Distributed Translucent Volume Rendering

for Hewlett-Packard Scalable Visualization Array

November, 2008

Budapest University of Technology and Economics

Department of Control Engineering and Information Technology

Contents

Chapter 1

Introduction

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 document 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 or camera- 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 display node in the third pass.

Chapter 2

Algorithmic background

2.1 Volume Rendering Background

Using traditional direct volume visualization, the classical volume-rendering integral is numerically computed by evaluating finite number of samples along the viewing rays [3]. 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 [3], 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.

Translucent volume rendering [2], 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. Our application is an efficient parallel implementation scheme for translucent volume rendering of largescale volumetric data sets.

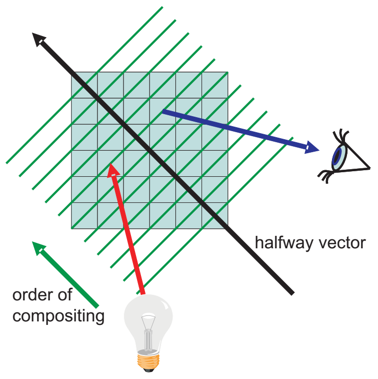

Figure 2.1: Translucent volume rendering on a single GPU using front-to-back compositing.

2.2 Translucent Volume Rendering Algorithm

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 2.1.

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.

The quality of the volumetric shadows can be enhanced with increasing the resolution of the shadow map. If the angle between the viewing direction V and the direction of the light source L is less than 90 degrees then the halfway vector H is calculated as H = (L+V)/2 and a back-to-front compositing should be used. However during the examination of a data set it is usually preferred to have the light source somewhere near our viewpoint, so use a light source that has a position fixed relative to the camera. This way this angle is always greather than 90 degrees and the halfway vector is calculated as H = (L-V)/2. In this case a front-to-back compositing is performed (see Figure 2.1).

Chapter 3

Parallel Implementation

3.1 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. 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.

In our approach a three-pass, object-parallel algorithm was used with parallel pipeline compositing. It performs the following 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 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 3.1 b.). 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

