RTPara – Distributed Ray-Casting Application

for Hewlett-Packard Scalable Visualization Array

June, 2007

Budapest University of Technology and Economics

BME IT^2

2007 Budapest University of Technology and Economics,
Innovation and Knowledge Center of Information Technology.
http://www.it2.bme.hu/

Contents

1 Introduction **3**

2 Algorithmic Background **4**
2.1 Ray Casting on a Single Node 6
2.1.1 Empty Space Skipping 7
2.1.2 Front-to-Back Evaluation 7
2.1.3 Intersection Refinement 8
2.1.4 Fast Third Order Filtering 9
2.1.5 Multidimensional extension 12
2.2 Gradient Estimation and Shading 14
2.3 Curvature Estimation for NPR (Non-Photorealistic Rendering) . 15

3 Parallel Implementation **21**
3.1 Parallelization Approach 21
3.1.1 Screen space 21
3.1.2 Object space 22
3.2 ParaComp 22

4 Installation and Usage **24**
4.1 Installation 24
4.1.1 Library Dependencies 24
4.2 Execution and Usage 25

5 Program Structure **28**
5.1 Algorithm 29
5.2 Parallelization 30
5.3 Display 30
5.4 Volume loader 30

6 Results **32**

Chapter 1

Introduction

This document gives an overview of the parallel isosurface ray-casting application called *RTPara* developed by BME IT^2 [1]. The application allows the visualization of volume datasets in a distributed environment by rendering isosurfaces using screen space or object space decomposition approaches. Rendering parameters like the viewing direction, isosurface threshold, etc. can be interactively modified, the program provides an immediate visual feedback.

In the *second chapter* of this document the implemented isosurface ray-casting algorithm is presented in detail. The *third chapter* discusses the parallelization approaches (screen space and object space) and presents their realization using the ParaComp library [2]. The *fourth chapter* provides an overview of the installation and usage of the application. The *fifth chapter* describes the structure of the program and details the relevant implementation aspects. The *sixth chapter* contains numerical results using various datasets and scenarios.

[1] `http://www.it2.bme.hu`

Chapter 2

Algorithmic Background

Volume rendering is a technique used to display a 2D projection of a 3D discretely sampled data set (sampled representation of a 3D continuous signal). A typical 3D data set is a group of 2D slice images acquired by a CT (Computed Tomography) or MRI (Magnetic Resonance Imaging) scanner. Usually these are acquired in a regular pattern (e.g. one slice in every millimeter) and usually have a regular number of image pixels in a regular pattern. This is an example of a regular volumetric grid, with each volume element or voxel represented by a single value.

To render a 2D projection of the 3D data set, we need to assign an opacity value and a color value to every voxel. These are usually defined using a function that determines the RGBA (red, green, blue, alpha) values for every possible voxel value (*transfer function*).

A volume may be rendered by displaying surfaces of equal values (*isosurface*). Isosurfaces can be extracted in two fundamentally different ways, which are represented by *direct* and *indirect* methods. Indirect volume-rendering methods create an intermediate geometrical representation of an isosurface from the volumetric data, which can be rendered by using the traditional surface-rendering techniques. Apart from the rough approximation, the most important drawback of such an indirect method is that the computationally expensive preprocessing has to be repeated whenever the user modifies the isosurface threshold.

In contrast, using direct volume rendering, an isosurface can be implicitly extracted by resampling the volume data along the viewing rays at evenly located sample points. Rays are cast from the view point through the center of each pixel and the first point where the ray intersects the isosurface is determined.

While volume data is a sampled representation of a continuous signal, the evaluation of a density sample at an arbitrary sample position needs the recovery of this continuous signal from its samples (*reconstruction*). The Whittaker-Shannon interpolation formula states that under certain limiting conditions the continuous signal can be recovered exactly from its samples by an ideal low-pass filter. An ideal low pass filter can be realized mathematically (theoretically) by multiplying the signal of samples by the rectangular function in the frequency domain or, equivalently, by the convolution with a sinc function (see Figure 2.1) in the time domain.

The sinc function is not practical since it has infinite support, has negative values, and may result in ringing. Thus for practical cases approximations of

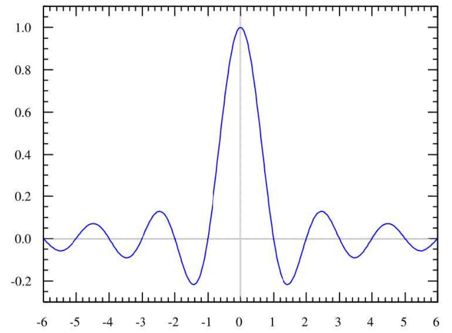


Figure 2.1: The normalized sinc function ($\frac{sin(\pi x)}{\pi x}$)

the ideal low-pass filter are used, which are not negative and have finite support. The graphics hardware directly supports piece-wise constant (box filter) and tri-linear filtering. These correspond to zero-order and first-order filter schemes. However, these filters are very far from the ideal reconstruction filter, thus their application results in sampling artifacts. For more accurate reconstruction, higher order polynomial filters are needed.

In addition to the determination of the visible point of the isosurface, the point has to be shaded, therefore a local surface characteristics are calculated for each intersection point. These characteristics include the surface normal, that is the derivative of the data, and might include curvatures associated with the second derivatives. While the reconstruction can be expressed as a convolution, from the properties of convolution comes that the reconstruction of the derivative by a given filter can be replaced by the reconstruction of the original data with the derivative of the filter. Note that this also means that the appropriate order derivatives of the filter are needed so the box filter cannot be applied when derivatives are reconstructed, and the tri-linear filtering is also bad if second order derivatives (i.e. curvatures) need to be obtained. Considering the restrictions mentioned above and for the sake of efficiency, we use the partial derivatives of a third-degree, i.e. cubic reconstruction filter, which is applied for density reconstruction and during its implementation the bi-linear filtering units of graphics hardware can be exploited (see Section 2.1.4).

In this document we present the implementation of a direct volume rendering algorithm. The flow of the algorithm is:

- compute the intersection points of the rays with the implicit isosurface using ray-casting,
- compute first and second order derivatives for the found intersection points,
- generate (sub)image applying the chosen shading model.

Since the pixel colors of the final image are computationally independent of each other (intersection points, derivatives, etc. may be computed independently) the computationally intensive parts can be implemented on the GPU (Graphics Processing Unit) exploiting the performance and parallelism offered by the modern, programmable GPUs. In general, when decomposing multipass

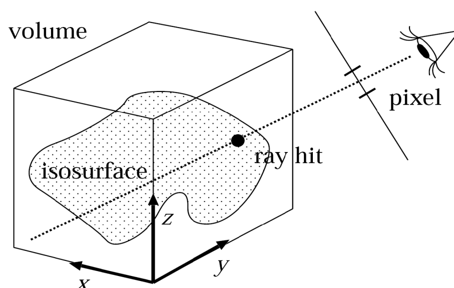


Figure 2.2: Ray casting volume isosurfaces

algorithms for GPU implementation, partial results, precomputed data, etc. are stored in textures and accessed by the corresponding vertex and fragment programs of the subsequent phases. Various rendering and computation phases employ different vertex and shader programs for accomplishing the necessary computations. The following sections detail the main steps.

2.1 Ray Casting on a Single Node

The basic concept of GPU-based ray casting is to store the volume data in a 3D texture map and resample it along the rays in the fragment program [6] (see Figure 2.2). Suppose that the texture is the sampled representation of a scalar field $f(\mathbf{x})$ that assigns a scalar density value to every 3D point $\mathbf{x}$.

Isosurface rendering selects a surface from the scalar field, where the density equals to a user specified threshold s. Thus the points $\mathbf{x}$ of the surface to be rendered is given by equation

$$f(\mathbf{x}) = s. \tag{2.1}$$

If a virtual camera is place in the same space where the scalar field is defined, the eye position and the pixels of camera window define rays. The ray of pixel x, y is defined by the following parametric equation:

$$\mathbf{p}(t, x, y) = \mathbf{c} + \mathbf{d}(x, y)t, \tag{2.2}$$

where x and y are the screen-space coordinates of the given pixel, $\mathbf{c}$ is the center of the camera, $\mathbf{d}(x, y)$ is the direction of the corresponding ray, and t is the ray parameter.

The ray contains those points that are projected onto the respective pixel. Since we are interested in visualizing the isosurface, the displayed point should also be on the isosurface of equation 2.1. Substituting the ray equation into this surface equation, we obtain:

$$f(\mathbf{p}) = f(\mathbf{c} + \mathbf{d}(x, y)t) = s.$$

The objective of ray casting is to solve this equation for unknown point $\mathbf{p}$ and non-negative ray parameter t. If there are more than one solutions, that is, the ray intersects the isosurface more than once, we need the visible intersection that has the minimal, non-negative ray parameter.

