method for transfer function generation and the two approaches still require fine-tuning by the user, large scale data visualization is time consuming work without interactivity.

In this paper, we suggest a 3D texture based parallel volume rendering system using a PC cluster with off-the-shelf components (COTS). The main goal of our research is interactive visualization of large volume datasets, which are beyond the capabilities of a single PC or a single-pipe graphics machine, using general-purpose PC graphics hardware, such as the GeForce 2 and GeForce3. The rendering system is an interactive direct volume rendering system, which is limited in the size of the data it can visualize by its available texture memory.

The VolumePro is a special purpose graphics card designed for interactive volume rendering [Pfister et al. 1999]. [Lombeyda et al. 2001] parallelized the VolumePro hardware with Sepia-2 blending hardware and achieved high performance. We suggest a similar approach to get interactive rendering performance using cheaper, general-purpose hardware. We also suggest a method to remove seam planes between rendered image pieces at each node in the final composition step. The seam planes are caused by the framebuffer's precision, which has only a byte for each components of red, green, blue and alpha(RGBA).

The rendering system consists of two parts. The first part is a client interface, which is running on Windows 2000. It adjusts transfer functions and view parameters interactively with decimated data, which fits into texture memory. It sends viewing information to the rendering server. The second part is a parallel 3D texture based rendering server on RedHat Linux Machines.

The rest of this paper is organized as follows: Section 2 describes the previous work. In section 3, we explain the parallel rendering algorithm and seam planes removing algorithm. In section 4, the performace and rendering results are presented. Finally, we make a conclusion in section 5.

2 Previous Work

Hardware accelerated, parallel volume rendering research can be divided into two groups. In the first group, parallel machines are used such as SGI and FUZION. Kniss et al. [Kniss et al. 2001b] developed texture-based volume rendering algorithm using 16-pipe SGI Origin 2000 with IR-2 graphics hardware. Sub-volumes are loaded into texture memory of each graphics pipe. Sub-images rendered by the graphics pipes are composed from back to front in software. [Park et al. 2001] suggested proportional brick assignment algorithm according to the number of raster managers of each graphics pipe. They used SGI Onyx2 machine with 6 InfiniteReality2 graphics pipes. [MeiBner et al. 2001] implemented a parallel ray casting algorithm using the FUZION architecture. In the second group, PC graphics hardware is used, such as Geforce and VolumePro. [Lombeyda et al. 2001] have achieved high performance using special-purpose volume rendering hardware, the VolumePro for volume rendering and a Sepi-2 board for image blending. They also proved that the front-to-back final composition formula of intermediate images, which are composed individually, is identical to the front-to-back composition method without breaking them up. [Magallon et al. 2001] used multiple Geforce2 graphics processors in parallel, which allow only 2D texture mapping.

However, nobody has considered the composition errors caused by the framebuffer's precision. To use the texture based parallel volume rendering, we must use the framebuffer of each node. The framebuffer's low precision causes roundoff errors that are most noticable when intermediate images are composited into the final image.

3 Parallel Algorithm

The rendering system consists of two parts, the Client Interface and the Rendering Server. The Client Interface is running on Windows2000, while the Rendering Server is running on RedHat Linux. Both of them use Geforce3 processors except the root node in figure 1. The root node uses a Geforce2 instead of the more expensive GeForce3, since 2D texture mapping is enough for the hardware accelerated final composition,

A very large dataset can be visualized in a single PC after it is decimated to fit into the texture memory of the PC. In that case, we

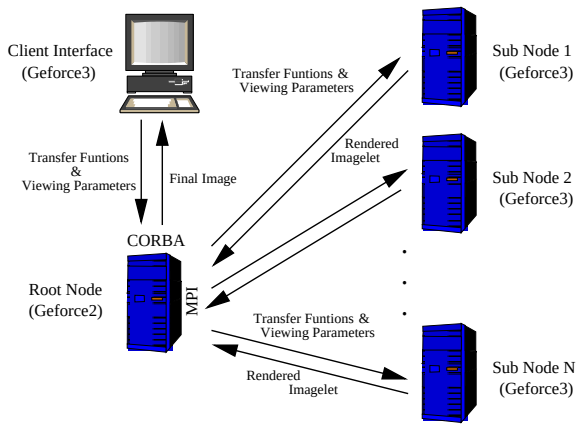


Figure 1: The Rendering System Diagram: The transfer functions and viewing parameters are edited using the Client Interface and sent to the rendering server through CORBA. The root node broadcasts the information to sub-nodes and get rendered imagelets from the sub-nodes. Finally, The images are composed into a final image by the root node and the final image is sent to the Client Interface.

