A Fast Method for Applying Rigid Transformations to Volume Data

J. Fischer, A. del Río
WSI/GRIS
University of Tübingen
Sand 14,
D-72076 Tübingen, Germany
{fischer, anxo}@gris.uni-tuebingen.de

ABSTRACT

Volume rendering is a widespread method for displaying graphical models in fields such as medical visualization or engineering. The required image information is stored in a volume dataset, which is a threedimensional array of voxel intensities. Sometimes it is necessary to transform a given volume dataset by geometric transformations like rotation, scaling or translation. In this process, a new volume dataset containing the transformed volume is generated. Straightforward algorithms with an emphasis on minimizing the resampling error are the standard approach to computing such transformations. In this paper we present a new algorithm, which is significantly faster than these. Since it is based on a simple interpolation scheme, it is useful for generating fast previews. Depending on their size and voxel bit depth, it can even provide realtime transformation of volumes.

Keywords
Volume Data, Rigid Transformations, Interpolation, Bresenham algorithm.

1. INTRODUCTION

Unlike many approaches for realtime display of 3D computer graphics, direct volume rendering is not based on drawing polygonal models. Instead the graphical data to be rendered is structured as a threedimensional array of image elements, the so-called voxels. Each voxel contains a value describing the material properties (e.g. density) at its location in the graphical model. Such volume datasets are usually acquired and preprocessed before rendering and remain static afterwards. During the display of the dataset, only the viewer position and orientation, as well as parameters for image generation, are altered. Direct volume rendering has been described by Levoy [Lev88]. A more recent overview is available in [SML98].

In some applications, though, it is necessary to apply changes to the volume data itself outside of the rendering process. In many cases this means applying geometric transformations like rotation, scaling or translation to the graphical model. Sometimes several datasets are combined to form a new one using a process called constructive volume geometry. A description of constructive volume geometry is given by Leu and Chen in [LC99].

In this work we investigate several methods for generating a new volume dataset from an existing one by application of a geometric transformation. Along with describing standard approaches for this task, which try to achieve small resampling errors, we also suggest a new algorithm designed to shorten computing times significantly. Since our algorithm is only capable of processing rigid transformations (i.e. only linear transformations, no scaling), we will limit our considerations to these.

In order to apply a geometric transformation to a volume dataset, two major steps have to be performed. At first, the transformed position has to be determined for every voxel. Then the new voxel intensities are computed for all grid positions in the resulting volume. The latter step is done by using an interpolation scheme to approximate the voxel values.

Different methods for performing and optimizing the two steps of volume transformation exist. They are discussed in detail in Section 2. The new algorithm, which combines both parts in a novel way, is described in Section 3.

In addition to the software-based methods that this paper focuses on, some research has also been done into special hardware for volume transformation. Examples of this related work include [TQ97] and [CK00].

2. STANDARD APPROACH

In this section the standard approach to transforming volume data is described. An implementation of this method can be found in the VTK software library [SML98]. We assume here that rigid transformation A is to be applied to a volume data set V. A is given as a 4-by-4 matrix containing a combination of rotations, translations and coordinate inversions.

$$A \in \mathbb{R}^{4x4}$$

$$V = \{v_{ijk}\}, \quad \begin{array}{c} i=1,\ldots,I \\ j=1,\ldots,J \\ k=1,\ldots,K \end{array} \tag{1}$$

As shown in Equation 1, the volume to be transformed, V, has K slices, each comprising J lines of I voxels. Without loss of generality we assume that the lower left corner of the volume dataset is located at the coordinate origin. The slices are to be parallel to the x-y-plane and to have increasing z-coordinates, starting at zero. We also require the volume to be isotropic and have a regular spacing of 1.0 as shown in Equation 2.

Given voxel index (i, j, k) for voxel v_{ijk}:

$$p_{ijk} = \begin{pmatrix} x_{p_{ijk}} \\ y_{p_{ijk}} \\ z_{p_{ijk}} \\ w_{p_{ijk}} \end{pmatrix} = \begin{pmatrix} i-1 \\ j-1 \\ k-1 \\ 1 \end{pmatrix} \tag{2}$$

In Equation 2, p_{ijk} denotes the position of voxel v_{ijk} in world coordinates. Also note that we have already introduced a homogeneous coordinate $w_{v_{ijk}}$, which is required for multiplication by a 4-by-4 matrix.

