Parallel 3D Texture Mapping Volume Renderer

for Hewlett-Packard Scalable Visualization Array

July, 2007

Budapest University of Technology and Economics

Department of Control Engineering and Information Technology

Contents

Chapter 1

Introduction

In real-time graphics, three-dimensional objects are mostly modeled with polygons or parametric surfaces. Besides, we define the optical properties of the modeled surface like color, transparency, reflection and refraction coefficients, etc. *Rendering* calculates the projections of these surfaces onto the image plane. This conventional approach is referred to as *surface rendering*.

Contrarily, *volume rendering* can be applied for direct visualization of three-dimensional scalar and vector fields. The main difference between surface and volume renderings is that no explicit geometry of these fields is given that could be easily visualized.

Volume rendering techniques were originally developed for effective visualization of measured data and simulation results. The typical scopes of volume visualization are medical applications (such as computer tomography, magnetic resonance imaging, positron emission tomography, and three-dimensional ultrasound), computational fluid dynamics, rendering geological and seismical data, visualization of abstract mathematical results or financial calculations.

In this document implementation details of a parallel volume rendering application is presented. This application uses static object-space distribution of the data among the rendering nodes and parallel compositing to get the final results.

In the second chapter of this document, the algorithmic background of the *3D texture mapping volume rendering* method is introduced. The third chapter gives a short introduction to parallel rendering techniques, it describes the parallelization approach and the details of the implementation for HP-SVA graphics clusters. Chapter 4 presents the installation and usage instructions for the mentioned platform. In Chapter 5 the structure of the code is summarized. The generated images and the measured frame rates are reported in the last chapter.

Chapter 2

Algorithmic Background

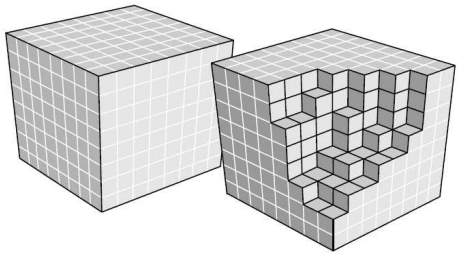


Figure 2.1: Voxels of a volumetric data sampled on a regular grid.

According to the current practice, volumetric data are generally sampled on a regular grid. In this way, the volume can be imagined as the set of small cubes (*voxels*[1]) as illustrated in Figure 2.1.

Although, the model of voxels with extent is very expressive, intrinsically, voxels are discrete samples of a continuous three-dimensional signal:

$$f_{i,j,k} = f(\mathbf{x}_{i,j,k}), \qquad f(\mathbf{x}) \in \mathbb{R}, \mathbf{x} \in \mathbb{R}^3$$

Theoretically, from these samples the original signal can be exactly reconstructed if the sampling frequency is greater than twice the signal bandwidth using an ideal low-pass filter. Unfortunatelly, the ideal low-pass filter has infinite support in spatial domain. Therefore, in practice the reconstruction is performed using only the close neighbor voxels of a sample point. For instance, *box filters*, *bilinear* (2D texture), or *trilinear filters* (3D texture) can be used for this purpose. In practical volume rendering applications, the most popular reconstruction filter is the trilinear filter, since it represents a reasonable trade-off between the quality and the rendering speed and it is natively supported by recent consumer graphics hardware which enables interpolated texture reads addressed "between" the previously stored voxels.

The volume visualization techniques can be classified into two different approaches.

1. In the first group of methods the volumetric data are first converted into a set of polygonal iso-surfaces (e.g. using the *Marching Cubes* algorithm [19]) and subsequently rendered with surface rendering hardware. This approach is referred to

[1] voxel is a portmanteau of the words volumetric and element just like pixel = picture + element

as *indirect volume rendering*. The most important disadvantage of this method is that the computationally expensive preprocessing step has to be repeated whenever the isosurface is modified.

2. On the other hand, volumetric data can be directly rendered without an intermediate conversion step. This is called *direct volume rendering*. There are four fundamentally different approaches for direct volume rendering [24].

   (a) *Image-order* methods calculate the color of each pixel separately. *Ray casting* [17] follows the inverse way of the light from the eye of the observer back to a light source.

   (b) In contrast, *object-order* methods calculate the projection of each voxel. *Splatting* [31] represents the volume as an array of overlapping basis functions, e.g. Gaussian kernels with amplitudes scaled by the voxel values. The image is then generated by projecting these basis functions to the screen.

   (c) *Shear-warp factorization* [14] had been originally proposed as a fast software implementation of direct volume rendering.

   (d) The *texture mapping* method exploits the resampling functionality of the texturing hardware on graphics cards. The resampling of the volume is performed by rendering several overlapping polygons. As most of the recent consumer graphics cards support 3D texture mapping, this approach has become one of the most popular volume-rendering techniques.

Ray casting, which can produce the highest quality of rendered images, has been implemented on different parallel architectures using either image-space or object-space partitioning [15, 20, 21, 1, 28, 30]. Similarly, the classical object-order splatting technique has also been adapted to multiprocessor environments [3, 10, 18]. The shear-warp algorithm was parallelized on an SGI system [13]. The most important drawback of 3D texture mapping is that due to limited texture memory large-scale volume data cannot be rendered without swapping sub-volumes between the main memory and the local texture memory [6]. 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 [11, 22, 5].

This document summarizes the implementation aspects of the texture mapping method parallelized on an I/O connected, message driven distributed memory architecture: the HP SVA graphics cluster.

2.1 Volume Rendering Equation

The fundamental element in volume rendering is the *volume rendering integral* that describes the light transport within the volume. For our volume rendering application we used the simplified, single-scattering optical model [23] assumes that a certain light ray only scatters once before leaving the volume. The single-scattering model computes the amount of light coming from ray direction $\mathbf{r}$ in length L to the image surface point $\mathbf{x}$:

$$I_\lambda(\mathbf{x},\mathbf{r}) = \int_0^L C_\lambda(s)\mu(s)\mathrm{e}^{-\int_0^s \mu(t)\mathrm{d}t}\mathrm{d}s, \tag{2.1}$$

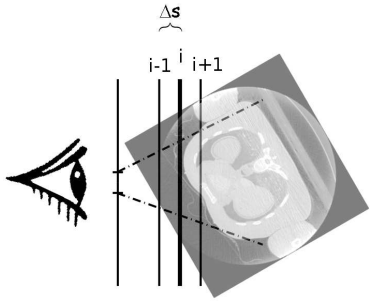


Figure 2.2: Calculating and compositing volume layers in *texture mapping volume rendering*

where $C(s)\mu(s)$ denotes the *emitted, transmitted, or reflected intensity* from a sample point of distance s in the direction of r. This intensity is then attenuated by the material between s and the image plane. The atternuation if governed by $\mu(t)$ which is the *differential absorption coefficient*. This calculation has to be performed for the whole ray.

Since Equation (2.1) cannot be evaluated analytically for the general case [23], in practical volume rendering algorithms the integral should be approximated using a discrete Riemann sum with a sample distance Δs:

$$I(\mathbf{x},\mathbf{r}) = \sum_{i=0}^{L/\Delta s} C(s_i)\mu(s_i)\Delta s \prod_{j=0}^{i-1} \mathrm{e}^{-\mu(s_j)\Delta s} \tag{2.2}$$

By approximating the exponential term with the first two terms of its Taylor series expansion we get the *discretized volume rendering equation* (DVRI):

$$I(\mathbf{x},\mathbf{r}) = \sum_{i=0}^{L/\Delta s} C(s_i)\alpha(s_i) \prod_{j=0}^{i-1} (1-\alpha(s_j)), \tag{2.3}$$

where $C(s_i)$ is emitted or reflected intensity of the discrete volume element s_i and $\alpha(s_i) = \mu(s_j)\Delta s$ is its *opacity*. Opacity is the discretized version of the absorption coefficient, which fits well to the architecture of the graphics hardware which handles an additional alpha channel besides the R, G, B color channels.

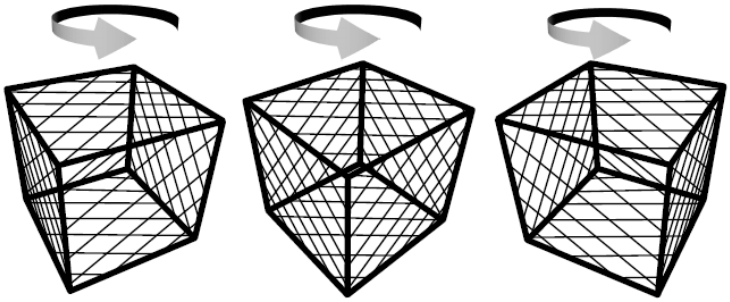
The accumulated intensity on image surface point $\mathbf{x}$ in pixel (u,v) can be calculated using Equation (2.3) where the ray is directed from the eye through the center of the pixel. This ray casting approach evaluates the sums of one after another. However, these sums can be evaluated in parallel calculating the intermediate images for $i = 1,2,\ldots,L/\Delta s$:

$$C\big(S_{i+1}(u,v)\big) = C\big(L_i(u,v)\big)\alpha\big(L_i(u,v)\big) + C\big(S_i(u,v)\big)\Big(1-\alpha\big(L_i(u,v)\big)\Big), \tag{2.4}$$

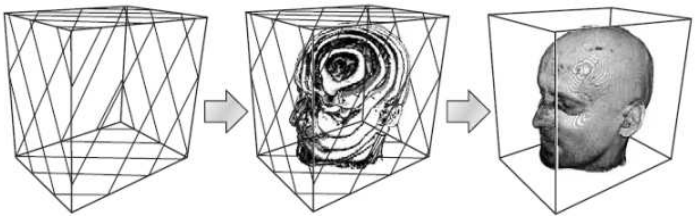
where S_i is the compound of intensity and opacity composited taking layers L_0, L_1, …, L_i in sequence (see Figure 2.2). Notation L_i stands for the *layers* to be composited and S_i indicates the *composited* color and opacity. In this way the rendering of the volume can be calculated by $C\big(S_{L/\Delta s}(u,v)\big)$.

