University of West Bohemia in Pilsen
Faculty of Applied Sciences
Department of Computer Science and Engineering

DIPLOMA THESIS

Pilsen, 2002

Petr Šereda

University of West Bohemia in Pilsen
Faculty of Applied Sciences
Department of Computer Science and Engineering

Diploma Thesis

Direct Visualization of Iso-Surfaces in Volume Data

Plzeň, 2002 Petr Šereda

I declare that this diploma thesis was written by myself with help of the literature listed in the references.

Pilsen, 20th August 2002 ……..

Petr Šereda

Acknowledgements

Special thanks to Prof. Ing. Václav Skala, CSc. for his advices, literature support and for the organisation of the Socrates programme and the computer graphics conferences that helped us to get valuable international experiences

I would also like to thank to my parents for their general support during my long studies, to my brother who helped me to keep my body fit and to my friends for the moral support and making the studies be easier.

Abstract

This thesis contains a basic introduction to scalar volume data and iso-surfaces. Three direct iso-surface visualization methods are described: ray-casting, discrete ray-casting and the shear-warp method. An alternative acceleration technique is described that uses the 3D hardware for an effective space-leaping in the ray-casting methods. Finally, the methods are compared.

Keywords: volume visualization, iso-surface rendering, ray-casting, shear-warp, space-leaping, hardware acceleration, OpenGL.

Contents

1 Introduction to Volume Data

1.1 Surface Representation

Different approaches can be taken to represent objects in space. Intuitive and thus commonly used are techniques that use geometric primitives. These primitives include boxes, spheres, cylinders, cones and others. The point is to choose the right combination of primitives in order to give the true picture of the object. This can be rather efficient way for simple objects. However, in case of complex objects this often becomes unfeasible and some approximation has to be chosen.

Very popular is representation of surfaces using triangles. Triangles form a triangle mesh that can be used to approximate every shape. Thanks to the hardware support that is given by modern graphic cards, the visualization of the triangle meshes becomes very fast. But the hardware has some limitations for the number of triangles that can be effectively rendered. These limits often prevent us from choosing sufficiently accurate representation.

Surface can be also described directly using mathematic functions. The advantage is that large variety of shapes can be described precisely. However, operations with such objects have rather mathematical than purely geometrical character and involve solution of nontrivial equations. Furthermore, some objects can not be described without compositing large number of functions and thus it looses the efficiency.

Generally speaking, the surface representation has a serious disadvantage. It is suitable as long as we are interested just in the shape of an object. Once we are interested in the interior as well, we find it useless.

1.2 Volume Representation

Unlike the surface representations mentioned above, volume representation does not describe space as set of objects. It describes whole section of space (usually limited by a bounding box) using sample points. In the simplest case the sample points create a 3-dimensional grid. In every node of this grid a sample is taken that reflects somehow properties of the material in that point.

We can illustrate the basic idea of this approach on an example of a 2-dimensional object.

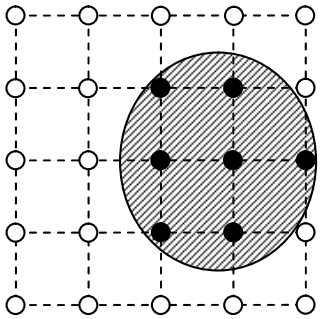


Figure 1.1: Binary classification of a 2D object.

In the Figure 1.1 we can see the grid. Samples are taken in the nodes. In this case the binary classification is used. That means that value in the node is either 1 (the black nodes lying inside an object) or 0 (white nodes that lie outside the object).

The node in 3D grid is in the terminology of volume data visualization called *voxel* (volume element) which is a 3D analogy of *pixel* (picture element). Voxel represents a unit of volume.

The voxels can have different shapes and sizes. These properties often depend on the way the data are acquired. Most common are uniform rectangular voxels which are also easy to handle. This kind of voxels is also used further in this thesis.

1.3 Character of Volume Data

Data sampled in the grid nodes can have different meaning. Most commonly used are the following types:

Binary data – the voxels contain either ones or zeroes depending on whether they lie inside or outside an object respectively. This representation describes shapes of the objects. We can consider this data as a special case of scalar data.

Scalar data – each voxels contains one value. Typical meaning of the value is the density of material where greater values represent higher densities than the lower values.

Vector data – each voxel contains one-dimensional array of values. The vector can represent e.g. fluid velocity.

Tensor data – each voxel contains a matrix of values. This data can represent more complex physical measurements.

Further in this thesis we will be dealing with the scalar data only.

1.4 Sources of Volume Data

A source can be the sampled data of real objects, artificial data produced by a computer simulation or voxelization of a geometric model (a synthetic model is converted into volume representation).

Examples of applications that generate sampled data are medical imaging (CT – computer tomography, MRI – magnetic resonance imaging, ultrasonography), geoscience (seismic measurements, oil resources explorations) and industry (CT inspections).

Applications that generate computed datasets include for example computational fluid dynamics and meteorology.

Voxelized terrain models as well as voxelized objects like buildings and vehicles are also used in computer simulators. Specific properties of terrain models can be used for designing special visualization algorithms that are more efficient that those for arbitrary models [Cohe96].

2 Volume Visualization

Volume visualization can be defined as a process that projects a tree-dimensional dataset (array of voxels) onto a two-dimensional plane (array of pixels). This process is supposed to give the user an understanding of the structure contained within the data. Volume visualization is also called volume rendering.

We can interpret scalar volume data in two different ways. Either as voxels - cubes having the same value in the whole volume or as cells- cubes that have eight different values in the vertices and value in the volume is interpolated using trilinear interpolation, see Figure 2.1.

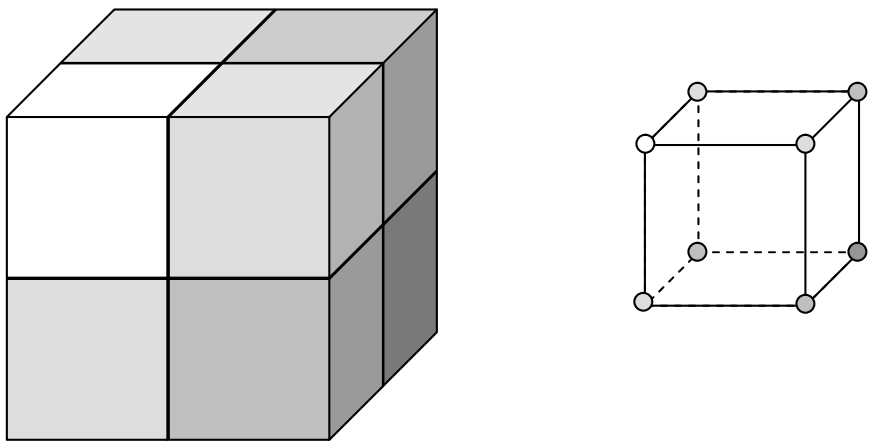


Figure 2.1: Volume data interpreted as voxels (left) and cells (right)

There are two basic groups of techniques used for volume data visualization.

2.1 Volume Rendering Techniques

This group includes techniques that visualize the whole volume at once using the alpha-blending. The whole volume is thus projected onto the projection plane. As the result of these techniques, we get an image that has a similar look to the traditional X-ray picture.

An X-ray picture gets its look due to the different material translucencies towards the X-rays. The ray that passed through more translucent tissues appears as a darker dot on the negative because more energy penetrates those tissues.

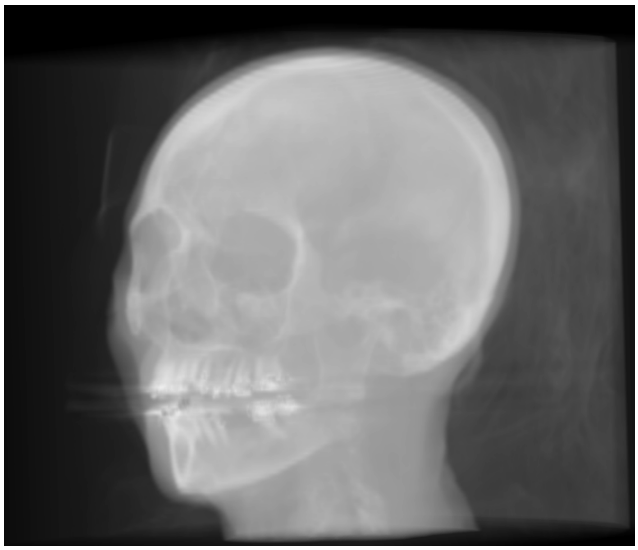


Figure 2.2: Image of a human head generated using volume rendering.

2.2 Surface Rendering Techniques

Surface rendering is used when visualization of a tissue surface is required.

2.2.1 Iso-Surfaces in Volume Data

An iso-surface is defined as a set of points having the same property (i.e. material density in our case). These points form a surface. The iso-surface is a 3D analogy of the iso-line in Figure 2.3. The dark and white dots in the figure represent values lying above and below the iso-value respectively. Thus the iso-line must be somewhere between these values.

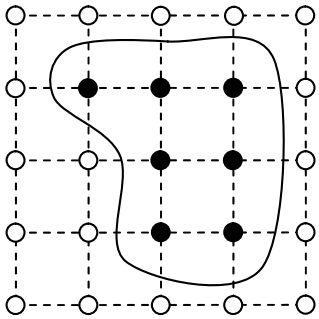


Figure 2.3: Iso-line in 2D scalar data.

2.2.2 Indirect Methods

The word “indirect” means that the iso-surface in the volume data is not being visualized directly from the data set onto the user screen, but there is an additional step. In this step the iso-surface is converted into surface representation. This process is called *extraction of iso-surface*. The triangle mesh is used to represent the surface. When the surface (triangle mesh) is extracted, the power of graphic hardware is used to display (render) it onto the screen (see Figure 2.4).

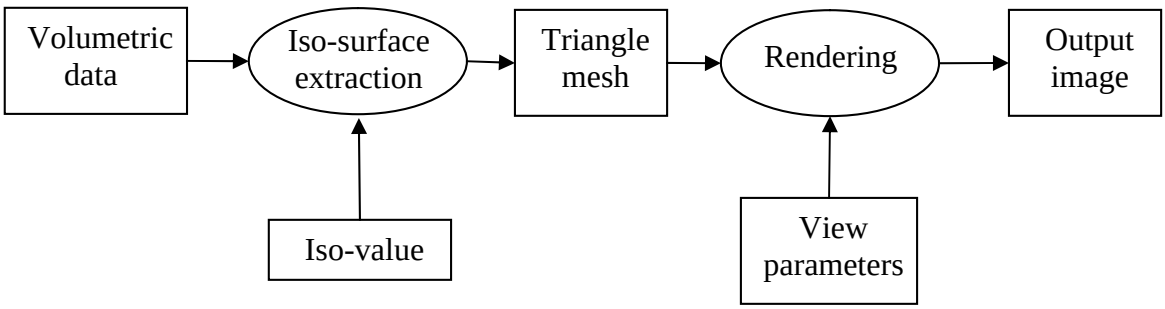


Figure 2.4: Indirect visualization process.

The main advantage of these methods is in the speed of the rendering step. Once the triangle mesh is extracted, changing the viewpoint (rotating the data) means just changing the view parameters and rendering the mesh again using the fast graphic hardware.

However, when an interactive change of iso-value is needed, the surface extraction becomes a bottleneck as it is relatively very slow. In other words, as long as we have a surface extracted and want to rotate it to explore it from the other side, it is fast. As soon as we need to keep the view direction and want to see other surface (e.g. the skull instead of the skin), other iso-surface has to be extracted and we have to wait for the resulting image.

Moreover, in the case of large datasets and complex surfaces, the number of generated triangles can be so high that can not be interactively rendered using the available graphic hardware. The problem is that many triangles are generated that can not be seen on the output image (are hidden by other triangles).

2.2.3 Direct Methods

Compared to the indirect methods, the direct methods render the volume data directly into the result image.

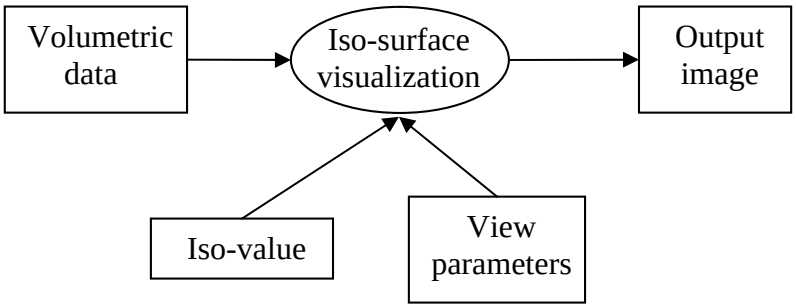


Figure 2.5: Direct visualization process.

As we can see in the Figure 2.5, the direct visualization process is dependent on both the iso-value and the view parameters. No matter if the iso-value or the view parameters are changed, the visualization works in the same way.

2.3 Image-order Methods

There are two approaches used in the visualization.

The first one, the image-order approach, is driven by the image. It takes the image pixels one after the other and finds objects that are projected into this pixel. Depending on the technique, either the closest object to the observer or a combination of objects using alpha-blending is then projected into this pixel.

A typical image-order technique is the ray-casting

2.4 Object-order Methods

On the contrary, the object-order approach takes the objects from the object space and project them into the image pixels. This process is called rasterization. To solve the visibility either the objects are taken in FTB (front to back) or BTF (back to front) order or a z-buffer is used.

3 Ray-Casting

3.1 The Principle of Ray-Casting

Ray-casting is a direct visualization method. It is a typical the image-order method.

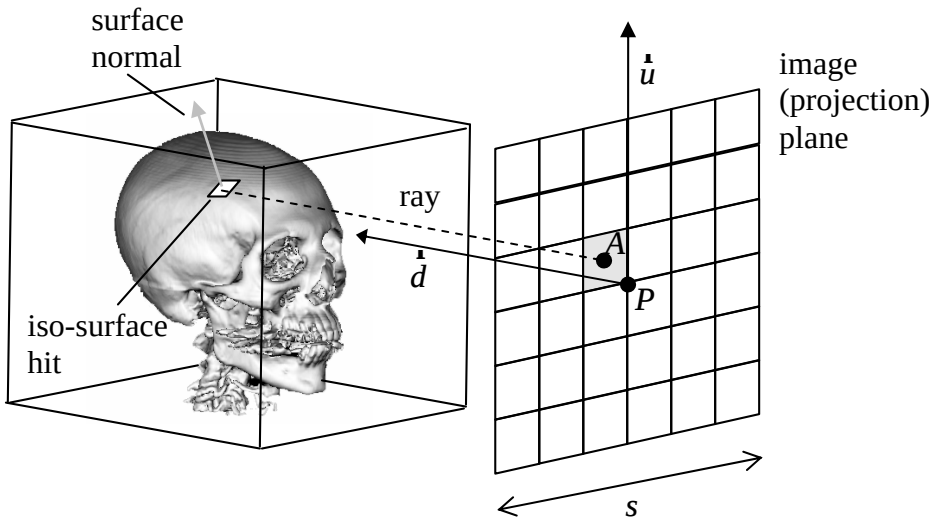


Figure 3.1: The principle of ray-casting

The Figure 3.1 shows the basic principle of the ray-casting technique. The image plane on the right side corresponds to the resulting image. The goal is to acquire a colour value for every single pixel of the image. This is an image-order technique because the visualization process depends on the image plane properties. Every pixel of the result image corresponds to a point on the image plane. A viewing ray (3D line) is cast through that point in the direction perpendicular to the image. The ray is perpendicular because we consider just parallel projection where all viewing rays are parallel to each other. This ray is then traced along its path through the data volume until an iso-surface is hit or the ray gets out of the volume (there was nothing but empty space along the path). If the surface is hit, a surface normal has to be computed in order to perform shading. After the hit point is shaded, the resulting colour is set as the value for the corresponding pixel. If no surface is hit, the pixel value is set to background colour. This process has to be performed for every pixel.

Size of the image plane in this implementation is given by the volume bounding box diagonal length. Therefore the projected data fits the plane in any position. Smaller projection plane could be used for rendering a defined region of interest.