Possibilities for removing the isotropy and spacing preconditions are discussed in Section 5. The expected configuration of the volume dataset is depicted in Figure 1.

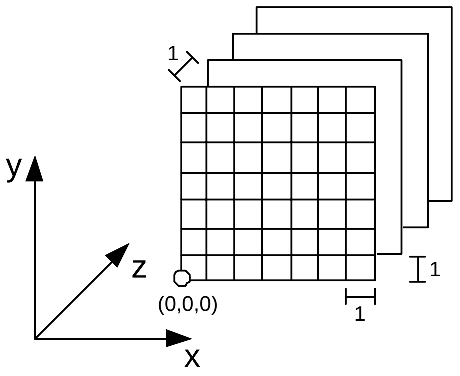


Figure 1: *Assumed isotropic and regular spacing of slices.*

2.1. Transformation of Voxel Positions

In order to transform the volume, it is processed sequentially. The slices are dealt with from front to back. Within each slice, first the voxel lines and then single voxels are considered in an order of increasing coordinates.

Each voxel position (i, j, k) that is generated in this process is transformed according to the given transformation matrix. This can be done by simply performing a matrix-vector multiplication.

$$\bar{p}_{ijk} = A \cdot p_{ijk} = A \cdot \begin{pmatrix} i-1 \\ j-1 \\ k-1 \\ 1 \end{pmatrix} \tag{3}$$

Equation 3 shows the transformation of voxel position p_{ijk} to its transformed position $\bar{p}_{ijk}$. Computing the matrix-vector product requires up to 16 multiplications and 12 additions, depending on the kind of transformations that are permitted. Even given the restrictions to rigid transformations and constant homogeneous coordinates, still at least 9 multiplications and 9 additions have to be performed. Note that the matrix-vector product must be calculated for every single voxel position in the entire volume. Thus it contributes significantly to the overall computational complexity.

However there is an effective optimization for this step of the process. It can be observed that the column vectors of matrix A represent the directions of the transformed unit vectors. This leads to a faster replacement of the matrix-vector product.

$$A = \left(\begin{array}{c|c|c|c} a_1 & a_2 & a_3 & a_4 \end{array} \right) \tag{4}$$

In Equation 4, matrix A is defined to consist of the column vectors a_1 to a_4. In the following discussion we will denote the coordinate origin as point p_0 and the coordinate system's unit vectors as e_1 through e_3.

$$p_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \; e_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \; e_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \; e_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

$$\begin{aligned} \bar{p}_0 &= A \cdot p_0 = a_4 \\ \bar{e}_i &= A \cdot e_i = a_i \qquad {\scriptstyle i \in \{1,\ldots,3\}} \end{aligned} \tag{5}$$

As shown in Equation 5, both the transformed origin and the transformed unit vectors correspond to columns of matrix A. Now every transformed voxel's position can be obtained through a linear combination of $\bar{e}_1$, $\bar{e}_2$, $\bar{e}_3$ and $\bar{p}_0$.

$$\bar{p}_{ijk} = \bar{p}_0 + ({\scriptstyle i-1}) \cdot \bar{e}_1 + ({\scriptstyle j-1}) \cdot \bar{e}_2 + ({\scriptstyle k-1}) \cdot \bar{e}_3 \tag{6}$$

Equation 6 is essentially a reshaping of the matrix-vector product described above. It can be evaluated incrementally while traversing the volume, as shown in the following code fragment:

```
curPos := $\bar{p}_0$
for k := 1 to K do
  curPos := curPos + $\bar{e}_3$
  for j := 1 to J do
    curPos := curPos + $\bar{e}_2$
    for i := 1 to I do
      curPos := curPos + $\bar{e}_1$
      computeInterpolation(curPos)
    done
  done
done
```

This optimization reduces the computational costs to 3 additions in the inner loop, as well as the same amount in each of the outer loops.

