# Interactive Distributed Translucent Volume Rendering

Balázs Domonkos Balázs Csébfalvi
Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
Magyar tudósok krt. 2.
H-1117, Budapest, Hungary
domonkos@ik.bme.hu, cseb@iit.bme.hu

**ABSTRACT**

Translucent volume rendering is a robust and efficient direct volume-rendering technique for capturing optical effects, like subsurface scattering, translucency, and volumetric shadows. However, due to the limited computing and memory resources of the recent consumer graphics hardware, high-resolution volume data can still hardly be interactively visualized by this method. In this paper we present the theoretical aspects and implementation details of a parallelization scheme for translucent volume rendering. Our method is a three-pass parallel rendering algorithm with parallel compositing, based on object-space distribution of the data among the rendering nodes. In the first pass the 2D shadow maps are computed and sent to the effected nodes. In the second pass the nodes render their associated subvolumes by sequential translucent volume rendering. The generated framelets are then visualized by a dedicated display node in the third pass.

**Keywords:** Volume Rendering, Volumetric Shadows, Parallel and Distributed Graphics

## 1 INTRODUCTION

Using traditional direct volume visualization, the classical volume-rendering integral is numerically computed by evaluating finite number of samples along the viewing rays [Lev88]. Optical properties, like color and opacity are assigned to the samples by mapping the density and optionally the gradient magnitude with a transfer function. The color samples are shaded according to the normalized direction of the estimated gradient, which is treated as a normal of an isosurface. In a nearly homogeneous region, however, the variation of the densities is presumably due to the noisy data acquisition. Therefore the gradient estimation yields stochastic normal directions in the originally homogeneous regions. As there are no well-defined isosurfaces in these regions, the evaluation of a local shading model is not physically plausible. This problem is usually avoided by modulating the opacities by the gradient magnitude [Lev88], which enhances the well defined isosurfaces contained in the volume. Another drawback of the classical direct volume rendering model is that it relies on accurately estimated gradient directions. However, the gradient is usually calculated from quantized density values, so it can represent only a limited number of surface normals. Furthermore, the ideal gradient estimation cannot be efficiently implemented, therefore it is only approximated in practical applications. Because of these two reasons, images rendered by the traditional direct volume-rendering approach typically contain staircase artifacts.



*WSCG 2007 conference proceedings, ISBN 1213-6964*
***WSCG'2007, January 29 – February 1, 2007***
*Plzen, Czech Republic.*


Translucent volume rendering [KPHE02], which is based on a fundamentally different optical model, does not rely on estimated gradients at all. In this case, the colors are also assigned to the samples by a transfer function, but they are not shaded by evaluating an explicit local shading model. Instead, the color of each sample is multiplied by the intensity of an attenuated light ray coming from the light source into the given sample position. Furthermore, with a Gaussian perturbation, this approach can also be used for a rough approximation of forward scattering. Despite its robustness and optical modeling potential, the literature on translucent volume rendering is relatively narrow. In this paper we aim at an efficient parallel implementation scheme for translucent volume rendering of large-scale volumetric data sets. For the classical direct volume rendering of high-resolution data sets several researchers proposed efficient hierarchical data distribution and parallelization schemes. According to our best knowledge, however, a parallel translucent volume rendering algorithm has not been published yet.

In Section 2 the previous work related to distributed and parallel volume rendering is reviewed. In Section 3 we briefly overview the traditional sequential implementation of translucent volume rendering on a single GPU. Our parallelization scheme and its implementation are presented in Section 4 and Section 5 respectively. Images generated by our algorithm and the performance measurements are reported in Section 6,

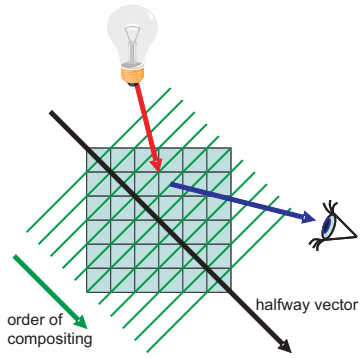


Figure 1: Translucent volume rendering on a single GPU using back-to-front compositing.