Equation (2.4) describes the back to front compositing technique frequently applied in *texture mapping* volume rendering techniques:

$$C_{i+1} = \alpha C + (1-\alpha)C_i \tag{2.5}$$



(a) image-aligned slicing polygons



(b) rendering volume slices

Figure 2.3: Proxy geometry for 3D texture mapping volume rendering

which is referred to as the *over* operator. In OpenGL, it can be set using

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

2.2 Single-Node Implementation

The main steps of a hardware accelerated 3D texture volume rendering algorithm can be summarized as follows.

1. The volume data is *loaded into a 3D texture*. This is done once for a particular volume. When the application uses the 3D texturing capabilities of the graphics hardware, viewport-aligned slices can be rendered and the trilinear interpolation is done by the hardware.

2. The *point of view* and the *view direction* are calculated from the modelview matrix.

3. A series of polygons are computed cutting the volume perpendicularly to the viewing direction in order to define layers introduced in Equation (2.4). This is called *proxy geometry* that allows using surface rendering hardware for direct volume rendering (see Figure 2.3). The vertices of these cutting polygons are the intersection points of the bounding box of the volume and the series of planes perpendicular to the direction of view. The distance between the planes equals with the sampling distance (Δs) that usually matches the voxel dimensions of the volumetric data.

4. Each slice is *rendered as a textured polygon*, from back to front. Alpha blending operation with over operator (Equation (2.5)) is performed for each slice.

5. As the viewpoint or the direction of view change, the *slicing polygons are recomputed* before rendering a new frame.

2.2.1 Fast Proxy Geometry Calculation

When calculating proxy geometry the vertices of the generated polygons must be ordered either in clockwise or in counterclockwise order to form a valid non-self-intersecting shape. Since the number of vertices varies from zero to six, the intersection calculation can be terminated after six intersection point is found. Using an additional integer this ordering procedure can be fastened. If the plane intersects the ith edge of the bounding cube the ith bit is set of this integer. Finally, the proper ordering of the vertices can be retrieved using a single fetch from a *precalculated lookup table*. This table is indexed by a 12-bit mask where each bit is considered as a boolean variable that indicates intersection with an edge of the bounding box.

Therefore, the computational cost of a polygon is $6 \dots 12$ intersection calculation and an additional array read.

2.2.2 3D Texture Limits

The recent graphics cards have a maximal value for 3D texture resolutions which is typically 512. So the size of the largest data block that can be handled at once is $512 \times 512 \times 512$ which means that maximum 128 Megabytes of data can be handled using 1 byte/voxel data type. However, the amount of the graphics memory in these hardware elements are higher[2] than this value.

To overcome this limitation, the data is partitioned into blocks handled as separate 3D textures and rendered sequentially. However, fitting these blocks together is not trivial, since multiple samples should be avoided at the borders. In order to evaluate the interpolation in the border of the blocks, the bounding voxels of the neighbor blocks should be added to each block (Figure 2.4). But, in this way the multiplicated voxels would be composited twice or more in the final image. Therefore, the texture coordinates originally defined in $[0,1]^3$ should be cut in the

$$\left[\frac{1}{R}, 1-\frac{1}{R}\right) \times \left[\frac{1}{S}, 1-\frac{1}{S}\right) \times \left[\frac{1}{T}, 1-\frac{1}{T}\right) \tag{2.6}$$

half-closed half-opened interval, where $R \times S \times T$ is the size of the block in voxels. This can be performed in the fragment shader (see later in Listing 2.2).

2.2.3 Transfer Function

In practice, it usually not worth using the resampled data directly for shading. Applying an appropriate function can enhance the features of the data and provide more meaningful and more impressive images by mapping data values to optical properties (RGB+A color and opacity channels). This mapping function is called the *transfer function* of the visualization.

[2] typically 256 MB - 1 GB in the first half of 2007

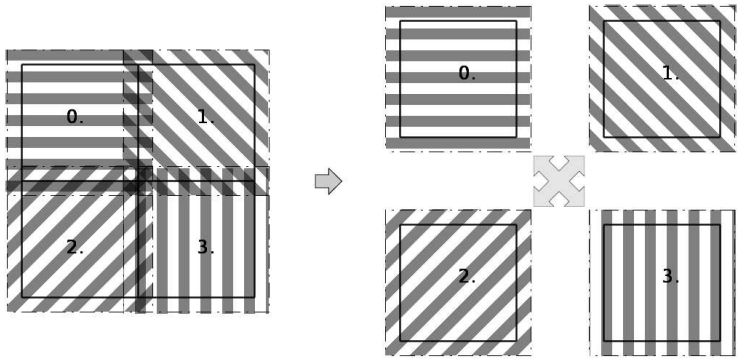


Figure 2.4: Sharing voxels between neighbor volume blocks

Such function for volumes storing scalar value can be the *pseudo-coloring*. A pseudo-color image is derived from a grayscale image by mapping each pixel value to a color. A typical example is the encoding of altitude in cartographic maps using hypsometric tints in relief maps, where negative values (below sea level) are usually represented by shades of blue, and positive values by greens and browns.

2.2.4 GPU Implementation

Texture-mapping techniques were developed to fit well to the architecture of graphics hardware. It is straightforward to implement the clipping of the texture coordinates and the shading as shader programs. Note that no texture coordinate generation is necessary on CPU when using a GPU with programmable pipeline since the object space coordinates can be copied as texture coordinates in the vertex shader. Converting to clipping coordinates are performed as usual (see Listing 2.1).

```
struct VertIn {
    float4 pos          : POSITION;
};

struct VertOut {
    float4 pos          : POSITION;
    float4 tcoord0      : TEXCOORD0;
};

VertOut main(
    VertIn IN,
    uniform float4x4 modelViewProj
)
{
    VertOut OUT;
    OUT.pos = mul(modelViewProj, IN.pos); // Calculate object coordinates in clipping space
    OUT.tcoord0 = IN.pos; // Use object space position as texture coordinates

    return OUT;
}
```

Listing 2.1: Vertex shader for texture mapping volume rendering

Listing 2.2 shows a pseudo-color transfer function implemented in the fragment shader that maps the density values from $[0,1]$ to $[0,1]^4$ of the RGBA space. The color is calculated using the HSI $\rightarrow$ RGB transformation, where the Hue equals the density to be mapped while both the Saturation and the Intensity are fixed constants. The opacity value is calculated as the power of the density using a tunable exponent. With higher exponent, the low-density samples can be removed and only the high density structure is visualized, which is usually the skeleton structure of the data.

```
float3 hsi2rgb(float H, float S, float I) {
    float r, g, b;
    float PI = 3.14159;

    if (H < 1.0/3.0) {
        b = (1-S)/3;
        r = (1+S*cos(2*PI*H)/cos(PI/3-2*PI*H))/3.0;
        g = 1 - (b + r);
    } else if (H < 2.0/3.0) {
        H = H - 1.0/3.0;
        r = (1-S)/3;
        g = (1+S*cos(2*PI*H)/cos(PI/3-2*PI*H))/3.0;
        b = 1 - (r+g);
    } else {
        H = H - 2.0/3.0;
        g = (1-S)/3;
        b = (1+S*cos(2*PI*H)/cos(PI/3-2*PI*H))/3.0;
        r = 1 - (g+b);
    }

    return float3(I * r, I * g, I * b);
}

// Note angle: 0.0->1.0
float rotate(float value, float angle) {
    value += angle;
    return value - floor(value);
}

struct fragIn {
    float3 tcoord0    : TEXCOORD0;
};

float4 main(fragIn IN,
            uniform sampler3D texture,
            uniform float colorRotation,
            uniform float alphaExponent,
            uniform float3 voxelSize) : COLOR {

    // Clip texture coordinates to avoid multiple sampling
    // on the borders
    if (IN.tcoord0.x < voxelSize.x || IN.tcoord0.y < voxelSize.y ||
        IN.tcoord0.z < voxelSize.z || IN.tcoord0.x >= 1.0-voxelSize.x ||
        IN.tcoord0.y >= 1.0-voxelSize.y || IN.tcoord0.z >= 1.0-voxelSize.z)
            discard;

    // Resample density at the sample point
    float tcol = tex3D(texture, IN.tcoord0).r;

    // Apply transfer function
    // Color: maximal saturation and intensity
    float3 color = hsi2rgb(rotate(tcol, colorRotation), 1.0, 1.0);
    // Alpha is the power of the sampled density
    float alpha = pow(tcol, alphaExponent);

    return float4(color, alpha);
}
```

Listing 2.2: Pseudo-color transfer function in a fragment shader (HSI→RGB transformation)

Chapter 3

Parallel Implementation

The process of rendering using recent consumer graphics hardware can be divided into two main stages. The *geometry processing* operates on the polygon vertices and the *rasterization* transforms these polygons into corresponding 2-dimensional points on the screen and fill in the transformed 2-dimensional triangles as appropriate. These operations are performed sequentially in a well defined order called *rendering pipeline* that introduces *functional parallelization* [4].

On the other hand, parallelization can either mean distributed handling of data and sharing them between GPUs, graphics cards, and computing nodes. *Load balancing* becomes very important when using this approach called *data parallelization*.

Rendering methods involved in data parallelism can be classified based on the graphics pipeline. In this way, the parallelization of the rendering can be defined as a data distribution and data sorting problem [25]. Sorting can be performed (1) in the beginning of the pipeline, (2) between the geometry processing and the rasterization step, and (3) at the end of the pipeline, after the rasterization. The location of the sorting fundamentally determines the required hardware architecture and communication infrastructure.