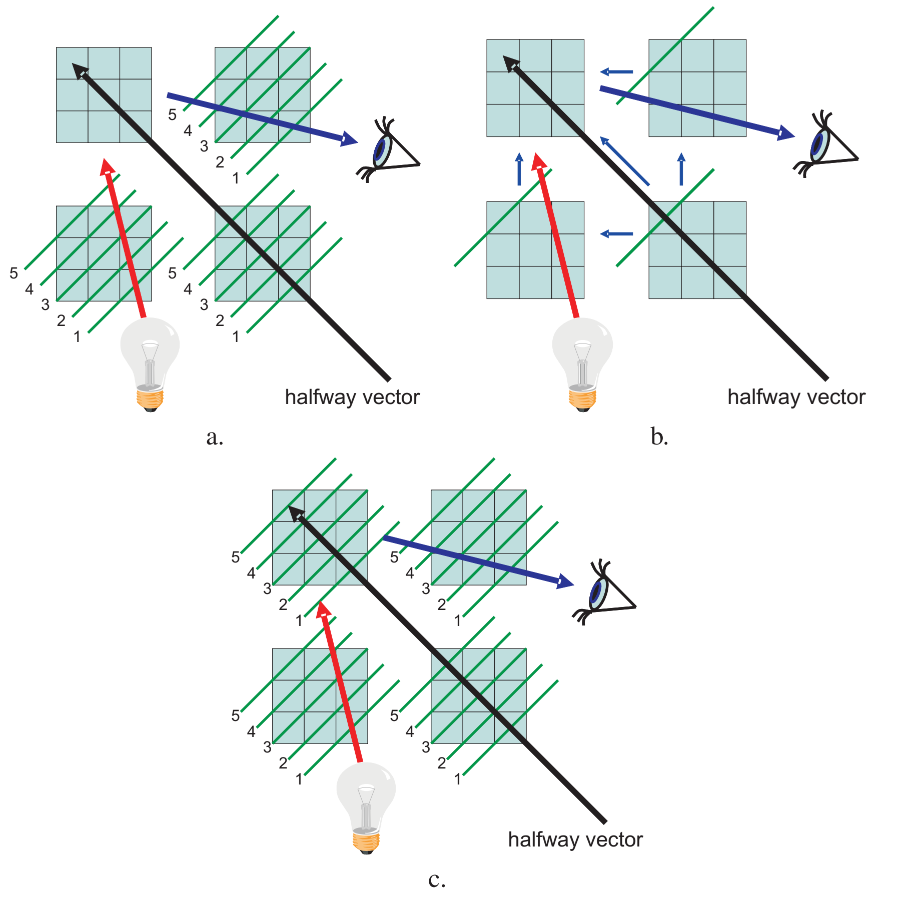

Figure 3.1: Distributed translucent volume rendering algorithm: a.:Parallel calculation of 2D shadow maps on the separate nodes. The numbers represent the different time steps. b.: Sharing the 2D shadow maps with the effected nodes for parallel compositing. c.: Parallel translucent volume rendering on the separate nodes. The numbers represent the different time steps.

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. The resulting framelets are split up, composited simultaneously, and the portions are sent to a dedicated node, which is responsible for displaying (see Figure 3.1 c.). In this third step depth-sorting is necessary before compositing, since the alpha-blending evaluation is order-dependent.

In cases of small data sets and few rendering nodes it can be more effective if the full original volumetric data is loaded by each node and a decomposition along the half way vector is used (see figure 3.2). The main reason behind the performance gain achieved with this subdivision is that the rendering speed of each node strongly depends on the render target switch time needed after rendering one slice. Though with this method the number of rendered pixels on a node does not decrease with increasing the numbers of rendering nodes, the number of render slices do. Note that in case of a larger number of rendering nodes (8 or more), the volume can be uniformly subdivided along the three main axes and fewer render slices are needed for all nodes and for arbitrary view directions.

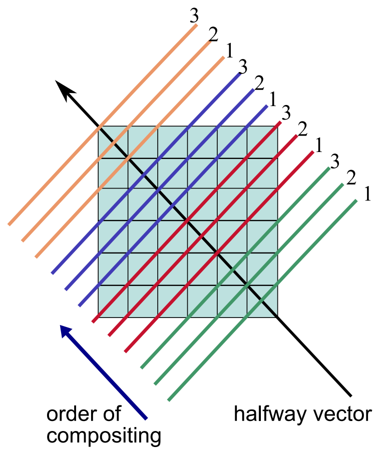

Figure 3.2: Parallel rendering of a volume using subdivision along the halfway vector. The numbers represent the different time steps. The slices with common color run on one rendering node.

3.2 Implementation Details

The volume is rendered with a texture slicing method with slices perpendicular to the halfway vector. These slices are rendered to a half precision floating point render target to enable high precision alpha blending. Shadow map rendering is performed the same way, the only difference is in the view and projection matrices (the shadow map is rendered from the lights point of view).

```
struct VS_OUT {
    float4  position : POSITION;
    float3  coord: TEXCOORD0;
    float4  lightSpacePos:TEXCOORD1;
};

float4  color_ps(VS_OUT In,
                 uniform sampler3D density,
                 uniform sampler1D transferCurve,
                 uniform sampler2D shadowMap,
                 uniform float2  halfPixel ):COLOR0
{
 if(In.coord.x < 0 || In.coord.x > 1 ||
    In.coord.y < 0 || In.coord.y > 1 ||
    In.coord.z < 0 || In.coord.z > 1 )
    discard;

 float4  color = 0;
 float  d = tex3D(density, In.coord).r;
 if(d < 0.01)
   discard;

 color = tex1D(transferCurve, d);

 float2 Lpos = In.lightSpacePos.xy / In.lightSpacePos.w;
 Lpos = Lpos * 0.5 + 0.5;
 Lpos += halfPixel;
 float  I = tex2D(shadowMap, Lpos).r;

 color.rgb *= I;
 color.rgb *= color.a;

 return color;
}
```

Listing 3.1: Front-to-back compositing fragment shader code.