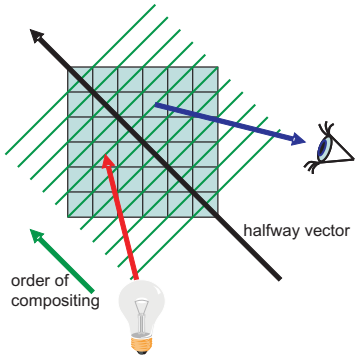


Figure 2: Translucent volume rendering on a single GPU using front-to-back compositing.

while in Section 7, we summarize the contribution of this paper.

## 2 RELATED WORK

There are four fundamentally different approaches for direct volume rendering: ray casting [Lev88], splatting [Wes90], shear-warp factorization [LL94], and texture mapping [CCF94, WE98]. The sequential or single-processor implementations of these methods are usually used to visualize data sets of moderate resolution. Practical data sets, however, continue to drastically increase in size, therefore different parallelization and data distribution schemes have been proposed for all the four basic algorithms.

Ray casting, which can produce the highest quality of rendered images, has been implemented on different architectures using either image-space or object-space partitioning [LY96, MPH93, MPHK94, BIPS00, PTT98, RPS99]. Similarly, the classical object-order splatting technique has also been adapted to multiprocessor environments [Elv92, JG95, LWM97]. The shear-warp algorithm, which had been originally proposed as a fast software implementation of direct volume rendering, was parallelized on an SGI system [Lac96]. As most of the recent consumer graphics cards supports 3D texture mapping, the texture-slicing approach became one of the most popular volume-rendering techniques. Nevertheless, its most important drawback is that due to limited texture memory large-scale volume data cannot be rendered without swapping subvolumes between the main memory and the local texture memory [GWGS02]. In this case, the bottleneck is the bandwidth of data transferring rather than the performance of the GPU. Therefore implementations for parallel architectures have been proposed by several researchers to overcome this limitation [KMM01, MHE01, GPH04].

The previously published parallel systems usually support only the classical volume-rendering model. However, a more sophisticated optical model, which includes volumetric shadows and forward scattering, requires a more complicated communication between the parallel processing units. In this paper we adapt the sequential translucent volume rendering method to parallel computing nodes. We demonstrate that, using a static data distribution scheme and parallel image compositing with high-speed communication channels, intermediate image data can be efficiently transferred between the nodes. As a consequence, the communication overhead is negligible compared to the rendering cost.

## 3 TRANSLUCENT VOLUME RENDERING

Translucent volume rendering can be efficiently implemented exploiting the 3D texture mapping capability of recent graphics cards. Unlike traditional slice-based direct volume rendering techniques, in this case volumetric shadows are calculated simultaneously with the compositing of the resampling slices. In order to avoid the computation of a 3D shadow map, the slicing is performed perpendicular to the halfway vector between the viewing direction and the direction of the light source as illustrated in Figure 1. For each pixel covered by the projection of a slice, the intensity of the light has to be determined, which reaches the intersection point between the slice and the corresponding viewing ray. Therefore each slice is simultaneously projected onto a plane perpendicular to the direction of the light source. In this way, a 2D shadow map can be maintained, which corresponds to the current stage of compositing. Whenever a pixel is processed in the pixel shader it is determined where the corresponding intersection point is projected onto the current shadow map and its color is modulated accordingly. As the resolution of the shadow map is the same as the resolution of the frame buffer, accurate volumetric shadows can be calculated by this method. If the angle between the viewing direction $\mathbf{V}$ and the direction of the light source $\mathbf{L}$ is less than 90 degrees then the halfway vector $\mathbf{H}$ is calculated as $\mathbf{H} = (\mathbf{L} + \mathbf{V})/2$ and a back-to-front compositing is performed (see Figure 1). Otherwise the halfway vector is

calculated as $\mathbf{H} = (\mathbf{L} - \mathbf{V})/2$ and a front-to-back compositing is performed (see Figure 2).

## 4 DISTRIBUTED TRANSLUCENT VOLUME RENDERING

As our major goal is to interactively render large-scale data sets using the translucent shading model, the basic algorithm is adapted to a parallel computing environment. Parallelization schemes can be classified according to the type of entities, which are simultaneously processed. Single-threaded software renderers take graphics primitives one after another and the pixels corresponding to these primitives are also processed sequentially. In contrast, recent graphics cards have multiple graphics pipelines, therefore more vertices and pixels can be processed at the same time. Pixel-based parallelization can also be performed when multiple graphics cards are used for creating tiles of the overall output image and the rendering queue is branched into multiple pipes. On the other hand, when the data is divided in an initialization step, multiple subsets of graphics primitives and subvolumes can be processed at the same time.