The *sort-first* approach splits the image into non-overlapping pieces and assigns the incoming primitives to the designated renderers based on the positions of the primitives in the camera frame. The benefit of this method is the low communication cost. However, it is difficult to create good load balancing. For example, when using a programmable pipeline the shaders can dynamically modify the positions of the vertices. Therefore, it is difficult to calculate which pixels will be affected by a polygon before feeding the primitives into the local pipelines [26] [9].

In case of a *sort-middle* parallelization the screen is also split into non-overlapping pieces. However, the primitives are transformed into screen coordinates on the host when they were generated, they are clipped, and then redistributed for rasterization. In case of software renderers this is a “natural” break point in the pipeline between the geometry processing and the rasterization. However, the pipelines of modern graphics cards cannot be broken to retrieve the primitives. Thus, this method cannot be applied for hardware accelerated rendering [12].

The *sort-last* method transmits the primitives through the local rendering pipelines and defers sorting after rasterization. In this case, one fraction of processes (*renderers*) are assigned to different subsets of primitives. The other type of processes (*compositors*) are assigned to areas of pixels in the output image. This method assures more treatable load balancing, but its network communication is higher than in the previous

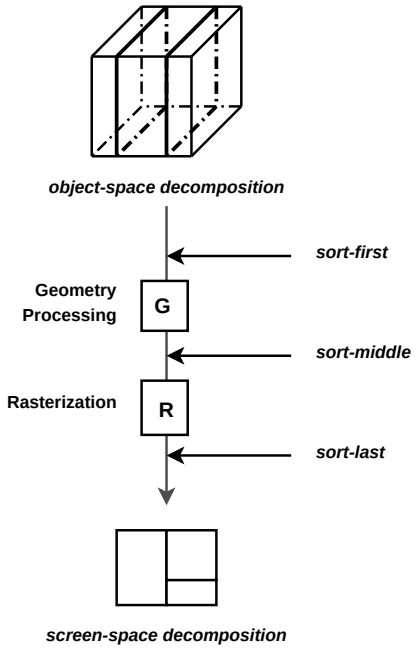


Figure 3.1: Classification of parallel rendering methods based on the *graphics pipeline* and the *simultaneously processed entities*

two approaches. This disadvantage can be moderated when only the modified subimage of the screen is transmitted to the compositors (*sparse merging*).

Another classification scheme of data-parallel rendering methods is actually based on the type of data simultaneously processed (i.e. the entities that enter and the entities that leave the conceptual, global pipeline). 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. This is called *pixel-parallel rendering*. Pixel-based parallelization can also be performed when the upper and the lower parts of the screen are rendered by different GPUs (like in NVIDIA SLI Technology) or when multiple graphics cards are used for creating tiles of the overall output image. In general this is called *screen-space decomposition*. On the other hand, when the data is divided in an initialization step, multiple subsets of graphics primitives can be processed at the same time. This is called *object-space decomposition*.

Using screen-space decomposition the image fragments can be easily joined, however object-parallel rendering needs the combination of the subsets of pixels corresponding to different objects, which is called *image compositing*. This is a simple procedure, which involves processing of pixel attributes.

Originally alpha colors were introduced as a pixel coverage model for compositing digital images [29]. The higher opacity (alpha value) the voxel has, the more it dominates the final pixel color. Besides opacity-based compositing, spatial covering can be also carried out comparing depth values, when a subset of the Z-buffer is transmitted

with the color values [2]. In this case the closest pixel is visible on the final image.

Since these calculations have to be done for all pixels, the compositing could become a bottleneck for the overall rendering system and may make it unsuitable for interactive applications. However, when the compositing is also done in parallel, *interactive compositing* is possible. There are several approaches providing parallel image compositing on multiprocessor architectures including *direct send* [8, 27], *parallel pipeline* [16], and *binary swap* [21]. These algorithms are not detailed in this report. Nevertheless, the message passing library introduced in Section 3.2 implements the parallel pipeline algorithm.

Nowadays, there are two significant trends for interactive parallel rendering. One of them based on the sort-first approach *virtually merges multiple graphics cards* and provides a *single conceptual graphics pipeline*. In this way the incoming primitives are redirected to the corresponding rendering node right after their definitions. The benefit of this approach is that applications with originally non-parallel design can be executed in a distributed environment without source code modification or moreover without recompilation. The other solution uses the *sort-last method* with *object-space data distribution* and *image compositing*. The drawback of this method is that larger modifications or redesign are required for existing applications, however the advantage is that the load balance is more predictable and designable. Since this later approach is more favorable in volume rendering methods when the amount of data dramatically determines the overall performance, the sort-last method was used.

3.1 Parallelization Approach

For the 3D texture mapping algorithm introduced in Section 2.2, *object-space parallelization* is more favorable than screen-space partitioning. The reason is twofold. First, the amount of data determines the number of slicing planes and in this way the number of texture fetches that mainly affects the overall performance. On the other hand, the memory capacity of the graphics cards are limited hence, the data must be partitioned in any way.

With object-space data distribution the load is approximately proportional to the size of the data block. Approximately, this is because a block closer to the camera has a larger image than the far ones so the number of texture fetches is higher. However, handling dynamic sized volume blocks needs continuous texture updating which is expensive.

The network traffic can be reduced when only the pixels within the axis-aligned bounding rectangle of the projection of the volume block are transferred. When the application is interactive and the orientation is not fixed, the average size of this rectangular area can be minimized when *diagonals of the volume blocks* are the shortest.

Alpha blending compositing operator (over) should be set in the sort-last compositing. To avoid multiple sampling at the borders of the volume block the same scheme clipping should be applied as presented in Section 2.2.2.

3.2 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

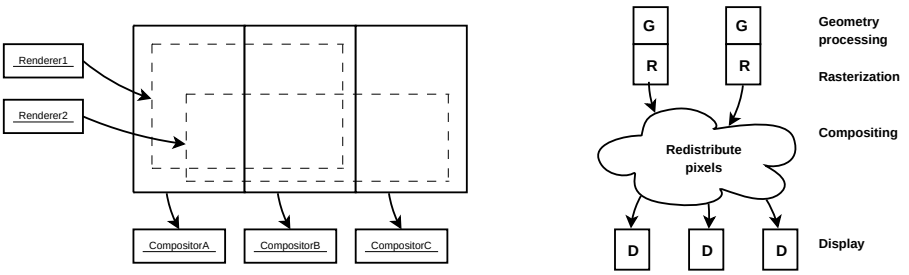


Figure 3.2: The operation of the *HP Parallel Compositing Library*

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 3.2). 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 [7].

Chapter 4

Installation and Usage of the Parallel 3D Texture Mapping Volume Renderer

The 3D texture volume rendering application was implemented based on a very thin graphics library called Minimalist OpenGL Environment. This library was designed to handle the common issues of the development of a visualization application with the possibly maximal code reusability. This library has a parallel extension that eases the implementation of a parallel visualization application.

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 volume rendering application:

- **mingle**: Minimalist OpenGL Environment library (version 0.11)
- **mingle-parallel**: the Parallel Rendering extension of MinGLE (version 0.11)
- **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
- **glu**: OpenGL Utility Library

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

- **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, CgGL, MinGLE, and MinGLE-parallel 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:

```
# 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
# rpm -i mingle-0.11-1.x86_64.rpm
# rpm -i mingle-devel-0.11-1.x86_64.rpm
# rpm -i mingle-parallel-0.11-1.x86_64.rpm
# rpm -i mingle-parallel-devel-0.11-1.x86_64.rpm
```

The `XXX-devel-YYY.rpm` packages are only needed when the volume renderer 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 3D texture volume renderer (`texturevr`) can be also installed from a prebuilt RPM[2] package in the same way:

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

4.1.3 Building from Sources

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

```
$ ./configure --with-inc-dir=<additional include directory> \
 --with-lib-dir=<additional library directory>
$ make
$ sudo make install
```

Since the only implemented parallel rendering support is the HP Parallel Compositing Library, it must be enabled. On a 64-bit HP XC platform the additional path values are the following:

- `<additional include directory>` = `/opt/paracomp/include`
- `<additional library directory>` = `/opt/paracomp/lib64`

MinGLE and MinGLE-parallel libraries can be also built from sources as follows.

[2] Red Hat Package Manager

Building MinGLE from Sources

The build system of Minimalist OpenGL Environment library is also based on GNU Autotools:

```
$ ./configure
$ make
$ sudo make install
```

Currently MinGLE supports only the GLUT windowing system. Hence, OpenGL headers and GLUT headers are needed. MinGLE is customizable, each feature can be disabled in the following way in the configuration step:

```
$ ./configure --disable-glew \
              --disable-devil \
              --disable-freetype
```

However, please note that the volume renderer uses OpenGL extensions, therefore OpenGL Extension Wrangler support should not be disabled. Please also note that the application has a graphical user interface that requires font rendering, so Developer's Image Library is also needed. Nevertheless, FreeType support can be disabled if necessary, since the fonts are read from precalculated image files.

Building MinGLE-parallel from Sources

The parallel extension can be built and installed with the following configuration options:

```
$ ./configure \
 --with-inc-dir=<additional include directory> \
 --with-lib-dir=<additional library directory>
$ make
$ sudo make install
```

The meaning of the path options is the same as the volume rendering application.

4.2 Usage

The volume renderer can be executed in non-parallel mode or in parallel mode using the SVA subsystem of the visualization XC clusters. The user interface of the program is simple; the navigation can be performed and the shader parameters can be set using the mouse device. The data to be visualized have to be a raw data file (containing only the data without extra format headers in the file). The data attributes can be described in an additional text file (see Section 4.2.5).