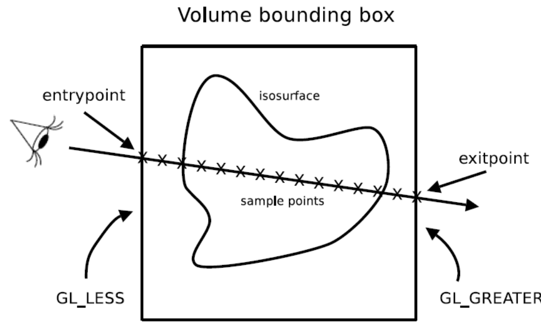


Figure 2.3: Ray marching between the entry (GL_GREATER) and (GL_LESS) exit points

2.1.1 Empty Space Skipping

Since the volume data fits into a cube, the sampling along a ray has to be done between the points where the ray enters and exits this bounding cube. In order to find the entry and exit points for the ray of each pixel of the final image, the bounding cube is rendered in two separate passes and the entry an exit point parameters are stored for the subsequent ray-casting phase in textures [1]. First, the front faces of the cube are rasterized enabling depth test with GL_LESS. We assign volume-space coordinates as three color components to the bounding boxes, therefore inside a scanline the color of each pixel represents entry point $\mathbf{p}_{entry}(x, y)$ of the corresponding ray into the cube (it is assumed that the graphics hardware performs non-distorted perspectively correct color interpolation, thus the vertex attributes are linearly interpolated in homogeneous space, and a homogeneous division is executed for each pixel inside the scanline). Similarly, exit points $\mathbf{p}_{exit}(x, y)$ of the rays are determined by rendering the back faces of the cube enabling the depth test with GL_GREATER. The ray direction $\mathbf{d}(x, y)$ is calculated as the difference between the exit point $\mathbf{p}_{exit}(x, y)$ and entry point $\mathbf{p}_{entry}(x, y)$ (see Figure 2.3).

2.1.2 Front-to-Back Evaluation

After having the entry and exit points of each ray, the rays are marched, i.e. evaluated in front-to-back order at evenly located sample positions started from point $\mathbf{p}_{entry}(x, y)$. Therefore the ith sample position $\mathbf{p}_i(x, y)$ along ray direction $\mathbf{d}(x, y)$ is calculated as follows:

$$\mathbf{p}_i(x, y) = \mathbf{p}_{entry}(x, y)(\mathbf{p}_{exit}(x, y) - \mathbf{p}_{entry}(x, y))i/N, \tag{2.3}$$

where N is the number of samples. Note that this approach evaluates the same number of samples between the entry and exit points no matter how far they are from each other. We can also normalize the ray direction to guarantee that all rays are marched with the same increments, and use smaller number of samples when the entry and exit points are close. The front-to-back evaluation is implemented as a loop in the fragment shader, which is performed separately for each pixel. This approach avoids multi-pass rendering for this phase, unlike the classical texture-slicing, which uses alternating p-buffers and a separate pass

for each resampling slice. The following simplified fragment shader program illustrates the principle of the ray marching process for finding the intersection points.

```
float4 intersection(
        in float2 unit_tc : TEXCOORD0, // shaded pixel in
                                       // texture coordinates
        uniform float isovalue,        // isovalue of the
                                       // surface to be
                                       //  rendered
        uniform sampler3D volume_tex,  // volume to be
                                       // visualized
        uniform samplerRECT entry_tex, // precomputed entry
                                       // points
        uniform samplerRECT exit_tex   // precomputed exit
                                       // points
      ) : COLOR0                       // ray - isosurface
                                       // intersection
{
  float4 entrypoint = tex2D(entry_tex, unit_tc);
  float4 exitpoint = tex2D(exit_tex, unit_tc);
  float4 raydir = exitpoint - entrypoint;
  float4 hit;
  bool intersectionFound = false;
  // advance along the ray to find the first ray-isosurface
  // intersection
  for(float t = 0; t <= 1.0f; t += dt) {
    float4 p = entrypoint + raydir * t;
    float v = tex3D(volume_tex, p).r;

    // value above isovalue: intersection is found
    if (v > isovalue && !intersectionFound) {
      hit = p;
      intersectionFound = true;
    }
  }
  return hit;
}
```

Listing 2.1: Fragment shader for finding ray–isosurface intersections

2.1.3 Intersection Refinement

Suppose that ray marching detects an intersection point between sample positions $\mathbf{p}_i(x, y)$ and $\mathbf{p}_{i+1}(x, y)$. This is possible if the volume density is below isovalues in the first point and above in the second, that is $f(\mathbf{p}_i(x, y)) < s$ and $f(\mathbf{p}_{i+1}(x, y)) \geq s$, where $f(\mathbf{p})$ denotes the density function at $\mathbf{p}$ and s is a threshold defining the isosurface.

A refined intersection point can be calculated by using the following secant root searching algorithm:

```
float3 intersectionRefinement(
            float3 pNear,
            float3 pFar,
            float isovalue,
```

```
                  uniform sampler3D volume_tex //volume to
                                               // be visualized
                 )
{
  float3 pNew;
  float fNear, fFar;
  int i;
  int iterations = 10;
  for(int i = 0; i < iterations; i++)
  {
    fNear = tex3D(volume_tex, pNear).r;
    fFar  = tex3D(volume_tex, pFar);
    pNew = (pFar - pNear) * (isovalue - fNear) /
           (fFar - fNear) + pNear;
    if(tex3D(volume_tex, pNew) < isovalue)
        pNear = pNew;
    else
        pFar = pNew;
  }
  return pNew;
}
```

Listing 2.2: Intersection refinement

Function `intersectionRefinement` is called by arguments `pNear` $= \mathbf{p}_i(x, y)$ and `pFar` $= \mathbf{p}_{i+1}(x, y)$.

2.1.4 Fast Third Order Filtering

Regarding the quality of isosurface rendering the applied resampling technique is crucial. Generally the wider the support of the reconstruction filter, the better its quality is. On the other hand, by increasing the support of the filter kernel a convolution with it is getting more and more expensive computationally. In practical volume-rendering applications the most popular filter is the tri-linear filter, since it represents a reasonable trade-off between quality and the rendering speed, and it is directly supported by the graphics hardware. The most important drawback of tri-linear interpolation, however, is that it produces discontinuous derivatives. Alternatively, gradients can be calculated at grid points using a more sophisticated estimation technique, and they can be tri-linearly interpolated between the grid points. Nevertheless, this approach drastically increases the storage requirements. Furthermore, some of the color transfer functions and non-photorealistic volume-rendering techniques take also second derivatives into account, which can hardly be estimated by a linear filter. Therefore, to make our implementation generally usable with different rendering models, we apply a high-quality third-order filtering technique [5]. The technique presented below is highly optimized for GPU implementation since it minimizes the required texture memory fetches.

To discuss how the bi-linear filtering units can be exploited to minimize the number of texture memory fetches when implementing a higher order filter, let us consider the problem for a one-dimensional signal. We shall see that the extensions to two-dimensional images and particularly for three-dimensional volumes are straightforward.

The cubic reconstruction of a 1D signal can be formulated at an arbitrary position x as a weighted sum of the signal values at the nearest four sample

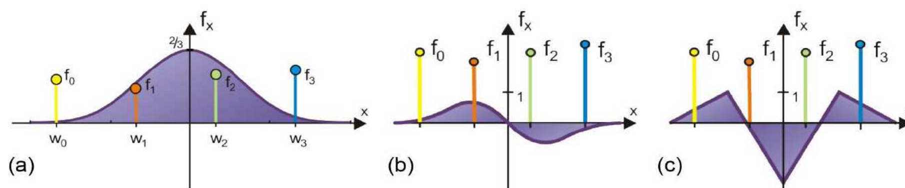


Figure 2.4: (a) Convolution of input samples f_i with filter weights $w_i(x)$. First-order (b) and second-order (c) derivatives of the cubic B-spline.

positions (see Figure 2.4/a):

$$f(x) \approx \tilde{f}(x) = w_0(x)f_{i-1} + w_1(x)f_i + w_2(x)f_{i+1} + w_3(x)f_{i+2}, \tag{2.4}$$

where i is the integer part of x and $f_i = f(i)$ are the samples of the original signal. The filter weights $w_i(x)$ for a cubic B-spline are periodic in interval $x \in [0, 1]$, that is $w_i(x) = w_i(\alpha)$, where $\alpha = x - \lfloor x \rfloor$ is the fractional part of x. The cubic B-spline filter kernel is defined as

$$\begin{aligned} w_0(\alpha) &= \frac{1}{6}(-\alpha^3 + 3\alpha^2 - 3\alpha + 1), \\ w_1(\alpha) &= \frac{1}{6}(3\alpha^3 - 6\alpha^2 + 4), \\ w_2(\alpha) &= \frac{1}{6}(-3\alpha^3 + 3\alpha^2 + 3\alpha + 1), \\ w_3(\alpha) &= \frac{1}{6}\alpha^3. \end{aligned}$$

If we applied these weights naively, a $4 \times 4 \times 4$ neighborhood of the sample point should be fetched from the volume texture, which would be too slow. Fortunately, we can take advantage of the linear interpolation unit of the texturing hardware to decrease the number of necessary fetches.