The main idea behind translucent volume rendering is to alternate between shadow map render target and the final color render target slice by slice. First a slice is rendered to the color buffer then this slice is also rendered to the shadow map using a multiplicative blending simulating light absorption by the actual slice.

At the beginning the shadow map is initialized to a white color representing the unoccluded light intensity of the light source. While rendering to the color buffer the actual value of the shadow map is used to darken the shaded color which simulates volumetric shadows. After rendering the last slice the shadow map stores the absorbed light by the whole volume and the color render target stores the final image seen from the camera.

In both rendering steps the light absorbtion property of the shaded point should be determined. 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. The four channel transfer function stores the color and opacity values associated to the density values.

Listing 3.1 shows the fragment shader used while rendering to the color render target, and Listing 3.2 shows the fragment shader used for shadow map rendering.

```
struct VS_OUT {
    float4  position : POSITION;
    float3  coord: TEXCOORD0;
    float4  lightSpacePos:TEXCOORD1;
};

float4  illum_ps(VS_OUT In,
                 uniform sampler3D density,
                 uniform sampler1D transferCurve):COLOR0
{
 if(In.coord.x < 0 || In.coord.x > 1 ||
    In.coord.y < 0 || In.coord.y > 1 ||
    In.coord.z < 0 || In.coord.z > 1 )
        discard;

 float4  color = 1;
 float  d = tex3D(density, In.coord).r;
 color.a = tex1D(transferCurve, d).a;
 return color;
}
```

Listing 3.2: Shadow map rendering fragment shader code.

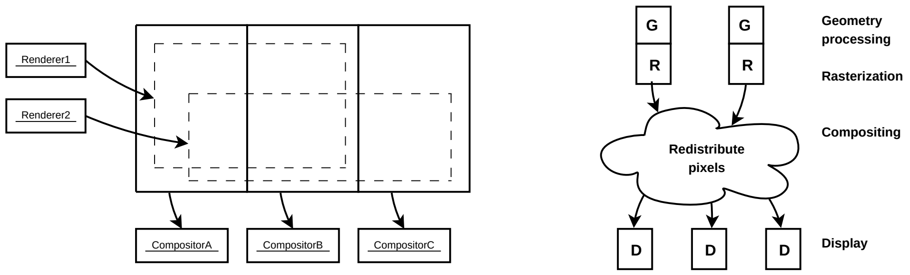

Figure 3.3: The operation of the *HP Parallel Compositing Library*

3.3 HP Parallel Compositing Library

The *HP Parallel Compositing Library (ParaComp)* is a *sort-last parallel compositing* API suitable for *hybrid object-space screen-space decomposition.* The API was originally developed by Computational Engineering International (CEI) to make its products run efficiently in a distributed environments. The latest version is based on the abstract Parallel Image Compositing API (PICA) designed by Lawrence Livermore National Lab, HP, and Chromium team.

ParaComp is a *message passing* library for graphics clusters enabling users to take advantage of the performance scalability of clusters with network-based pixel compositing without understanding its inner structure and operation. The library makes it possible for multiple graphics nodes in a cluster to collectively produce images, thus significantly larger data sets can be processed and larger images can be created than on any individual graphics hardware by distributing the load over multiple nodes.

However, there is no explicit data distribution so no load balancing is done by the API. The philosophy of the designers is keeping the API as thin as possible. Therefore, only a *global frame* is defined and one or more nodes can contribute pixels to this frame and one or more nodes can receive a specified subset of the frame. ParaComp controls the operation of the nodes based on their request; it takes the results of their renderings and generates the needed composited images (see Figure 3.3). According to the nomenclature of the API a sub-image contribution is called *framelet* and the received image area is called the *output.* These framelets and the outputs can overlap each other without any restriction to their origin or destination nodes. The attributes of a framelet are the following:

- **horizontal and vertical position** in the global frame;
- **width and height** of the framelet in pixels;
- the **data source** which can be both the system memory and the frame buffer; and
- the **depth order** of the framelets which is needed by non-commutative compositing operators like alpha blending.