3.2 Casting a Ray

The viewing parameters for the parallel projection in this implementation (see Figure 3.1) are given by:

P – a point in the object space defining the centre of the projection plane

s – size of the projection plane

m – number of rays being sent per one row/column of the projection plane, the plane is square shaped, together m^2 rays are sent

$\vec{d}$ – a normalized directional vector

$\vec{u}$ – a normalized vector perpendicular to the $\vec{d}$ vector, pointing in the direction "up"

In the ray generating process, we divide the projection plane using an equidistant grid *mxm*. Each field of the grid corresponds to one pixel in the resulting image. We generate the point A in every field of the grid. The rays are then sent from these points. The ray is given by the equation $X(t) = A + \vec{d} \cdot t$.

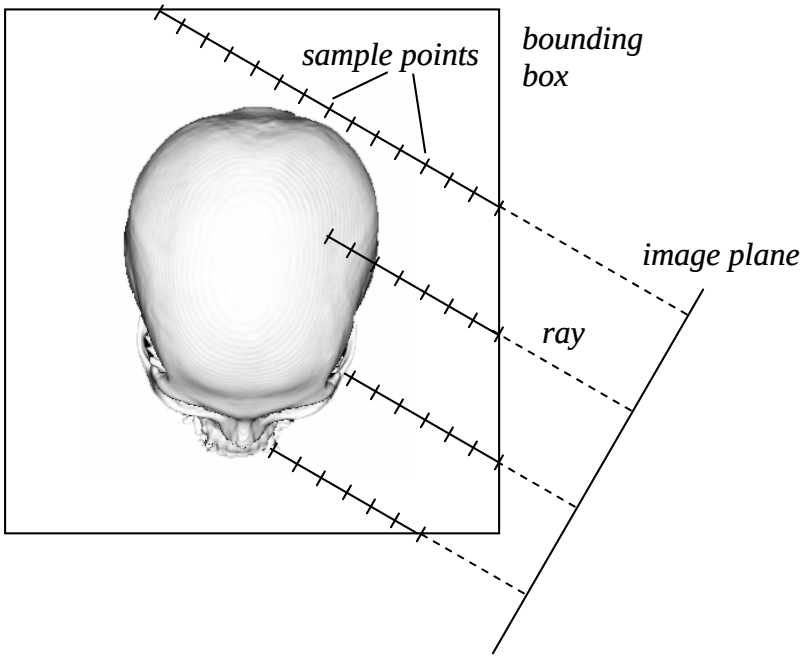


Figure 3.2: Sampling the volume along the rays.

The principle of ray sampling through the volume is shown in the Figure 3.2. The sampling must be restricted just to the volume bounding box. This is achieved by restricting the parameter t, i.e. we find t_{min} and t_{max} for each ray. The ray is then sampled in equal steps for $t \in < t_{\min}, t_{\max} >$.We need to choose the sampling step, i.e. dt, which is problematic.

Choosing bigger steps is one of the ways of accelerating the algorithm; on the other hand this may cause an undersampling and loosing small details. Reasonable is choosing the step as the shortest distance between two voxels.

3.3 Finding the Intersection

The sampling along the ray is done in order to find the iso-surface given by the iso-value (threshold value). The sample is taken inside the cells made of eight voxels (see Figure 3.1) using the tri-linear interpolation. As the result of the interpolation, we get a scalar value in the sampling point $f(X(t))$. We continue the sampling until the t_{max} is reached or until $f(X(t)) \geq treshold$. At this moment we know that the iso-surface is located between points ***X(t-dt)*** and ***X(t)***. As we know scalar values in both these points and the value for iso-surface, we can use the linear interpolation to get the precise location of the ray and iso-surface intersection:

$t_{iso} = (t - dt) + \frac{dt}{f(X(t_2)) - f(X(t_1))}(f_{iso} - f(X(t_1)))$, where f_{iso} is the iso-surface value.

The intersection is then at point $X(t_{iso})$.

3.4 Shading

Further step we need to do is to compute the surface normal in the intersection point. First we compute the normal vectors in the eight voxels creating the cell in which is the intersection point situated. The surface normal vector is given by the data gradient that can be approximated using the central difference. Gradient vector in the voxel at position [x, y, z] is computer as follows:

$$\nabla f(x,y,z) = \left(\frac{f(x+1,y,z) - f(x-1,y,z)}{2a}, \frac{f(x,y+1,z) - f(x,y-1,z)}{2b}, \frac{f(x,y,z+1) - f(x,y,z-1)}{2c} \right),$$

where a, b and c are distances between voxels in x, y and z directions respectively. Different methods of gradient estimation are described e.g. in [Bentum96] or [Neumann00].

The Phong shading was used. To perform the Phong shading means to compute the surface normal in the intersection point. As we know the intersection point position within the cell, the normal vector is computed from the eight gradient vectors using trilinear interpolation.

The central difference can not be used for voxels at the volume borders. To avoid checking if the voxels lies at the border or not, it is useful to add extra empty voxels around the volume.

3.5 Illumination Model

The Lambert diffuse illumination model with just one light was used:

$$I_V = I_A r_a + I_L r_d (\vec{L} \cdot \vec{N}),$$

where I_V is the final intensity reflected in the viewer direction, I_A is intensity of the ambient light, I_L intensity of the directional light we have, r_a and r_d are reflection constants that are specific for each material, $\vec{L}$ is the vector pointing from the surface point towards the light and $\vec{N}$ is the surface normal vector.

Just a white light and a white material are used. Therefore one grey-tone value results from the equation. As no materials are differentiated, the coefficients k_a, k_d are constants. The light rays are parallel (the light is in infinite distance) and in order to make the visualized data side be directly illuminated, the light rays are parallel to the viewing direction as well.

In the implementation, the normalized viewing direction vector $\vec{d}$ and the normalized gradient vector $\vec{g}$ are used instead of the vectors $\vec{L}$ and $\vec{N}$.

$$I_V = I_A + I_L (\vec{d} \cdot \vec{g})$$

To map the intensity value to one-byte grey value, the ambient and light intensities are chosen so that $I_A + I_L = 255$

3.6 Properties

The time needed for the rendering depends on the image size and on the number of samples taken. The complexity of this brute-force ray-casting algorithm is $O(m^2.n)$, where m is the number of pixels (rays sent) per one side of the result image and n number of voxels per one side of the volume. The number of samples taken along the ray path through the volume is proportional to n.

The memory requirements of implemented algorithm are $2N$ bytes, where N is number of voxels in the dataset. N bytes are taken for storing the dataset and N bytes for the auxiliary data structure used for the sampling speedup (see below).

3.7 Acceleration Techniques

It is obvious that the ray-casting algorithm spends most of the time sampling the scalar values along the ray until an iso-surface is hit. To reduce the costs caused by trilinear interpolation, an auxiliary data structure was introduced in this implementation. It has the same size as the volume. Every element of this 3D array represents a volume cell given by eight voxels. In this array, there are stored the maximum voxel values for each cell. Every

time the sample is going to be taken, the algorithm looks into this array first. If the maximum voxel value within the cell is lower than the threshold value, then the interpolation is not performed as the interpolated value will be always lower.

3.7.1 Adaptive Refinement

The adaptive refinement [Levoy89a] is a typical acceleration technique that uses the image coherency. As the time spent on one image is influenced by the number of sent rays, we can get the image faster when fewer rays are sent. This image has lower quality but faster rendering enables interactive frame rates. When the viewpoint is changed, a lower number of rays are sent from the pixels uniformly spread over the projection plane. The empty pixels are filled using bilinear interpolation. If the user needs to change the viewpoint again, new low-quality image is generated. If the viewpoint change is not requested, more rays are sent to improve the image quality. The refinement of the image can be done adaptively according to the low-quality image. When two neighbouring rays gave colour values whose difference is relatively high, then another ray is sent between them. The refinement continues until the user changes the viewpoint or until rays from all pixels are sent. However the refinement can continue in sub-pixel accuracy to perform antialiasing.

3.7.2 Octrees

Spatial coherency is often exploited using various spatial structures that divide the volume and enable empty space leaping. One of them is the octree structure. It is a tree structure in that does every node has eight sons. On the highest level, the root node represents the whole volume box. The eight sons represent the box divided into eight equal boxes. On the lowest level, the leaves represent single cells. Each node contains the maximal voxel value within the box so that the whole box can be skipped if the threshold value is greater that the maximal value. A serious disadvantage of this algorithm is that the ray spends considerable amount of time on changing the current level in the octree. More detailed description of using octrees for volume rendering can be found e.g. in [Levoy89b].

3.7.3 Proximity Clouds

The proximity clouds [Cohen93] represent another object order technique for space leaping. In the preprocessing step the volume is evaluated so that every voxel contains a distance to the closest iso-surface. In other words the values tell us how long sampling step can be made in any direction without hitting the surface. The advantage is that no tree like structure needs to be traversed. On the other hand the preprocessing is relatively time demanding and as it depends on the current threshold, interactive threshold changes are problematic.

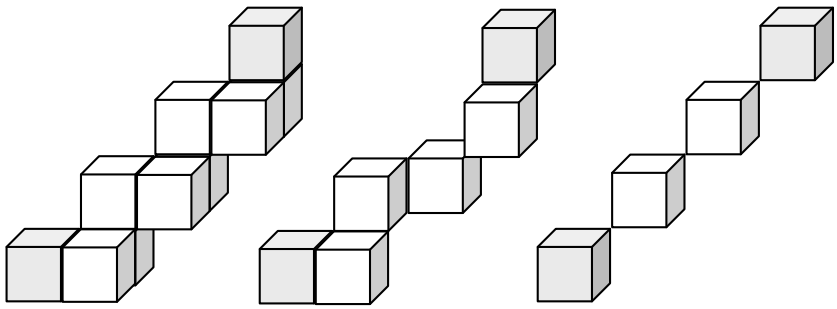
4 Discrete Ray-Casting

4.1 Principle

This technique is based on the same principle as the previous one. The rays are given by the parametric equation and the parameter *t* is restricted by the data bounding box. The difference is in the way the ray is traced inside the volume and in the way the data is treated. Instead of computing the precise data values inside the cells, the data is treated as an array of voxels (see Figure 2.1). Therefore we do not need to know the sample position within the cell, but just need to know which voxels (full boxes) the ray pierces. The advantage of the discrete ray-casting is that it can be achieved using simple integer arithmetic.

4.2 Path Generation

The path through the volume is generated using a 3D line algorithm. See three different line connections that can be chosen (Figure 4.1).



6-connected 18-connected 26-connected

Figure 4.1: Three path generation possibilities

If we look at the voxels as at boxes, we can say that two voxels are 6-connected if they share a face; they are 18-connected if they share a face or an edge and they are 26-connected if they share a face, an edge or a vertex.

Usually the 6-connected or 26-connected paths are used. The 6-connected path contains all voxels the ray pierces but can have up to three times more voxels than the 26-connected one (for diagonal direction). The 26-connected path was chosen for the implementation in order to have every sample along the ray in different volume slice. The sampling is thus similar to that used in the shear-warp implementation (see chapter 5).

We start the path generation at the first volume voxel pierced by the ray. The first voxel is given by the equation $X(t) = A + \vec{d} \cdot t_{\min}$ where t_{min} is the lower parameter bound (see chapter 3.2). Although a Bresenham algorithm modification for 3D could be used as well (see [Amanatid87]), the 3D DDA algorithm was implemented. This algorithm is based on adding differences to the coordinates:

$$x_{i+1} = x_i + \Delta x$$
$$y_{i+1} = y_i + \Delta y$$
$$z_{i+1} = z_i + \Delta z$$

This needs floating-point arithmetic and rounding operations to get integer coordinates. To use the integer arithmetic and avoid rounding operations, the numbers are represented as fixed-point numbers stored in 32 bit integers. The higher word is used for the whole part and lower 16 bits for the decimal part. Adding a difference thus means adding an integer number and no rounding is needed as the higher word represents the position. At the beginning the floating-point numbers are simply converted to fixed-point numbers by multiplying by 2^{16} and rounding to an integer.

The difference in the principal direction is always 1. Therefore in each step we traverse to the next data slice and the other two differences update the current position within the slice.

The maximal number of steps is computed in advance from t_{min} and t_{max}. The path is generated until a voxel having the value greater or equal the threshold value is reached or until the maximal number of steps is reached.

Finally the gradient and colour value are computed using the central difference and illumination model (see chapters 3.4 and 3.5).

4.3 Properties

The complexity of this algorithm is $O(m^2.n)$.

The memory requirements of the algorithm are N bytes, where N is number of voxels in the dataset. This memory is taken for storing the dataset.

As the algorithm treats the data as full voxels and thus no inter-cell interpolation is done, by sending more rays per voxel the image quality does not improve. The algorithm is therefore not suitable for rendering with sub-voxel precision.

The 26-connectivity results in several problems. Firstly, a thin surface having in discrete space 26-connected holes can be missed. The second problem is in the way the gradient vector is computed. The ray may penetrate into the object “too quickly”, and all the voxels

taken into account in the central difference may belong to the object and have same values which results in a zero vector (see Figure 4.2).

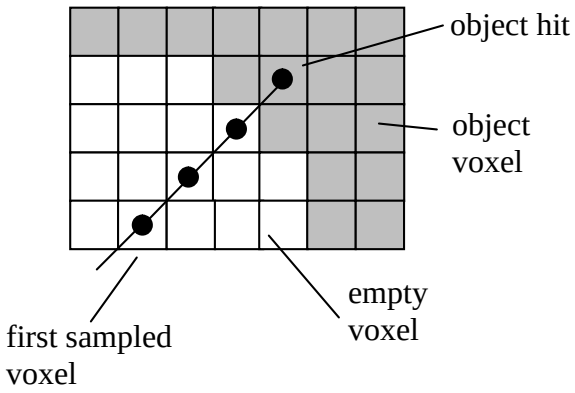


Figure 4.2: An example of hitting an object following a 26-connected path

5 Shear-Warp Factorization

This method presented in [Lacroute95] is an object-order method. When visualizing a volume data, we use a view transformation that rotates the object space according to user's needs. When using the classic approach, the volume needs to be traversed in arbitrary direction which is not very efficient. Random accesses to the volume data mean evaluating data indexes as well as memory access that is slow compared to the speed of CPU (see results in chapter 7.5). More efficient is traversing the volume in storage order. Data is accessed sequentially and the advantage is taken of the memory caching.

5.1 Principle

The arbitrary view transformation can be achieved using 3D shearing and 2D warping. In Figure 5.1, we can see the shear-warp process.

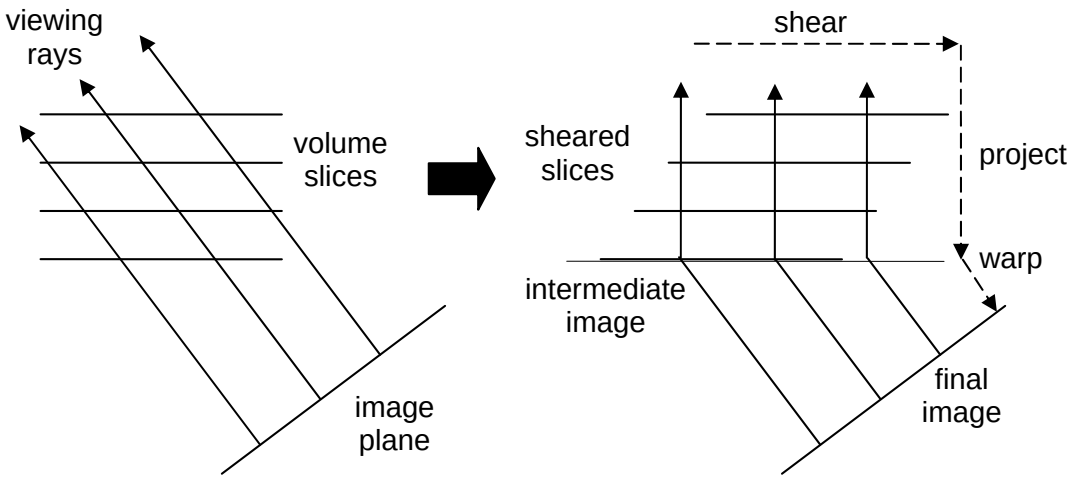


Figure 5.1: Volume is transformed to sheared object space by translating slices.

The shearing direction is chosen parallel to the set of slices that are most perpendicular to the viewing rays (viewing direction). The volume is sheared so that the viewing rays become parallel to the data slices. The shearing is done by translation of slices. Of course no slow physical data moving is done since the translation is achieved by index shifting during data resampling.