The main idea is the following. If the texture memory is fetched between two texel centers and bi-linear filtering is enabled, then we already get a weighted average of multiple texels. Thus a single fetch gives the weighted sum of two values for 1D textures, of four values of 2D textures, and of 8 values of 3D textures.

Concerning the cubic B-spline filtering of 1D textures, it means that instead of four texture memory fetches, the same result can be obtained with two fetches, where one fetch is between $i - 1$ and i, and the other fetch is between $i + 1$ and $i + 2$. Let us denote the difference of these locations and x by h_0 and h_1, respectively. The first fetch returns

$$f_{x-h_0(x)} = f_{i-1}(x - h_0(x)) + f_i(1 - x + h_0(x)).$$

The second fetch gives

$$f_{x+h_1(x)} = f_{i+1}(2 - x - h_1(x)) + f_{i+2}(x + h_1(x) - 1).$$

Values $f_{x-h_0(x)}$ and $f_{x+h_1(x)}$ are weighted and added to compute the desired filtered value $\tilde{f}(x)$. Let us denote the not yet known weights by $g_0(x)$ and $g_1(x)$, respectively. The weighted sum is then

$$w_0(x)f_{i-1} + w_1(x)f_i + w_2(x)f_{i+1} + w_3(x)f_{i+2} = g_0(x)f_{x-h_0(x)} + g_1(x)f_{x+h_1(x)}.$$

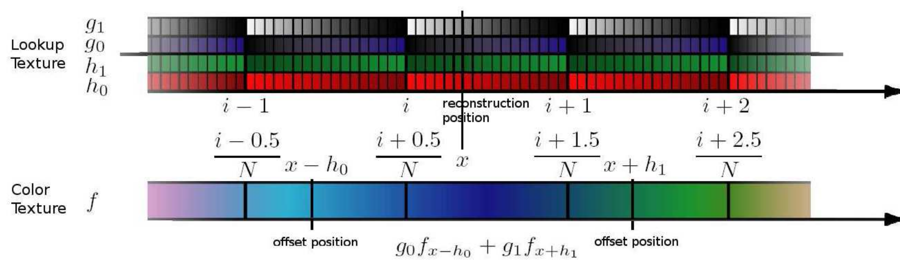


Figure 2.5: To reconstruct a color texture of size N, we first perform a linear transform of the reconstruction position x. This gives us the texture coordinates for reading offsets $h_i(x)$ and weights $w_i(x)$ from a lookup texture. Second, two linear texture fetches of the color texture are carried out at the offsets positions $x-h_0$ and $x+h_1$. Finally, the output color is computed by a linear combination of the fetched colors using weights $g_i(x)$ [1].

The fetch locations h_0 and h_1, as well as new weights g_0 and g_1 should be selected to make the weighted sum equal to the desired filtered value $\tilde{f}(x)$. Inspecting the factors of f_{i-1}, f_i, f_{i+1}, and f_{i+2} one by one, we get the following system of equations for unknown locations and weights:

$$\begin{aligned} g_0(x)(x - h_0(x)) &= w_0(x), \\ g_0(x)(1 - x + h_0(x)) &= w_1(x), \\ g_1(x)(2 - x - h_1(x)) &= w_2(x), \\ g_1(x)(x + h_1(x) - 1) &= w_3(x). \end{aligned}$$

Solving this equation, locations h_0, h_1 and weights g_0, g_1 can be obtained:

$$g_0(x) = w_0(x) + w_1(x), \qquad h_0(x) = 1 - \frac{w_1(x)}{w_0(x) + w_1(x)} + x, \tag{2.5}$$

$$g_1(x) = w_2(x) + w_3(x), \qquad h_1(x) = 1 - \frac{w_3(x)}{w_2(x) + w_3(x)} - x.$$

Since weights $g_0(x)$, $g_1(x)$ and locations $h_0(x)$, $h_1(x)$ are also periodic, they can be stored in a lookup texture. The schematic of one-dimensional cubic filtering is shown in Figure 2.5. In multi-dimensional spaces the cubic reconstruction kernel is evaluated separately along the major axes and the resulting weights are simply multiplied (tensor product extension).

The Cg code of the fragment program for one-dimensional cubic filtering is shown in Listing 2.3. Note that B-splines fulfill the partition of unity, i.e. $\sum w_i(x) = 1$, and so do the two weights since $g_0(x) + g_1(x) = 1$. Therefore, we do not need to actually store $g_1(x)$ in addition to $g_0(x)$, and the final weighting is again a convex combination carried out with a single `lerp()` instruction.

```
float4 FuncRec1D (float coord_source : TEXCOORD0,
                  uniform sampler1D tex_source, // source texture
                  uniform sampler1D tex_hg,     // filter texture
                                                // (offsets and
```

```
                                                        //  weights)
                    uniform float e_x,               // source texel
                        size
                    uniform float size_source        // source texture
                        size
                   ) : COLOR
{
  // calc filter texture coordinates where [0, 1] is a single
     texel
  float coord_hg = coord_source * size_source - 0.5f;

  // fetch offsets and weights from filter texture
  float3 hg_x = tex1D(tex_hg, coord_hg).xyz;

  // determine linear sampling coordinates
  // hg_x.x = h1(x), hg_x.y = h0(x), hg_x.z = g0(x)
  float coord_source1 = coord_source + hg_x.x * e_x;
  float coord_source0 = coord_source - hg_x.y * e_x;

  // fetch two linearly interpolated inputs
  // tex_source0 = f_{x-h0(x)}, tex_source1 = f_{x+h1(x)}
  float4 tex_source0 = tex1D(tex_source, coord_source0);
  float4 tex_source1 = tex1D(tex_source, coord_source1);

  // weight linear samples
  tex_source0 = lerp(tex_source0, tex_source1, hg_x.z);

  return tex_source0;
}
```

Listing 2.3: Cubic B-spline filtering of a one-dimensional texture

The fragment shader parameters would be initialized as follows assuming a 1D source texture with 256 texels:

```
e_x = float (1 / 256.0f);
size_source = float (256.0f);
```

The `e_x` parameter corresponds to the size of a single source texel in texture coordinates, which is needed to scale the offsets fetched from the filter texture to match the resolution of the source texture. The `size_source` parameter simply contains the size of the source texture, which is needed to compute filter texture from source texture coordinates so that the size of the entire filter texture corresponds to a single texel of the source texture.

2.1.5 Multidimensional extension

The extension of one-dimensional cubic-filtering is straightforward due to fact that the cubic reconstruction kernel can be evaluated separately along the major axes and the resulting weights are simply multiplied as mentioned before. In the implementation, this relates to multiple fetches from the same one-dimensional lookup texture. The final weights and offsets are then computed using:

$$g_{\mathbf{i}}(\mathbf{x}) = \prod g_{i_k}(x_k), \quad \mathbf{h}_{\mathbf{i}}(\mathbf{x}) = \sum \mathbf{e}_k h_{i_k}(x_k),$$

where index k relates to the directions. The Cg code of the fragment program for two-dimensional cubic filtering is shown in Listing 2.4.

```
float4 FuncRec2D(float2 coord_source : TEXCOORD0,
                 uniform sampler2D tex_source, // source texture
                 uniform sampler1D tex_hg,     // filter texture
                                               // (offsets and
                                               // weights)
                 uniform float2 e_x,           // texel size in
                                               // x direction
                 uniform float2 e_y,           // texel size in
                                               // y direction
                 uniform float2 size_source    // source texture
                                               // size
                ) : COLOR
{
  // calc filter texture coordinates where [0,1] is a single
     texel
  float2 coord_hg = coord_source * size_source - float2(0.5f, 0.5
     f);

  // fetch offsets and weights from filter texture
  // hg_x.x = h1x(xx), hg_x.y = h0x(xx), hg_x.z = g0x(xx)
  // hg_y.x = h1y(xy), hg_y.y = h0y(xy), hg_y.z = g0y(xy)
  float3 hg_x = tex1D( tex_hg, coord_hg.x ).xyz;
  float3 hg_y = tex1D( tex_hg, coord_hg.y ).xyz;

  // determine linear sampling coordinates
  float2 coord_source10 = coord_source + hg_x.x * e_x;
  float2 coord_source00 = coord_source - hg_x.y * e_x;
  float2 coord_source11 = coord_source10 + hg_y.x * e_y;
  float2 coord_source01 = coord_source00 + hg_y.x * e_y;
  coord_source10 = coord_source10 - hg_y.y * e_y;
  coord_source00 = coord_source00 - hg_y.y * e_y;

  // fetch four linearly interpolated inputs
  float4 tex_source00 = tex2D( tex_source, coord_source00 );
  float4 tex_source10 = tex2D( tex_source, coord_source10 );
  float4 tex_source01 = tex2D( tex_source, coord_source01 );
  float4 tex_source11 = tex2D( tex_source, coord_source11 );

  // weight along y direction
  tex_source00 = lerp( tex_source00, tex_source01, hg_y.z );
  tex_source10 = lerp( tex_source10, tex_source11, hg_y.z );

  // weight along x direction
  tex_source00 = lerp( tex_source00, tex_source10, hg_x.z );

  return tex_src00;
}
```

Listing 2.4: Cubic B-spline filtering of a two-dimensional texture

The fragment shader parameters of Listing 2.4 would be initialized similarly to the 1D case. Filtering in three dimensions is a straightforward extension of Listing 2.4.