can get interactive visualization, but we lose data accuracy. Transfer functions and viewing parameters can be edited using the decimated data interactively with the Client Interface. After the transfer functions and viewing parameters are sent to the rendering server, the server generates accurate images. The rendering system provides interactivity and accuracy at the same time for large dataset while the cost of the server is minimized.

3.1 Parallel Rendering Algorithm

There are three steps to render images. First, the Client Interface loads decimated data. The transfer functions and viewing parameters are interactively edited using the Client Interface which uses a Geforce3 for interactive volume rendering. There is some trade-off between the image quality and interactivity in the Interface. In the second step, the Client Interface requests a high-resolution images from the rendering server. If the Client Interface requests the first execution from the rendering server, then each node of the server start to load sub-volume, which size should be less than or equal to the texture memory size of each node. Right after the sub-volume data are downloaded into texture memory, each node renders it's sub-volume. Rendered images are read from framebuffer and sent to the root node. The root node composes the images using hardware-accelerated 2D texture mapping. Finally, the server reads the composed image from framebuffer and sends it to the Client Interface. The Client Interface displays the two images, low and high resolution at the same time. The figure 2 shows the data flow of the rendering system. The processor 1 through 4 have Geforce3 processors and the server has a Geforce2 processor. The root node calculates view direction and position and decides the composition order of the rendered images taken from the sub-nodes. There are only 2 ways to compose the images, from the number 1 image to the number k image or from the number k image to the number 1 image.

The hardware-accelerated composition is faster than the software composition, but it has a serious problem. Since each component of RGBA has only 1 byte in the framebuffer, each component can cover only from 0 through 255 values. When the alpha value (0-1) is less than 1/255, nothing would be accumulated on the framebuffer, since the framebuffer tries to omit decimals. Even though tons of planes are drawn to the framebuffer, we can see only back-

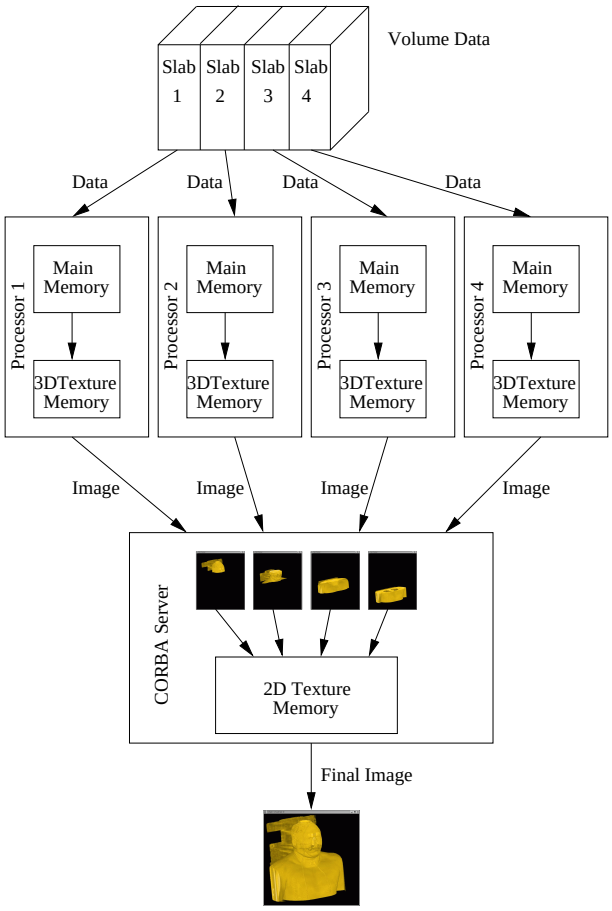


Figure 2: The Parallel Algorithm of the Rendering Server: In the first excecution, the each node of the rendering server reads the sub-volume data and downloads them to the texture memory. After generating sub-image, each node sends it to the root node. The root node puts them into 2D texture memory and blends them into a final image.

ground color. In the next section, we suggest a technique to reduce the blending errors. The main idea of our method is to increase the alpha values to compensate the blending errors, while decreasing the number of sampling planes. The next section explains the details of the mathematical calculation.

