Visualization of Very Large Oceanography Time-Varying Volume Datasets

Sanghun Park[1], Chandrajit Bajaj[2], and Insung Ihm[3]

[1] School of Comp. & Info. Comm. Engineering, Catholic University of Daegu
Gyungbuk 712-702, Korea
mshpark@cu.ac.kr
http://viscg.cu.ac.kr/~mshpark

[2] Department of Computer Sciences, University of Texas at Austin
TX 78731, USA
bajaj@cs.utexas.edu
http://www.cs.utexas.edu/~bajaj

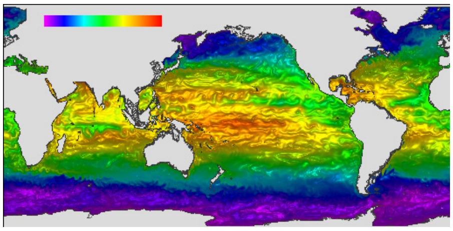
[3] Department Computer Science, Sogang University
Seoul 121-742, Korea
ihm@sogang.ac.kr
http://grmanet.sogang.ac.kr/~ihm

Abstract. This paper presents two visualization techniques suitable for huge oceanography time-varying volume datasets on high-performance graphics workstations. We first propose an off-line parallel rendering algorithm that merges volume ray-casting and on-the-fly isocontouring. This hybrid technique is quite effective in producing fly-through movies of high resolution. We also describe an interactive rendering algorithm that exploits multi-piped graphics hardware. Through this technique, it is possible to achieve interactive-time frame rates for huge time-varying volume data streams. While both techniques have been originally developed on an SGI visualization system, they can be also ported to commodity PC cluster environments with great ease.

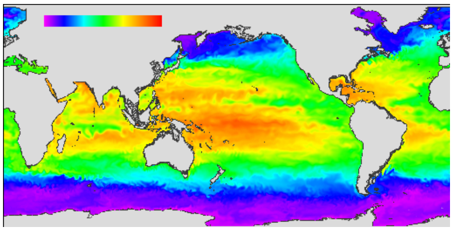
1 Introduction

Understanding the general circulation of oceans in the global climate system is critical to our ability to diagnose and predict climate changes and their effects. Recently, very high quality time-varying volume data, made of a sequence of 3D volume data, were generated in the field of oceanography. The model has a resolution of 1/6 degree (2160 by 960 points) in latitude and longitude and carries information at 30 depth levels. It includes several scalar and vector field data sets at each time step: temperature, salinity, velocity, ocean surface height, and ocean depth. The datasets are from a 121 day oceanographic simulation. The time step interval is 300 seconds beginning on Feb-16-1991 at 12:00:00. Each scalar value of voxel is stored in four bytes, and the total size of the data is about 134 GB (refer to Table 1).

Usually, oceanographers have used pre-defined color-mapped images to visualize and analyze changes in an ocean. Because there is a one-to-one correspondence between colors in color maps and voxel values, these images are quite intuitive. However, they appear as a simple 2D plane, which leads to difficulties in understanding dynamic changes between the time steps. On the other hand, images created by 3D rendering techniques,



(a) Volume rendering scheme combined with on-the-fly isocontouring



(b) Color mapping scheme

Fig. 1. Comparison of two visualization schemes on T data

such as ray-casting and splatting, may be less intuitive in case illumination models are applied carelessly. However, 3D-rendered images of high quality have an advantage that detailed changes between time steps are described effectively. Figure 1 compares two visualized images produced using the volume rendering/isocontouring method and the color-mapping method, respectively, for the temperature data.

In this paper, we discuss two different approaches designed for efficient rendering of the huge time-varying oceanography datasets on high performance graphics architectures. First, we present our off-line parallel rendering method to produce high-resolution fly-through videos. In particular, we propose a hybrid scheme that combines both volume rendering and isocontouring. Through parallel rendering, it effectively produces a sequence of high quality images according to a storyboard.

Secondly, we explain an interactive multi-piped 4D volume rendering method designed for the same datasets. Although this technique is reliant on the color-mapping method and does not create 3D rendered images of high quality, it is possible to freely control camera and rendering parameters, select a specific time step, and choose a color map in real-time through graphical user interfaces.