2.2 Gradient Estimation and Shading

After having an accurate intersection point for each ray, the shading computations are performed for each corresponding pixel. Using the classical Phong-Blinn reflection formula for shading, the normal vector of the isosurface needs to be calculation. The surface normal can be determined as the first derivative of the density field. Therefore a new fragment program has to be loaded, which takes the result of the ray casting as a 2D texture storing the x, y, and z coordinates of the intersection points, and a subvolume stored in a 3D texture. The fragment program resamples the 3D texture at the given intersection point, where gradient

$$\mathbf{g} = \nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right)^T$$

has to be determined for the shading computations. The gradient components are calculated by filtering the volume data with the partial derivatives of the 3D reconstruction kernel (see Figure 2.4/b). For efficient derivative reconstruction, the same fast filtering scheme can be used as for the function reconstruction with the following modifications. The difference is that now all the filter kernel weights sum up to zero instead of one (derivative of constant value is 0), $\sum w_i(x) = 0$. In comparison to Listing 2.3, where the two linear input samples were weighted using a single `lerp()`, we obtain the second weight as the negative of the first one, that is, $g_1(x) = -g_0(x)$, which can be written as a single subtraction and subsequent multiplication, as shown in Listing 2.5.

```
float4 DerivRec1DX (float coord_source : TEXCOORD0,
                    uniform sampler1D tex_source ,  // source
                         texture
                    uniform sampler1D tex_hg ,      // derived
                         filter
                                                    // texture (
                                                       offsets
                                                    // and weights)
                    uniform float e_x ,             // source texel
                         size
                    uniform float size_source       // source
                         texture
                                                    // size
                   ) : COLOR

  // ... unchanged from Listing 2.3

  // weight linear samples
  // g0 * f_{x-h0(x)} - g0 * f_{x+h1(x)} = g0 * (f_{x-h0(x)} - f_{x+h1(x)})
  tex_source0 = hg_x.z * (tex_source0 - tex_source1);

  return tex_source0;
}
```

Listing 2.5: First-derivative cubic B-spline filtering of a one-dimensional texture

To compute the gradient in higher dimensions, we obtain the corresponding filter kernels via the multiplication of 1D derived cubic B-splines for the axis of derivation, and 1D (nonderived) cubic B-splines for the other axis (for the two-dimensional case, see Listing 2.6).

```
float4 DerivRec2DX (float2 coord_source : TEXCOORD0,
                    uniform sampler2D tex_source , // source
                        texture
                    uniform sampler1D tex_hg ,     // filter
                        texture
                                                   // (offsets and
                                                   // weights)
                    uniform sapmler1D tex_dhg      // derived
                        filter
                                                   // texture (
                                                     weights)
                    uniform float2 e_x ,           // texel size in
                                                   // x direction
                    uniform float2 e_y ,           // texel size in
                                                   // y direction
                    uniform float2 size_source     // source
                        texture
                                                   // size
                   ) : COLOR

  // ... unchanged from Listing 2.4

  // fetch offsets and weights from filter and derived filter
      texture
  float3 dhg_x = tex1D(tex_dhg , coord_hg.x).xyz;
  float3 hg_y = tex1D(tex_hg , coord_hg.y).xyz;

  // ... unchanged from Listing 2.4

  // weight linear samples
  tex_source00 = dhg_x.z * (tex_source00 - tex_source10);

  return tex_source0;
}
```

Listing 2.6: First partial derivative cubic B-spline filtering along the x-axis of a two-dimensional texture

The other components of the gradient can be determined analogously. Afterwards the gradients are normalized to use them as surface normals for the shading computations.

2.3 Curvature Estimation for NPR (Non-Photorealistic Rendering)

The normalized gradient can be used for all shading models that require a surface normal, like the Phong-Blinn shading. More sophisticated non-photorealistic or illustrative shading models can be based on surface curvatures. The curvature of a surface is defined by the relationship between small positional changes on the surface, and the resulting changes in the surface normal, thus curvatures are associated with the derivative of the surface normal. The surface normal is the normalized gradient of the volume, or its negative, depending on the notion

of being inside/outside the object:

$$\mathbf{n} = \pm \frac{\mathbf{g}}{|\mathbf{g}|},$$

where $\mathbf{g}$ is the gradient of the surface $f(\mathbf{x})$.

The derivative of the gradient, i.e. the second derivative of the volume is the Hessian matrix:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial x \partial z} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} & \frac{\partial^2 f}{\partial y \partial z} \\ \frac{\partial^2 f}{\partial z \partial x} & \frac{\partial^2 f}{\partial z \partial y} & \frac{\partial^2 f}{\partial z^2} \end{bmatrix} \tag{2.6}$$

The mixed derivatives in $\mathbf{H}$ (the off-diagonal elements) can be computed using the previously described fast filtering approach for the first derivatives, because the 1D filter kernels are derived only once in this case (so it needs the first derivatives of the filter).

For the diagonal elements of $\mathbf{H}$, however, the filter has to be derived two times. The second derivative of the cubic B-spline is a piecewise linear function (see Figure 2.4/c). Listing 2.7 shows how the convolution sum can be evaluated with this filter using three linearly interpolated input samples.

```
float4 DerivRec1DXX (float coord_source  : TEXCOORD0
                     uniform sampler1D tex_source , // source
                         texture
                     uniform float e_x              // source
                         texel
                                                    // size
                    ) : COLOR
{
  // determine additional linear sampling coordinates
  float coord_source1 = coord_source + e_x;
  float coord_source0 = coord_source - e_x;

  // fetch three linearly interpolated inputs
  float4 tex_source0 = tex1D( tex_source , coord_source0 );
  float4 tex_sourcex = tex1D( tex_source , coord_source );
  float4 tex_source1 = tex1D( tex_source , coord_source1 );

  // weight linear samples
  tex_source0 = tex_source0     2 * tex_sourcex + tex_source1;

  return tex_source0;
}
```

Listing 2.7: Second-derivative cubic B-spline filtering of a one-dimensional texture

To calculate the second derivatives in higher dimension, the three samples along the axis of derivation can be computed by a subsequent convolutions along the other axis (so filter texture is needed because of the other directions). For the two-dimensional case see Listing 2.8.

```
float4 DerivRec2DXX (float2 coord_source : TEXCOORD0,
                     uniform sampler2D tex_source, // source
                         texture
                     uniform sampler1D tex_hg,     // filter
                         texture
                                                   // (offsets and
                                                   // weights)
                     uniform float2 e_x,           // texel size
                         in
                                                   // x direction
                     uniform float2 e_y,           // texel size
                         in
                                                   // y direction
                     uniform float2 size_source    // source
                         texture
                                                   // size
                   ) : COLOR
{
  // calc filter texture coordinates where [0,1] is a single
      texel
  float2 coord_hg = coord_source * size_source - float2(0.5f, 0.5
      f);

  // fetch offsets and weights from filter texture in direction y
  float3 hg_y = tex1D(tex_hg, coord_hg.y)xyz;

  // determine linear sampling coordinates
  float3 coord_source1 = coord_source + hg_y.x * e_y;
  float3 coord_source0 = coord_source - hg_y.y * e_y;

  float3 coord_source10 = coord_source1 - e_x;
  float3 coord_source00 = coord_source0 - e_x;
  float3 coord_source11 = coord_sourec1 + e_x;
  float3 coord_source01 = coord_source0 + e_x;

  // fetch four linearly interpolated inputs
  float4 tex_source00 = tex2D(tex_source, coord_source00);
  float4 tex_source01 = tex2D(tex_source, coord_source01);
  float4 tex_source10 = tex2D(tex_source, coord_source10);
  float4 tex_source11 = tex2D(tex_source, coord_source11);

  // two additional texture fetches
  float4 tex_source0 = tex2D(tex_source, coord_source0);
  float4 tex_source1 = tex2D(tex_source, coord_source1);

  //weight along the y-axis
  tex_source00 = lerp(tex_source00, tex_source10, hg_y.z);
  tex_source01 = lerp(tex_source01, tex_source11, hg_y.z);
  tex_source0  = lerp(tex_source0, tex_source1, hg_y.z);

  // convolution with the second partial derivatives along the x-
      axis
  tex_source0 = tex_source00 - 2.0f * tex_source0 + tex_source01;

  return tex_source0;

}
```

Listing 2.8: Second partial derivative cubic B-spline filtering (along the x-axis) of a two-dimensional texture

The Hessian matrix has all the information needed for determining the cur-

vatures, we just have to extract it. The difficulty is that the Hessian matrix depends on the coordinate frame while curvatures are independent of the actual coordinate system. Thus we project information of the Hessian matrix into a coordinate system that is assign to the surface and is independent of the global frame.