3.2 Removing Seam Planes Algorithm

There are two methods to blend texture-mapped planes into an image. One is back-to-front composition method and the other is front-to-back composition method which are used in [Kniss et al. 2001b] and [Lombeyda et al. 2001] respectively. The following equations (1) are used for back-to-front composition [Porter and Duff 1984] :

$$\begin{aligned} C_d &= \alpha_s C_s + (1-\alpha_s) C_d \\ \alpha_d &= \alpha_s + (1-\alpha_s)\alpha_d \end{aligned} \quad (1)$$

where C_d and α_d are the accumulated color and alpha value in framebuffer respectively and C_s and α_s are the source color and alpha value composed into the framebuffer respectively.

The next equations (2) are for front-to-back composition.

$$\begin{aligned} C_d &= C_d + (1-\alpha_d)\alpha_s C_s \\ \alpha_d &= \alpha_d + (1-\alpha_d)\alpha_s \end{aligned} \quad (2)$$

Both of them can be used for intermediate image blending method in each sub-node. In the case of back-to-front composition, we can use the same function (1) for the final image blending, but in the case of the front-to-back composition, the following formula should be used :

$$\begin{aligned} C_d &= C_d + (1-\alpha_d)C_s \\ \alpha_d &= \alpha_d + (1-\alpha_d)\alpha_s \end{aligned} \quad (3)$$

We use the back-to-front composition method with pre-multiplied $\alpha_s \times C_s$ values, since it does not need the destination alpha values and the front-to-back composition method is more susceptible to roundoff errors. If $\alpha_s \times C_s$ is less than $1/255$ in (1), where each value of α_s and $C_s$is from 0 to 1, then nothing will be accumulated in framebuffer, even though very many planes are blended. For example, if C_d is initialized with black color, 0 and $\alpha_s = 0.35$ and $C_s = 0.01$, then $C_d = \alpha_s \times C_s = 0.0035$, which will be 0 after it is accumulated into framebuffer, which has only 1 byte for each component of RGBA values. After the blending is repeated 450 times, we should get $C_d = 0.3462$ and $\alpha_s = 0.9891$. However, if they are accumulated into the framebuffer, which has only 1 byte precision, then the values will be $C_d = 0.0/255$ and $\alpha_s = 155/255$. We will see only black background color. One or two colors among red, green and blue(RGB) could disappear. The figure 7(a) shows the problem. Unexpected planes are visible in the final image after the final blending. The same problem happens in front-to-back blending formula (2).

There are a couple of solutions to remove the seam planes. Software high precision blending is left as a resort because of it's low performace. First, one solution is to increase the precision of the framebuffer. However it is not possible on the GeForce3. Second, the seam planes are generated due to the framebuffer's precision, which would omit decimals. We can blend texture mapped planes twice using the scaled trimed numbers. After rescaling the image and we can add it to the final image. This method will cut the blending performance in half, since it blends twice. Finally, we suggest the following method: We can increase the alpha values to reduce the blending errors, while the number of texture mapping planes decreases. The blending errors usually happen when the $\alpha_s \times C_s$ value is too small. We do not change color values, C_s, to recover the rounding off errors, since the rendering server should give the same colored images to the client. The following formula represents the relationship between α_s and the number of mapping planes [Kniss et al. 2001b]:

$$\alpha_{new} = 1 - (1-\alpha_{old})^{\frac{sr_{old}}{sr_{new}}} \quad (4)$$

where sr_{old} is the sample rate used with α_{old} and sr_{new} is the new sample rate used with α_{new}. When α_s is increased, the number of mapping planes should be decreased nonlinearly. After we decided the new α_s value, the number of planes can be calculated using (5).