Table 1. Statistics on the oceanography datasets used

Name	Dataset	Attribute	Dimension	Time Step	Size (GB)
T	Temperature	Scalar	$2160 \times 960 \times 30$	121	28.0
S	Salinity	Scalar	$2160 \times 960 \times 30$	115	26.7
Vel	Velocity	Vector (U, V, W)	$2160 \times 960 \times 30$	113	78.6
PS	Surface Height	Scalar	2160×960	114	0.9

2 Offline Parallel Rendering

2.1 Basic Algorithm

Our off-line parallel volume rendering of a data stream was designed to quickly produce high-resolution fly-through movies using parallel processors. It is based on our parallel rendering algorithm that tried to achieve high performance by minimizing, through data compression, communications between processors during rendering [1]. The algorithm was also extended to uncompressed data streams as input data and implemented on a Cray T3E, SGI Onyx2, and a PC cluster. In our parallel scheme, the image screen is partitioned into small, regularly spaced pixel tiles, which form a pool of tasks. During run time, processors are assigned to tiles from the pool of tasks waiting to be executed. The processors perform a hybrid rendering technique, combining volume rendering and isocontouring as explained below, repeatedly on tiles until the task pool is emptied. Load balancing is carried out dynamically during rendering.

2.2 Combining Volume Rendering and Isocontouring

It is very useful to extract isosurfaces at chosen data values to analyze information in volume data, but isocontouring of large volume data generates a large number of isocontour polygon sets. We designed a new rendering algorithm, merging volume ray-casting and isocontouring. In this scheme, isocontour extraction during the rendering process does not require large sets of polygons to be created and stored. When cast rays intersect volumes and isosurfaces, shading and composing are properly applied on the fly. The algorithm takes advantage of visualizing interesting materials and isosurfaces simultaneously, and using higher order interpolation and approximation. In our implementation, the ocean floors and continents were rendered by isocontouring at a selected function value, and the other parts were visualized by volume ray-casting (see Figure 1 (a)).

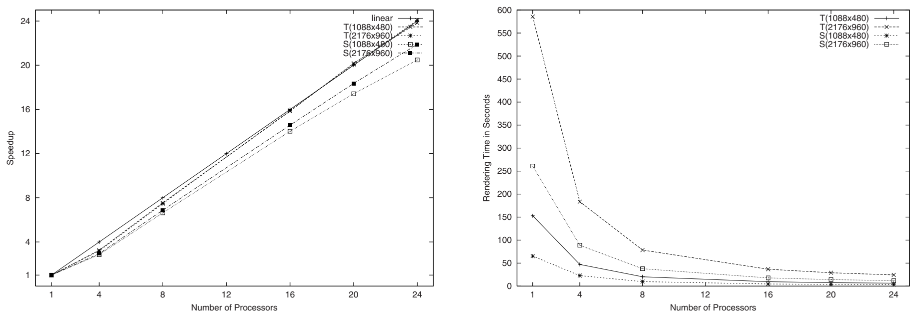
To create meaningful, high quality 3D images that match pre-defined color maps, we modified Phong's illumination model, which determines the final colors from the sum of three components, ambient color, diffuse, and specular: $I = k_a I_a + \sum_{i=1}^{n} I_{l_i}\{k_d(N \cdot L_i) + k_s(N \cdot H_i)^{n_s}\}$. In this basic illumination formula, ambient color $k_a I_a$ is ignored and diffuse and specular coefficients k_d and k_s are substituted for a color k_c from a pre-defined color map according to the data values at the sampling points. The effects showing 3D appearance and dynamic changes between time steps are demonstrated well through the calculation of the gradient N used in diffuse and specular terms: $\overline{I} = \sum_{i=1}^{n} I_{l_i} k_c\{N \cdot L_i + (N \cdot H_i)^{n_s}\}$.

In traditional volume rendering, it is important to use appropriate transfer functions based on gradient magnitude and functional values, as materials are visualized with specific ranges. In the visualization of oceanography data, defining the transfer functions is not a critical issue because all voxels are rendered regardless of their values. We used uniform transfer functions defined over whole values. To minimize mismatches between colors in the final images and the pre-defined color maps, caused by improper composition, our algorithm maintains very dense sampling intervals and uses the 'over' operator for color composition.