Let us first note that if a vector is multiplied by the following matrix composed of unit vector $\mathbf{n}$,

$$\mathbf{n}\mathbf{n}^T = \begin{bmatrix} \mathbf{n}_x^2 & \mathbf{n}_x\mathbf{n}_y & \mathbf{n}_x\mathbf{n}_z \\ \mathbf{n}_x\mathbf{n}_y & \mathbf{n}_y^2 & \mathbf{n}_y\mathbf{n}_z \\ \mathbf{n}_x\mathbf{n}_z & \mathbf{n}_y\mathbf{n}_z & \mathbf{n}_z^2 \end{bmatrix}$$

then this operation projects the vector onto the line of direction $\mathbf{n}$. Since multiplying a vector by unit matrix $\mathbf{I}$ does not change the vector at all, the multiplication by matrix $\mathbf{P} = \mathbf{I} - \mathbf{n}\mathbf{n}^T$ leads to a projection that keeps everything except for direction $\mathbf{n}$. This operation is the projection onto a plane that is perpendicular to $\mathbf{n}$ (the complement of the line of $\mathbf{n}$). Since $\mathbf{n}$ is the normal vector of the isosurface, this projection projects onto the tangent plane of the isosurface.

Remember that the Hessian matrix is the derivative of the gradient, thus it tells how the gradient vector changes as a function of displacements away from the point at which its measured. Those displacements, and the resulting gradient changes, are both vectors in three dimensions. But if we post-multiply the Hessian matrix by projection matrix $\mathbf{P}$, then we are only looking at changes in the gradient vector due to displacements in the tangent plane, i.e. on the isosurface. Similarly if we pre-multiply the Hessian matrix by projection matrix $\mathbf{P}$, then we obtain the projection of the gradient vector changes onto the surface. If the Hessian matrix is both pre-multiplied and post-multiplied by $\mathbf{P}$, then $\mathbf{PHP}$ will express the tangent plane change of the gradient vector when a movement along the surface is made, which is directly proportional to the curvature along the given direction on the surface. So to get the principle curvatures the eigenvalues of matrix

$$\mathbf{G} = -\frac{\mathbf{PHP}}{|\mathbf{g}|}$$

need to be determined [3]. Matrix invariants provide the possibility to extract the eigenvalues, including curvature values κ_1 and κ_2 from $\mathbf{G}$ (the third eigenvalue is zero). The trace of $\mathbf{G}$ is $\kappa_1 + \kappa_2$. The Frobenius norm of $\mathbf{G}$, notated $|\mathbf{G}|_F$ and defined as $\sqrt{trace(\mathbf{G}\mathbf{G}^T)}$, is $\sqrt{\kappa_1^2 + \kappa_2^2}$. κ_1 and κ_2 are then found by solving a second order equation.

So the steps needed to compute curvature at an arbitrary point in a scalar field:

1. Measure the first partial derivatives comprising the gradient $\mathbf{g}$. Compute $\mathbf{n} = -\frac{\mathbf{g}}{|\mathbf{g}|}$, and $\mathbf{P} = \mathbf{I} - \mathbf{n}\mathbf{n}^T$.

2. Measure the second partial derivatives comprising the Hessian $\mathbf{H}$ (see Equation 2.6). Compute $\mathbf{G} = -\mathbf{PHP}/|\mathbf{g}|$.

3. Compute the trace T and Frobenius norm F of $\mathbf{G}$. Then,

$$\kappa_{1,2} = \frac{T \pm \sqrt{2F^2 - T^2}}{2}.$$

After the computation of these differential surface properties (see Listing 2.9), they can be mapped onto final pixel colors using different non-photorealistic shading models. RTPara uses direct color mapping. There is a predefined color lookup texture which is addressed by a given function (for example, by the mean curvature ($(\kappa_1 + \kappa_2)/2$) or the Gaussian curvature ($\kappa_1 \kappa_2$)).

```
float4 curvature(in float2 unit_tc : TEXCOORD0,   //shaded pixel
    in
                                                   // texture
                                                       coordinates
                uniform sampler2D grad_tex,        // texture of
                    gradients
                uniform sampler2D hess_diag_tex,   // texture of
                    diagonal
                                                   // elements of
                                                       Hessian
                unifomr sampler2D hess_mixed_tex   //texture of
                    mixed
                                                   // derivatives

               ) : COLOR0

{
    // get first and second derivatives
    float3 g = tex2D(grad_tex, unit_tc);
    float3 H_diag = tex2D(hess_diag_tex, unit_tc);
    float3 H_mixed = tex2D(hess_mixed_tex, unit_tc);

    // normalize gradient
    float l = dot(g,g);
    l = rsqrt(l);
    float3 n = g*l;

    // compute P = I - nn^T
    float3 I = {1.0, 0.0, 0.0};
    float3 P1 = -n*n.x + I.xyz;
    float3 P2 = -n*n.y + I.zxy;
    float3 P3 = -n*n.z + I.yzx;

    // compute Hessian
    float3 H1 = {H_diag.x, H_mixed.x, H_mixed.y};
    float3 H2 = {H_mixed.x, H_diag.y, H_mixed.z};
    float3 H3 = {H_mixed.y, H_mixed.z, H_diag.z};

    // compute G = PHP/|g|
    float3 G1 = {dot(P1, H1)*(-1), dot(P1, H2)*(-1), dot(P1, H3)
        *(-1)};
    float3 G2 = {dot(P2, H1)*(-1), dot(P2, H2)*(-1), dot(P2, H3)
        *(-1)};
    float3 G3 = {dot(P2, H1)*(-1), dot(P3, H2)*(-1), dot(P3, H3)
        *(-1)};

    G1.x = dot(G1, P1); G1.y = dot(G1, P2); G1.z = dot(G1, P3);
    G2.x = dot(G2, P1); G2.y = dot(G2, P2); G2.z = dot(G2, P3);
    G3.x = dot(G3, P1); G3.y = dot(G3, P2); G3.z = dot(G3, P3);

    float3 GGT1 = {dot(G1, G1), dot(G1, G2), dot(G1, G3)};
    float3 GGT2 = {dot(G2, G1), dot(G2, G2), dot(G2, G3)};
    float3 GGT3 = {dot(G3, G1), dot(G3, G2), dot(G3, G3)};
```

```
    //compute trace and Frobenius norm^2 of G
    float T, F;

    T = G1.x + G2.y + G3.z;
    F = GGT1.x + GGT2.y + GGT3.z;

    float4 color;

    color.r = T + sqrt(2*F - T);
    color.g = T - sqrt(2*F - T);

    return color;

}
```

Listing 2.9: Fragment shader program of curvature computations

Chapter 3

Parallel Implementation

3.1 Parallelization Approach

Rendering methods belonging to the ray-casting or ray-tracing family can easily be implemented in a distributed environment. Our implementation (application) contains both the *object space decomposition*, which partitions the data to be rendered among the participating computing resources (nodes), and the *screen space approach*, which assigns parts of final the image to be rendered to the nodes. In both cases the resulting partial images need to be composited together to form the final image.

The distributed compositing infrastructure of RTPara is based on ParaComp library (see Section 3.2) which implements the sort-last compositing technique [4].

3.1.1 Screen space

Using screen space decomposition the different nodes load the same data but they render only a part of the final image (different tiles of the output image). Thinking in terms of the classical camera analogy and OpenGL a so called frustum pyramid (*viewing frustum*) determines what part of the scene will be rendered. The parameters of the viewing frustum can be set by the `glFrustum()` function in OpenGL which describes a perspective matrix that produces a perspective projection. The parameters of `glFrustum()` (`Left, Right, Bottom, Top, Near, Far`) determine two rectangles, which are the top and bottom base areas (on the near and far clipping planes) of the frustum pyramid. The objects contained by the frustum pyramid will be rendered. The lower left (`Left, Bottom, -Near`) and upper right corner (`Right, Top, -Near`) of the top rectangle are mapped to the lower left and upper right corners of the window, respectively, assuming that the eye is located at $(0, 0, 0)$. The parameter `Far` specifies the location (`-Far`) of the bottom base plane (it is parallel with the top base plane). The corners of the bottom base rectangle are the intersection of the bottom base plane and the lines determined by the corners of the top base rectangle and the eye location (see Figure 3.1). The window where the top base rectangle is mapped can be set by the `glViewport()` function.

In screen space decomposition the compositing node has to place the tiles one aside the other (see Figure 3.2/a). While nodes render only a part of the

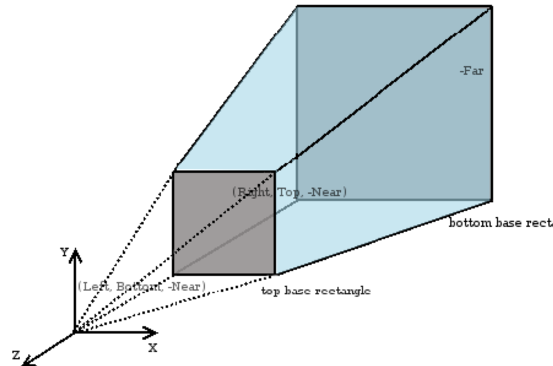


Figure 3.1: Viewing frustum

entire screen, the number of the rays to be computed by one node is divided by the number of the rendering nodes. So one node has to compute less number of rays, which means that screen space decomposition induces higher frame rate (*performance scalability*) because computing the ray–isosurface intersection is the most computational expensive part of a ray-casting algorithm (could be up to 70%).