The size of the output does not necessarily equal the size of the global frame. For example, each tile can be connected to a separate node in a multi-tile display. The attributes of an output are:

- **horizontal and vertical position** in the global frame;

- **width and height** of the output in pixels; and
- the **pixel data** to be returned (RGB, RGBA, RGBA+depth).

For details see the official documentation of the HP Parallel Compositing Library [1].

Chapter 4

Installation and Usage of the Translucent Volume Rendering application

This program is the SVA implementation of the translucent volume rendering algorithm. It uses NVidia's Cg toolkit for rendering and HP's Paracomp for compositing.

4.1 Installation

Both source and prebuilt versions of the application and the library can be found on the web site of the project[1].

4.1.1 Library dependencies

The following libraries are required by the application:

- **paracomp**: Hewlett Packard implementation of the Parallel Compositing API (version 1.0-beta1 or later)
- **devil**: Developer's Image Library (version 1.6.7)
- **glew**: OpenGL Extension Wrangler library (version 1.3.4 or later)
- **Cg** and **CgGL** : NVIDIA Cg library
- **gl**: library implementing OpenGL API
- **glut**: OpenGL Utility Toolkit

There are prebuilt packages for HP XC V3.2 RC1 platform for AMD64 architecture on the web site of the project for Developer's Image Library, OpenGL Extension Wrangler, Cg and CgGL libraries. If one of them is missing from the target system, it can be installed in the usual way using the `rpm` package manager program:

[1]`http://amon.ik.bme.hu/translucentvr/`

```
# rpm -i devil-1.6.7-1.x86_64.rpm
# rpm -i devil-devel-1.6.7-1.x86_64.rpm
# rpm -i glew-1.3.4-1.x86_64.rpm
# rpm -i glew-devel-1.3.4-1.x86_64.rpm
# rpm -i Cg-1.5.x86_64.rpm
```

The `XXX-devel-YYY.rpm` packages are only needed when the visualization application is built from sources. Otherwise, only the shared libraries are to be installed.

The other libraries like the Parallel Compositing library, the standard C/C++ libraries, and the OpenGL libraries are platform specific and have to be installed based on the actual software stack.

4.1.2 RPM Package

The translucen volume renderer (`translucentVR`) can be also installed from a prebuilt RPM[2] package in the same way:

```
# rpm -i translucentvr-0.1-1.x86_64.rpm
```

4.1.3 Building from Sources

The build system of the volume rendering application application is based on CMake. So, it can be built with the usual procedure:

```
$ cmake .
$ make
$ sudo make install
```

4.2 Usage

4.2.1 SVA Startup Script

A SLURM[3] startup script is provided to use `translucentVR` for parallel rendering. It can be invoked with the following command:

```
$ translucentVR.sh -r <renderers> -cf <descriptor-file>
```

The startup script has two parameters that should be set. The first one (`-r`) tells SLURM the number of *additional render nodes* to be allocated. The later one (`-cf`) sets the volume descriptor file. The config file describes the path of the volume file to load, the rgba transfer curve files, render slice count, shadow map resolution and supports a command to mirror the data set along its z axis (this was required by some

[2]Red Hat Package Manager

[3]SLURM is an abbreviation for Simple Linux Utility for Resource Management. It is an open-source resource manager designed for Linux clusters of all sizes. This software solution is used for HP-XC clusters.

of our test volumes). If ”-r n” is not given 2 rendering nodes will be set up. If ”-cf file” is not given the application will terminate.