$$sr_{new} = sr_{old} \times \frac{log(1-\alpha_{old})}{log(1-\alpha_{new})} \quad (5)$$

However, it is not easy to decide the new α_s value to recover the rounding off errors when integer calculation is used, since α_s and C_s are not constant and round-off errors happen at every blending step. Fortunately, the weird final blending of 7(a) happen, only when we use nearly transparent planes in most cases. So, we can make assumption that α_s is small values and C_s is a constant to make it easy to calculate. The C_d is initialized with black color, 0, so after the first blending, we can get C_d like following:

$$C_d = \alpha_s \times C_s + (1-\alpha_s) \times \alpha_s \times C_s \quad (6)$$

C_d should be greater than $1/255$ to prevent drawing nothing. α_s should be between the two values:

$$\left(1 - \sqrt{1 - \frac{1}{255C_s}},\ \ 1 + \sqrt{1 - \frac{1}{255C_s}}\right) \quad (7)$$

(7) is a restriction for the new α_s value. After blending a enough number of planes using real numbers, C_d are equal to C_s in (1). The following equation is true at some point:

$$\alpha_s C_s = \alpha_s C_d. \quad (8)$$

However, if we use the integer precision, then the C_d value is less then the C_s value. For example, if $\alpha_s = 0.04$ and $C_s = 0.35$, then $(\alpha_s \times C_s \times 255) = 3.57$. After we omit decimals, the $(\alpha_s \times C_s \times 255)$ value will be 3. In our program, since we use pre-multiplied $(\alpha_s \times C_s)$ values, the C_d values of (8) is less than what it should be. If the α_s value is increased by 0.01, then $(\alpha_s \times C_s \times 255) = 4.4625$. Integer value of $(\alpha_s \times C_s \times 255)$ is 4, which is bigger than 3.57, but more than 0.43 values are trimed by the low precision blending. Here is the C like algorithm to decide the new α_s value and the new sample rate :

```
Calculating_New_As_&_SR_Values() {
 As = Average of the Alpha Values;
 Cs = Average of the Color Value;
 AsCs = As*Cs;
 Cd = (double)((int)(AsCs*255))/(As*255);
 do {
    Cd = (double)((int)(AsCs*255))/(As*255);
    if (Cd > Cs) break;
    else AsCs += 0.001;
 } while(1);
 As_new = AsCs/Cs + Estimated_Error
 if (As_new > 1 - sqrt(1 - 1/(255*Cs) ) )
     sr_new = sr_old*log(1-As)/log(1-As_new);
 if (sr_new ≥ sr_old ) {
     /* Do not change the values */
     sr_new = sr_old;
     As_new = As;
 }
}
```

The do-while loop compensates for the errors caused by the pre-multiplication $\alpha_s \times C_s$ and the `Estimated_Error` adds the roundoff errors of the low precision blending in each sub-node. The `Estimated_Error` depends on the number of texture mapping planes. In our implementation, the real numbers from 1/255 to 2/255 are good values for the `Estimated_Error`.

4 Implementation and Results

Our implementation platform is a cluster of Compaq PCs, each of which has 800MHZ Pentium III processors and GeForce3 graphics processors with 256M main memory. One of them has a GeForce 2 processor. The Client Interface is running on Windows2000 and

the rendering server is running on RedHat 7.2 Linux. The Client Interface communicates with the rendering server through CORBA. The root node and sub-nodes of the rendering server talk to each other using MPI. All these machines are connected by 100Mb/s Ethernet.

4.1 The Client Interface and The Rendering Server

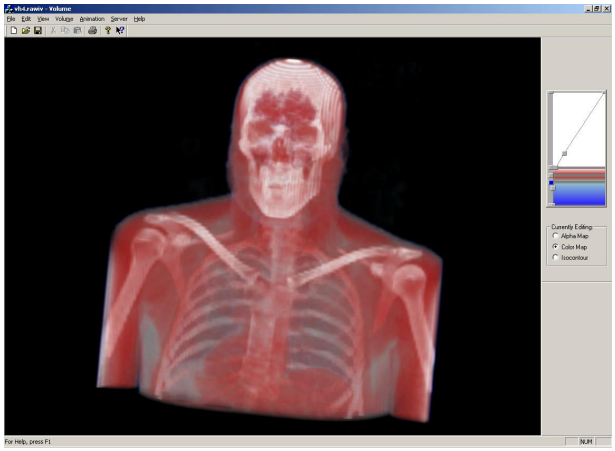


Figure 3: The Client Interface: The user manipulates the transfer function using the graphical control on the right.