3.1.2 Object space

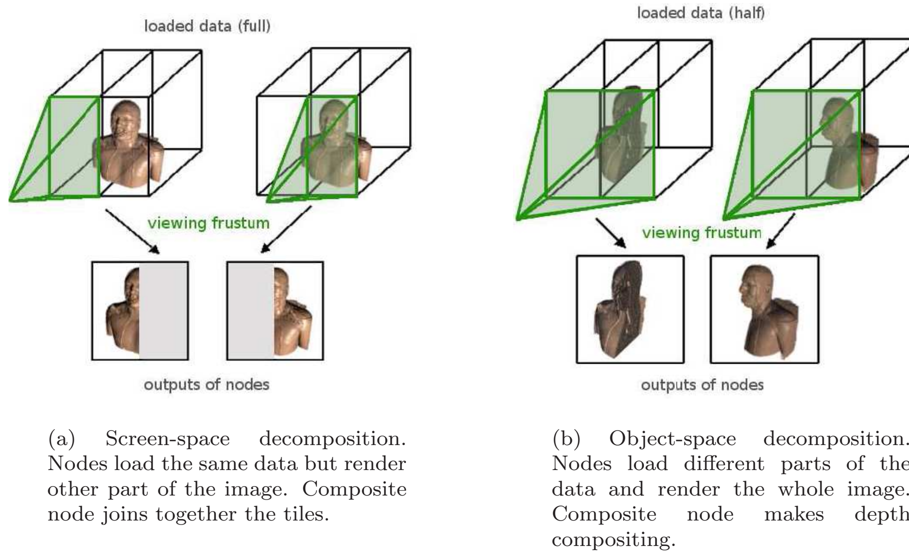
In object space decomposition the data is divided into blocks. Rendering nodes load one of the blocks and render the whole image of it. The result of the ray casting on a single node is an image, which contains in each pixel the ray parameter (i.e. depth) of the corresponding intersection point and its shaded color. The output image is produced by depth compositing these partial images (see Figure 3.2/b). While nodes load different partition of data, bigger datasets can be visualized at the same time without frame rate drop (*data scalability*).

3.2 ParaComp

The *Parallel Compositing Library* (ParaComp) has been developed by HP and Computational Engineering International (CEI). The addition of this library simplifies the development and use of parallel applications on graphics clusters and allows high performance computing users to interactively render and visualize huge data sets. The HP Parallel Compositing Library does for graphics clusters what MPI did for compute clusters. It enables users to take advantage of the inherent performance scalability of clusters with network-based pixel compositing.

The ParaComp library was designed in order to create a single image from a collection of partial images generated by multiple sources. The sources can be located on one or more machines and can be threads of execution on a single machine. The library was designed to hide the network layer from the caller and provide a graphics pixel abstraction. For more information see [2].

ParaComp library is able to perform depth and alpha compositing. The



(a) Screen-space decomposition. Nodes load the same data but render other part of the image. Composite node joins together the tiles.

(b) Object-space decomposition. Nodes load different parts of the data and render the whole image. Composite node makes depth compositing.

Figure 3.2: Screen space and object space parallelization

RTPara application uses the depth compositing mode in both parallelization cases. The used pixel format is

```
PC_PF_BGR8 | PC_PF_Z32I,
```

which means that the color of a pixel consists of three bytes in BGR (blue/-green/red) order and there are 32 bits for depth data. If depth compositing is performed, this pixel format setting provides the best performance of ParaComp (see [2]). In screen space decomposition, framelets (part of the entire frame) do not overlap each other while in object space decomposition framelets correspond to the whole output image.

Chapter 4

Installation and Usage

This chapter explains the usage of RTPara. The first section contains installation instructions. Section 4.2 describes how RTPara can be started and used.

4.1 Installation

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

The build process is the well-known configure, make, make install method (after downloading and extracting):

```
$ cd src
$ ./configure --with-incdir=<additional include directory> \
  --with-libdir=<additional library directory>
$ make
$ sudo make install
```

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`

RTPara can be also installed from a prebuilt RPM package[2]:

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

4.1.1 Library Dependencies

The following libraries are needed by RTPara:

libparacomp is the Hewlett Packard implementation of the *Parallel Compositing API*,

[1] http://amon.ik.bme.hu/rtpara
[2] Red Hat Package Manager

libGL library implements OpenGL API,

libCgGL is the Cg (C for Graphics)[3] runtime library (for OpenGL),

libCg is the Cg library runtime library,

libglut is the OpenGL Utility Toolkit (GLUT).

4.2 Execution and Usage

RTPara is developed and designed to run under HP XC V3.2 Scalable Visualization Array (SVA) System. After installing RTPara, it can be started with the `rtpara.sh` command. This script can start a program on several nodes, allocate resources, launch services, and run the visualization job. It uses SVA[4] resource management and job scheduling scripts, which are based on SLURM[5] (Simple Linux Utility for Resource Managment). The script works with RGS[6] (Remote Graphics Software) and TurboVNC[7] (Turbo Virtual Network Client) if the execution of the application from a remote machine is required. Two groups of parameters can be specified for the script. Parameters of the first type are processed by the script itself, others are processed by the program:

```
rtpara.sh [[−r <node count>] [−g <tile geometry>] [−p <partition>]
          −a [−i <volume data path>] [−t [object|screen]]
          [−s <volume dimension>] [−q <true|false>]
          [−w <window sizes>]]
```

where

-a indicates that the rest of the parameters are going to be processed by the application.

-r sets the number of nodes which perform rendering. Its default value is 2.

-p request resources from the given SLURM partition. The default partition is used if no partition is named.

-g sets the tile geometry that each of tiles will use. It can be any of the supported resolutions on the cluster.

-i sets the path of the volume data file. If a wrong path to the volumetrical data is given, the program will exit. Its default value is "./data/volume.dat".

-t defines the type of decomposition, its value can be either "screen" or "object". Its default value is "screen".

-s specifies the expected dimensions of the volume for loading ($x \times y \times z$). This part of the data will be displayed if it is possible. Its default value is $512 \times 512 \times 512$.

[3] `http://developer.nvidia.com/object/cg_toolkit.html`
[4] http://www.docs.hp.com/en/A-SVARN-4A/ch01s01.html?btnPrev=%AB%A0prev
[5] `http://www.llnl.gov/LCdocs/slurm/`
[6] `http://h20331.www2.hp.com/Hpsub/cache/286504-0-0-225-121.html`
[7] `http://www.virtualgl.org/About/TurboVNC`

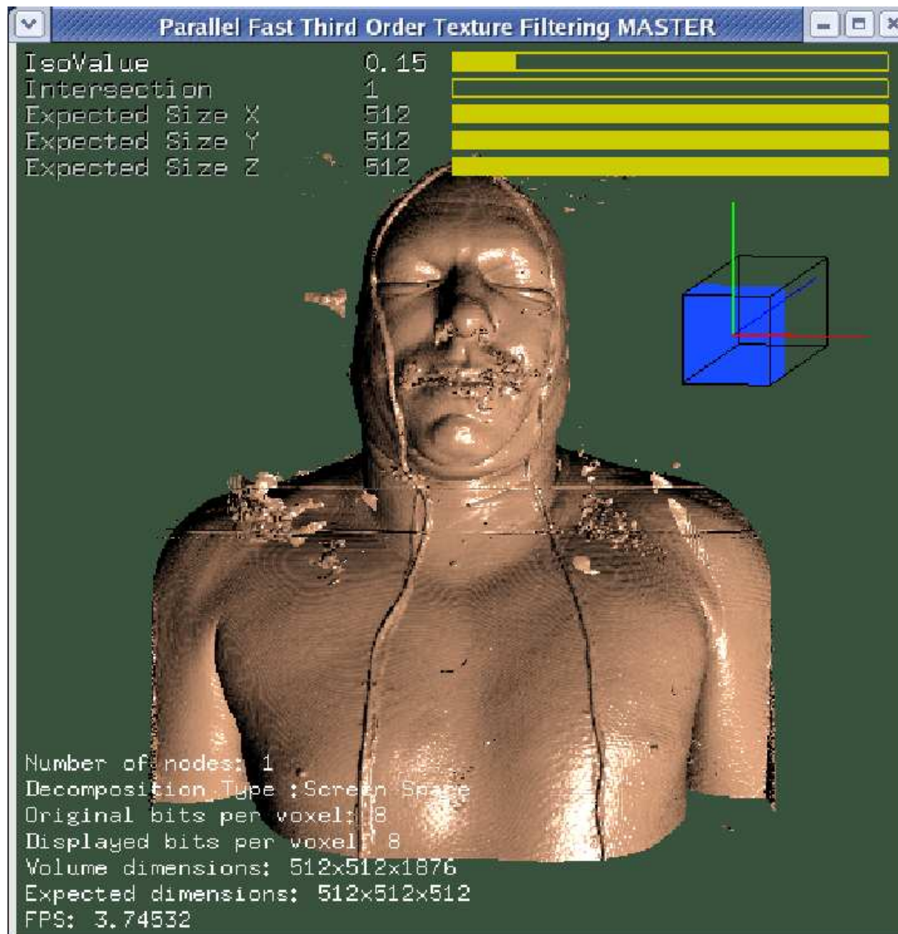


Figure 4.1: GUI of RTPara

-q indicates whether the program should quantize the data to eight bit precision. Its default value is true.