4.2.2 Configuration File

The config file has the following format:

```
Volume File Path
Red   Channel Transfer Curve File Path
Green Channel Transfer Curve File Path
Blue  Channel Transfer Curve File Path
Alpha Channel Transfer Curve File Path
Render Slice Count
Shadow Map Resolution
"flipped" / "not flipped"
"objdiv" / "halfvecdiv"
```

Listing 4.1: Config File Format

Using the above format a sample config file would look like:

```
/home/DATA/stagbeetle.volume
/home/DATA/BeetleRed.curve
/home/DATA/BeetleGreen.curve
/home/DATA/BeetleBlue.curve
/home/DATA/BeetleAlpha.curve
600
512
 flipped
 halfvecdiv
```

Listing 4.2: Sample Config File

4.2.3 Volume Descriptor File

The volume descriptor file has two main sections. In the first one, there are name-value pairs for setting different parameters like resolution, physical size, and voxel type. In the second part the data files are listed in a sequence. The list of parameters is the following:

- `width`, `height`, and `depth` describe the dimensions of the volumetric data, i.e. the number of voxels in each dimension,
- `voxeltype` specifies the data type of the volumetric data. Currently the following values are accepted:
  - `unsigned-char` sets byte/voxel data type,
  - `unsigned-short` sets word/voxel data type,
  - `float-msb` sets IEEE 754 float/voxel data type;
- `sizex`, `sizey`, and `sizez` sets the sizes of the bounding box.

See Listing 4.3 for a sample descriptor file. The volume descriptor files for the Visible Human and the McMaster University’s data sets can be also downloaded from our data server.

```
# Resolution
width=256
height=256
depth=159

# Physical size
sizex=1
sizey=1
sizez=0.621

# Voxel type
voxeltype=unsigned-char

# Data files
kopf
```

Listing 4.3: Sample Volume Descriptor File

4.2.4 User Interface

The virtual camera can be rotated around the examined volume by holding the left mouse button and moving the mouse. The camera can be moved closer or further with the ”W” and ”S” keys. Pressing SPACE will show or hide the transfer curves.

The transfer curves can be adjusted with the mouse. Clicking on them will select the closest control point or create a new control point if no point exists nearby. Selected control points can me moved by holding the left mouse button and dragging the mouse. Selected control points can be deleted with the right mouse button.

The actual transfer curves can be saved into files by pressing ”P”, and the last saved curves can be loaded with the ”L” key.

Chapter 5

Results

For our experiments we used a *Hewlett-Packard's Scalable Visualization Array* consisting of four computing nodes. Each node has a dual-core AMD Opteron 246 processor, an nVidia 8800GTX graphics controller, and an InfiniBand network adapter.

Table 5.1 shows our results. We can conclude that distributed rendering is more effective in case of larger data sets. In case of the *Head* data set two node gives worse performance as the single computer implementation because of the network overhead, while using three render nodes this overhead becomes less significant.

In case of larger data sets - where more render slices are needed - distributed rendering unambiguously shows its benefits. The *Head* and *Beetle* data sets were subdivided along the halfway vector while in case of the *Human* data set static object space subdivision was used.

Figure 5.1, 5.2, 5.3 and 5.4 show our rendering results for some test data sets.

Data Set	Resolution	1 node	2 node	3 node
Head	$256 \times 256 \times 159$	36 FPS	33 FPS	37 FPS
Beetle	$416 \times 416 \times 247$	20 FPS	22 FPS	29 FPS
Human	$512 \times 512 \times 1877$	5 FPS	7.7 FPS	10.5 FPS

Table 5.1: Performance results.