In the Figure 5.2 the translation of slices and projection onto the intermediate image are shown. The slices are taken in front-to-back order and resampled using bilinear interpolation. Since the slices are just translated, the resampling weights are the same for the whole slice (Figure 5.3).

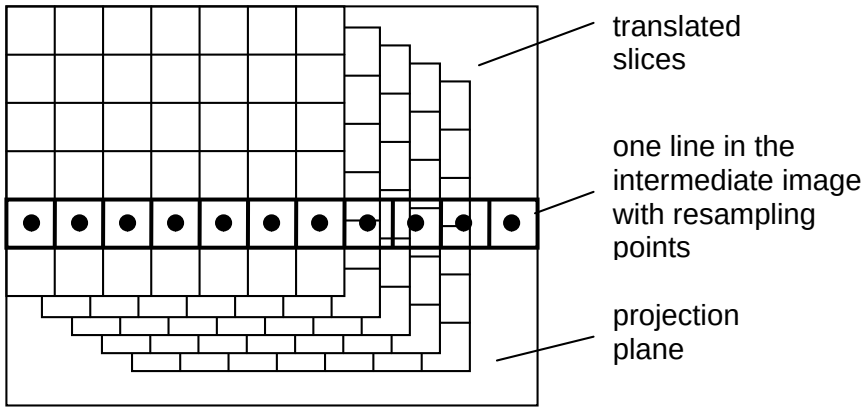


Figure 5.2: Projection of translated slices using resampling

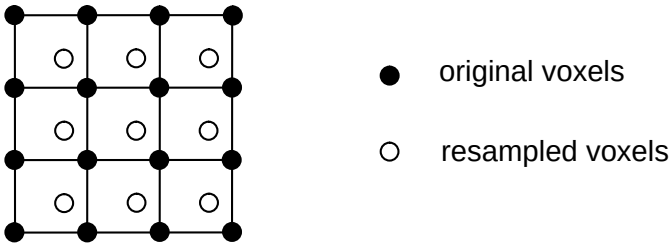


Figure 5.3: The resampling weights are same within the whole slice

5.2 Derivation

Four different coordinate systems are used in the shear-warp algorithm. See Figure 5.4.

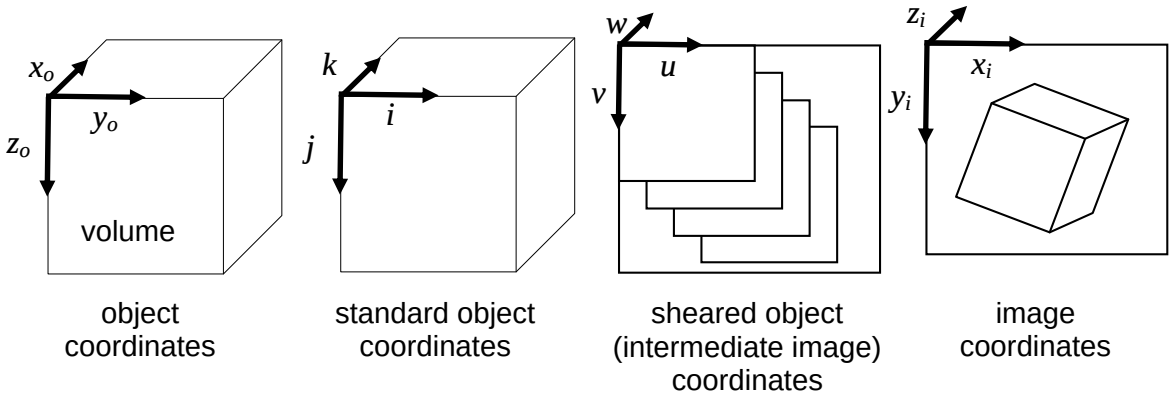


Figure 5.4: Four coordinate systems used in the shear-warp factorization

The viewing transformation is given by a view matrix M_{view} that transforms points from the object space into the image space.

$$\begin{bmatrix} x_i \\ y_i \\ z_i \\ w_i \end{bmatrix} = M_{view} \begin{bmatrix} x_o \\ y_o \\ z_o \\ w_o \end{bmatrix}$$

Our goal is to find a shear matrix M_{shear} and a warp matrix M_{warp} so that:

$$M_{view} = M_{warp} \cdot M_{shear} \cdot P$$

The shear matrix transforms the object space into the sheared object space. The warp matrix transforms the sheared object space into the image space.

The principal viewing axis is the axis of the object coordinate system that is most parallel to the view direction. To eliminate the three possible cases, we transform the object space into the standard object space. In the standard object coordinates, the principal axis is the *k* axis. We can get from the object coordinate system to the standard one by permuting the axes using a permutation matrix *P*. When the principal axis corresponds to the *z* axis, the permutation matrix is the identity matrix. When the principal axis is the *x* or *y* axis then

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ or } P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ respectively.}$$

The principal viewing axis can be found very easily. As we know the view direction vector v_o, we just find the vector element with the maximum absolute value.

The directional vector can be also computed as

$$\vec{v}_o = \begin{bmatrix} m_{12}m_{23} - m_{22}m_{13} \\ m_{21}m_{13} - m_{11}m_{23} \\ m_{11}m_{22} - m_{21}m_{12} \end{bmatrix} \text{ where } m_{ij} \text{ are elements of } M_{view}.$$

Let M'_{view} be a permuted view matrix that transforms standard object space into the image space:

$$M'_{view} = M_{view} P^{-1}$$

The view direction in the standard object space is:

$$\vec{v}_{so} = P \cdot \vec{v}_o$$

After transforming to the sheared object space, this direction must be perpendicular to the (i, j) plane. The shear matrix has the following form:

$$M_{shear} = \begin{bmatrix} 1 & 0 & s_i & t_i \\ 0 & 1 & s_j & t_j \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The shear coefficients s_i and s_j are equal to:

$$s_i = -\frac{v_{so,i}}{v_{so,k}} = \frac{m'_{22}m'_{13} - m'_{12}m'_{23}}{m'_{11}m'_{22} - m'_{21}m'_{12}},\ s_j = -\frac{v_{so,j}}{v_{so,k}} = \frac{m'_{11}m'_{23} - m'_{21}m'_{13}}{m'_{11}m'_{22} - m'_{21}m'_{12}},$$

where m'_{ij} are elements of M'_{view}.

The k-th slice (labeled $0..k_{max}$ in front-to-back order) will be translated relative to the previous one by s_i and s_j in the i and j direction respectively.

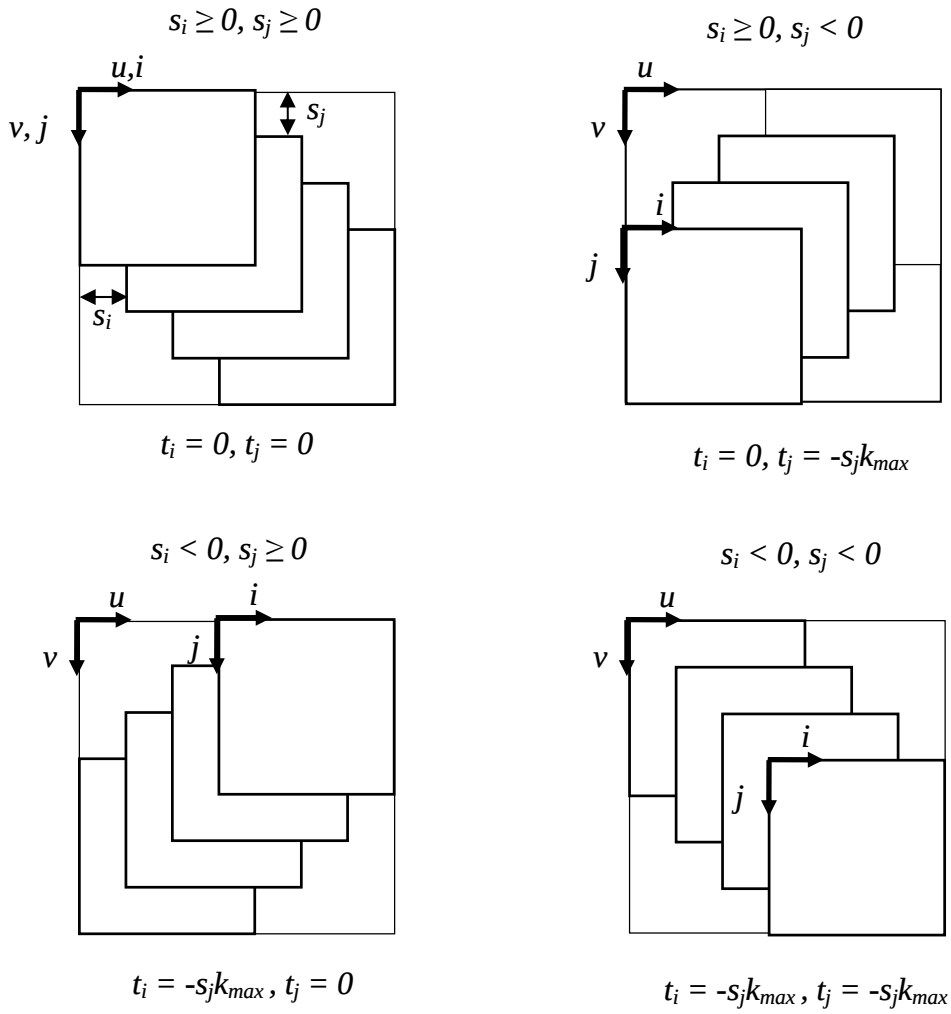


Figure 5.5: Four cases of displacement computation

The translation coefficients t_i and t_j in the shear matrix specify the difference between the origin of the standard object space and the sheared object space. Projection of the 0-th slice starts at the position $[t_i, t_j]$ in the intermediate image space. See Figure 5.5 for details.

The last thing we need is to compute is the warp matrix. We get it as:

$$M_{warp} = M'_{view} M^{-1}_{shear} = M'_{view} \begin{bmatrix} 1 & 0 & -s_i & -t_i \\ 0 & 1 & -s_j & -t_j \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Because the warp operation is performed in 2D, the third coordinates do not have to be taken into account. We can omit the third row and column and get a 3x3 warp matrix:

$$M_{warp2D} = \begin{bmatrix} m'_{11} & m'_{12} & m'_{14} - t_i m'_{11} - t_j m'_{12} \\ m'_{21} & m'_{22} & m'_{24} - t_i m'_{21} - t_j m'_{22} \\ 0 & 0 & 1 \end{bmatrix}$$

For a more detailed derivation see [Lacroute95].

5.3 Acceleration Using RLE

Traversing the volume in the storage order allows us to introduce further improvements of the shear-warp algorithm efficiency. Usually, the image-order algorithms are able to use the image coherency together with early ray termination to achieve faster visualization, but are not as efficient in exploiting the spatial coherency that involves traversing special data structures (e.g. octrees) which becomes slow as the same structure is often traversed multiple times. On the other hand the object-order algorithms can effectively use the spatial data structures, but can hardly take advantage of the image coherency.

The way of speeding up the shear-warp algorithm, as presented in [Lacroute95], uses the fact that the voxel scan lines are parallel to the scan lines of the intermediate image. This property enables taking advantage of both image and spatial coherency.

To exploit the spatial coherency, the voxel rows are encoded using the run the length encoding (RLE). In the preprocessing step, the voxels are classified as transparent or non-transparent according to the current threshold value and encoded. In [Lacroute95] the rows of voxels are encoded separately. When deciding whether to resample a voxel, information from both the neighbouring voxel rows is needed. If there is a non-transparent voxel run at least in one of the rows then the resampling proceeds, see Figure 5.6.

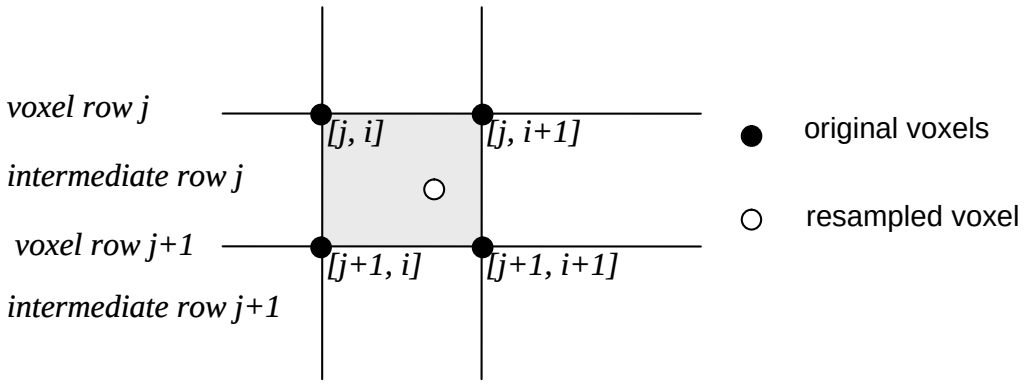


Figure 5.6: Resampling of a voxel using two neighbouring rows

To avoid taking two rows into account, a slightly different approach in encoding the data was chosen in this thesis. In this approach the intermediate rows are encoded (see Figure 5.6). The intermediate row consists of 2D cells. The cell at position [j, i] is created by voxels [j, i], [j, i+1], [j+1, i] and [j+1, i+1]. In the preprocessing we find the maximum scalar value of these four voxels. If the maximum value is lower then the threshold, the cell is encoded as a transparent one, otherwise as a non-transparent one.

In [Lacroute95] this resampling technique was used in combination with alpha-blending composition of computed colours to enable visualization of semi-transparent tissues. However, in this thesis just one fully opaque iso-surface is considered and thus no blending is used. Since just one colour value contributes to each pixel, the image quality considerably depends on its accuracy. After implementing this method, the tests showed that even tough the bilinear interpolation as shown in the Figure 5.6 should give better results than the nearest neighbour interpolation, the results were similar. It is caused by the absence of the interpolation between slices which would need an extension of current algorithm. Therefore the nearest neighbour interpolation was finally implemented and used for comparison with other methods. This simplification made the preprocessing and resampling step computationally less expensive. Due to the nearest neighbour interpolation, the data is treaded in the same way as in the discrete ray-casting algorithm (as voxels – see Figure 2.1) and the image quality is comparable as well.

The classic RLE encoding does not allow us to access the encoded data randomly. The simple data structure in Figure 5.7 enables a fast access to next non-transparent cell.

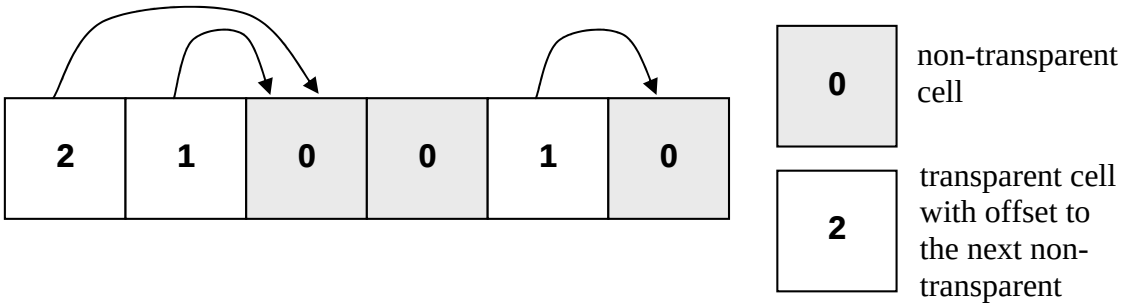


Figure 5.7: The data structure for a fast access to non-transparent cells

When projecting the volume slices onto the intermediate image, we switch between traversing the image (using image coherency, see further) and traversing the row of cells. When switching from the image traversal to the volume traversal, we have an index to the 2D cell to be processed. To avoid processing the transparent cells, we need to find the next non-transparent cell as fast as possible. This is achieved by looking into the data structure (Figure 5.7). The cell containing a zero value is non-transparent and needs to be resampled, otherwise the cell is transparent and the value is an offset to the next non-transparent cell.

As the way the volume is being sliced depends on the principal viewing axis, three such data structures need to be constructed in the preprocessing. As the viewing direction changes, we switch between them. Because one byte is used for each cell, this auxiliary data structures take three times as much memory as the volume data. However, this is bearable for small 256^3 volumes. In case of larger datasets the original structure used in [Lacroute95] should be used to save the memory since it does not store the transparent voxels.