-w gives the size of the window ($width \times height$). Its default value is 512×512.

RTPara can be started without parameters (all parameters have default values). In this case the program runs on two nodes and performs screen space decomposition.

If parameter **-r** is set to one, the program runs on one node and similarly to multinode rendering renders, adds the rendering results to the compositor, and composites.

The data files are in raw format (see Section 5.4) and contain the dimensions and data precision of the volume in the header. If the expected dimensions are greater than the real volume dimensions the program will set them to the possible biggest values. If they are less, then during execution we can set which part of the volume is displayed. The expected dimensions are also variable.

The application supports the following user actions (see Figure 4.1):

- **rotate the volume**: The displayed volume can be rotated by clicking somewhere on the window then moving the mouse.
- **set isovalue**: The value (density) which determines the isosurface. It can be set by the isovalue slider.
- **navigate in the data**: If not the whole data is loaded for rendering, the user has the possibility to interactively determine which part of the data should be rendered. To allow this selection, the program displays a wired box in the right upper corner of the window. Its sizes are proportional to the sizes of the whole volumetrical data. The superimposed blue box with a coordinate system shows which part of the data is actually loaded. The loaded part of the volume can be changed by moving the blue cube by clicking on one of the axes and moving the mouse.
- **change expected sizes**: We can change the sizes of the loaded part of the volume using the X, Y, Z expected size sliders.
- **"stripping" the data**: The user can set whether the application should display the first, second, etc. isosurface intersections with the aid of the intersection slider.

The program displays information about the current rendering parameters in the left bottom corner. The displayed information includes:

decomposition type can be either object space or screen space.

entire volume sizes are the sizes of data loaded into the operative memory.

displayed volume sizes are the sizes of the displayed data. This part of the whole data is loaded from the operative memory into the graphics card memory.

FPS is actual rendered frames per second.

number of nodes is number of rendering nodes.

original bits per voxel is the original number of bits per voxel.

displayed bits per voxel is the displayed number of bits per voxel.

Sliders, navigator, and rendering information can be hidden or made visible again by pressing the "m" key.

Chapter 5

Program Structure

This chapter provides a brief overview of the source code organization, providing the interested reader with guidelines where to find the code responsible for the main functional units. RTPara has four main parts which are responsible for different functionalities:

algorithm implements the isosurface ray-casting algorithm. It is responsible for rendering the given part of the volume.

parallelization manages the parallel rendering using ParaComp.

display creates the windows and displays the result of compositing.

volume loader loads the volume data.

The root directory (`RTpara`) of the source code contains directories of header files (`inc`), source files (`src`), shaders (`cgprograms`), and volume data files (`data`). The parts of the program are separated into different directories as illustrated in Figure 5.1:

cgwrapper (algorithm) contains classes for managing the Cg environment, vertex shader and fragment shader loading, their parameter settings, etc.

display (display) includes window handling classes. These classes are responsible for realizing the user interface with GLUT.

fto (algorithm) contains the core files of the isosurface ray-casting algorithm (see Chapter 2).

glh (display and algorithm) is the place of files which give additional helper functionalities for OpenGL and GLUT (for example interactors).

network (parallelization) is the directory of the RTPara network communication classes. They provide functionality for passing parameters/commands among the nodes.

para (parallelization) contains the wrapper class of ParaComp and the class specialized for the parallelization of the isosurface ray-casting algorithm.

paramgl (display) contains helper classes for interactive parameter display.

shared (algorithm) contains miscellaneous utility functions. Such as initialization of openGL extensions, basic class of pbuffer, etc.

voldata (volume loader) is the place of class for volume loading.

A distributed rendering task with RTPara means the execution of program instances on the nodes of the employed cluster. The instances of RTPara can be either master or slave. There must be one master instance and any number of slave instances. The master instance runs on the node (master node) where the program is started. Slave nodes create slave context of ParaComp API [2], render the appropriate part of the output and send the result to the compositing (master) node. In addition, a master node creates the master context of ParaComp API [2], composites, displays results, and provides GUI.

After starting the program the entire data is loaded into the operative memory and its specified part is displayed. The GUI is visible initially. The master instance handles user actions and sends them to the slave instances.

5.1 Algorithm

The core class of the algorithm part is called `Renderer`. Its main methods are as follows:

init() initializes the shaders, some lookup textures, OpenGL extensions, loads the volume data and allocates buffers.

initSpaceFramelet() sets the parameters that determine what has to be rendered. It sets `glFrustum()` and `glViewport()` attributes as explained in Chapter 3.

initScaling() sets the texture scaling factors, texture translation values and the sizes and offsets of the bounding cube. Since volume data is stored in a 3D texture map, the coordinates of the density samples are mapped into texture coordinates, initially into the $[0, 1]$ interval in the direction x, y, z. So if the volume data is not cubic the scaling and translation of the texture coordinates are needed. The goal of this translation is different in the two parallelization approaches. While in the case of screen space decomposition its purpose is only to display the data in the center of the screen, it is a crucial step during object space parallelization (all nodes render the entire image but load only a part of the data which has to be put in the appropriate position, see Section 3.1.2). Since the entry and exit points of the rays are calculated by projecting the bounding cube of the displayed data (see Section 2.1) its parameters have to be set properly.

onRender() performs rendering.

The first two functions are called once during an execution, `onRender` is executed once for each rendered frame. The `initScaling()` function is called every time when the sizes of the displayed data are changed.

5.2 Parallelization

`Para` and `RTPara` classes are responsible for managing the parallel execution (they can be found in the para directory). `Para` provides a comfortable interface for using ParaComp. It solves every ParaComp initialization step using a node list and its type (master or slave). These two classes assume that windows are already created. `Para` has two virtual functions for specializing the class for a given task:

- `initRenderingJob()`,
- `renderingJob()`.

The `initRenderingJob()` function of `RTPara` calculates the sizes and offsets of framelets and calls the initialization functions of the `Renderer` class. Its `renderingJob()` function calls not only the rendering function of `Renderer` but also refreshes its rendering parameters. In addition, the master node sends these parameters to every nodes through a broadcast message mechanism.

5.3 Display

This part of the program creates windows, handles user inputs and passes them to the `RTPara` class. It creates the instance of `RTPara` and calls its appropriate methods during execution. It processes the input arguments and provides the GUI on the master node. The program is based on GLUT and does not use any widget sets.

5.4 Volume loader

The class `VolData` loads and stores the volume data. Its constructor sets the user defined expected sizes, data path, quantize bit (see Section 4.2) and default values. Data loading is performed by the `loadRawVolume()` function which modifies the values of the expected sizes if it is needed and quantizes if it is demanded. The `getData()` function returns a pointer to the required data. The used raw data format is the following:

- the first 3×2 bytes are the sizes in directions X, Y, Z, respectively,
- the seventh byte is the bits per voxel value.

The data is continuous within the file in direction Z. The data storing and arbitrary data access need two memory regions (see Figure 5.2). The first memory region stores the loaded data, the other one is used by the `getData()` function. If there are X and/or Y offsets or the expected sizes in the direction X and/or Y are not equal to the appropriate data sizes, the corresponding region from the first memory is copied into the second memory and `getData()` function returns with a pointer to the second memory region.