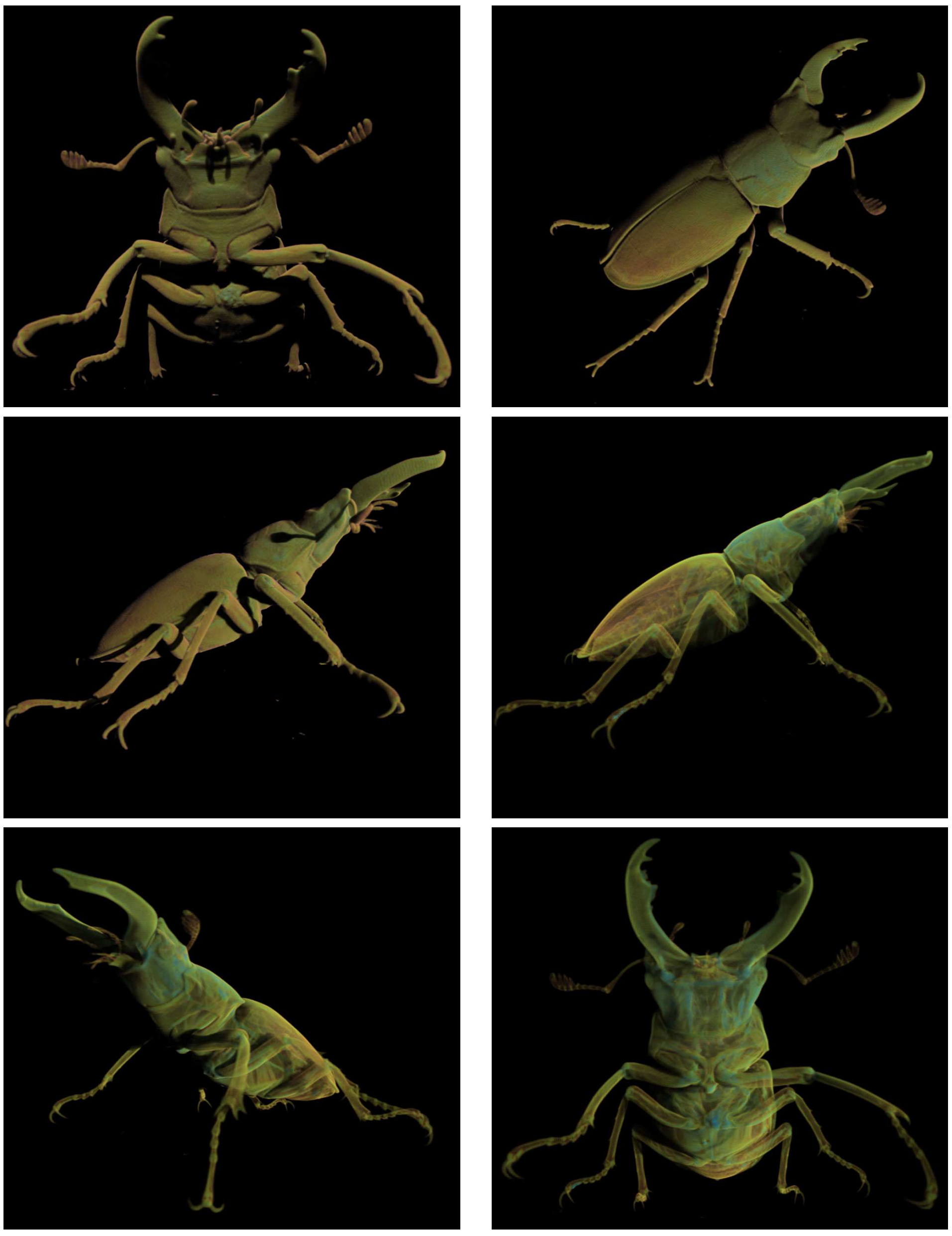

Figure 5.1: The *Stagbeetle* data set rendered with translucent volume rendering.