To exploit the image coherency, the similar structure is used. At the beginning all the pixels in the intermediate image are empty. As the projection of the volume slices progresses, the pixels get a colour value. After a pixel gets a colour value, all other parts of volume that would be projected onto this pixel are hidden (the slices are taken in front-to-back order). Thus we do not need to resample the volume in the point that projects onto the full pixel. Because the pixel scan lines are parallel to the rows of cells as they are traversed, by using an auxiliary data structure we can easily skip the cells that would be projected into the full pixels. The only problem is that the data structure is not constant as in the case of the volume, but needs to be updated each time a pixel is coloured. We need to add a new full pixel as quickly as possible but want to maintain the structure of offsets at the same time. Marking the pixel as full is simply done by putting the offset value 1 instead of 0 which meant that the pixel was empty. The offset 1 points at the next pixel. Adding a

new pixel to the existing run of full pixels using this simple algorithm creates a kind of tree where the root is the first empty pixel behind the full pixels (see Figure 5.8 a)).

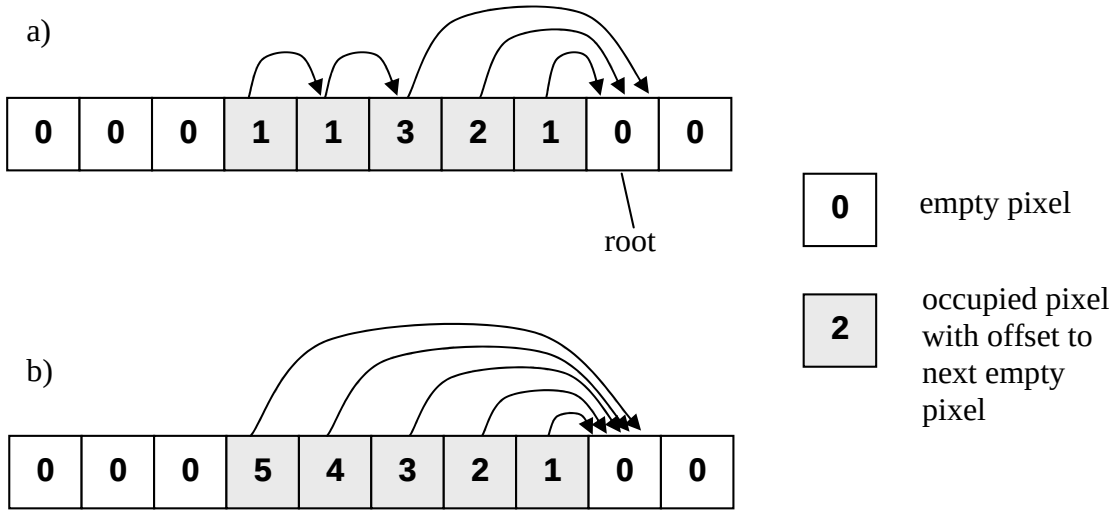


Figure 5.8: Auxiliary data structure for pixel traversing

To maintain this structure suitable for the fast empty pixel search (i.e. to make the full pixels point directly at the root), an additional step is taken when searching for the root. Each time the algorithm goes through the list of pixels pointing at each other, after finding the root, it goes through the pixels again and changes the offsets so that they point at the root (Figure 5.8 b)).

The projection algorithm can be written as follows:

```
FOR each slice in FTB order
    Compute slice displacement within the intermediate image
    Compute resampling coefficients
    FOR each slice row
        pCell ß 1st row cell
        pPixel ß corresponding pixel
        WHILE not end of row
            IF IsFull(pPixel)
                Skip full pixels and corresponding cells
            ELSE IF IsTransparent(pCell)
                    Skip transp. cells and corresponding pixels
                ELSE
                    Image[pPixel]=ResampleAndShade(pCell);
                    SetPixelAsFull(pPixel);
        END WHILE
    END FOR
END FOR
```

The `ResampleAndShade(pCell)` step was finally due to the nearest neighbour interpolation simplified to `Shade(pVoxel)`. The gradient is computed using the central difference and the colour value is obtained as shown in chapter 3.5.

5.4 Warping the Image

The final step towards the resulting image is warping the intermediate image. There are two possibilities of warping the image. Fist one is the direct application of the 2D warping matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = M_{warp2D} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix},$$

where $[x, y]$ are coordinates of a pixel in the intermediate image and $[x', y']$ the coordinates in the final image to which is the pixel mapped. The advantage of this solution is that we do not have to transform the pixels that have the background colour. The intermediate image must be large enough to fit the volume projected in any possible position and as the result, in some positions less than 25% of the image is used. The intermediate image size is *2n*x*2n*, where n is the maximal number of voxels per side.

This solution has also a serious disadvantage. For some pixels of the final image may happen that no source pixel is mapped to them. This results in holes in the final image that are usually uniformly spread and create a pattern.

To avoid this effect, an inverse mapping must be applied. In other words, we need to find a source pixel for every destination (final image) pixel. This means finding an inverse matrix to the warping marix.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = M^{-1}_{warp2D} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

When warping the image, we go through every final image pixel and find the corresponding coordinates in the intermediate image. As the coordinates $[x, y]$ are floating-point numbers, the source point lies generally somewhere between four pixels. We can use this for bilinear interpolation. However, the tests I made showed, that the warping process must be carefully optimised otherwise it becomes a bottleneck. Thus just the nearest neighbour interpolation was used together with integer arithmetic and look-up tables to get most of the computations out of the loop and make the warping time be a fraction of the total time.

5.5 Properties

The complexity of the brute-force shear-warp algorithm is $O(n^3)$ where n is the number of voxels per one side of the volume. Since it is an object-order algorithm, the complexity does not depend on the image size. For the accelerated shear-warp technique as described here, the worst case complexity is still $O(n^3)$. However, for tested datasets the observed complexity (see the table in Figure 7.6) is approximately $O(n^2)$. The reason is that even though every voxel scan line (n^2 scan lines) needs to be processed, the number of voxels that contribute to the image is proportional to n^2 and thanks to the long voxel and pixel runs the other can be effectively skipped

The memory requirements of the algorithm are $4N$ bytes, where N is number of voxels in the dataset. N bytes are taken for storing the dataset and N bytes for each of the three auxiliary data structures used for the encoded volume.

5.6 Other Acceleration Techniques

There are other acceleration techniques that use the shear-warp factorization. For example in [Csébfalvi98] a fast method is described that classifies the volume into a binary volume and encodes each voxel as one bit in 32bit integers. The 32bit strips are always perpendicular to the projection plane and help to efficiently find the first ray-surface intersection voxel. Although the classified data is physically sheared, the moving is fast as 32 voxels are handled at the same time. The presented algorithm is efficient for small shearing steps, i.e. rotation by small angles

Other method based on shear and warp was presented in [Csébfalvi99]. A great amount of work is done in the preprocessing step where the volume is classified and voxels that create the surfaces are selected. It means that voxels that are transparent and voxels that are inside the objects and hence are not be visible from any viewpoint are discarded. Just the voxels creating the surfaces are stored and rendered slice by slice in back-to-front order using painter's algorithm to solve visibility.

6 Ray-Casting Acceleration Using 3D Hardware

The ray-casting algorithm is a very simple method that enables to easily implement various modifications like cutting planes. The algorithm is also very easy to parallelize and no preprocessing is needed when the threshold value is changed. However, the algorithm is relatively slow due to the long time is spends traversing the volume until the iso-surface is hit. A space leaping technique is presented in this paragraph that I developed to quickly find the limits in that is the ray going to be sampled and thus to avoid the empty space sampling. The standard 3D graphic hardware is used for this purpose. The implementation was done using the OpenGL library.

6.1 Creating the Super-Cells

We divide the volume into super-cells. Each super-cell is a cube made of k^3 cells ($(k+1)^3$ voxels), $\lceil (n-1)/k \rceil$ super-cells are created per one side of the volume, where n is the number of voxels per volume side. If the volume size is not a whole multiple of k, the super-cells at borders are smaller.

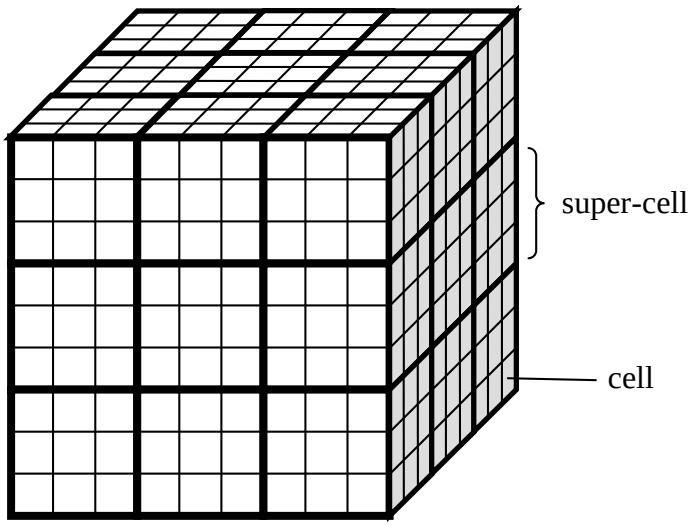


Figure 6.1: An example of super-cells created by 3^3 cells (4^3 voxels)

The super-cells actually play the role of the bounding boxes in this acceleration. The super-cells are created in the preprocessing. In the auxiliary 3D array, one value is stored for every super-cell – the maximum scalar value of the voxels that create the super-cell. This data structure is independent on the current threshold value.

6.2 Rendering the Super-Cells

As we know from the section 3.2, the ray is given by the equation $X(t) = A + \vec{d} \cdot t$. This algorithm tries to restrict the parameter *t* so that the ray is then sampled along a shorter path. The restriction is achieved with the help of the hardware depth buffer.

The super-cells that have the maximal value greater or equal the threshold value are rendered using OpenGL. They are rendered twice. Once with the depth function set to less (closer objects hide the more distant) and the second time with the function set to greater. After each rendering, the depth buffer is retrieved into the buffers in the main memory. The buffers have the same size as the rendered image. The buffer values are then used as bounds for the cast rays. The super-cells are rendered as blocks using rectangles.

Of course not all super-cells with value greater than the threshold need to be rendered as some may be hidden. Neither all six faces are rendered because just three can be visible at the same time. These visibility aspects are considered in another preprocessing made every time the threshold value changes. In the Figure 6.2, we can see the six principal directions. The faces that create the super-cells are perpendicular to these directions. In the preprocessing six lists of visible faces are created. Each list corresponds to one principal direction and contains faces perpendicular to this direction with respect to visibility (just the front face is selected). The lists are stored in GL lists to be fast rendered afterwards.

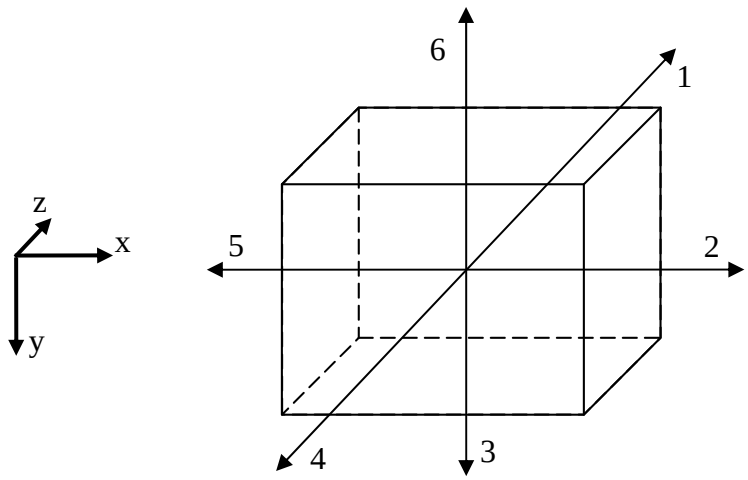


Figure 6.2: The six principal directions

An arbitrary viewing direction can be composed of three principal directions. For example the viewing direction with all positive coefficients is composed of principal directions 1, 2 and 3 (see Figure 6.2). We first render the lists corresponding to these three directions and store the depth buffer values into the first array. Then the other lists corresponding to the opposite directions (4, 5 and 6) are rendered with depth function set to greater (we render what would be seen from the opposite side of the volume) and the depth buffer values are

saved into the second array. Values from the buffers are used as restriction bounds for the rays. Therefore the ray enters the volume in the first nonempty super-cell along its path and unless the iso-surface is found it leaves the volume after the last nonempty super-cell is sampled.

6.3 Solving the Visibility of the Faces

In order not to visualize the hidden faces a simple algorithm was used. It does not remove all hidden faces but on the other hand it is quick.

The visibility is taken into account when the lists of faces are compiled. The face is added to one of the six lists if one empty and one non-empty super-cell share the face. It is obvious that this approach also selects the faces that create the interior surface (see Figure 6.3). An improved algorithm should be used to remove the interior faces. However, this would cost more time in the preprocessing.

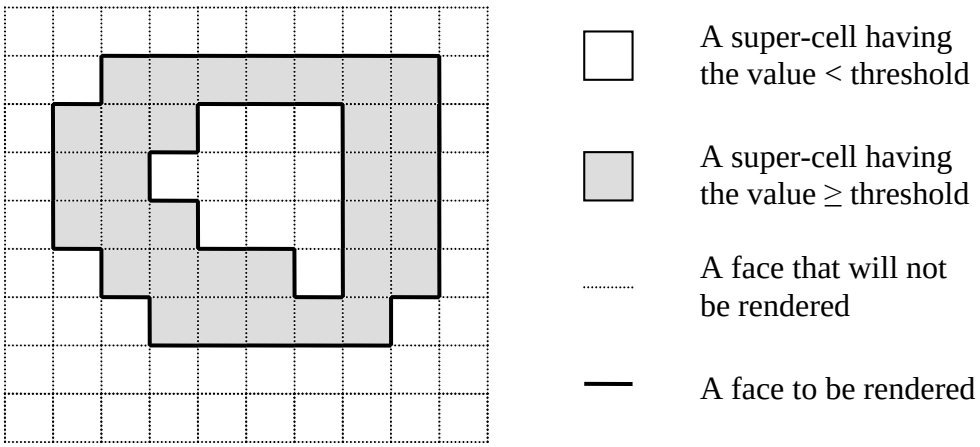


Figure 6.3: Visibility of the super-cell faces

6.4 Properties

The worst-case asymptotic complexity of any of the the ray-castings from chapters 3 and 4 with using the hardware acceleration is $O(f+m^2n)$ where n is the number of voxels per one side of the volume, m is the image size and f is the number of rendered faces. This is for the case when the super-cells do not help at all to restrict the ray path. However, the expected complexity that more corresponds to the results in Figure 7.6 is $O(f+m^2k)$, where k is the size of the super-cell. The number of samples taken per ray is expected to be proportional to k.

The accelerating technique takes at most N additional bytes for the super-cell values.

7 Results

This chapter includes the tests that were made in order to compare speed of implemented methods.

The methods had been tested on a PC with Intel Celeron processor running at 900MHz, 640MB of main memory and a GeForce2 graphic card under the Windows XP operating system.

7.1 Tested Data

The methods were tested on real datasets.

Dataset	Size
syn_64.vol	64x64x64
ctmayo.vol	128x128x128
cthead.vol	256x256x113
engine.vol	256x256x110
bentum.vol	256x256x256

New dataset	Size
cthead_64.vol	64x64x64
cthead_128.vol	128x128x128
cthead_256.vol	256x256x256
cthead_512.vol	512x512x512

Figure 7.1: The tested datasets. The left table shows original datasets. The right table shows sizes of resampled datasets.

Although there is a variety of original data sizes, the datasets can not be compared with each other since the datasets include different objects. Furthermore some datasets have other scale than 1 along any of the axis. To compare properties of the algorithms on different data sizes the cthead dataset having originally 256x256x113 voxels was resampled to 64^3, 128^3, 256^3 and 512^3 volumes (see Figure 7.1).

The resampling was simply done by the trilinear interpolation. A 3D grid of the new resolution was introduced into the original volume and new scalar values were computed from the eight nearest original values using trilinear interpolation. The trilinear interpolation was chosen for its simplicity. The disadvantage is that it makes the step artefacts more visible; a more sophisticated filter should be used to avoid that. However, it is sufficient for the testing purposes.