Figures 1 (a) and (b) were created by the proposed rendering method and the color-mapping method, respectively, where the first time step volume of temperature T data at a depth of 100 meters was visualized. Although there are some mismatches between colors of the ocean and the color map in Figure 1 (a), the figure clearly shows dynamic changes in temperature when a movie for all time steps is played.

2.3 Results of Offline Parallel Rendering

Here, we only present results for uncompressed data to avoid the degradation resulting from using compressed volume data. The MPI (Message Passing Interface) toolkit was



(a) Speedup (b) Rendering time

Fig. 2. Results on speedup and rendering time

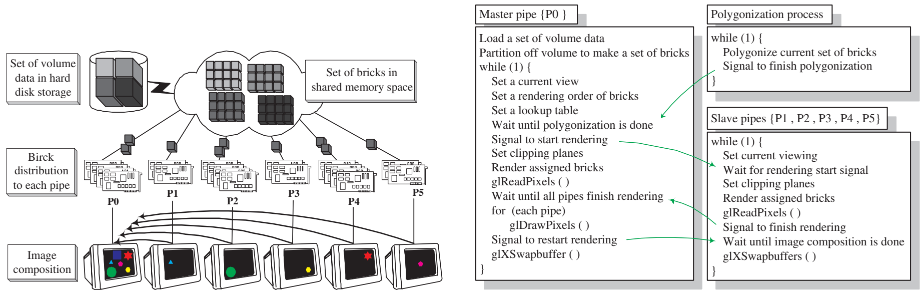
used as an interprocessor communication library, and timings were taken in seconds for generating 2176×960 and 1088×480 perspective images using 32×32 tiles on a SGI Onyx2 with 24 processors and 25 GB of main memory. Figure 2 shows the performances of the average speedup and rendering times on T and S data for one time step. The data loading times from secondary disks to main memory were not considered in these results. When 24 processors were used, it took 3.19 seconds to render a 1088×480 image and 6.42 seconds to render a 2160×960 image, both of S data. On the other hand, it took 65.33 and 260.96 seconds to generate the same images on a uniprocessor. The primary reason for the increased speed is that our scheme minimizes the data communication overhead during rendering. Only communication for task assignments and image segment collection is necessary.

3 Interactive Multi-pipe 4D Rendering

3.1 Implementation Details

We have also developed an effective multi-pipe rendering scheme for the visualization of time-varying oceanography data on the SGI Onyx2 system that has six InfiniteReality2 graphics pipes with multiple 64MB RM9 Raster Managers. The graphics system can be tasked to focus all pipelines on a single rendering window, resulting in near-perfect linear scalability of visualization performance [2]. It is optimized for rendering polygon-based geometric models, not for visualizing volume data. Most large volume datasets contain much more voxels than can be stored in texture mapping hardware. To take advantage of graphics hardware acceleration, volume datasets are partitioned into sub-volumes called bricks. A brick is a subset of voxels that can fit into a machine's texture mapping hardware. In general, the optimal size of bricks is determined by various factors such as texture memory size, system bandwidth, and volume data resolution. In our application, the size of the bricks is dynamically determined to minimize texture loading.

Texture-based volume rendering includes a setup phase and a rendering phase (see Figure 3 (a)). The setup phase consists of volume loading and data bricking whose computational costs depend on disk and memory bandwidth. This process does not


Set of volume data in hard disk storage
Set of bricks in shared memory space
Birck distribution to each pipe
P0
P1
P2
P3
P4
P5
Image composition

```
Master pipe {P0 }
Load a set of volume data
Partition off volume to make a set of bricks
while (1) {
    Set a current view
    Set a rendering order of bricks
    Set a lookup table
    Wait until polygonization is done
    Signal to start rendering
    Set clipping planes
    Render assigned bricks
    glReadPixels ( )
    Wait until all pipes finish rendering
    for  (each pipe)
        glDrawPixels ( )
    Signal to restart rendering
    glXSwapbuffer ( )
}
```

```
Polygonization process
while (1) {
    Polygonize current set of bricks
    Signal to finish polygonization
}
```

```
Slave pipes {P1 , P2 , P3 , P4 , P5}
while (1) {
    Set current viewing
    Wait for rendering start signal
    Set clipping planes
    Render assigned bricks
    glReadPixels ( )
    Signal to finish rendering
    Wait until image composition is done
    glXSwapbuffers ( )
}
```