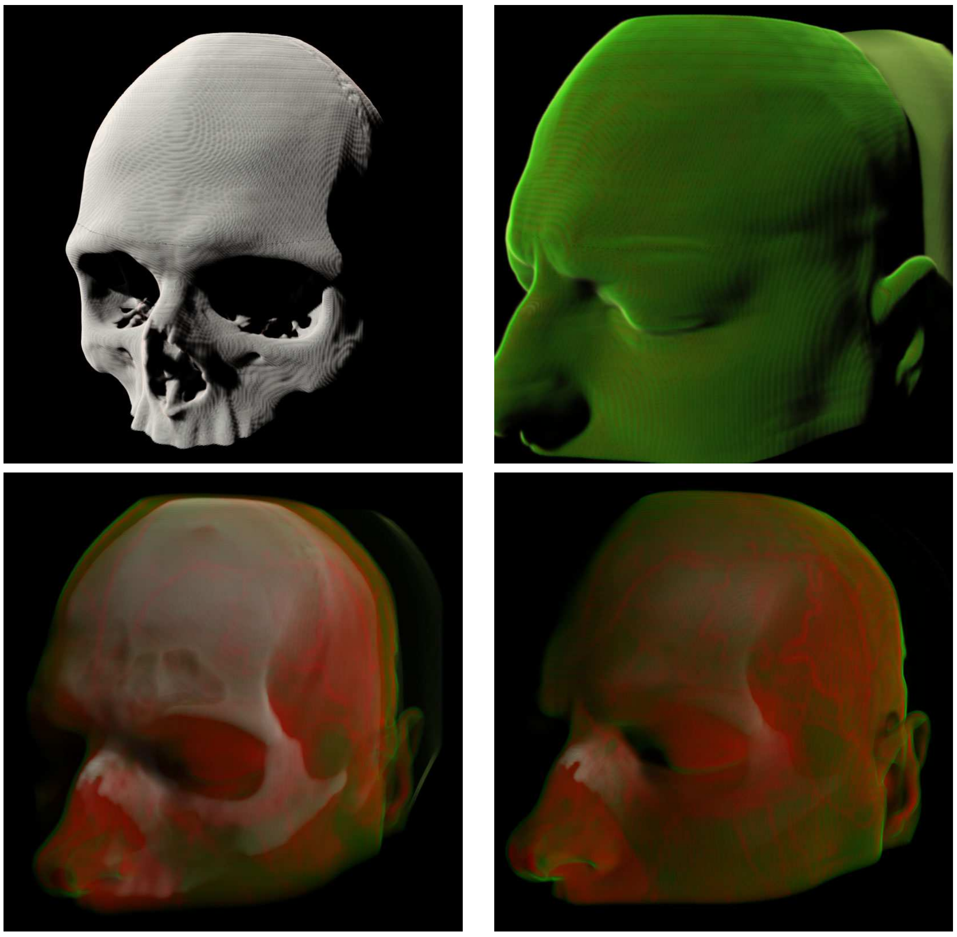

Figure 5.2: The *Head* data set rendered with translucent volume rendering.