7.2 Testing Principles

Since the visualization process depends on many conditions, the conditions have to be as equal as possible for all tested methods in order to get correct results. See the testing conditions in this chapter for details about performed tests.

The cthead resampled volumes (see the right table in the Figure 7.1) had been used for the performace tests. In next chapters they are referred as 64^3, 128^3, 256^3 and 512^3 volumes respectively.

Since the shear-warp method (chapter 5) is able to generate only intermediate images that are twice as big as is the size of the volume (e.g. 512^2 size images for 256^3 volume), the image sizes in tests are set to double size as well. This helps getting similar conditions for all methods.

The threshold value for visualization was set to 50 which corresponds to the skin surface.

Because the visualization depends on the viewing parameters, it is sensible to make the following tests:

Test A

The rotation around the *z* axis as shown in the Figure 7.2 a). This test was designed for the testing of the data storage order dependency. This test does not make any sense for the shear-warp method since the rotation around viewing direction is achieved by the final image warp.

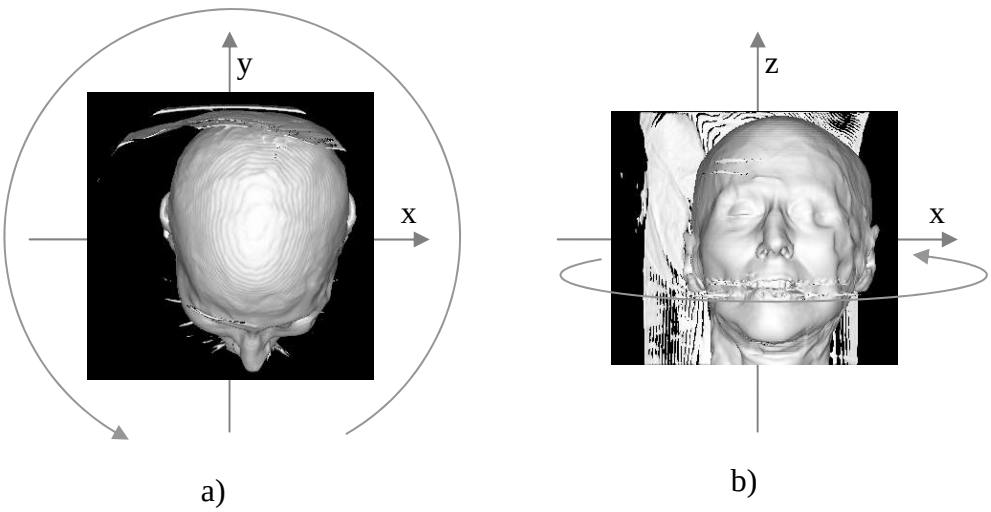


Figure 7.2: An illustration of performed tests, the volume is rotated around z axis, two different viewing directions are used: a) negative z axis; b) positive y axis

TestB

The rotation around *z* axis as shown in the Figure 7.2 b). The 360 degree rotation was done in 10 degree steps. In this test minimal, maximal and average times were computed.

Face View

If the face view is referred, it means that the positive *y* axis viewing direction is used as shown in the Figure 7.2 b).

For simplicity, the tested methods are referenced using following abbreviations:

RC – the brute-force ray-casting method described in chapter 3

DRC – the discrete ray-casting as described in chapter 4

RC HW – the RC method with hardware acceleration from chapter 6

DRC HW – the DRC method with hardware acceleration from chapter 6

S&W – the method using shear and warp factorization and RLE acceleration as described in chaper 5

The times were measured using the hardware performance counter. In all cases, the times include time needed for shading and storing the colour values into the image buffer; times for S&W include warping of the intermediate mage into the frame buffer. The times do not include drawing from the image buffer into the user window.

7.3 Hardware Acceleration

To test the acceleration method described in chapter 6 and compare it with other methods, the super-cell size has to chosen. Choosing the size is quite problematic. With decreasing super-cell size, the number of super-cells increases and there is more work for hardware to render them. On the other hand larger super-cells mean longer paths to sample until a surface is hit. See Figure 7.3 for an example of number of faces used to visualize the super-cells.

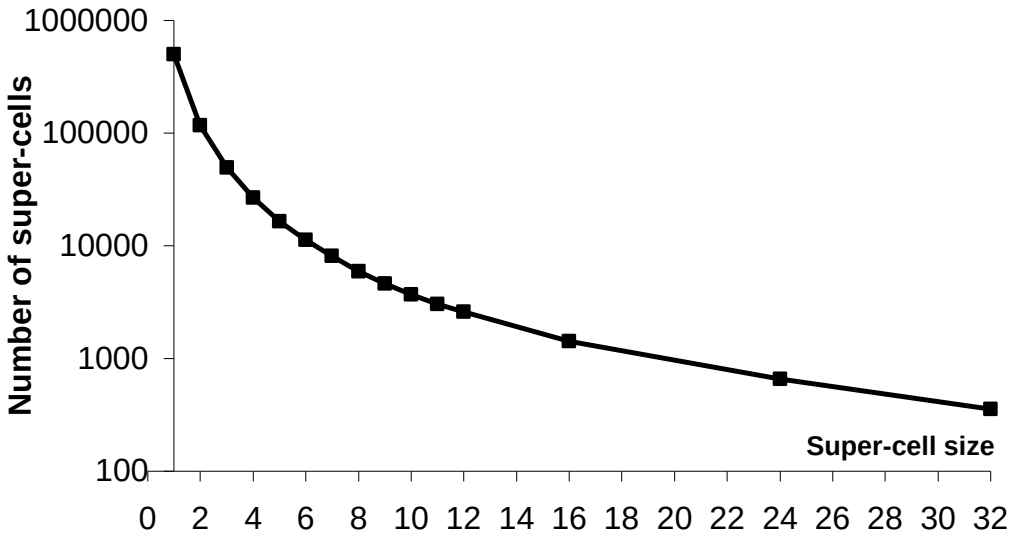


Figure 7.3: Number of generated faces to be rendered, shown in dependency on the super-cell size. The 256^3 volume was used.

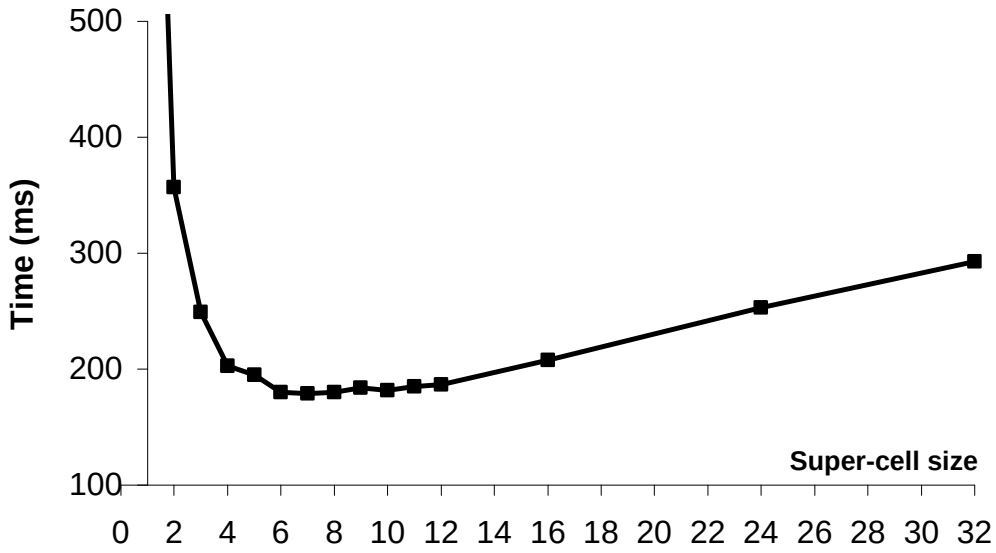


Figure 7.4: An example of rendering time dependency on the super-cell size. The DRC HW method and face view were used. The 256^3 volume was rendered, the image was 512^2. The best time (179ms) was achieved for the 7^3 super-cell.

The cell size for which the fastest rendering is achieved depends on used hardware, dataset being visualized, threshold value, used rendering method and image size. For tested datasets it was observed that the most efficient cell size does not depend on current viewing direction.

See Figure 7.4 for an example of the time dependency on the super-cell size. This dependency was investigated for both RC HW and DRC HW methods and all tested datasets. For the increasing number of super-cells, the hardware becomes the bottleneck. The resulting best super-cell sizes are listed in the table in Figure 7.5.

Data size	**Image size**	**RC HW**		**DRC HW**	
		Cell size	**Time (ms)**	**Cell size**	**Time (ms)**
64^3	128^2	3^3	32	8^3	11
128^3	256^2	3^3	128	6^3	42
256^3	512^2	3^3	577	7^3	179
512^3	1024^2	4^3	2580	7^3	735

Figure 7.5: The optimal super-cell sizes for different data volumes and corresponding image sizes.

The time courses look similar to that in the Figure 7.4. The differences between the rendering time for the optimal size and times for the neighbouring sizes are minor. Thus we come to an observation that the optimal super-cell sizes for the given rendering method and dataset resampled to different sizes does not change as long as the data size and image size ratio is constant.

7.4 Rendering Speed Comparison

This chapter contains speed comparison of implemented methods. For the following tests, the rotation Test B was used (see chapter 7.2 for details). In the Figure 7.6 the minimal, maximal and average times are shown that were achieved during the rotation.

	$63^3/128^2$			$128^3/256^2$			$256^3/512^2$			$512^3/1024^2$		
	min	max	**avrg**	min	max	**avrg**	min	max	**avrg**	min	max	**avrg**
RC	66	82	**75**	360	557	**470**	2502	4288	**3453**	17206	31932	**24956**
RC HW	31	43	**36**	110	161	**144**	490	714	**637**	1933	3129	**2721**
DRC	13	18	**15**	64	89	**76**	364	593	**473**	2308	4650	**3246**
DRC HW	10	14	**11**	39	49	**45**	149	209	**186**	624	863	**789**
S&W	3	4	**4**	12	18	**16**	59	101	**80**	286	528	**406**

Figure 7.6: The table showing result times in milliseconds. For methods RC HW and DRC HW, the optimal super-cell sizes (as shown in Figure 7.5) were used.

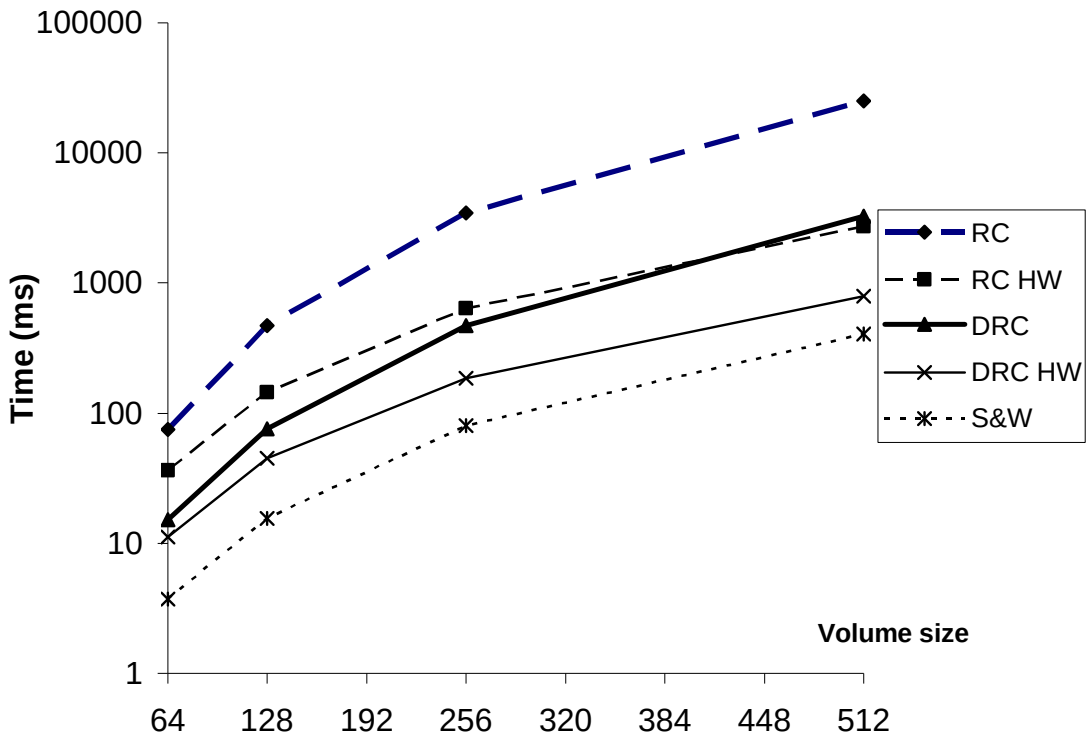


Figure 7.7: A graph showing the rendering time dependency on the volume size. The average values from the table in the Figure 7.6 were used.

See the Figures 7.8 and 7.9 that show the acceleration achieved for RC and DRC methods respectively.

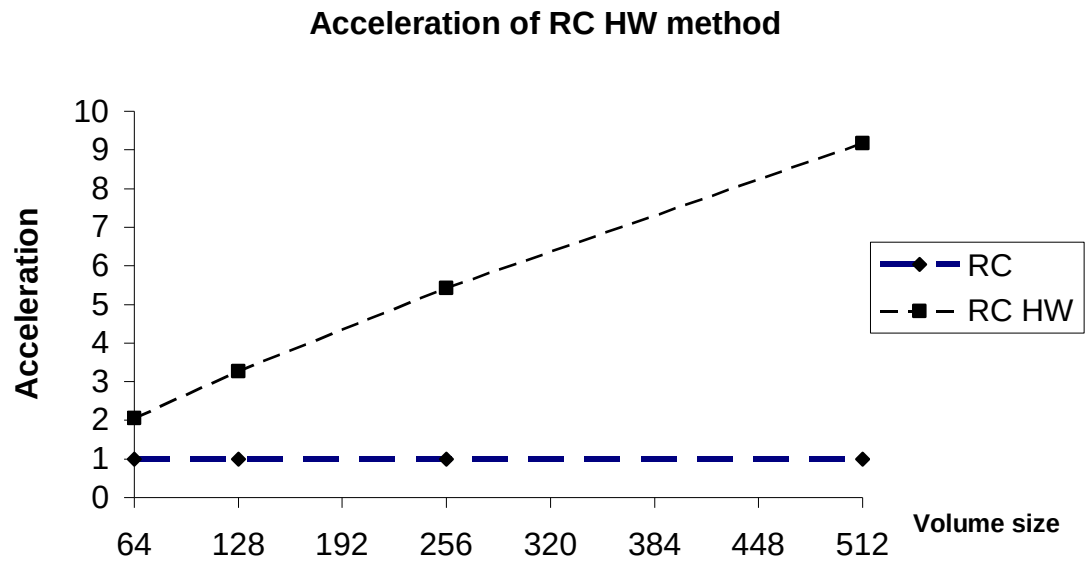


Figure 7.8: An acceleration of the RC HW method against the brute-force ray-casting.

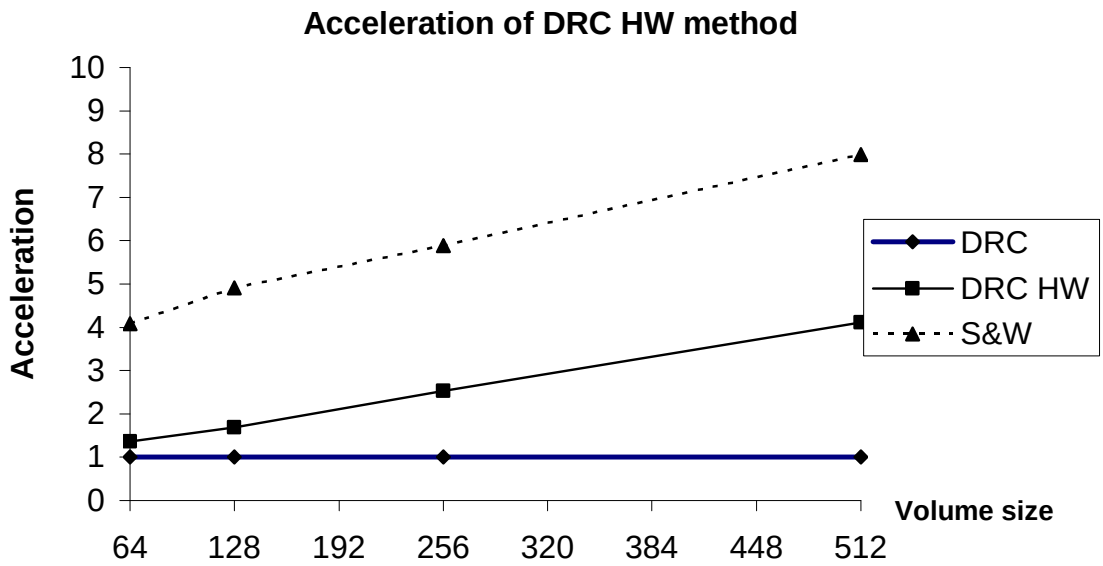


Figure 7.9: An acceleration of the DRC HW method against the brute-force discrete ray-casting. The shear-warp method is shown for comparison.