(a) Rendering architecture (b) Applied algorithm

Fig. 3. Our multi-pipe visualization scheme

affect the actual frame rates in run-time rendering because the entire voxels of each test set are loaded into shared memory.

The rendering phase involves texture loading from shared memory space to texture memory, 3D texture mapping in Geometry Engines and Raster Managers, and image composition. It is important to optimize each step in the rendering phase to achieve interactive-time frame rates. Because the maximum texture download rate is 330 MB/second from host memory, it takes at least 0.19 seconds ($\frac{64}{330}$) to load one 64 MB brick. Despite the ability that downloads and draws textures simultaneously, allowing textures to be updated on the fly, the cost is so expensive that real-time frame rates are hard to achieve. When volume datasets, much larger than the amount of texture memory on the graphics platform, are visualized, the inevitable texture swapping hinders real-time rendering [5], hence should be minimized. Actually, without texture swapping, it was possible to create over 30 frames per second using the Onyx2. Multiple graphics pipes are involved in our scheme. As each pipe loads different bricks, the amount of texture swapping can be minimized.

As mentioned, partitioning large volume data into bricks in hardware accelerated rendering is necessary. It is important to assign them to the respective pipes carefully because the number of Raster Managers of our system varies pipe-by-pipe (see Figure 3 (a) again). It takes 0.08 seconds (12.24 frames per second) to render a volume dataset of 64 MB using a pipe in $\{P_0, P_2, P_4\}$ with four Raster Managers. On the contrary, if a pipe in $\{P_1, P_3, P_5\}$ with two Raster Managers is used to render the same data, the rendering time is almost doubled (0.15 seconds: 6.81 frames per second). We were able to improve the speedup by assigning additional bricks to pipes with more Raster Managers.

Figure 3 (b) gives an overview of our algorithm based upon the object-space division method to minimize texture swapping. The master pipe $\{P_0\}$ plays an important role in controlling the slave pipes $\{P_1, P_2, P_3, P_4, P_5\}$ and composing sub-images. The slave pipes render assigned bricks and write sub-images to shared memory space under the control of the master pipe. The polygonization process as a separate thread process continues to generate sequences of polygons perpendicular to the viewing direction until the program is finished.

Once the current view is set, a rendering order of bricks is determined. Each pipe starts creating sub-images for the assigned bricks and then the master pipe composes the sub-