4.2.1 Stand-Alone Execution

After installing `texturevr`, it can be started with the `texturevr` command in *stand-alone* (non-parallel) mode, which the following command:

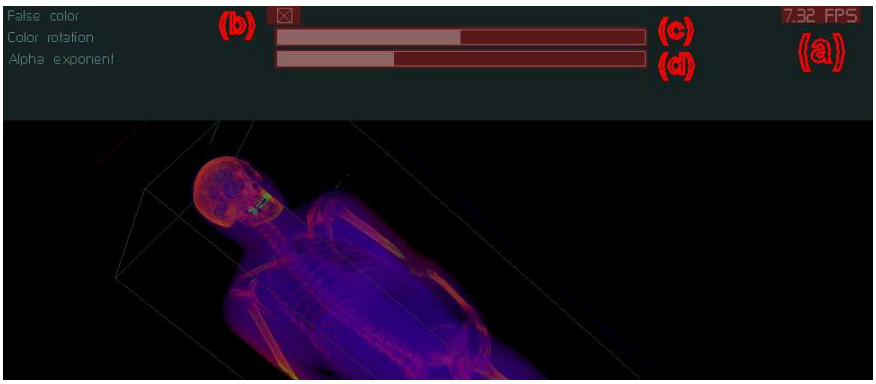


Figure 4.1: Elements of the user interface of the volume renderer

```
$ texturevr <volume-descriptor-file>
```

In this command the *volume descriptor* file is a text file that stores additional properties of the data set. This is a flexible solution to integrate different raw data sets into this visualization system. For more details on the structure of this file see Section 4.2.5.

Please note that this non-parallel rendering mode cannot be applied for visualization of large data sets. To find the maximum size the amount of graphics memory should be considered, which typically varies from 256 MB to 1 GB for recent graphics cards.

4.2.2 SVA Startup Script

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

```
$ texturevr-hpxc.sh -r|--render <renderers> --volume <descriptor-file>
```

The startup script has two parameters that should be set. The first one (`--render`) tells SLURM the number of *additional render nodes* to be allocated. The later one (`--volume`) sets the volume descriptor file, just like with the stand-alone version.

4.2.3 User Interface

The user interface of the volume renderer is rather simple (see Figure 4.1). There is a *frame rate indicator* (a) on the right side which displays the frame rates for the last second. There are three other widgets in the center of the user interface:

- a checkbox for toggling on/off the *pseudo-color* transfer function (b),
- a slider for rotating the *hue offset* for pseudo-coloring (c), and

[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.

- another slider for setting the *alpha exponent* (d).

The opacity (alpha) value is calculated as d^{exp}, where d is the resampled density value in the data and exp is the alpha exponent that can be tuned by the user interactively.

The navigation can be performed using the mouse in the following way:

- the *left button* can be used for *rotating* the scene,
- the *right button* is for *zooming*, and
- the *middle button* can be used for *translating* the scene.

The following hotkeys are defined:

- `Esc` quits from the program,
- `Tab` toggles the user interface,
- `Ctrl` + `P` creates a screen shot, and
- `Ctrl` + `R` starts/stops recording a frame sequence that can be used for creating videos.

4.2.4 Test Volumes

We used the following data sets for presenting the application. *The Visible Human Data Set* can be downloaded from the web site of the U.S. National Library of Medicine[4]. The *astrophysical data set* was created at the McMaster University[5]. The *hydrodynamical data set* was provided by the Hewlett-Packard. The other data sets are classical volume rendering testing data sets. The *present*, the *Christmas tree*, and the *stag beetle* data sets were created at the Vienna University of Technology[6]. The preprocessed version of these data sets that can be visualized with this volume rendering application can be downloaded from our data server[7].

4.2.5 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,

[4] `http://www.nlm.nih.gov/research/visible/visible_human.html`
[5] `http://www.mcmaster.ca/`
[6] `http://www.cg.tuwien.ac.at/`
[7] `http://visdata.ik.bme.hu/`

```
# Resolution
width=1024
height=1024
depth=1024

# Voxel type
voxeltype=unsigned-char

# Physical size
sizex=1.0
sizey=1.0
sizez=1.0

# Data files
# values from 0 to 255
den1020.10033.01-unsigned-char
den1020.10033.02-unsigned-char
den1020.10033.03-unsigned-char
den1020.10033.04-unsigned-char
den1020.10033.05-unsigned-char
den1020.10033.06-unsigned-char
den1020.10033.07-unsigned-char
den1020.10033.08-unsigned-char
```

Listing 4.1: Sample Volume Descriptor File (McMaster University's astrophysical data set, unsigned char 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 4.1 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.

Chapter 5

Program Structure

The main parts of the 3D texture volume renderer application are the following:

- there is a specific **scene renderer** like presented in Listing A.1 (see Appendix A) in order to fit to the MinGLE system,
- there are specific classes responsible for **volume data loading** and **volume chopping** into desired sized blocks,
- **proxy geometry** calculation is accelerated using a previously calculated lookup table,
- the **shader handling** and **3D texture handling** are performed in designated classes,
- the **Cg shader sources** are in text files, and finally
- a **GUI sheet** is responsible for tuning the shader parameters.

5.1 Scene Renderer

The scene renderer is responsible for rendering the volume. This class inherits from the `MinGLE::WindowListener` class that can be added to the listener queue of a `MinGLE::Window`. The scene renderer class does not perform any rendering itself it only calls the `render()` method of the volume object.

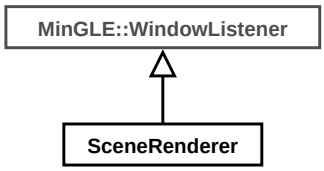


Figure 5.1: Inheritance diagram for the scene renderer

Files: `SceneRenderer.[h|cpp]`

5.1.1 Volume Object

The volume object is an administrative class referencing the *proxy geometry*, the *volume data* provider, the corresponding *3D texture*, and the applied *shader* objects. In order to overcome the 3D texture size limit of the graphics hardware the abstract `Volume` class is implemented in two ways. `SimpleVolume` can render one volume block. `VolumeSet` is a set of `SimpleVolumes`. When a volume set is called it renders all of its simple volumes in back to front order.

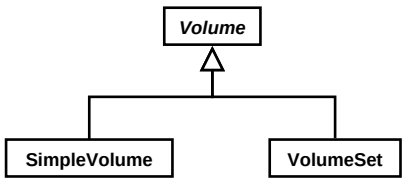


Figure 5.2: Inheritance diagram for the volume class

Files: `SceneRenderer.[h|cpp]`

5.1.2 Volume Chopping

`VolumeChopper` is a class that can produce volume blocks. It supports two chopping operations:

- it can partition a volume to N pieces with equal number of voxels and minimal diameters, and
- a volume can be chopped to smaller parts in order to satisfy the texture size limitation.

Files: `SceneRenderer.[h|cpp]`

5.1.3 Volumetric Data

In the current implementation only regular sampled volumes are supported. For efficiency, the class responsible for volumetric data loading is a template class. It is parameterized with multiple types (unsigned char, float etc.); the type of the volume loader is chosen in run time based on the volume descriptor file.

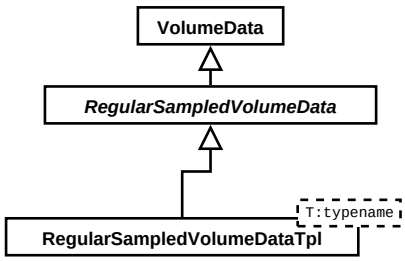


Figure 5.3: Inheritance diagram for the volumetric data

Files: `VolumeData.[h|cpp]`

5.1.4 Proxy Geometry

This implementation of the TextureVR enables using multiple type of proxy geometries. In current version, the most popular box bounded geometry is implemented with parallel slicing polygons and constant sampling distance.

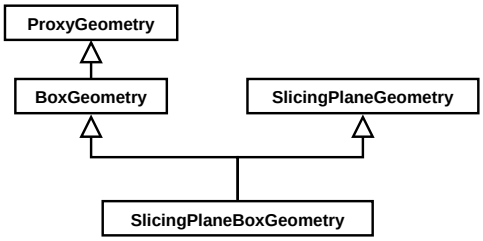


Figure 5.4: Inheritance diagram for the proxy geometry

Files: `ProxyGeometry.[h|cpp]`

5.1.5 3D Texturing

Similarly to the volumetric data loading, the 3D texture class is a template class as well that is parameterized with the type of the data and the corresponding OpenGL texture format constant.

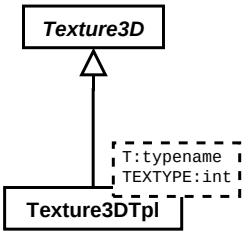


Figure 5.5: Inheritance diagram for the 3D texturing class

Files: `Texture3D.[h|cpp]`

5.1.6 GPU Shaders

Two small wrapper classes are defined for easing interaction between the host C++ code and the Cg shader code. These classes are then inherited for the specific purposes of the texture mapping volume rendering. For instance, the `TextureVRFragmentShader` class contains the parameters of the pseudo color transfer function.

Files: `Shaders.[h|cpp]`

5.2 GUI Renderer

`GuiRenderer` class manages the user interactions described in Section 4.2.3.

Files: `GuiRenderer.[h|cpp]`

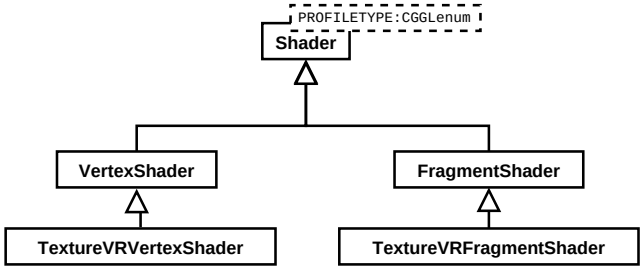


Figure 5.6: Inheritance diagram for the GPU shader classes

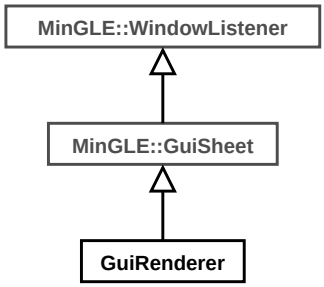


Figure 5.7: Inheritance diagram for the GUI renderer

5.3 Extending the application

The application can be easily modified for two purposes.