The Client Interface runs on Windows2000. It is implemented using the MFC. The user manipulates the transfer function using the graphical control on the right. After chosing the transfer function and view orientation, the user can request a high resolution image from the rendering server. The rendering server is implemented using C++. The OpenGL and GLUT libraries are used for rendering. One problem we encountered in our implementation is that the CORBA is not compatible with GLUT library, since both of them use the infinite event loop. We solved the problem by removing the GLUT event loop. The sub-nodes of the server use GeForce3 processors to do 3D texture mapping, while the root node uses GeForce2 for final image blending using 2D texture. Both of the GeForce2 and GeForce3 have 64MB graphics memory.

4.2 Scalability and Performance

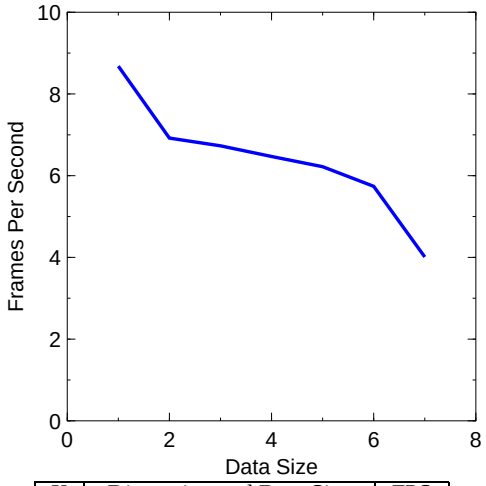
We present experimental results of the parallel rendering system in terms of performance and image quality. Table 1 shows the two datasets we used for our experiments.

Name	Data Dimension	Size
Visible Human Male	$512 \times 512 \times 512$	128MB
Visible Human Female	$512 \times 512 \times 512$	128MB

Table 1: The sizes of our test datasets

We also used decimated datasets for the Client Interface. The dimension of the decimated datasets is $128 \times 128 \times 128$. The original datasets are larger than table 1, but we resize the original datasets to fit them into our rendering system. Both of the datasets are from the visible human project of the National Library of Medicine.

First, we tested the rendering performance. We used a 512×512 drawing window and made the object big enough to fit the window. Figure 4 shows the the minimum frame rates that we achieved. X axis of figure 4 represents the data size and the Y axis is the frame rates.



X	Dimension and Data Size	FPS
1	$512 \times 512 \times 8$ (2 MB)	8.68
2	$512 \times 512 \times 16$ (4 MB)	6.92
3	$512 \times 512 \times 32$ (8 MB)	6.73
4	$512 \times 512 \times 64$ (16 MB)	6.47
5	$512 \times 512 \times 128$ (32 MB)	6.22
6	$512 \times 512 \times 256$ (64 MB)	5.74
7	$512 \times 512 \times 512$ (128 MB)	4.01

Figure 4: The Performance and Scalability According to the Data Size: The performance includes the network delay, rendering time, and hardware accelerated final blending time. We use GeForce3 processors for the rendering intermediate images and a GeForce2 processor for the final blending.

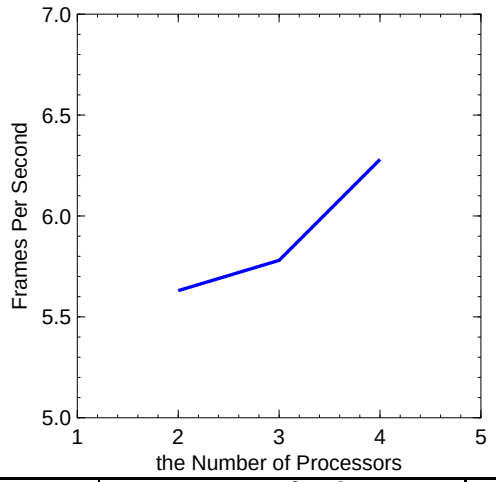
When the data size is from 4MB to 64MB, the performance of our rendering system remains almost constant, but the performance drops down at 128MB. The reason is that, as the data size grows, the 3D rendering time takes the larger part of whole generation time.