7.5 Data Storage Coherency

A test was made that shows how the order in which the data is accessed influences the rendering times. This issue is critical since the slow memory access becomes a bottleneck in traversing the data volume. As the CPU is much faster than the main memory, there is the local CPU cache that helps the processor to get the data faster. When there is a memory read request at an address, the neighbouring data is stored in the cache as well so that later it can be accessed faster. However, this assumes that the data is accessed in the storage order. Otherwise the cached data is replaced by data from other addresses since the cache size is relatively small.

To measure the access order dependency, the test A was used (see chapter 7.2 for details). The test is based on the fact that the same viewing direction is used and the view is being rotated around the viewing axis. This means that the same rays are sent however in different order.

The data voxels are stored in rows in the *x* direction (see Figure 7.2). The rows are stored in the *y* direction and create data slices. Finally, slices are stored in the *z* direction.

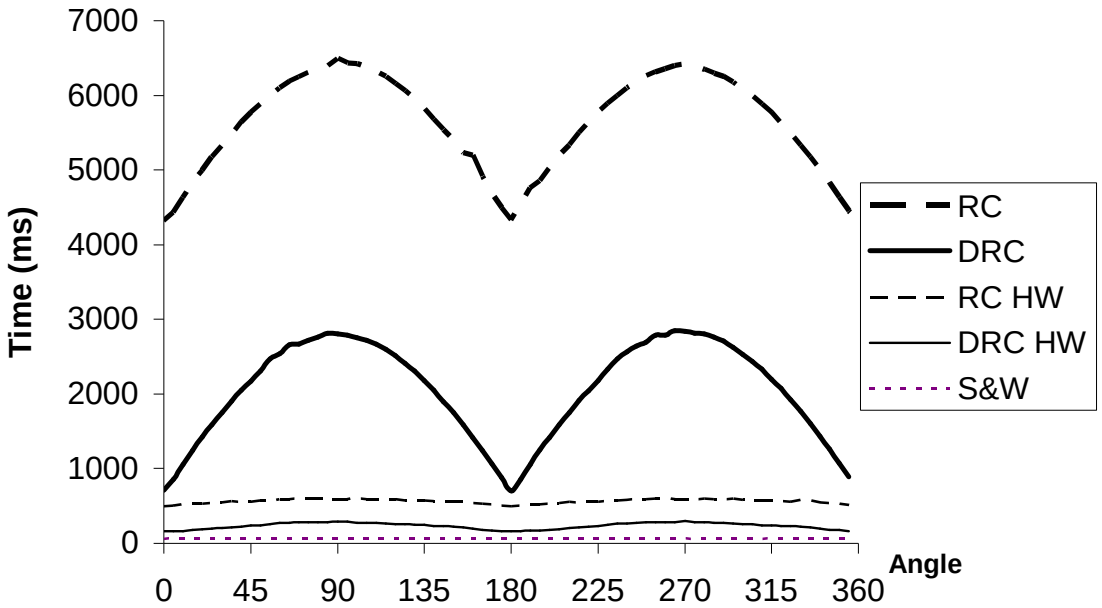


Figure 7.10: Rendering times vary depending on the rotation angle. It was measured for the 256^3 dataset in 5 degree steps.

The Figure 7.10 shows that the fastest times were achieved for 0° and 180° angles as the image rows were parallel to the data rows and the ray could take advantage of the data

cached by any of the previous rays. The slowest times were measured at 90° and 270° for image rows perpendicular to the data rows.

In the Figure 7.11 the previous results are expressed as a relative slowdown. The most noticeable slowdown can be observed for the discrete ray-casting method. In this method the time spent on one sample is minimized since no interpolation is done. The sampling loop thereby relies on the memory speed much more then the ray-casting method that does more computations in one step. Using the acceleration, the relative slowdown decreases since the rays do not traverse such long distances.

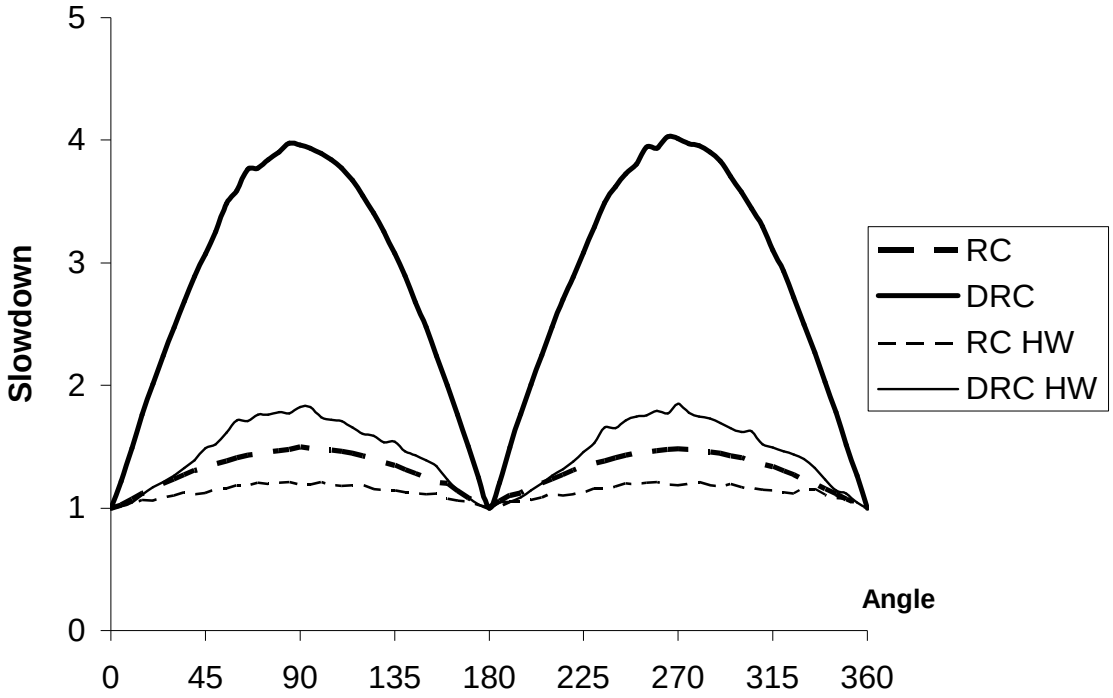


Figure 7.11: The relative slowdown expressed relatively to the time measured at 0 degrees.

7.6 Preprocessing Speed Comparison

Besides the rendering times that were discussed in previous chapters, there is another issue that has to be taken into account – the time needed for preprocessing. The preprocessing is used in RC HW, DRC HW and S&W methods each time the threshold value changes. The preprocessing time is not very important as long as the user does not need an interactive change of the threshold value. However, it becomes critical when rather threshold value changing than data rotation is needed. See Figure 7.12 for resulting preprocessing times. It can be observed that the preprocessing in the shear-warp method becomes with growing dataset relatively very slow. It is caused by the fact that it has to go through the whole

volume. The other two methods process just the super-voxels to determine which will be visible for the given threshold.

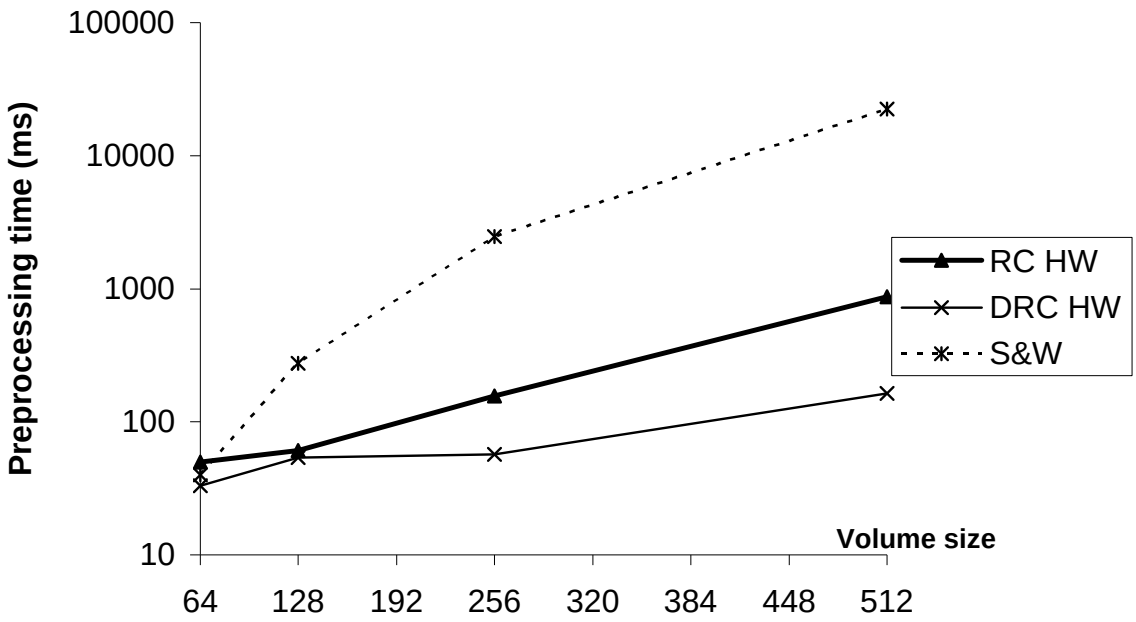


Figure 7.12: Times needed for preprocessing measured for 64^2, 128^2, 256^2 and 512^2 volumes and threshold equal to 50. For the RC HW and DRC HW methods, the rendering optimal super-cell sizes from Figure 7.5 were used. The RC and DRC methods are not included since they do not use any preprocessing.

However, in practice, when the user changes the threshold value, both the preprocessing and rendering must be done in order to give a response. We thus need to consider both times to see how quick the response for a threshold change is.

The Figure 7.3 shows the sum of the preprocessing times from Figure 7.12 and the average rendering times from Figure 7.7. Interesting results can be seen in that graph. Even though the RC and DRC methods do not need any preprocessing at all, the acceleration gain of the RC HW and DRC methods can easily compensate the low preprocessing costs.

However, notice that the high preprocessing costs of the S&W method make it unsuitable for interactive threshold changes since the total time becomes almost as slow as is that for the RC method.

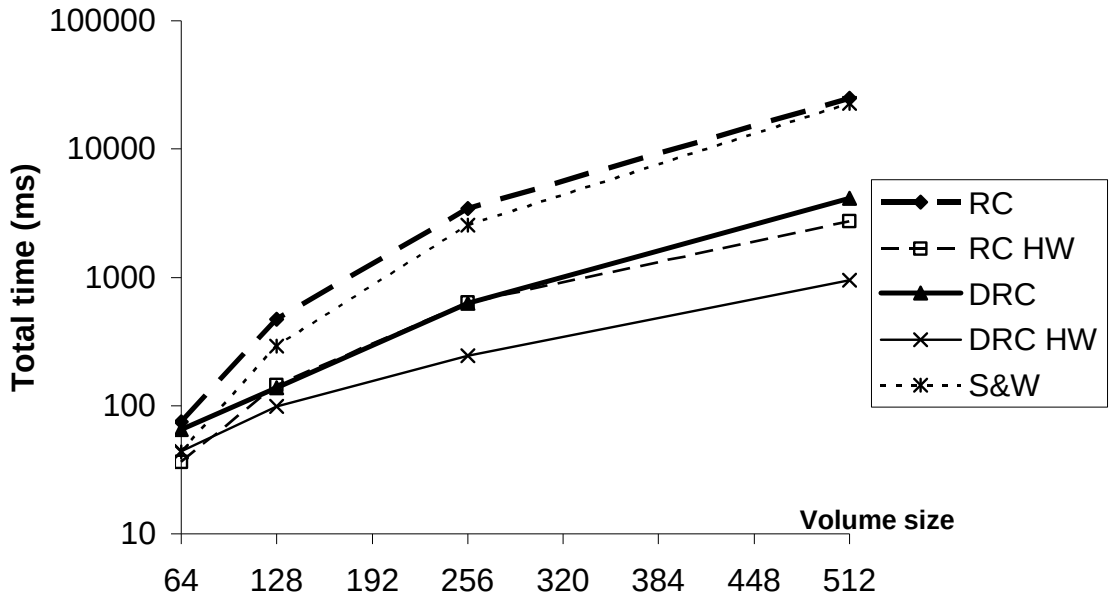


Figure 7.13: Times needed for a threshold change and following rendering.

Finally see Figure 7.14 for response speeds expressed in frames per second.

	Rotation (FPS)				Threshold change (FPS)			
	64^3	**128^3**	**256^3**	**512^3**	**64^3**	**128^3**	**256^3**	**512^3**
RC	13,33	2,13	0,29	0,04	13,33	2,13	0,29	0,04
RC HW	27,44	6,95	1,57	0,37	27,44	6,95	1,57	0,37
DRC	65,69	13,14	2,12	0,31	15,33	7,29	1,59	0,24
DRC HW	89,11	22,19	5,36	1,27	22,61	10,10	4,11	1,05
S&W	268,66	64,52	12,45	2,46	22,87	3,44	0,39	0,04

Figure 7.14: Table showing the frame rates for volume rotation in the left part (computed from time values in Figure 7.6) and for the threshold change response (include the threshold change and rendering).

7.7 Image Quality

This thesis is orientated to the rendering speed rather than to the output image quality. However, the image quality is a very important issue in volume data visualization. It depends primarily on the way the surface normal is computed and on the interpolation technique.

As two different interpolation approaches were used, the resulting images have different quality. The RC method gives the best results thanks to the trilinear interpolation it uses. The DRC and S&W methods use the nearest neighbour interpolation which trades the image quality for faster rendering times. The differences in the image quality are getting more obvious when more rays per one voxel/cell are sent. Furthermore the nearest neighbour interpolation used in the 2D warp in the S&W method causes the image causes slightly worse image quality than is that of the DRC method.

The acceleration technique used in the RC HW and DRC HW methods does not influence the output quality.

See Appendix C for a visual comparison of RC, DRC and S&W methods. There are also the RC method sample outputs for other datasets.

8 Conclusion

The ray-casting, discrete ray-casting and shear-warp methods were described and implemented in this thesis. The ray-casting method gives best image quality thanks to the inter-cell interpolation. The discrete ray-casting method is less computationally expensive and thus gives faster rendering times. However, the image quality is worse due to the nearest neighbour interpolation. An accelerating technique was introduced and implemented that uses the 3D graphic hardware to speedup the ray-casting. The acceleration does not influence the image quality as it only helps to skip the empty space.

It was shown that as the number of memory accesses increases, the algorithm speed becomes influenced by the memory access time. The speed of CPU is then not important as it has to wait for the slow memory. The data should be accessed so that advantage of the memory caching is taken.

The implemented shear-warp method achieved the fastest rendering times thanks to using both the image and data coherency and thanks to the effective way the data is accessed. However, the basic implementation does not enable the inter-cell interpolation that would have to be introduced in order to produce higher quality images. The shear-warp method has considerably slower preprocessing than the hardware accelerated methods which makes is less suitable for interactive iso-surface changing.

In practice, it is useful to have a combination of visualization techniques: a fast technique to interactively explore the data and another one that is slower but uses more precise interpolation and surface normal computation.