1. Data loader classes (`RegularSampledVolumeData` and `RegularSampledVolumeDataTpl<T>`) have to be modified to support other data formats for cases when the raw data support is not applicable.

2. To define different shaders, the Cg shader code has to be implemented, a new subclass of the `FragmentShader` or the `VertexShader` class has to be derived for handling parameter interchange between the host code the shader code and optionally a new GUI sheet can be derived from the `MinGLE::GuiSheet` class.

The overall class diagram is illustrated in Figure 5.8.

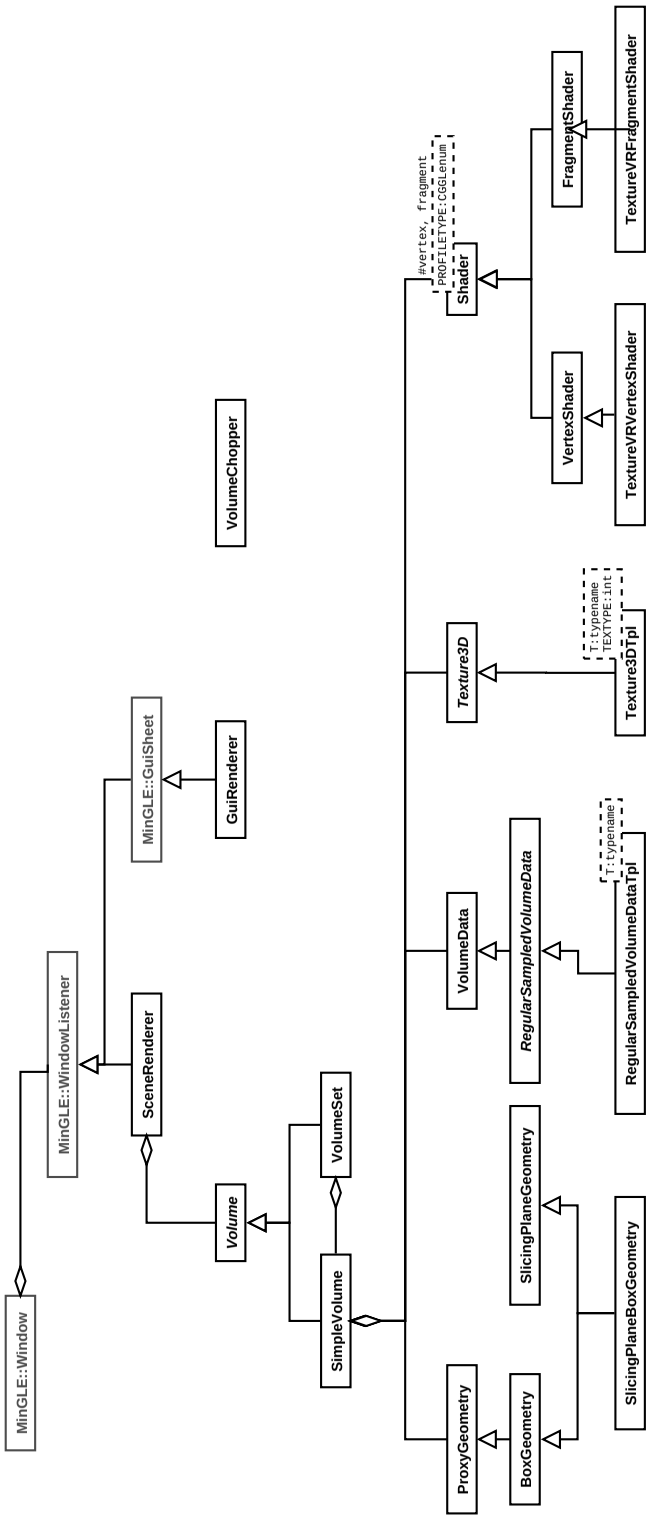


Figure 5.8: Overall class diagram of the Parallel 3D Texture Volume Renderer

Chapter 6

Results

In this section execution results are presented for a five-node HP SVA cluster. Each node had a dual-core AMD Opteron 246 processor and NVIDIA Quadro FX3450 graphics cards. The software environment was HP XC V3.2 RC1. Because of the incompatibility of the ParaComp implementation and the XC software stack only Gigabit Ethernet interconnection was available.

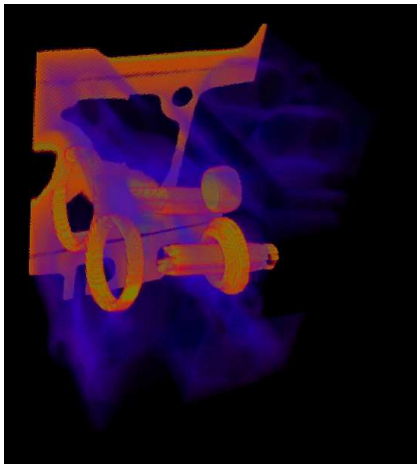
Screen shots of the execution are presented in Figures 6.2-6.6. The lobster, the engine, the frog, the head, the present, the Christmas tree, and the stag beetle illustrated in Figures 6.2-6.3 are de facto standard data sets in volume graphics. Each pixel has 2 bytes of grey tone. Figure 6.4 presents renderings of a floating point CFD data source. High resolution data sets with 8-bit depth are illustrated in Figures 6.5-6.6: *The Visible Human Male Frozen CT Data Set* which is a medical data set created from a CT scan of a real human and an astrophysical simulation generated by the McMaster University to simulate the formation of the large-scale structure of the Universe.

Frame rates for a 1024×768 viewport are presented in Table 6.1. The number of nodes was increased from 1 to 4 where it was possible. For the stagbeetle and Visible Human data set, the lower limit was 2 since the capacity of the texture memory was 256 MB for each nodes. For the same reason the astrophysical data set could be visualized only on four nodes and could not be used for scalability measurements. A rendering node was used for displaying therefore the final output when using one-node case the parallel compositing library was not used.

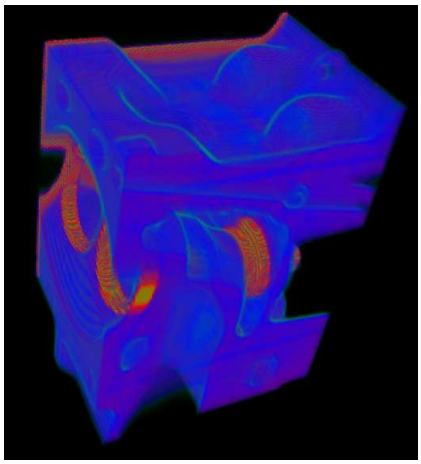
data set	size	type	data	1 node	2 nodes	3 nodes	4 nodes
lobster	$120^2 \times 34$	16-bit	≈1 MB	35.9	33.64	34.46	24.99
engine	$256^2 \times 110$	16-bit	14 MB	9.55	11.5	17.64	23.46
frog	$500 \times 470 \times 136$	16-bit	61 MB	7.69	10.06	12.23	15.8
head	$256^2 \times 159$	16-bit	20 MB	5.94	8.57	14.62	15.5
present	$492^2 \times 442$	16-bit	205 MB	1.74	2.57	3.67	4.44
xmas tree	$512 \times 499 \times 512$	16-bit	250 MB	1.4	2.46	3.34	4.53
beetle	$832^2 \times 494$	16-bit	653 MB	N/A	2.12	3.49	4.17
hydro	512^3	float	512 MB	N/A	1.63	2.87	3.45
VHP	$512^2 \times 1877$	8-bit	470 MB	N/A	0.62	2.16	2.07
McMaster	1024^3	8-bit	1 GB	N/A	N/A	N/A	0.71

Table 6.1: Frame rates and scalability (N/A indicates insufficient memory for the selected settings)

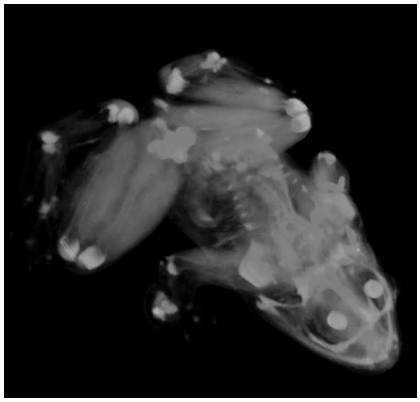
(a) lobster ($c_{rot} = 0.08, exp = 0.90$)



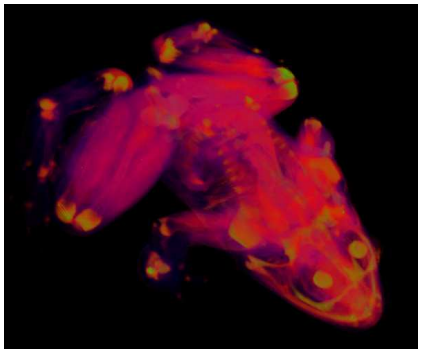
(b) engine ($c_{rot} = 0.15, exp = 7.19$)



(c) engine ($c_{rot} = 0.15, exp = 2.52$)