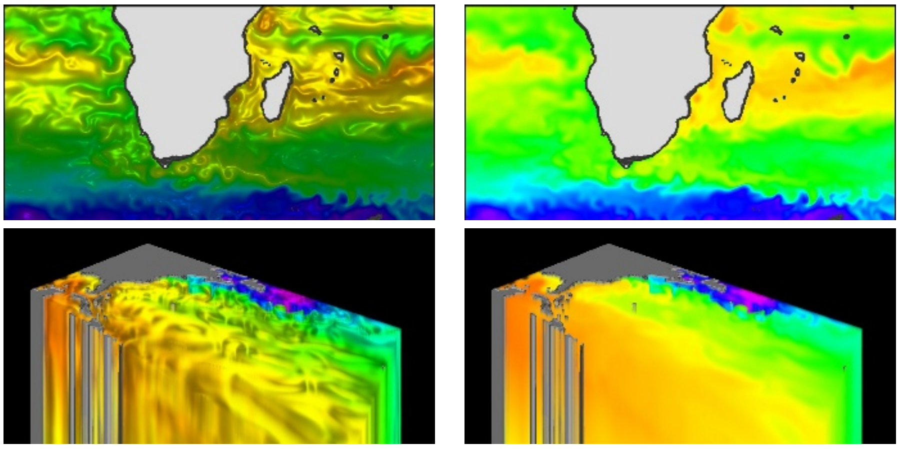

(a) Hybrid method (b) Color mapping method

Fig. 4. Visualization results on T data

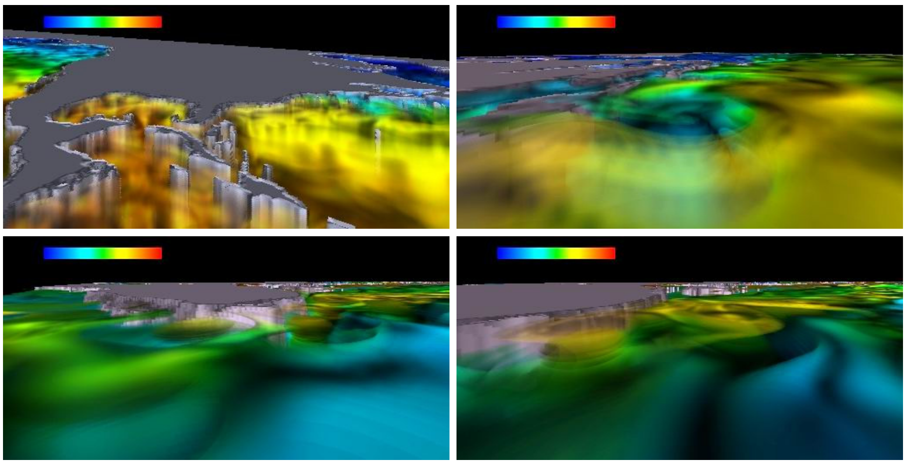


Fig. 5. Images of T data created by combining volume rendering and isocontouring

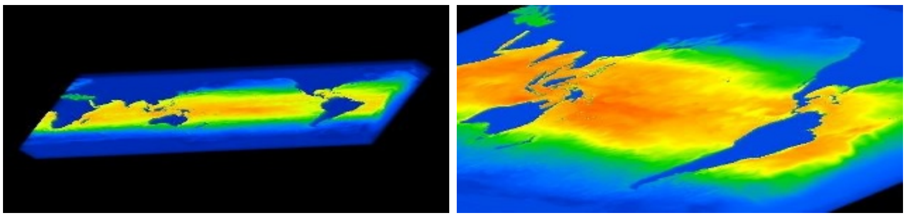
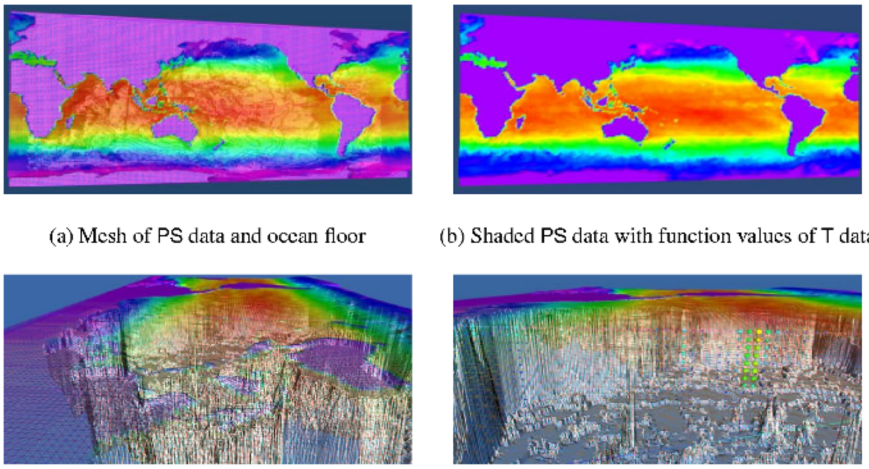


Fig. 6. Interactive multi-pipe time-varying volume visualization of T data



(a) Mesh of PS data and ocean floor

(b) Shaded PS data with function values of T data

(c) Fly-through under the ocean

(d) Spheres for showing Vel data

Fig. 7. A fly-through using the SGI Performer

images according to the order. Our algorithm requires synchronization for every frame between the master pipe and the slave pipes. Because the master pipe must wait until all slave pipes have written sub-images in shared memory space, the actual frame rate is dictated by the slowest pipe. We tried to solve this problem by proportionally assigning bricks. It is inevitable that the rendering of time-varying data requires additional texture swapping. To reduce the swapping cost, the algorithm routinely checks to make sure that the next brick to be rendered is already stored in texture memory.