References

[Amanatid87] Amanatides, J., Woo, A.: *A Fast Voxel Traversal Algorithm for Ray Tracing*, Dept. of Computer Science, University of Toronto, 1987

[Bentum96] Bentum, M.: *Interactive Visualization of Volume Data*, Ph.D. thesis, University of Twente, 1996

[Carr96] Carr, J.: *Surface Reconstruction in 3D Medical Imaging*, University of Canterbury, New Zealand, 1996

[Cohen96] Cohen-Or, D., Rich, E., Lerner, U. and Shenkar, V.: *A Real-Time Photo-Realistic Visual Flythrough*, 1996

[Cohen93] Cohen, D., Sheffer, Z., *Proximity Clouds – An Acceleration Technique for 3D Grid Traversal*, Technical report FC 93-01, Math & Computer Science, Ben Gurion University, Beer-Sheva, 1993

[Csébfalvi98] Csébfalvi, B.: *An Incremental Algorithm for Fast Rotation of Volumetric Data*, Proceedings of Spring Conference on Computer Graphics, 1998

[Csébfalvi99] Csébfalvi, B., König, A., Gröller, E.: *Fast Surface Rendering of Volumetric Data*, 1999

[Kauf98] Kaufman, A.: *Volume Visualization: Principles and Advances*, State University of New York at Stony Brook, 1998

[Lacroute95] Lacroute, P., G., Levoy, M.: *Using a Shear-Warp Factorization of Viewing Transformation*, Technical Report, 1995

[Levoy89a] Levoy, M: *Volume Rendering by Adaptive Refinement*, Computer Science Department, University of North Carolina, 1989

[Levoy89b] Levoy, M: *Efficient Ray Tracing of Volume Data*, Computer Science Department, University of North Carolina, 1989

[Neumann00] Neumann, L., Csébfalvi, B., König, A., Gröller, E.: *Gradient Estimation in Volume Data using 4D Linear Regression*, Eurographics 2000, Volume 19, Number 3

[Žára98] Žára, J., Beneš, B., Felkel, P.: *Moderní počítačová grafika*, Computer Press, 1998

Appendix A – User's Guide

Running the Application

The application was tested and should run at Microsoft Windows NT/2000/XP/98. A 3D graphic card and installed OpenGL are required as well.

For a non-problematic testing of supported methods, it is recommended that the system main memory is at least 4 times as large as the largest dataset to be tested.

To run the application, execute the `\public\IsoVolume.exe` file that can be found on the enclosed CD.

Loading the Data

Choose the *File/Open* from the main menu. The open dialog enables loading the `*.vol` files. Go e.g. to the `\public\data` directory to get a sample dataset.

Setting the Viewing Parameters

When a dataset is loaded, the window containing user controls appears. Choose the *View* page to set the view parameters.

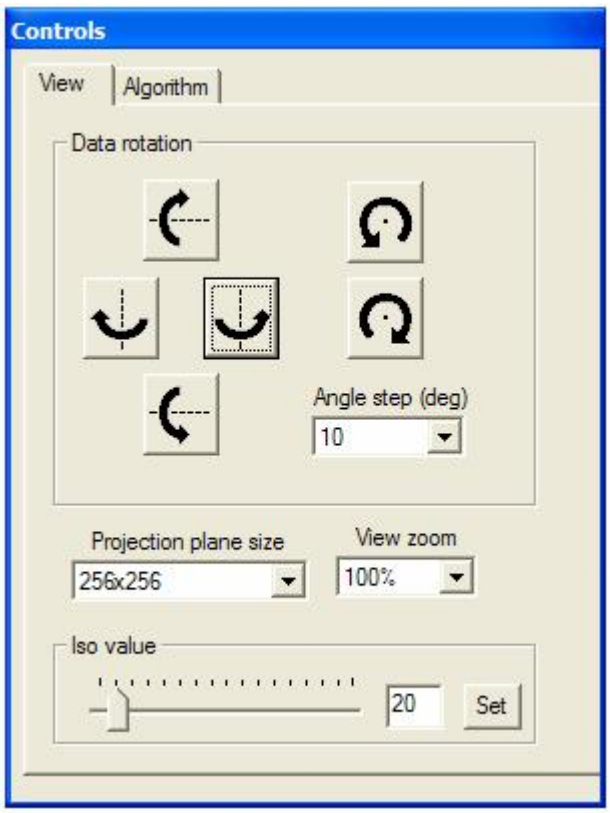


Figure A.1: Controlling the view parameters

Data rotation

To rotate the data, use the six rotation buttons as shown in the Figure A.1. The functionality is obvious from the button icons. In the left part of the panel, there are the four buttons that enable the rotation around axes perpendicular to the viewing direction. By pressing the other two buttons in the left part, the data rotates around the viewing axis.

Keeping the buttons pressed causes continuous rotation.

You can choose the rotation step to change the rotation speed. Choose the appropriate angle step in degrees from the combo box placed close to the rotation buttons.

Projection plane size

This combo box sets the number of rays that are sent per image. The default size is 256x256 which means that 256 rays are sent per image side. Together 256^2 rays are sent. If the zoom is set to 100%, resulting image size is 256x256 as one ray is sent per resulting image pixel.

This parameter considerably influences the image quality as well as the rendering speed. However, for the shear-warp algorithm the rendering time is not influenced as the internal size is twice the size of the dataset (e.g. 256^2 size for a 128^3 dataset) and using other image sizes means a warp from the optimal size to the required size.

View zoom

To resize the image, select the percentage value from the combo box. Selecting the zoom factor causes stretching the image to required size. It does not influence the rendering speed as no additional rays are sent.

Setting the threshold

In the lower part, the threshold value can be set. The control part is marked as *Iso value* (see Figure A.1). You can either use the slider or write the required value into the edit box and then press the *Set* button. The value can be in the range <1..255>.

Choosing the Algorithm

To select the current rendering algorithm, go to the *Algorithm* control page as shown in the Figure A.2.

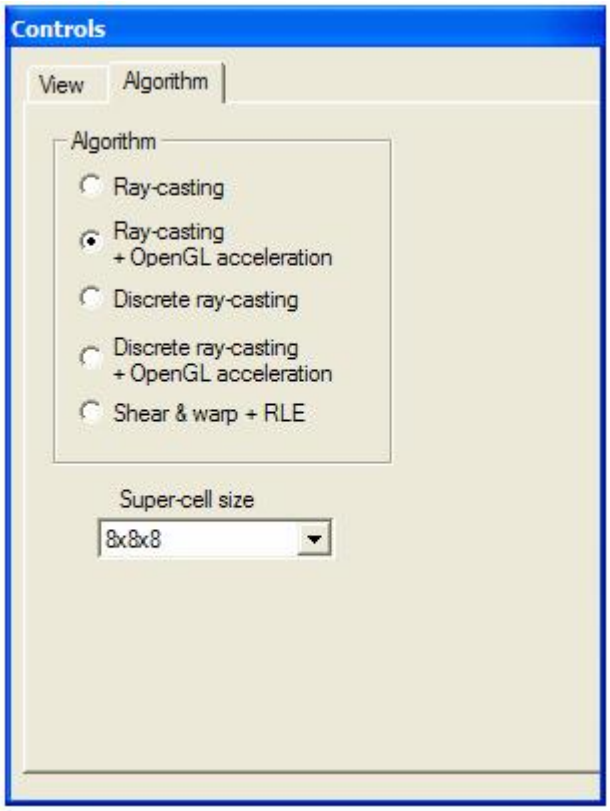


Figure A.2: Choosing the algorithm

In the *Algorithm* block use the radio buttons to choose current rendering algorithm. By changing the algorithm, the view parameters remain constant.

For the OpenGL accelerated algorithms use the combo box marked as ***Super-cell size*** to set the size of used super-cells.

Other Functions

See Figure A.3 that shows the rendering window with the main menu.

Saving the image

Use the *View/Save to BMP* command from the main menu to save the current image to a BMP file in current directory. The image is saved in original size, i.e. the zoom factor is not taken into account.

Logging the times

Use the *Debug/Start log times* and *Debug/Stop log times* to open and close the *times.log* file in the current directory. When the time logging is active, each time a frame is rendered the rendering time is written into the file.

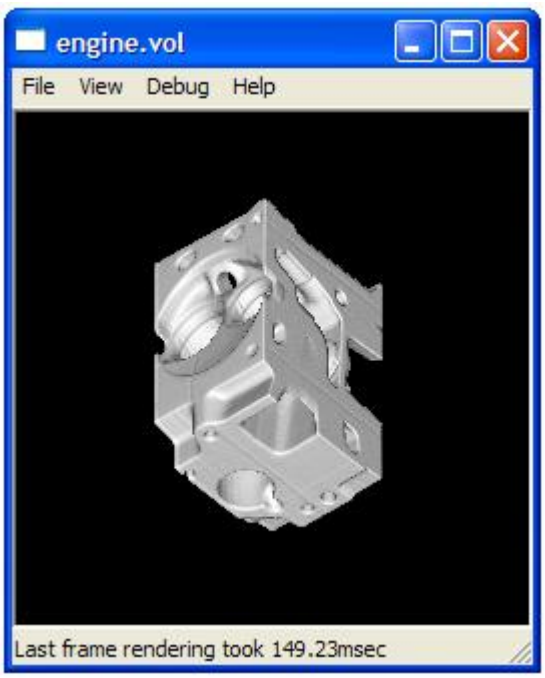


Figure A.3: The rendering window

Appendix B – Programmer's Guide

The implementation was done for MS Windows in C++ language using Visual C++ 6.0 and MFC. The source code can be found in the `\private\src` directory on the enclosed CD.

The project modules can be divided into following logical groups:

Framework classes

Class *CIsoVolumeApp* is placed in `IsoVolume.h` and `IsoVolume.cpp` files. This class was automatically generated by the project wizard. It includes basic application initialization as well as new *OnIdle* message handler used to perform the continuous rotation.

Class *CMainFrame* is placed in `MainFrm.h` and `MainFrm.cpp` files. It includes handlers for the main menu commands and messages sent to the main window frame. This class also creates instances of the *CChildView* and *CProperties* classes.

Class *CChildView* is placed in files `ChildView.cpp` and `ChildView.h`. This class handles the window painting and scrolling messages. It creates instances of the renderers according to the selected algorithm. It drives the visualization process. It contains instances of *CData*, *CCamera* and *CScreenBuffer* classes.

Class *CProperties* is placed in file `CProperties.h`. It is the frame for the window *Controls* and includes instances of *CCameraProperties* and *CAlgorithmProperties* classes.

Class *CCameraProperties* is placed in files `CameraProperties.cpp` and `CameraProperties.h`. This class creates the controls contained in the *View* page and handles messages from these controls.

Class *CAlgorithmProperties* is placed in files `AlgorithmProperties.cpp` and `AlgorithmProperties.h`. This class creates the controls contained in the *Algorithm* page and handles the messages from these controls.

Renderers

The abstract class *CAbstractTracer* is placed in file `AbstractTracer.h`. It defines which methods will be included in the renderers that are derived from this abstract class.

Class *CTracer* is placed in files `Tracer.cpp` and `Tracer.h`. It includes the methods that perform the brute-force ray-casting algorithm from chapter 3.

Class *CTracer3DLine* is placed in files `Tracer3DLine.cpp` and `Tracer3DLine.h`. It includes the discrete ray-casting algorithm from chapter 4.

Class *CTracerGL* is placed in files `TracerGL.cpp` and `TracerGL.h`. It includes the rendering ray-casting methods as implemented in the classes *CTracer* and *CTracer3DLine* together with the OpenGL acceleration from chapter 6. It contains an instance of the *CReducedVolume* class.

Class *CTracerSW* is placed in files `TracerSW.cpp` and `TracerSW.h`. It includes the shear-warp rendering algorithm from chapter 5.

Supporting classes and files

Class *CCamera* is placed in files `Camera.cpp` and `Camera.h`. It includes the view parameters and methods for view rotation.

Class *CData* is placed in files `Data.cpp` and `Data.h`. It includes the dataset as well as methods for reading the dataset from a file and adding borders to the dataset.

Class *CReducedVolume* is placed in files `ReducedVolume.cpp` and `ReducedVolume.h`. This class contains the super-cells data structure as well as methods for creating this data structure. It also includes a method that solves the face visibility and compiles the GL lists according to current threshold value.

Class *CScreenBuffer* is placed in files `ScreenBuffer.cpp` and `ScreenBuffer.h`. This class is used as an output for the renderers. Ii includes the screen buffer as well as methods for drawing the buffer into the window and storing the buffer into the BMP file.

The file `Def.h` contains various macros mostly for vector operations.

The file `DefMatrix.h` contains matrix operations that are used in the shear-warp renderer.

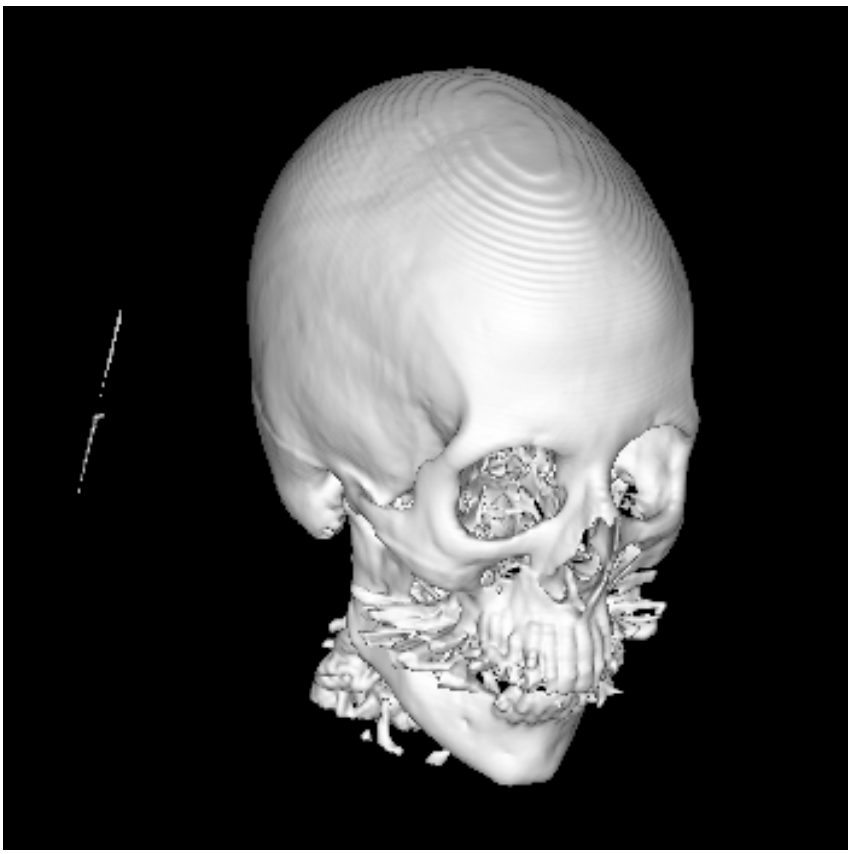
Files `OpenGL.cpp` and `OpenGL.h` include functions that initialize the OpenGL .

Files `Timer.cpp` and `Timer.h` support time measurements.

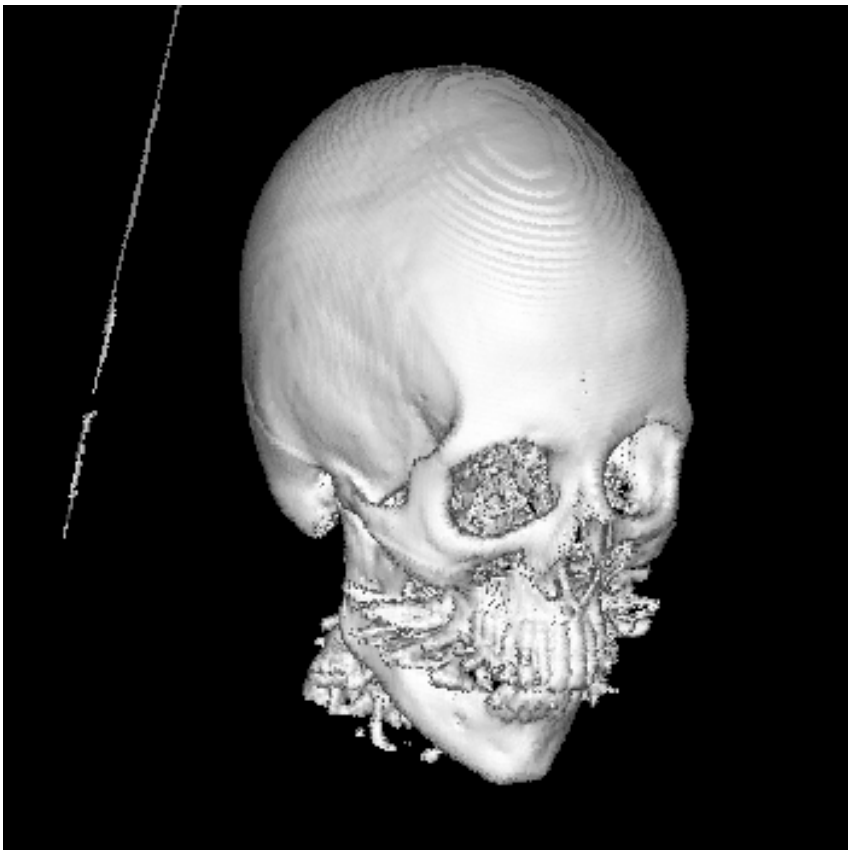
See the comments in the source code for more detailed information.