In our case the original volumetric data is decomposed into subvolume blocks using axis-aligned subdivision. These blocks are distributed among the computing nodes. This static data distribution scheme is more favorable than pixel-level partitioning, because of two reasons. (1) To create an equivalent rendering model, initial shadow maps are needed to start the rendering of a subset of the volume. This inter-node shadow communication can be more easily performed using a fixed object-space subdivision rather than image-space decomposition, which deals with non-axis-aligned connection surfaces. (2) Furthermore the nodes can efficiently render only subvolumes of moderate resolution without swapping because of their limited texture memory.

This object-parallel approach needs compositing the subsets of pixels corresponding to different subvolumes. This procedure involves processing of alpha and depth values for each pixel, therefore compositing can be a bottleneck of the overall rendering system and unsuitable for interactive applications. However, when the compositing is also done in parallel, interactive compositing is possible. There are several algorithms providing parallel image compositing on multiprocessor architectures including direct send, parallel pipeline [LRN96], and binary swap [MPHK94].

In our approach a three-pass, object-parallel algorithm was used with parallel pipeline compositing. It performs the following steps:

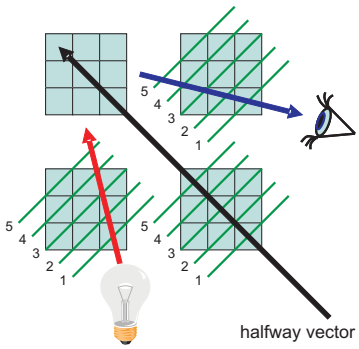


Figure 3: Parallel calculation of 2D shadow maps on the separate nodes. The numbers represent the different time steps.

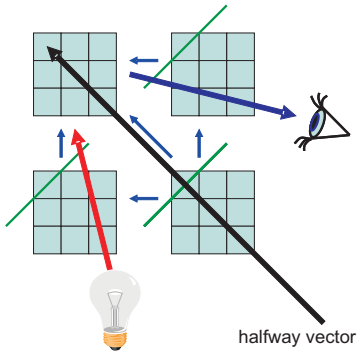


Figure 4: Sharing the 2D shadow maps with the effected nodes for parallel compositing.

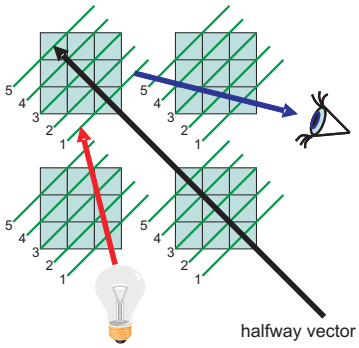


Figure 5: Parallel translucent volume rendering on the separate nodes. The numbers represent the different time steps.

1. pass: Each node computes its 2D shadow map and shares it with the effected nodes for parallel compositing.

2. pass: After compositing the received 2D shadow maps, each node performs translucent volume rendering as in the basic algorithm. The images of subvolumes are shared among all nodes.

3. pass: The portions of the final image are also composited in parallel and sent to the display node.

The first step is necessary, because each node needs an initial shadow map in order to start its effective rendering process. This map comes from the composited

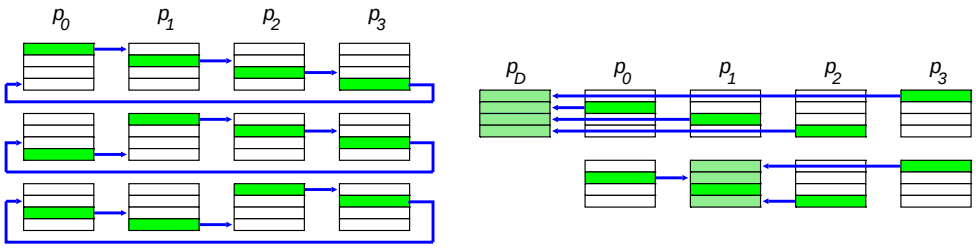


Figure 6: Parallel pipeline algorithm on distributed memory architectures. Left: framelet transfer for four compositing processes performed in $N-1$ steps, where $N$ is the number of compositors. Right: collecting final framelets for an external (display) process or for an internal process (compositing shadow maps). This is performed in one step [LRN96].