2.2. Reversal of Processing Direction

Until now we have assumed in our discussion that the voxels of the original volume are transformed into the target volume's coordinate system. Whereas this method is suitable for performing the first step of the process, it is not practical for the subsequent voxel value interpolation.

By transforming a voxel position according to matrix A, in most cases a non-integer position in the target volume is obtained. This makes it very difficult to distribute the original voxel's intensity value among its neighbours in the target coordinate system. Moreover it can not be ensured that every target voxel is assigned a value, especially when using nearest-neighbour interpolation.

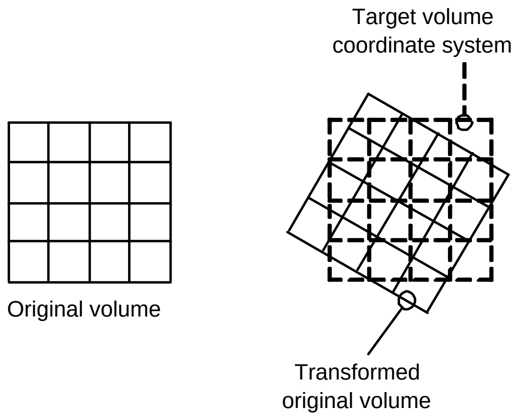


Figure 2: *Relation between transformed, original and target coordinate systems. (Corners and crossings correspond to voxel positions.)*

An illustration of how a transformed original volume might lie within the target volume's coordinate grid is shown in Figure 2. Note that the depicted 2D grids are only a graphical representation in the plane of the actual 3D relationships. They do not directly correspond to any slices in the volume dataset.

In order to make an easy interpolation of voxel intensities possible, the processing direction is inverted. Instead of the original volume, the *target* volume is traversed sequentially. Then, for every target voxel, the *backprojection* into the original coordinate system is computed. Within the original volume dataset, standard interpolation methods can be easily applied.

Target volume $\{w_{ijk}\}$ with voxel positions q_{ijk}:

$$\bar{q}_{ijk} = A^{-1} \cdot q_{ijk}, \quad \begin{smallmatrix} i=1,\ldots,I \\ j=1,\ldots,J \\ k=1,\ldots,K \end{smallmatrix} \tag{7}$$

Equation 7 shows how each voxel position q_{ijk} in the target volume can be backprojected into the original coordinate system. Due to the type of transformations permitted for matrix A, it is always invertible. The optimization of the matrix-vector product discussed in Section 2.1 can also be applied to the backprojection. Here, the columns of A^{-1} are used for incrementally calculating the voxel positions in the original coordinate system.

2.3. Common Interpolation Methods

Given the backprojected position $\bar{q}_{ijk}$ in the original volume, the new value for voxel w_{ijk} has to be computed now. A number of common methods exist for interpolating voxel intensities. A detailed discussion of various interpolation techniques can be found in [Wol90]. In [LCN98] specialized variations for volume rendering are described.

The simplest and cheapest approach is the nearest-neighbour interpolation. For all coordinates of the given threedimensional position, the nearest integer number is determined. Then the voxel value at this integer location is assigned to the target voxel.

$$\begin{aligned} i' &= \lfloor x_{\bar{q}_{ijk}} + 0.5 \rfloor \\ j' &= \lfloor y_{\bar{q}_{ijk}} + 0.5 \rfloor \\ k' &= \lfloor z_{\bar{q}_{ijk}} + 0.5 \rfloor \end{aligned} \tag{8}$$

$$w_{ijk} := v_{i'j'k'}$$

In addition to the calculations shown in Equation 8, in practice a boundary check has to be performed. If the resulting integer position is outside of the original volume, a default background intensity has to be returned instead.

Since the resampling errors caused by the nearest-neighbour algorithm are usually large, often more sophisticated interpolation schemes are used. A widespread method is the so-called trilinear interpolation.