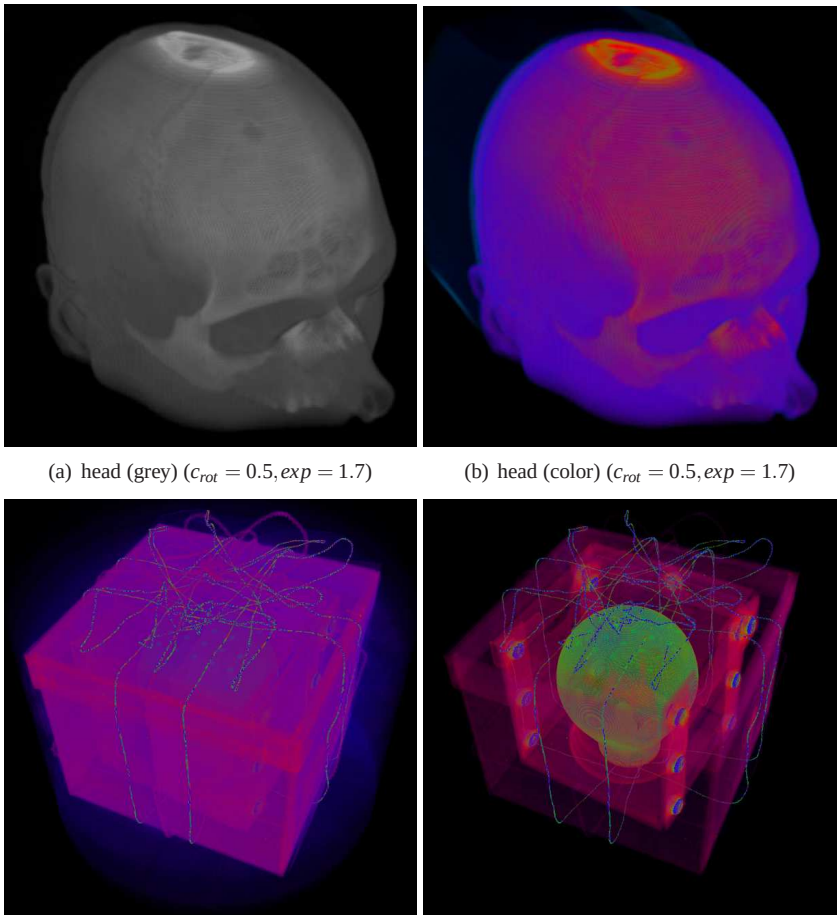


(d) frog (grey) ($c_{rot} = 0.5, exp = 4.34$)



(e) frog (color) ($c_{rot} = 0.5, exp = 4.34$)

Figure 6.1: Rendering results for the lobster, the engine, and the frog data sets



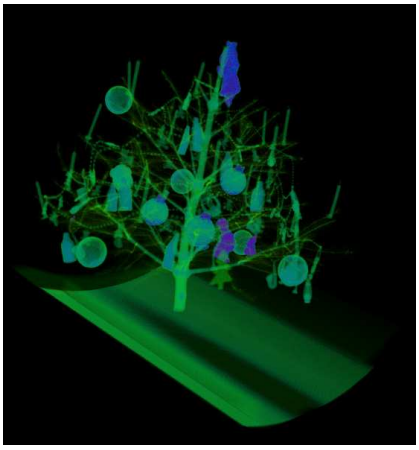
(a) head (grey) ($c_{rot} = 0.5, exp = 1.7$)

(b) head (color) ($c_{rot} = 0.5, exp = 1.7$)

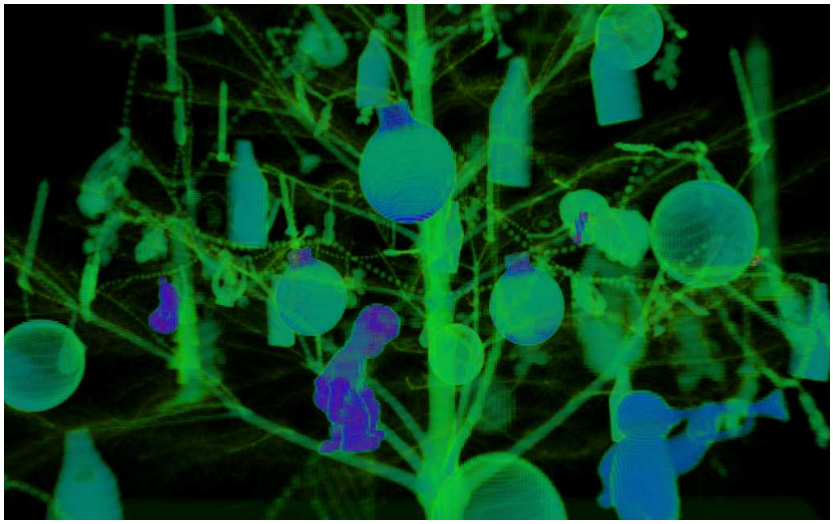
(c) present ($c_{rot} = 0.71, exp = 1.19$)

(d) present ($c_{rot} = 0.71, exp = 2.21$)

Figure 6.2: Rendering results for the head and the present data sets



(a) xmas tree ($c_{rot} = 0.21, exp = 1.38$)

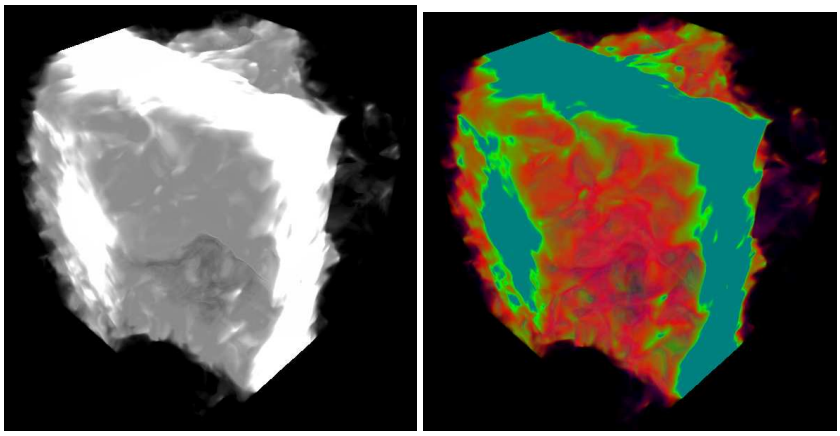


(b) xmas tree closeup ($c_{rot} = 0.21, exp = 1.38$)

(c) beetle ($c_{rot} = 0.79, exp = 1.55$)

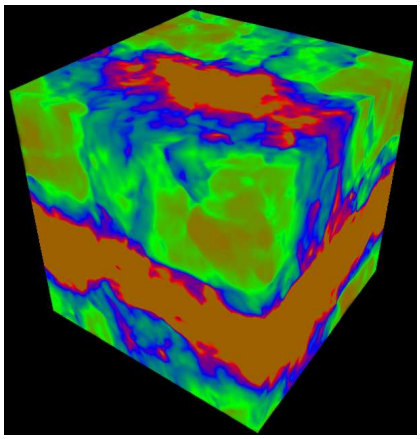
(d) beetle ($c_{rot} = 0.79, exp = 2.79$)

Figure 6.3: Rendering results for the chrismas tree and the stag beetle data sets

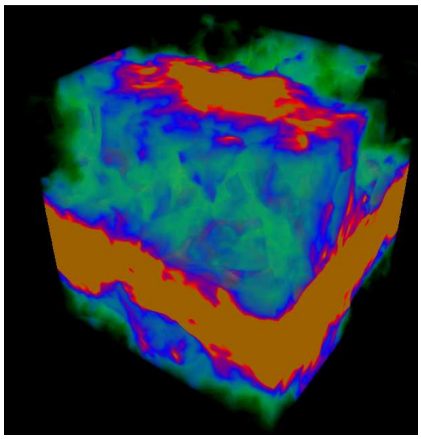


(a) hydro (grey) ($c_{rot} = 0.5, exp = 5.00$)

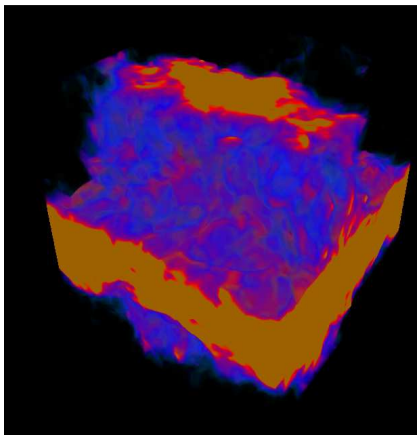
(b) hydro (color) ($c_{rot} = 0.5, exp = 5.00$)



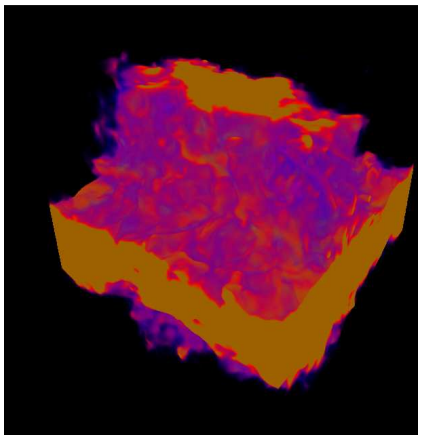
(c) hydro ($c_{rot} = 0.11, exp = 0.56$)



(d) hydro ($c_{rot} = 0.11, exp = 3.59$)