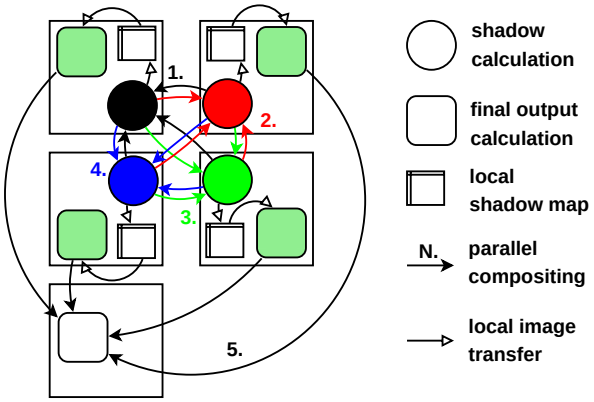


Figure 7: Parallel translucent volume rendering scheme for four rendering-compositing nodes and a composting-only display node. Shadow map calculation requires four parallel compositing contexts while a single context is needed for final image compositing. The participants of the final image context are fixed, while the members of the shadow maps are dynamically calculated based on the orientation of the volume. Note that the filled arrows point from the rendering sources to the destinations.

```
global p_i, N

function parallel_pipeline(i, target, frame) {
  def f_k ← frame.sub_image(0,k*frame.height/N,
        frame.width, (k+1)*frame.height/N−1)
  p_next ← p_{i+1} mod N
  p_prev ← p_{i−1} mod N

  for j ← 0, ..., N−2
    f_send ← f_{i−j} mod N
    f_recv ← f_{i−j−1} mod N
    send f_send → p_next
    receive f_recv ← p_prev
    compose f_recv with f_send

  if i ≠ target
    send f_i →p_target
  else
    for j ← 0, ..., N−1 (j ≠ i)
      receive f_j ← p_j
}
```

Listing 1: Pseudo-code of the parallel pipeline algorithm on distributed memory architectures for process $p_i$. The variables are coded as follows: $p_0 \ldots p_{N-1}$ are the compositing processors, $f_0 \ldots f_{N-1}$ are the image framelets, and *target* is the index of the target process.

shadow maps produced by the nodes associated to the covering subvolumes. Rendering these maps can be performed very efficiently on the graphics hardware, since only one multiplication has to be executed per pixel in the pixel shader code. Moreover, the nodes can simultaneously generate the shadow maps of their corresponding subvolumes, as illustrated in Figure 3, without waiting for each other. After having the shadow maps calculated they are shared with the effected nodes for parallel compositing (see Figure 4). In the second step, the nodes first have to composite the received images to produce an initial shadow map for the rendering. Depth information is not required here, as only an accumulated light attenuation needs to be evaluated for each pixel. Afterwards the nodes simultaneously perform the traditional translucent volume rendering for their assigned subvolumes (see Figure 5). The resulting framelets are split up, composited simultaneously, and the portions are sent to a dedicated node, which is responsible for displaying. In this third step depth-sorting is necessary before compositing, since the alpha-blending evaluation is order-dependent.

## 5 IMPLEMENTATION

We implemented our algorithm on a GPU cluster, a shared memory parallel rendering and compositing environment using the ParaComp[1] library. This software solution uses the parallel pipeline compositing algorithm consisting of two parts (see Figure 6 and Listing 1). The images to be composited are divided into

[1] Parallel Compositing library specified for multiprocessor systems by Hewlett-Packard in collaboration with Computational Engineering International

$N$ framelets, which is the number of the compositing processes. In the first part of the algorithm these framelets flow around through each node in $N-1$ steps, each consisting of a compositing and a communication stage. After $N-1$ steps each processor will have a fully composited portion of the final frame. The framelets are collected for an external display node or for an internal node in the second part in one step. The clear benefit of this compositing scheme is that the amount of data transferred on the network is independent of the number of compositing processes.

The detailed scheme of our algorithm is illustrated in Figure 7. Each rendering-compositing node being responsible for a subvolume has two separate processes and a local shadow buffer. The shadow calculation process renders the shadow image of the corresponding subvolume and belongs to multiple compositing contexts. The number of these contexts equals to the number of rendering nodes, which is four in Figure 7. Each compositing context has an explicit target process: the one which will use the composited shadow image. The other processes provide framelets to the context if the subvolumes of them occlude the subvolume corresponding to the target process. The composited shadow map is the input of another process of the node, which performs the final output calculation. The final image is composited in an additional context similarly in parallel – this is the fifth context in Figure 7. The parallel image compositing is illustrated with filled arrows while the empty arrows stand for direct local image transfer.