Appendix C - Output Examples

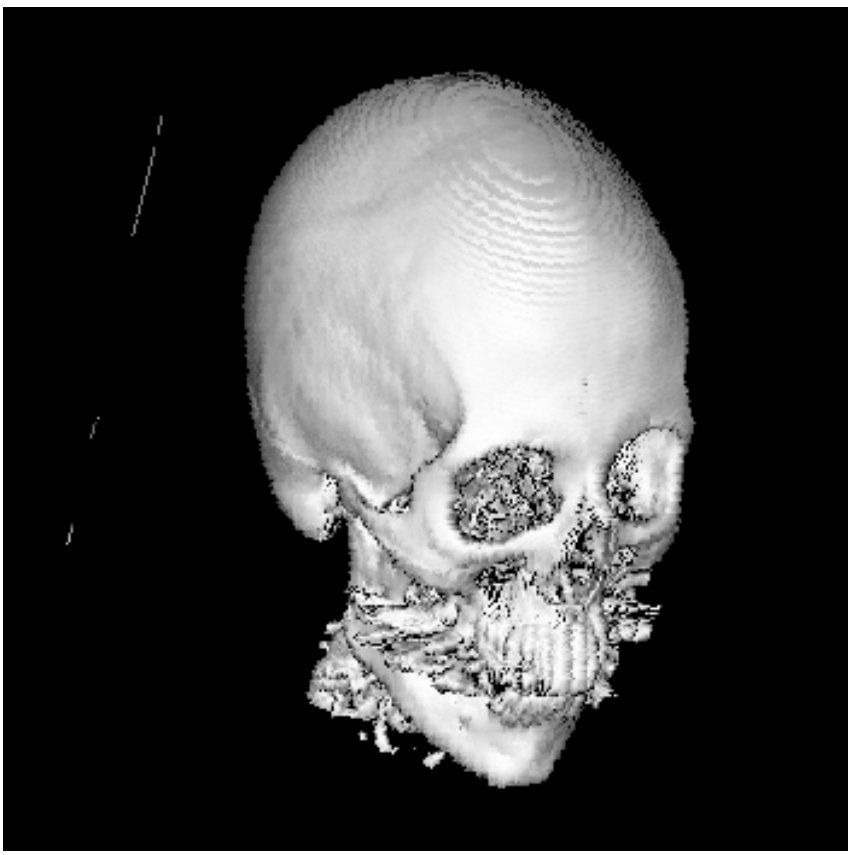
The next three images show outputs generated by the RC (ray-casting), DRC (discrete ray-casting) and S&W (shear-warp) methods respectively. The cthead_256.vol dataset was used (256^3 size). The threshold value was set to 100. The projection plane size was 512x512. The three images you can see below were obtained from the rendered images by cutting off part of the black background. Otherwise the skulls would be relatively smaller and the quality differences would be harder to recognize.



Ray-casting

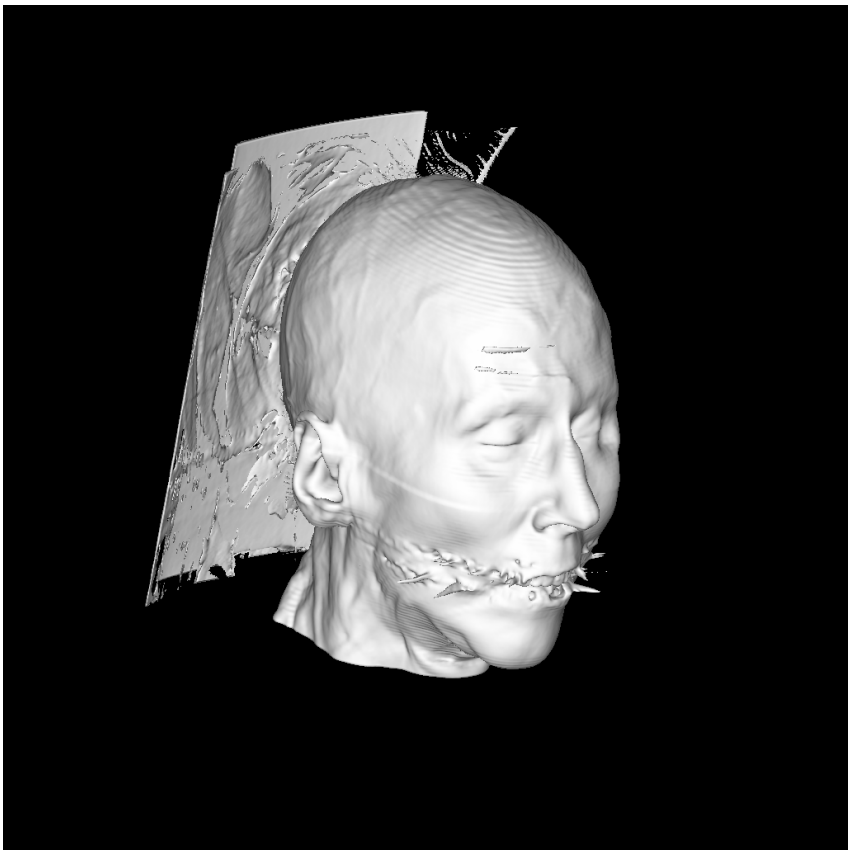


Discrete ray-casting

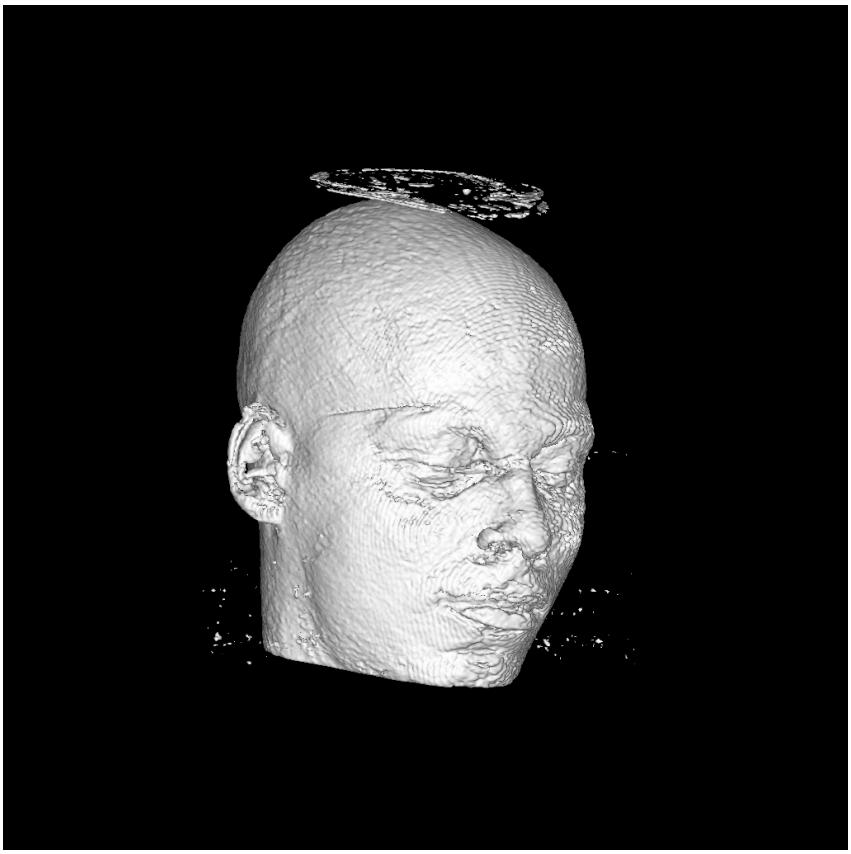


Shear-warp

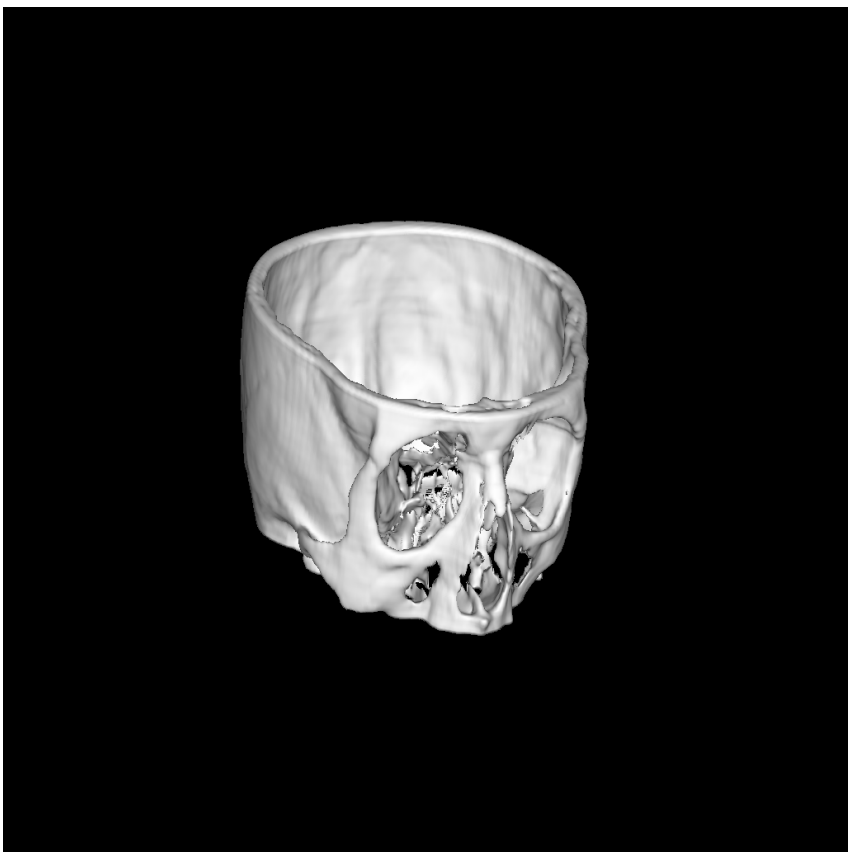
The following sample images were generated using the ray-casting method. The generated image sizes were 1024x1024. No cut-offs were applied. The source file name, dataset size and used threshold value are written under each image.



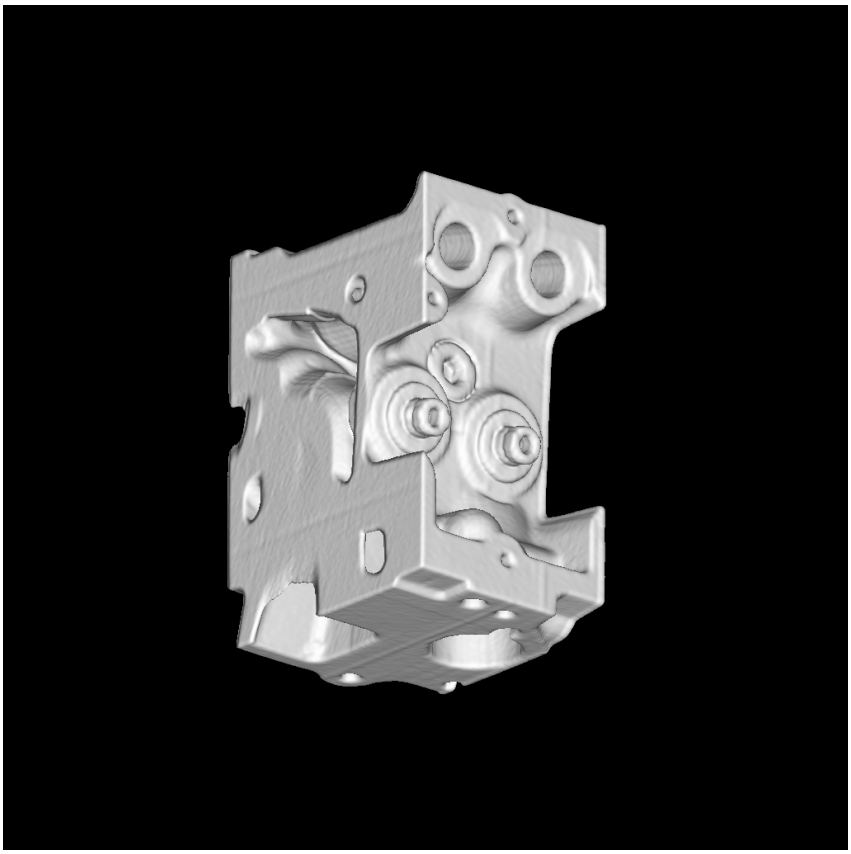
cthead_256.vol (256^3), threshold = 50



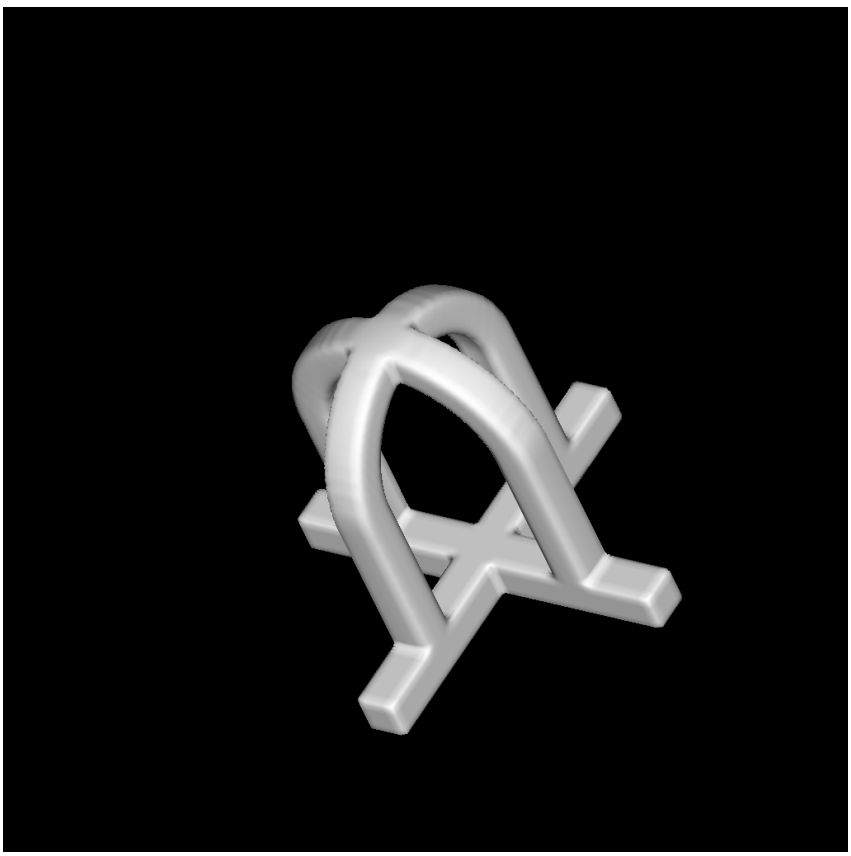
bentum.vol (256^3), threshold = 20



ctmayo.vol (128^3), threshold = 120



engine.vol (256x256x110), threshold = 50



syn_64.vol (64^3), threshold = 50

Evidenční list

Souhlasím s tím, aby moje diplomová práce byla půjčována k prezenčnímu studiu v Univerzitní knihovně ZČU v Plzni.

Datum: Podpis:

Uživatel stvrzuje svým čitelným podpisem, že tuto diplomovou práci použil ke studijním účelům a prohlašuje, že ji uvede mezi použitými prameny.

Jméno	Fakulta/katedra	Datum	Podpis