Figure 5 show the speed up for multi-PCs. The texture memeory allows only the power of two data size, so we can get enough speedup when the number of processors are doubled. The 2 and 3 processors have no big difference, but we can say that the 4 processors are faster than the 2 and 3 processors.

The Client Interface achieves 10-12FPS, with the $128 \times 128 \times 128$ dataset. We exploit the GeForce3 processor to get interactive visualization in the Client Interface.

5 Conclusion and Future Work

In this paper, we present a parallel multi-PC volume rendering system using off-the-shelf commodity components such as GeForce2 and GeFoce3 graphics processors. We exploit the hardware accelerated 3D texture mapping of the GeForce3 processors in both the Client Interface and the rendering system. A single GeForce3 processor is good enough for visualization of small volume datasets. However, when multiple GeForce3 processors are used for parallel rendering, we encountered a couple of problems. First, the final performance is affected by network delay and final composition time in addition to the rendering time. We could not guarantee that the performace increases continuously, as we use more GeForce3 processors. The frame rates we got using multiple GeForce3 processors in parallel was below interactive frame rates. However, our parallel multi-PC rendering system can solve the problem. The Client Interface provides the interactive visualization using decimated datasets and it requests high quality images from the rendering server. Since



# Processos	Data Dimension of Each Processor	FPS
2	$512 \times 512 \times 128$ (32 MB)	5.63
3	$512 \times 512 \times 128$ (32 MB)	5.78
4	$512 \times 512 \times 64$ (16 MB)	6.46

Figure 5: Speedup for Multiple-PCs:

the rendering server genetates high quality images within a second, we can achieve both of interactivity and high quality images.

We also present a technique to reduce the blending errors on the final images. The errors are not noticeable on the framebuffer of each node, but after the intermediate images are blended into an image, the blending errors result in seam planes. It is impossible to remove the seam planes perfectly because of the lack of the framebuffer's precision, but our technique can reduce the effect of the blending error.

We also ported our parallel rendering server to a PC cluster using software-based 3D texture mapping. The PC cluster has 32 Compaq PCs each of which has 256 MB main memory. They are connected by 100 MB/s Ethernet. The performace of the software-based server is definitely lower than the hardware-accelerated server, but the software-based server can deal with larger datasets. When we tested it using only 32 PCs, it was able to render 4GB volume datasets.

The GeForce3 processors provide per pixel volume shading techniques. We have tested the technique on the Client Interface. In near future, we will add the volume shading to the rendering server.

References

BAJAJ, C. L., PASCUCCI, V., AND SCHIKORE, D. R. 1997. The Contour Spectrum. *IEEE Visualization '97* (November), 167–175.

CHIANG, Y., FARIAS, R., SILVA, C. T., AND WEI, B. 2001. A Unified Infrastructure for Parallel Out-Of-Core Isosurface and Volume Rendering of Unstructured Grids. *IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics* (October).

KINDLMANN, G., AND DURKIN, J. W. 1998. Semi-Automatic Generation of Transfer Functions for Direct Volume Rendering. *1998 Symposium on Volume Visualization* (October), 79–86.

KNISS, J., KINDLMANN, G., AND HANSEN, C. 2001. Interactive Volume Rendering Using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets. *IEEE Visualization.*

KNISS, J., MCCORMICK, P., MCPHERSON, A., AHRENS, J., PAINTER, J., KEAHEY, A., AND HANSEN, C. 2001. Interactive Texture-Based Volume Rendering for Large Data Sets. *IEEE Computer Graphics and Applications 21*, 4 (July/August), 52–61.

LOMBEYDA, S., MOLL, L., SHAND, M., BREEN, D., AND HEIRICH, A. 2001. Scalable interaction volume rendering using off-the-shelf components.