It follows that $N+1$ frames have to be composited applying $N$ parallel rendering-compositing nodes. According to this scheme, for interactive rendering the algorithm could be optimal when the rendering and compositing time of the shadow map calculation equals to the time required for producing the final output.

## Node-level aspects

As a bottleneck of our algorithm could be the efficiency of the sequential translucent volume rendering performed on the separate nodes, we optimized the fragment programs corresponding to this pass (see Listing 2). In order to perform high-precision compositing, we use alternating floating-point render targets or ping-pong buffering [KPHE02]. The previous state of the frame buffer is stored in a 2D texture denoted by `frameBuffer`. In the first step the color of the current pixel (`colorIn`) is read from this 2D texture. As the opacity channel represents the accumulated opacity, we can terminate the execution of the fragment program if it is already higher than a predefined threshold. This is similar to the well-known early ray termination [DH92].

The volumetric data is loaded into the texture memory as a 3D density array. The trilinearly interpolated density values are used for addressing a look-up table representing the current transfer function. Using this post-classification approach, the transfer function can be interactively modified on the fly. If the opacity of a sample (`color.a`) read from the look-up table is less than a predefined threshold, its contribution to the current pixel is negligible. Therefore the execution is terminated in this case as well. For shading the colors assigned to the samples, the attenuation of the light coming from the light source to the given sample has to be calculated. This information is stored in a 2D texture map denoted by `shadowMap`. However, to read the appropriate pixel of the shadow map, the object-space position of each sample has to be projected onto an image plane perpendicular to the direction of the light source. This task can be shared between the vertex shader and the pixel shader. The vertex shader performs the projection for the vertices of the proxy geometry and the result is stored in texture coordinates assigned to the vertices according to Listing 3. First the vertex positions are transformed by the model-view and projection transformations. The role of `TexTrans` is to calculate the normalized 3D texture coordinates from the position of the given vertex. Transformations `LightView` and `LightProj` are initialized according to the position of the light source, and they are used to map the vertex position onto an image plane perpendicular to the direction of the light source.

It is assumed that the graphics hardware performs perspectively correct interpolation of the texture coordinates. However, `IN.TEX1` in the pixel shader represents the homogeneous coordinates of a sample point projected onto an image plane perpendicular to the direction of the light source. To calculate the corresponding Cartesian screen coordinates, a homogeneous division has to be performed in the pixel shader and afterwards the appropriate pixel position in the shadow map is calculated by translation and scaling operations. From this pixel position the intensity of the attenuated light is read (`light`) and the color of the current sample is modulated by this value. The rest of the fragment program is just responsible for the usual evaluation of front-to-back compositing. The fragment program for the back-to-front evaluation is almost the same, only the compositing operations are different.

For the simultaneous compositing of the shadow map another pixel shader is used. In each iteration step, first the current resampling slice is projected onto an image plane perpendicular to the viewing direction, and processed by the previously described fragment program. Afterwards the same slice is projected onto an image plane perpendicular to the direction of the light source, in order to update the shadow map for the next iteration (see Listing 4). Similarly to the previous case the volume data is available in a 3D texture map, while the transfer function is stored in a look-up table represented by a 1D texture. Here we also use alternating

```
fragout main(vf30 IN,
  uniform float halfWidth,
  uniform float halfHeight,
  uniform sampler3D data,
  uniform sampler1D transferFunction,
  uniform samplerRECT frameBuffer,
  uniform samplerRECT shadowMap)
{
  fragout OUT;

  float4 colorIn =
    f4texRECT(frameBuffer, IN.WPOS.xy);
  if(colorIn.a > 0.99) discard;

  float4 color =
    f4tex1D(transferFunction,
            f4tex3D(data, IN.TEX0).a);
  if(color.a < 0.01) discard;

  float4 position = IN.TEX1;
  position.xy /= position.w;
  position.x = (position.x+1.0)*halfWidth;
  position.y = (position.y+1.0)*halfHeight;
  float light = f4texRECT(shadowMap,
                          position.xy).a;

  float4 colorOut = colorIn;
  if(light > 0.01) colorOut.rgb += color.rgb
    * (color.a * (1.0 - colorIn.a) * light);
  colorOut.a += color.a - colorIn.a * color.a;
  OUT.col = colorOut;

  return OUT;
}
```