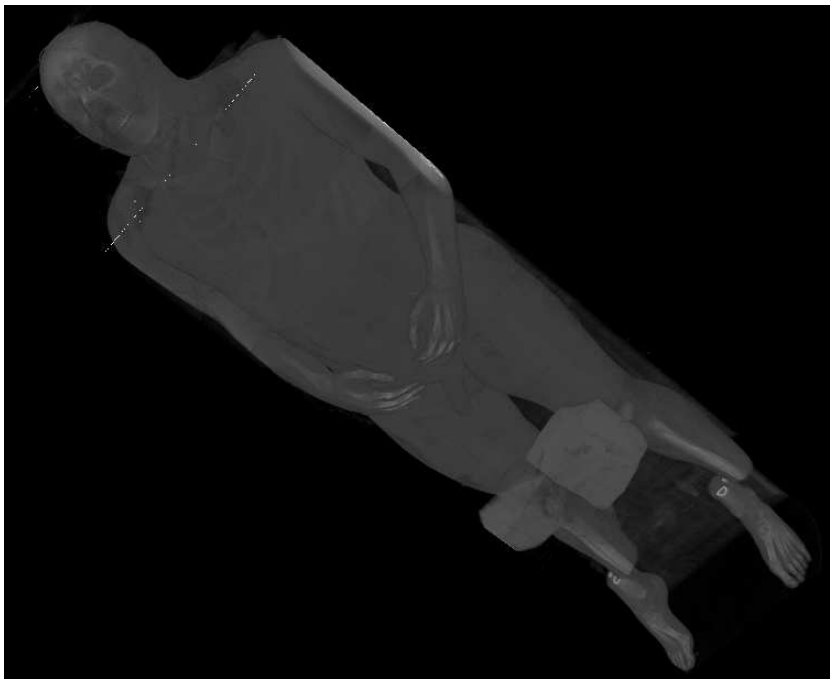


(e) hydro ($c_{rot} = 0.11, exp = 6.27$)

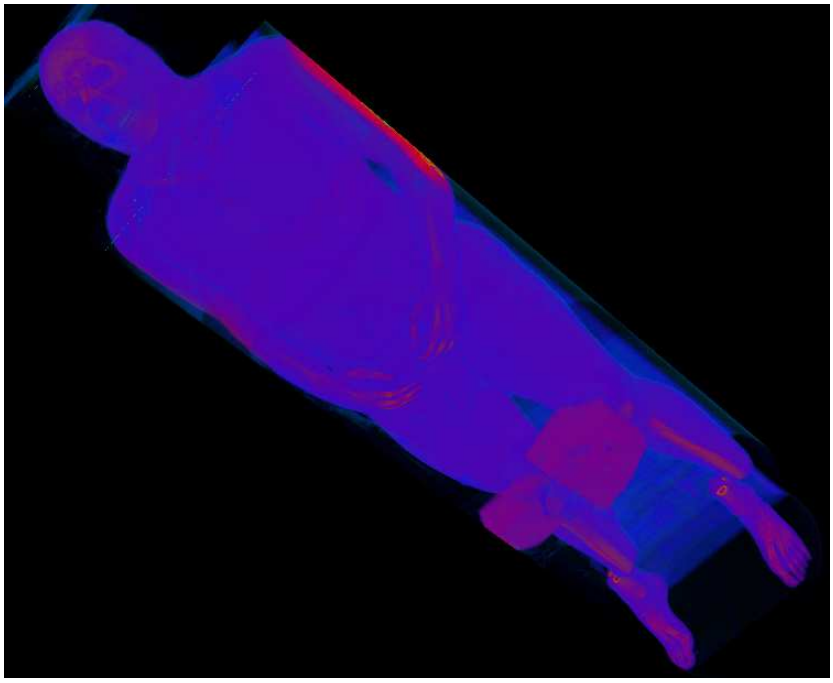


(f) hydro ($c_{rot} = 0.11, exp = 10.00$)

Figure 6.4: Rendering results for the hydrodynamical data set

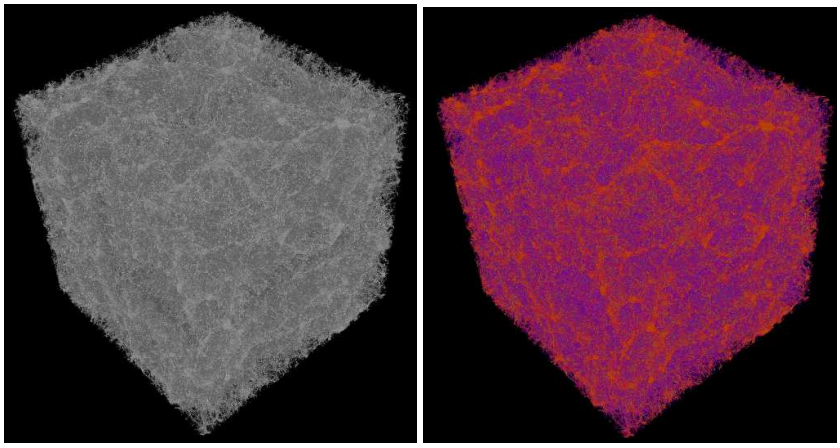


(a) VHP (grey) ($c_{rot} = 0.5, exp = 1.62$)



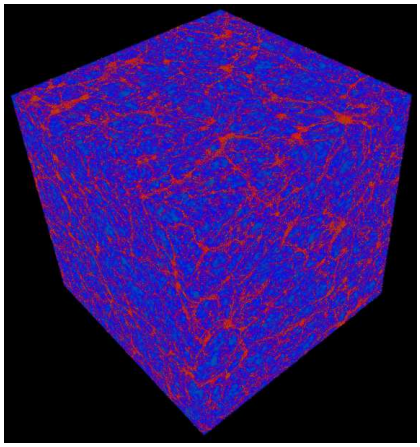
(b) VHP (color) ($c_{rot} = 0.5, exp = 1.62$)

Figure 6.5: Rendering results for The Visible Human data set

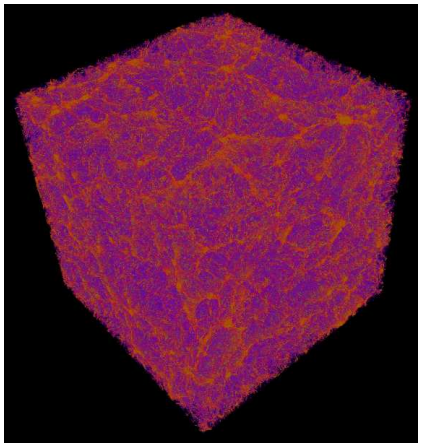


(a) mcmaster (grey) ($c_{rot} = 0.5, exp = 3.05$)

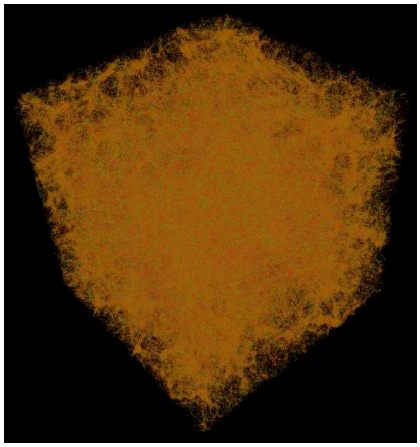
(b) mcmaster (color) ($c_{rot} = 0.5, exp = 3.05$)



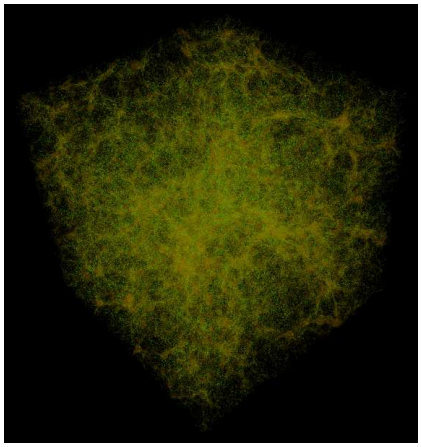
(c) mcmaster ($c_{rot} = 0.54, exp = 0.78$)



(d) mcmaster ($c_{rot} = 0.54, exp = 2.59$)



(e) mcmaster ($c_{rot} = 0.54, exp = 5.06$)



(f) mcmaster ($c_{rot} = 0.54, exp = 7.78$)

Figure 6.6: Rendering results for the McMaster University's astrophysical data set

Appendix A

MinGLE: Minimalist OpenGL Environment

The aim of this library is twofold. First, it forms a *thin object-oriented* window and event handling *wrapper layer* for GLUT, GLX, SDL and Windows systems. On the other hand, it contains basic helpers for some general common tasks involved in OpenGL based graphics applications, like camera handling and navigation, basic matrix operations, initializing OpenGL extensions, image handling, font rendering and simple user interface support, etc. Some of these features are implemented in-place and some of them use existing libraries. The overall goal is to provide platform independent aid for the very common tasks. A general API is defined, but at the moment only the GLUT platform is supported.

To get an impression of this library see the source code of a simple "Hello World!" application that renders a classic teapot object presented in Listing A.1. First, the singleton System object should be initialized. Next, a window is created to which several window listeners are added:

- our listener that overrides the `onRender()` method to render the teapot,
- a simple navigator that rotates, scales, and translates the scene based on the mouse interaction, and
- an application key handler that handles common keys for quitting, creating screen shots, and recording a frame sequence.

```
#include <mingle.h>
using namespace MinGLE;

#include <GL/glut.h>
#include <iostream>

// Custom window listener that does the rendering
class SceneRenderer : public WindowListener {
    protected:
        // This method is called when to render
        virtual bool onRender() {
            // Render a teapot using GLUT
            glutSolidTeapot (0.5) ;
```

```
            return true;
        }
};

int main(int argc, char **argv) {
    // Initializing the rendering system
    System:: initialize (&argc, argv);

    // Creating a window
    Window *win = System::createWindow();

    // Registering the window listener
    win->registerWindowListener(new SceneRenderer());

    // Adding a navigator
    ExaminerNavigator *navigator = new ExaminerNavigator();
    win->registerWindowListener(navigator);

    // Adding key handler
    ApplicationKeyHandler *keyHandler = new ApplicationKeyHandler();
    win->registerWindowListener(keyHandler);

    // Setting up OpenGL
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_SMOOTH);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

    GLfloat position [] = { 3.0f, 3.0f, 3.0f, 1.0f };
    GLfloat diffuse [] = { 0.8f, 0.8f, 0.8f, 1.0f };
    GLfloat specular [] = {1.0f, 1.0f, 1.0f , 1.0f};
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glLightfv (GL_LIGHT0, GL_POSITION, position);
    glLightfv (GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv (GL_LIGHT0, GL_SPECULAR, specular);

    // Entering the event handing loop
    System::enterMainLoop();

    return 0;
}
```

Listing A.1: Hello World! application using MinGLE

A.1 MinGLE Parallel: Parallel Extension for MinGLE

The parallel extension of MinGLE provides general support for sort-last parallel rendering in applications based on MinGLE. It *sets up compositing contexts, adds framelets, receives outputs*, and *transmits window events* to each application instance running in parallel.