MAGALLON, M., HOPF, M., AND ERTL, T. 2001. Parallel Volume Rendering using PC Graphics Hardware. *Pacific Graphics.*

MEIBNER, M., GRIMM, S., STRABER, W., PACKER, J., AND LATIMER, D. 2001. Parallel volume rendering on a single-chip SIMD architecture. *Proceedings of IEEE Symposium in Parallel and Large Data Visualization and Graphics.*

PARK, S., PARK, S., AND BAJAJ, C. 2001. Hardware Accelerated Multipipe Parallel Rendering of Large Data Stream. *CS and TICAM Technical Report.*

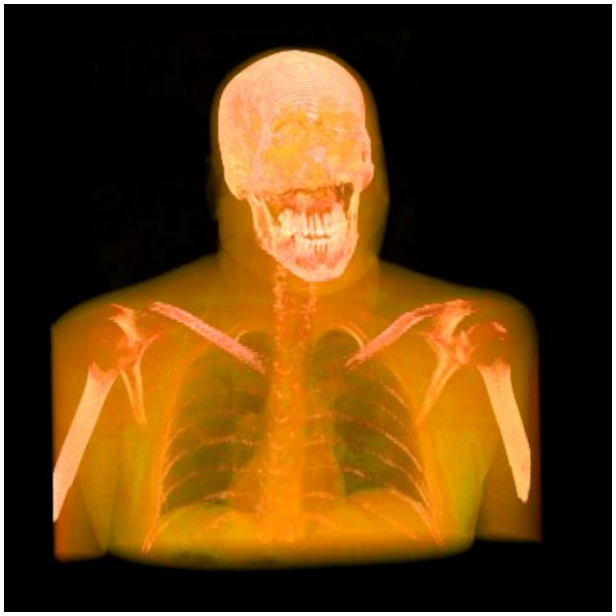
PFISTER, H., HARDENBERGH, J., KNITTEL, J., LAUER, H., AND SEILER, L. 1999. The VolumePro Real-Time Ray-Casting System. *Proceedings of SIGGRAPH 99* (August), 251–260.

PORTER, T., AND DUFF, T. 1984. Compositing digital images. *In SIGGRAPH 84*, 253–259.

The Visible Human Project. *http://www.nlm.nih.gov/research/visible/*.

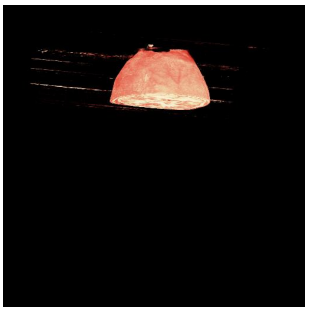
ZHANG, X., BAJAJ, C., AND BLANKE, W. 2001. Scalable Isosurface Visualization of Massive Datasets on COTS-Cluster. *Proc. of IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, 51–58.

(a)

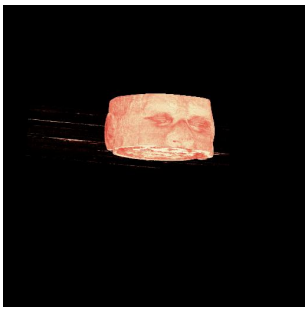


(b)

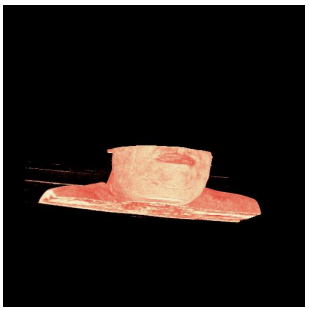
Figure 6: The images before and after applying the seam plane removing algorithm. (a) shows the image with seam planes, and (b) shows an image after the seam planes have been removed.



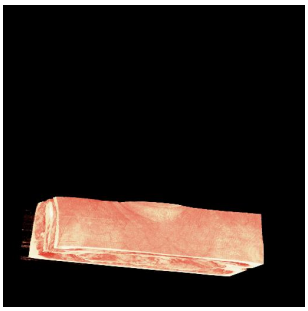
(a)



(b)



(c)



(d)

(e) Composed Final Image

Figure 7: The Visible Human Female Dataset. The sub-images have the same size as the final image. The back-to-front blending method is used for the intermediate images and the final image