Listing 2: Front-to-back compositing fragment shader code.

```
struct OUTPUT
{
  float4 HPosition     : POSITION;
  float4 TCoords0      : TEXCOORD0;
  float4 TCoords1      : TEXCOORD1;
};

OUTPUT main(float4 Position : POSITION,
  uniform float4x4 Proj,
  uniform float4x4 View,
  uniform float4x4 LightProj,
  uniform float4x4 LightView,
  uniform float4x4 TexTrans
  )
{
  OUTPUT output;

  float4 position  = mul(View, Position);
  output.HPosition = mul(Proj, position);
  output.TCoords0  = mul(TexTrans, Position);
  float4 light     = mul(LightView,Position);
  output.TCoords1  = mul(LightProj, light);

  return output;
}
```

Listing 3: Common vertex shader code of the three passes.

```
fragout main(vf30 IN,
  uniform sampler3D data,
  uniform sampler1D transferFunction,
  uniform samplerRECT shadowMap)
{
  fragout OUT;

  float transparency =
    f4texRECT(shadowMap, IN.WPOS.xy).a;
  if(transparency < 0.01) discard;

  float alpha =
    f4tex1D(transferFunction,
            f4tex3D(data, IN.TEX0).a).a;
  if(alpha < 0.01) discard;

  OUT.col.a = transparency * (1.0 - alpha);
  return OUT;
}
```

Listing 4: Shadow map compositing fragment shader code.

floating-point render targets for high-precision compositing. Therefore the previous state of the shadow map is obtained as an input 2D texture by the fragment program. In fact, the shadow map contains a transparency value in each pixel, which is first read by a 2D texture fetch. If this value is already less than a small predefined threshold, the current sample practically does not contribute to the given pixel, therefore the execution of the program is terminated. Otherwise the opacity of the current sample is read from the look-up table. If this opacity value is under the threshold, the sample practically does not have an influence onto the shadow map in this case either, so the execution is terminated as well. Generally, the transparency of the sample is calculated and it is multiplied by the previous value of the given pixel. This fragment program is used in the rendering pass and in the first initialization pass as well, when each node calculates the shadow map of its associated subvolume. Note that it is simpler than the one applied for the frame buffer compositing, therefore it is performed more efficiently. That is the reason why the cost of the initialization pass is relatively low compared to that of the second rendering pass.

## 6 RESULTS

For our experiments we used a Hewlett-Packard “Scalable Visualization Array” consisting of five computing nodes. Each node has a dual-core AMD Opteron 246 processor, an nVidia Quadro FX3450 graphics controller, and an InfiniBand network adapter.

To illustrate the scalability of our volume rendering system, configurations of one up to four rendering nodes were investigated for four different data sets. Each setup also contained an additional display node. The data sets were visualized on a viewport of resolution $512^2$. The volumes were rotated around two

| Number of nodes | Average frame rate | Average shadow fps | Shadow overhead |
|---|---|---|---|
| 1 | 8.15 | - | 0% |
| 2 | 9.67 | 13.53 | 28.0% |
| 3 | 11.02 | 15.73 | 29.4% |
| 4 | 12.67 | 18.29 | 30.1% |

Table 1: Scalability results using the human head data set (256x256x159). Average frame rate, average shadow frame rate for all compositing contexts, and relative shadow computation and communication overhead.

| Number of nodes | Average frame rate | Average shadow fps | Shadow overhead |
|---|---|---|---|
| 1 | 6.12 | - | 0% |
| 2 | 5.73 | 19.59 | 29.3% |
| 3 | 6.74 | 21.60 | 31.2% |
| 4 | 7.67 | 21.87 | 35.1% |

Table 2: Scalability results using the frog data set (500x470x136).

| Number of nodes | Average frame rate | Average shadow fps | Shadow overhead |
|---|---|---|---|
| 1 | 4.62 | - | 0% |
| 2 | 5.08 | 17.54 | 29.0% |
| 3 | 7.35 | 25.29 | 29.1% |
| 4 | 9.71 | 27.56 | 35.2% |

Table 3: Scalability results using the Christmas tree data set (512x499x512).