The parallelized version of the previous program is presented in Listing A.2. For simplicity neither the teapot nor the screen is divided, but when more rendering nodes are added the programs renders more colored teapots appear along a circle line. All generated mouse and keyboard events are automatically transferred to the slave nodes that receive these events as if they would have been generated by real user interaction. The overall feeling of the user is that one application is running that renders several teapots.

```
#include <mingle.h>
#include <mingle-parallel.h>
using namespace MinGLE;

#include <GL/glut.h>
#include <iostream>

#include <math.h>

class SceneRenderer : public WindowListener {
    protected:
        int mThisRenderer, mRendererCount;
        double mPosition[3];

    public:
        SceneRenderer(int thisRenderer, int rendererCount) :
            mThisRenderer(thisRenderer), mRendererCount(rendererCount) {
            // Setting up position of the object
            mPosition[0] = 0.5 * ::cos(2.0*M_PI/mRendererCount*mThisRenderer);
            mPosition[1] = 0.0;
            mPosition[2] = 0.5 * ::sin(2.0*M_PI/mRendererCount*mThisRenderer);
        }

        virtual bool onRender() {
            glMatrixMode(GL_MODELVIEW_MATRIX);
            glPushMatrix();
            glTranslatef(mPosition[0], mPosition[1], mPosition[2]);

            // Drawing teapot with unique color
            int i = mThisRenderer+1;
            glColor4f( (i&1) ? 1.0 : 0.0,
                       (i&2) ? 1.0 : 0.0,
                       (i&4) ? 1.0 : 0.0,
                       1.0/mRendererCount * i);
            glutSolidTeapot(0.1);
            glPopMatrix();

            return true;
        }
};

int main(int argc, char **argv) {
    // Initializing the rendering system with parallel support
    System::initialize(&argc, argv);
    ParallelRenderingSupport::initialize(&argc, argv);

    // Creating a window
    Window *win = System::createWindow();
    win->setRenderMode(Window::RENDER_WHEN_IDLE);

    // Adding parallel rendering support to the window
    // Master/slave mode is auto-detected using the command line arguments
    ParallelRenderingSupport::addParallelSupport(win);

    // Adding key handler
    win->registerWindowListener(new ApplicationKeyHandler());

    // Registering scene renderer
    SceneRenderer *sceneRenderer = new SceneRenderer(
        ParallelRenderingSupport::getThisRenderer(win),
        ParallelRenderingSupport::getRendererCount(win)
```

```
    );
    win->registerWindowListener(sceneRenderer);

    // Add navigator
    win->registerWindowListener(new ExaminerNavigator());

    // Setting up OpenGL
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_SMOOTH);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

    GLfloat lightPosition [] = { 3.0f, 3.0f, 3.0f, 1.0f };
    GLfloat diffuse [] = { 0.8f, 0.8f, 0.8f, 1.0f };
    GLfloat specular [] = {1.0f, 1.0f, 1.0f , 1.0f};
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_COLOR_MATERIAL);
    glLightfv (GL_LIGHT0, GL_POSITION, lightPosition);
    glLightfv (GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv (GL_LIGHT0, GL_SPECULAR, specular);

    // Entering the event handing loop
    System::enterMainLoop();

    return 0;
}
```

Listing A.2: Hello World! application using MinGLE-parallel

For implementing distributed applications a general API is defined, but at currently only the ParaComp library is supported. The applications based on the parallel extension library contains both the master and the slave parts of the visualization program. When writing the code this master-slave differentiation is hidden by the underneath MinGLE-parallel library. However, the application has to be executed in two different modes in order to exploit the benefits of parallel rendering power:

```
application <sessionid> <master> <slave1> <slave2> ... <slaveN>
```

for master mode, and

```
application <sessionid> <slave_i>
```

for slave mode. Note that using an application startup script the overall distributed startup can be done in one step, too. See Section 4.2.2 for the startup script of TextureVR designed for SVA and ParaComp.

Bibliography

[1] BAJAJ, C., IHM, I., PARK, S., AND SONG, D. 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* (2000), pp. 720–725.

[2] DUFF, T. Compositing 3-D rendered images. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1985), ACM Press, pp. 41–44.

[3] ELVINS, T. Volume Rendering on a Distributed Memory Parallel Computer. In *Proceedings of IEEE Visualization* (1992), pp. 93–98.

[4] FOLEY, J., VAN DAM, A., FEINER, S., AND HUGHES, J. *Computer Graphics, Principle and Practice*. Addison-Weseley, 1993.

[5] GRIBBLE, C., PARKER, S. G., AND HANSEN, C. 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).

[6] GUTHE, S., WAND, M., GONSER, J., AND STRASSER, W. Interactive Rendering of Large Volume Data Sets. In *Proceedings of IEEE Visualization 2002* (2002), pp. 45–52.

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

[8] HSU, W. M. Segmented ray casting for data parallel volume rendering. In *PRS '93: Proceedings of the 1993 symposium on Parallel rendering* (New York, NY, USA, 1993), ACM Press, pp. 7–14.

[9] HUMPHREYS, G., ELDRIDGE, M., BUCK, I., STOLL, G., EVERETT, M., AND HANRAHAN, P. WireGL: a scalable graphics system for clusters. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM Press, pp. 129–140.

[10] JOHNSON, G., AND GENETTI, J. Medical Diagnosis using the Cray T3D. In *Proceedings of Parallel Rendering Symposium* (1995), pp. 70–77.

[11] KNISS, J., MCCORMICK, P., AND MCPHERSON, A. Interactive Texture-Based Volume Rendering for Large Data Sets. *IEEE Computer Graphics and Applications 21*, 4 (2001), 52–61.

[12] L., W. J., AND E., H. R. A proposal for a sort-middle cluster rendering system. In *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2003. Proceedings of the Second IEEE International Workshop* (2003), pp. 36– 38.

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

[14] LACROUTE, P., AND LEVOY, M. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. *Proc. SIGGRAPH'94* (1994), 451–458.

[15] LAW, A., AND YAGEL, R. Multi-frame Thrashless Ray Casting With Advancing Ray-Front. In *Proceedings of Graphics Interface* (1996), pp. 70–77.

[16] LEE, T.-Y., RAGHAVENDRA, C. S., AND NICHOLAS, J. B. Image Composition Schemes for Sort-Last Polygon Rendering on 2D Mesh Multicomputers. *IEEE Transactions on Visualization and Computer Graphics 2*, 3 (1996), 202–217.

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

[18] LI, P. P., WHITMAN, S., AND MENDOZA, R. ParVox - A Parallel Splatting Volume Rendering System for Distributed Visualization. In *Proceedings of IEEE Symposium on Parallel Rendering* (1997), pp. 7–14.

[19] LORENSEN, W. E., AND CLINE, H. E. Marching Cubes: a High Resolution 3D Surface Construction Algorithm. *SIGGRAPH'87* (1987), 163–169.

[20] MA, K. L., PAINTER, J. S., AND HANSEN, C. D. A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering. In *Proceedings of Parallel Rendering Symposium* (1993), pp. 15–22.

[21] MA, K. L., PAINTER, J. S., HANSEN, C. D., AND KROGH, M. F. Parallel Volume Rendering using Binary-Swap Compositing. *IEEE Computer Graphics and Applications 14*, 4 (1994), 59–68.

[22] MAGALLON, M., HOPF, M., AND ERTL, T. Parallel volume rendering using PC graphics hardware. In *Proceedings of Ninth Pacific Conference on Computer Graphics and Applications* (2001), pp. 384–389.

[23] MAX, N. Optical Models for Direct Volume Rendering. *IEEE Trans. Vis. and Comp. Graph., vol. 1, no. 2* (1995), 99–108.

[24] MEISSNER, M., HUANG, J., BARTZ, D., MUELLER, K., AND CRAWFIS, R. A Practical Evaluation of Popular Volume Rendering Algorithms. *Volume Visualization and Graphics Symposium* (2000), 81–90.

[25] MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl. 14*, 4 (1994), 23–32.

[26] MUELLER, C. The sort-first rendering architecture for high-performance graphics. In *Symposium on Interactive 3D Graphics: Proceedings of the 1995 symposium on Interactive 3D graphics* (New York, NY, USA, 1995), ACM Press, pp. 75 – ff.

[27] NEUMANN, U. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *PRS '93: Proceedings of the 1993 symposium on Parallel rendering* (New York, NY, USA, 1993), ACM Press, pp. 97–104.

[28] PALMER, M. E., TOTTY, B., AND TAYLOR, S. Ray Casting on Shared-Memory Architectures: Memory-Hierarchy Considerations in Volume Rendering. *IEEE Concurrency 6*, 1 (1998), 20–35.

[29] PORTER, T., AND DUFF, T. Compositing digital images. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1984), ACM Press, pp. 253–259.

[30] RAY, H., PFISTER, H., AND SILVER, D. Ray Casting Architectures for Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics 5*, 3 (1999), 210–223.

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