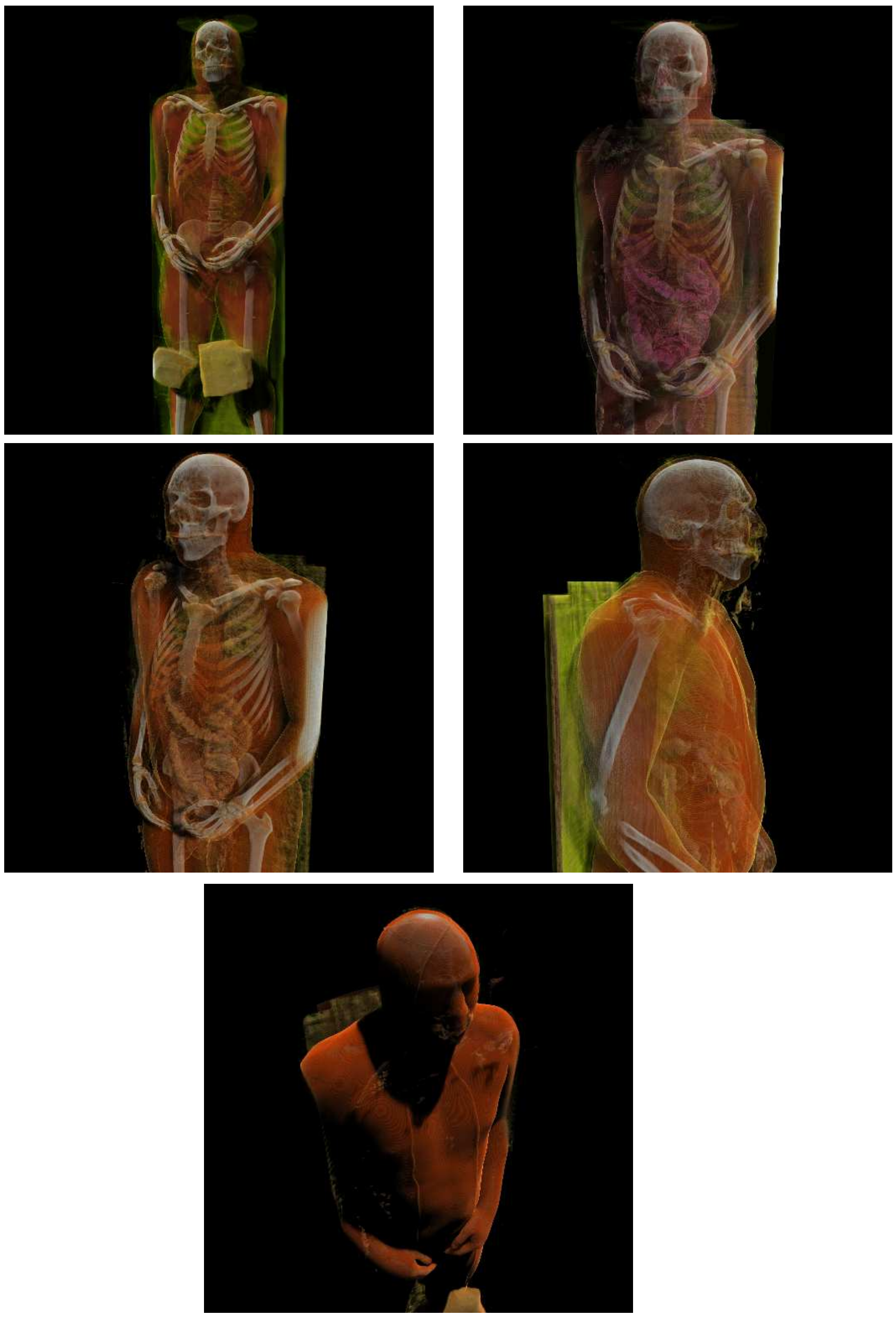

Figure 5.3: The *Visible Human* data set rendered with translucent volume rendering.

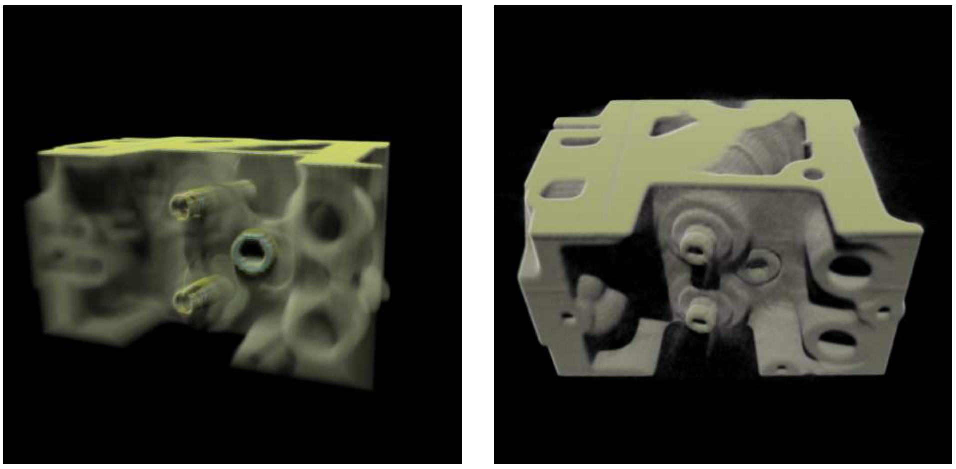

Figure 5.4: The *Engine Block* data set rendered with translucent volume rendering.

Bibliography

[1] HEWLETT PACKARD. *HP Scalable Visualization Array Parallel Compositing Library Reference Guide*, 2007.

[2] KNISS, J., PREMOZE, S., HANSEN, C., AND EBERT, D. Interactive translucent volume rendering and procedural modeling. In *Proceedings of IEEE Visualization* (2002), pp. 109–116.

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