| Number of nodes | Average frame rate | Average shadow fps | Shadow overhead |
|---|---|---|---|
| 1 | 7.78 | - | 0% |
| 2 | 8.68 | 35.03 | 24.8% |
| 3 | 10.03 | 35.70 | 28.1% |
| 4 | 11.95 | 40.99 | 29.1% |

Table 4: Scalability results using the stag beetle data set (416x416x247).

axes, in order to shuffle continuously the sorting order of the subvolumes. The average output frame rates, the average shadow calculation frame rates, and the relative shadow overheads are shown in Tables 1-4. The shadow calculation overhead is the ratio of the shadow rendering-compositing time and the overall rendering time. Note that the efficiency of the 2-node distribution might appear to be poor compared to the 1-node configuration. This illusory retrogression comes from the incoming shadow calculation overhead, which is not present in the 1-node setup. According to our measurements, the shadow computation overhead is around 30% including the communication on our hardware, however it is apparently growing by increasing the number of the partner nodes. The images of medical data sets rendered using 4 rendering-compositing nodes are shown in Figure 8.

# 7 CONCLUSION AND FUTURE WORK

In this paper we have demonstrated that using an object-space partitioning and high-speed communication channels between the rendering nodes, parallel volume rendering can be efficiently implemented on distributed memory architectures using even a more sophisticated optical model, which also takes light attenuation in translucent materials into account. Although our algorithm requires an initialization step to calculate, transfer, and composite the 2D shadow maps for each subvolume, due to the simple order-independent compositing operation this step can be efficiently performed. Furthermore, the high-speed channels are exploited to handle the additionally required communication overhead, which is theoretically not limited by the computing nodes when the compositing is also performed in parallel. Because of these two reasons the performance of our parallel translucent volume-rendering system is not set back by the additional computation and communication, therefore it can be improved by increasing the number of rendering nodes.

In our future work, we intend to further improve our algorithm. In the rendering phase each node has to wait until it receives all the necessary shadow maps. However, the nodes processing subvolumes shadowed by many other subvolumes have to wait longer, while the nodes corresponding to subvolumes closer to the light source can start rendering earlier. In order to improve load balancing, the calculation of consecutive frames can overlap, such that a node can render a framelet belonging to the next frame, while another node is still calculating a framelet belonging to the current frame.

# ACKNOWLEDGEMENTS

This work has been supported by the Postdoctoral Fellowship Program of the Hungarian Ministry of Education, Hewlett-Packard, and the Hungarian National Office for Research and Technology.

# REFERENCES

[BIPS00] C. Bajaj, I. Ihm, S. Park, and D. Song. Compression-Based Ray Casting of Very Large Volume Data in Distributed Environments. In *Proceedings of Fourth International Conference on High Performance Computing in the Asia-Pacific Region*, pages 720–725, 2000.

[CCF94] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 91–98, 1994.

[DH92] J. Danskin and P. Hanrahan. Fast Algorithms for Volume Ray Tracing. In *Proceedings of Workshop on Volume Visualization*, pages 91–98, 1992.

[Elv92] T. Elvins. Volume Rendering on a Distributed Memory Parallel Computer. In *Proceedings of IEEE Visualization*, pages 93–98, 1992.

[GPH04] C. Gribble, S. G. Parker, and C. Hansen. Interactive volume rendering of large datasets using the silicon graphics Onyx4 visualization system. *TR No. UUCS-04-003, University of Utah School of Computing*, 2004.

[GWGS02] S. Guthe, M. Wand, J. Gonser, and W. Straßer. Interactive Rendering of Large Volume Data Sets. In *Proceedings of IEEE Visualization 2002*, pages 45–52, 2002.