3.2 Performance of Multi-pipe Rendering

We have used the OpenGL and OpenGL Volumizer [4] to measure the timing performance in rendering images of size 640×640. As hardware-supported texture memory requires the dimensions of the bricks to be a power of two, each original time step of volume data was reduced to $2048 \times 1024 \times 32$ volume with 1-byte voxels for this experiment. The best rendering speed was achieved when three pipes $\{P_0, P_2, P_4\}$ were used. The oceanography data was rendered at an interactive frame rate of 12.3 frames per second with varying time steps.

4 Experimental Results and Concluding Remarks

When two videos, one created by our rendering scheme and the other by the color mapping method, were displayed at the same time in a single window, we found that it enables an oceanographer to easily analyze the information contained in the time-varying volume data. In Figure 4, interesting local regions are visualized to see the difference of the two rendering methods. The oblique images clearly show that our hybrid technique represents ocean changes better than the color mapping method does.

Figure 5 demonstrates animation frames taken from high density movies made by our off-line parallel scheme.

The images of Figure 6 are snapshots created by our multi-pipe technique at an interactive frame rate. As it creates images by rendering a sequence of resampling planes intersected with bricks, the volume data is also considered a geometric object. Therefore, it is easy to visualize volume data and other geometric objects simultaneously. It is also trivial to dynamically set cutting planes that allow us to investigate the inside of an opaque volume. Our multi-pipe rendering scheme can be easily transformed as a core rendering module for stereoscopic multi-screen display systems.

Time-varying oceanography data could be visualized using the OpenGL Performer [3]. Because the OpenGL Performer uses only polygonal models as input data and exploits optimized multi-pipe rendering, we had to extract geometric information from the time-varying volume dataset in a pre-processing stage. In Figure 7, the dynamic ocean surface polygons and the colors of the polygon vertices defined as functional values were generated from the PS data and the T data, respectively. The mesh dataset for the ocean floor was created by a simple polygonization algorithm. To visualize the velocity vector field, we made spheres from Vel. The radius of the spheres represents the magnitude of velocity at the sampling point, and the line segments attached to the spheres indicate the direction. During a fly-through at an interactive frame rate, we were able to effectively investigate the relationship between datasets in the dynamically changing ocean field.

In this paper, we have presented two visualization techniques to render huge time-varying oceanography data. We focused not only on creating high quality images using an off-line parallel rendering algorithm, but also on an interactive rendering algorithm that takes advantage of the multi-pipe feature of an SGI's high performance graphics system. Currently, our visualization techniques are planned to be ported to commodity PC cluster environments.

Acknowledgements. We wish to thank Prof. Detlef Stammer and Dr. Arne Biastoch of Scripps Institution of Oceanography for allowing access to the oceanography data. This work was supported by Korea Research Foundation Grant (KRF-2003-003-D00387).

References

1. C. Bajaj, I. Ihm, G. Koo, and S. Park. Parallel ray casting of visible human on distributed memory architectures. In *Proceedings of VisSym '99 (Joint EUROGRAPHICS-IEEE TCVG Symposium on Visualization)*, pages 269–276, Vienna, Austria, May 1999.
2. G. Eckel. Onyx2 Reality, Onyx2 InfiniteReality and Onyx2 InfiniteReality2 technical report. Technical report, Silicon Graphics, Inc., 1998.
3. G. Eckel and K. Jones. OpenGL Performer programmer's guide. Technical report, Silicon Graphics, Inc., 2000.
4. Silicon Graphics Inc. OpenGL Volumizer programmer's guide. Technical report, Silicon Graphics, Inc., 1998.
5. W. R. Volz. Gigabyte volume viewing using split software/hardware interpolation. In *Proceedings of Volume Visualization and Graphics Symposium 2000*, pages 15–22, 2000.