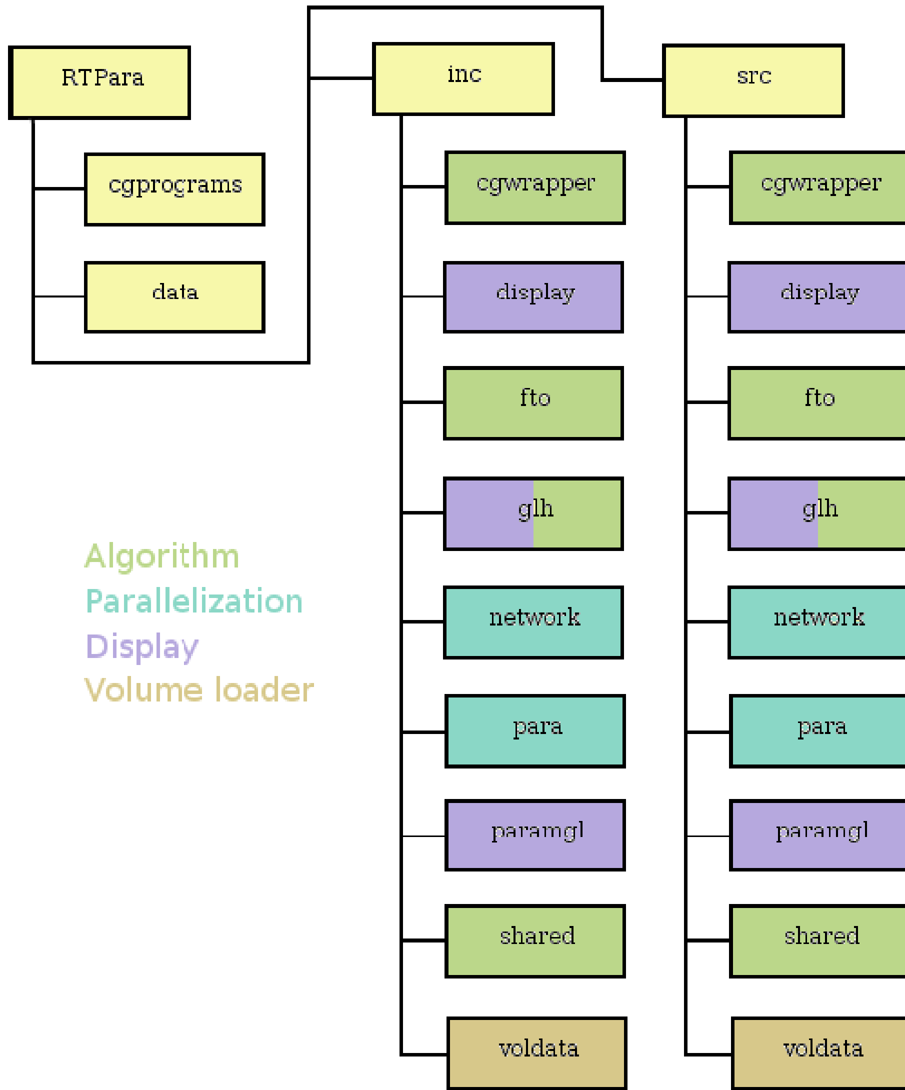


Figure 5.1: Directory structure of RTPara

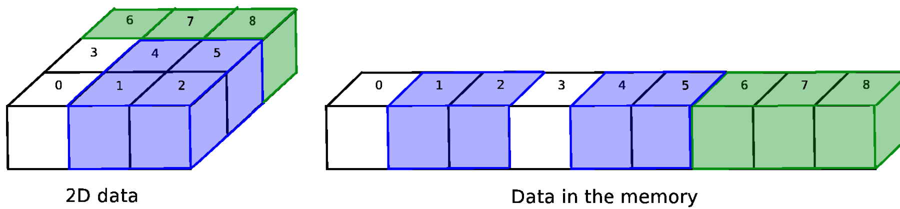


Figure 5.2: Continuous and non-continuous data access

Chapter 6

Results

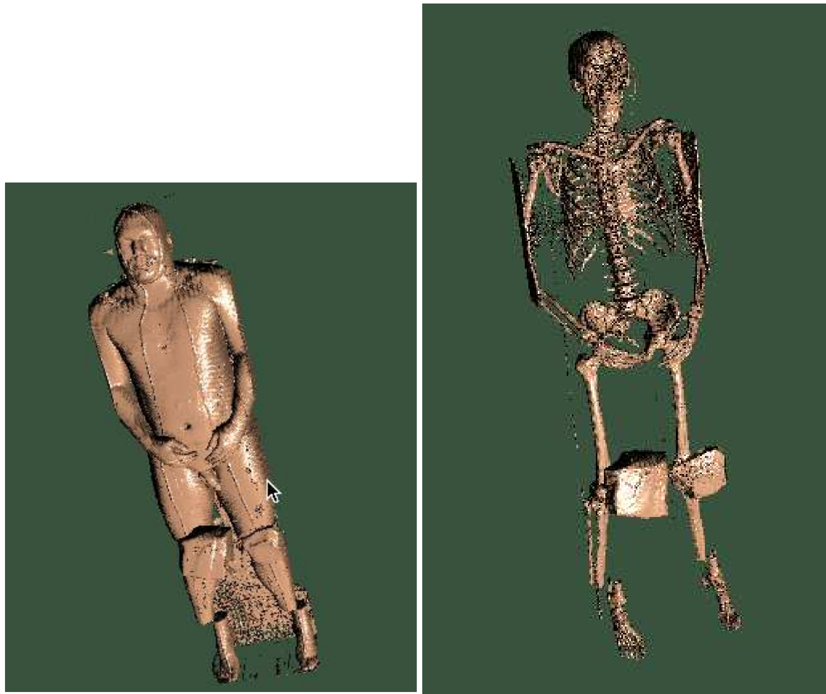
This chapter provides numerical results about the execution of RTPara on a five-node HP SVA cluster. Each node had a dual-core AMD Opteron 246 processor and NVIDIA Quadro FX3450 graphics card. Frame rates of the execution are presented in Table 6.1 and 6.2. Figures 6.1 show screen shots of the program.

Data set	Size (MB)	Resolution	1 node	2 nodes	3 nodes	4nodes
VHP	128	512×512	4.13	6.76	10.39	14.76
VHP	128	800×800	2.28	3.58	5.27	7.38
lobster	≈ 1	512×512	4.49	7.22	9.02	11.35
lobster	≈ 1	800×800	2.18	3.4	5.49	6.02
engine	14	512×512	2.64	5.62	7.01	11.34
engine	14	800×800	1.63	2.97	3.4	5.72

Table 6.1: Average frame rates using screen space decomposition

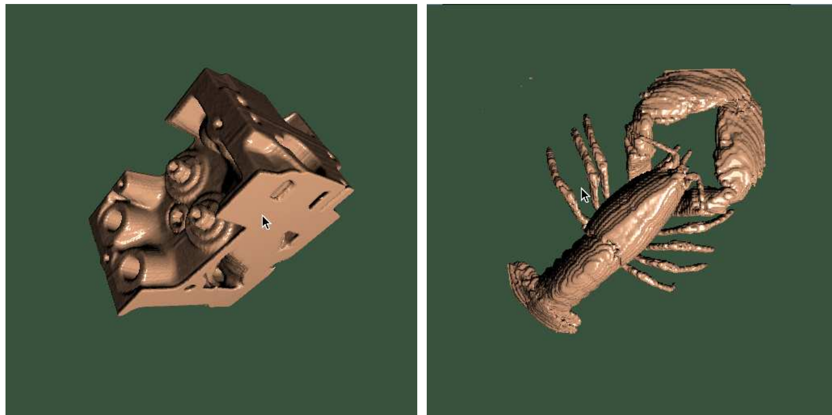
Data set	Resolution	1 (128 MB)	2 (256 MB)	3 (384 MB)	4 (470 MB)
VHP	512×512	3.87	4.19	5.42	4.43
VHP	800×800	2.14	0.97	1.43	2.06

Table 6.2: Average frame rates using object space decomposition



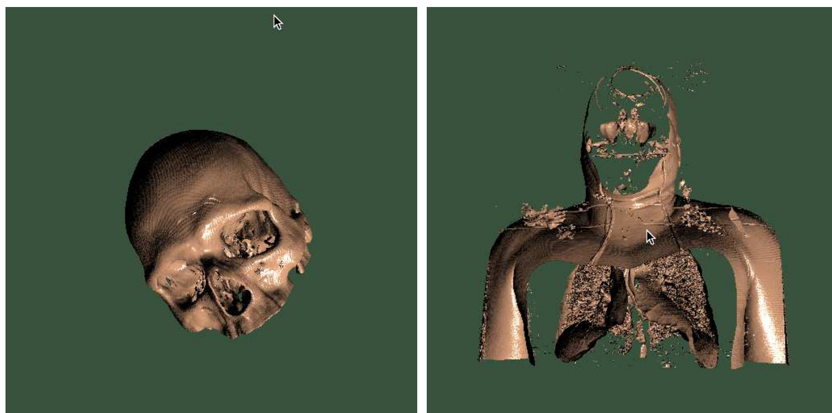
(a) VHP (isovalue = 0.15)

(b) VHP (isovalue = 0.32)



(c) engine (isovalue = 0.15)

(d) lobster (isovalue = 0.15)



(e) head (isovalue = 0.32)

(f) VHP (isovalue = 0.15)

Figure 6.1: Rendering results for DTB application

Bibliography

[1] Hadwiger, M., Sigg, C., Scharsach, H., Bühler, K., and Gross, M. Real-time ray-casting and advanced shading of discrete isosurfaces. *In Proceedings of EUROGRAPHICS 24*, 3 (2005), 303–312.

[2] Hewlett Packard. *HP Scalable Visualization Array Parallel Compositing Library Reference Guide*, 2007.

[3] Kindlmann, G., Whitaker, R., Tasdizen, T., and Möller, T. Curvature-based transfer functions for direct volume rendering: Methods and applications. *In Proceedings of IEEE Visualization* (2003), 513–520.

[4] Molnar, S., Cox, M., and Ellsworth, D. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (1994), 23–32.

[5] Sigg, C., and Hadwiger, M. Fast third-order texture filtering. *In GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation Matt Pharr(ed.), Addison-Wesley* (2005), 313–329.

[6] Westermann, R., and Ertl, T. Efficiently using graphics hardware in volume rendering applications. *In Proceedings of SIGGRAPH* (1998), 169–177.