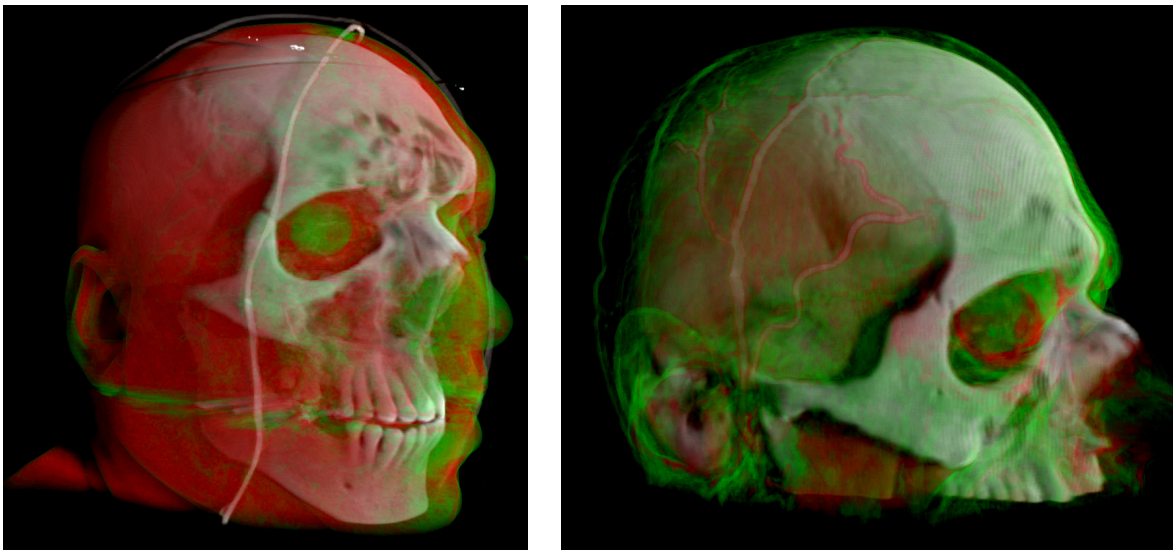

Figure 8: Images of medical data sets rendered by our parallel translucent volume rendering system.


[JG95] G. Johnson and J. Genetti. Medical Diagnosis using the Cray T3D. In *Proceedings of Parallel Rendering Symposium*, pages 70–77, 1995.

[KMM01] J. Kniss, P. McCormick, and A. McPherson. Interactive Texture-Based Volume Rendering for Large Data Sets. *IEEE Computer Graphics and Applications*, 21(4):52–61, 2001.

[KPHE02] J. Kniss, S. Premoze, C. Hansen, and D. Ebert. Interactive translucent volume rendering and procedural modeling. In *Proceedings of IEEE Visualization*, pages 109–116, 2002.

[Lac96] P. Lacroute. Analysis of a Parallel Volume Rendering System Based on the Shear-Warp Factorization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):218–231, 1996.

[Lev88] M. Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[LL94] P. Lacroute and M. Levoy. Fast Volume Rendering using a Shear-Warp Factorization of the Viewing Transformation. *Computer Graphics (Proceedings of SIGGRAPH '94)*, pages 451–457, 1994.

[LRN96] Tong-Yee Lee, C. S. Raghavendra, and John B. Nicholas. Image Composition Schemes for Sort-Last Polygon Rendering on 2D Mesh Multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):202–217, 1996.

[LWM97] P. P. Li, S. Whitman, and R. Mendoza. ParVox - A Parallel Splatting Volume Rendering System for Distributed Visualization. In *Proceedings of IEEE Symposium on Parallel Rendering*, pages 7–14, 1997.

[LY96] A. Law and R. Yagel. Multi-frame Thrashless Ray Casting With Advancing Ray-Front. In *Proceedings of Graphics Interface*, pages 70–77, 1996.

[MHE01] M. Magallon, M. Hopf, and T. Ertl. Parallel volume rendering using PC graphics hardware. In *Proceedings of Ninth Pacific Conference on Computer Graphics and Applications*, pages 384–389, 2001.

[MPH93] K. L. Ma, J. S. Painter, and C. D. Hansen. A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering. In *Proceedings of Parallel Rendering Symposium*, pages 15–22, 1993.

[MPHK94] K. L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel Volume Rendering using Binary-Swap Compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, 1994.

[PTT98] M. E. Palmer, B. Totty, and S. Taylor. Ray Casting on Shared-Memory Architectures: Memory-Hierarchy Considerations in Volume Rendering. *IEEE Concurrency*, 6(1):20–35, 1998.

[RPS99] H. Ray, H. Pfister, and D. Silver. Ray Casting Architectures for Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):210–223, 1999.

[WE98] R. Westermann and T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. *Computer Graphics (Proceedings of SIGGRAPH '98)*, pages 169–176, 1998.

[Wes90] L. Westover. Footprint Evaluation for Volume Rendering. *Computer Graphics (Proceedings of SIGGRAPH '90)*, pages 144–153, 1990.