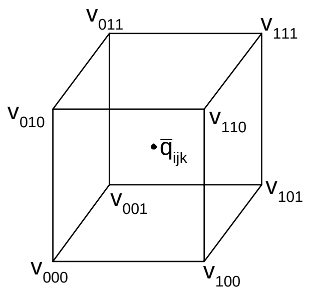


Figure 3: *Voxel cell for trilinear interpolation.*

$$x = x_{\bar{q}_{ijk}} - \lfloor x_{\bar{q}_{ijk}} \rfloor \quad y = y_{\bar{q}_{ijk}} - \lfloor y_{\bar{q}_{ijk}} \rfloor$$
$$z = z_{\bar{q}_{ijk}} - \lfloor z_{\bar{q}_{ijk}} \rfloor$$

$$\begin{aligned} w_{ijk} = \quad & v_{000} \cdot (1-x) \cdot (1-y) \cdot (1-z) + \\ & v_{100} \cdot x \cdot (1-y) \cdot (1-z) + \\ & v_{010} \cdot (1-x) \cdot y \cdot (1-z) + \\ & v_{001} \cdot (1-x) \cdot (1-y) \cdot z + \\ & v_{101} \cdot x \cdot (1-y) \cdot z + \\ & v_{011} \cdot (1-x) \cdot y \cdot z + \\ & v_{110} \cdot x \cdot y \cdot (1-z) + \\ & v_{111} \cdot x \cdot y \cdot z \end{aligned} \tag{9}$$

The voxel cell surrounding a backprojected voxel $\bar{q}_{ijk}$ is shown in Figure 3, with the neighbours denoted as v_{000} to v_{111}. The interpolated value for w_{ijk} is then computed as a weighted sum of the neighbours' intensities. The interpolation result depends on the position of the voxel backprojection within its surrounding voxel cell, denoted (x, y, z) in Equation 9.

Trilinear interpolation requires access to several voxels, as well as at least 7 multiplications and 14 additions, even in its most optimized form. Thus it is significantly more expensive than nearest-neighbour interpolation.

Moreover a number of interpolation schemes exist which take an even greater number of surrounding voxels into account. These methods, like cubic or quadratic interpolation, achieve even smaller resampling errors while requiring still more computation time (see [LCN98] and [Wol90]).

3. THE NEW ALGORITHM

In this paper we propose a novel algorithm for performing rigid transformations of a volume dataset. Unlike the standard approach discussed in Section 2, our method combines backprojection and interpolation into a single step.

The basic idea is to traverse a scanline of a slice in the target volume while *simultaneously* traversing its backprojection in the original volume. In order to efficently generate the backprojection of a line in the volume dataset, we employ the *3D Bresenham* line generation algorithm [KS86].

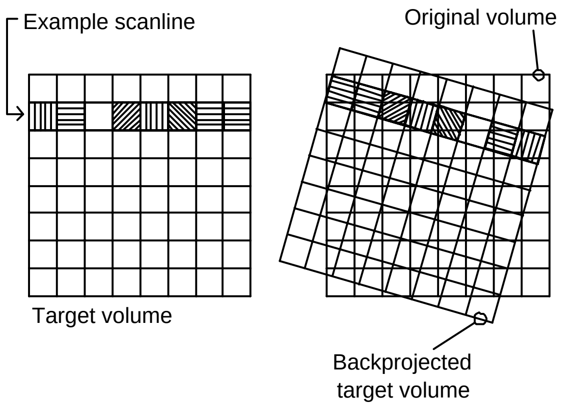


Figure 4: *Example of a scanline in the target volume and its backprojection into the original volume. In this figure, each square symbolises one voxel's intensity. Different filling patterns represent different intensities.*

Using this algorithm, integer voxel positions in the original volume corresponding to voxels in the target volume are determined. Voxel itensities can then be directly copied. This is similar to applying a nearest-neighbour interpolation (see Section 2.3), but eliminates the need for explicit computation of the backprojection and coordinate flooring operations.

Figure 4 illustrates how a target volume scanline corresponds to its backprojected equivalent in the original coordinate system. In the figure, the combination of a rotation around the volume center and a coordinate inversion is assumed as the rigid transformation. This is the same as mirroring the volume at a slightly inclined symmetry plane. Here again it has to be noted, that the grids shown in the diagram do not directly represent any slices of the volume data. Their purpose is to demonstrate the underlying 3D relationships in 2D.

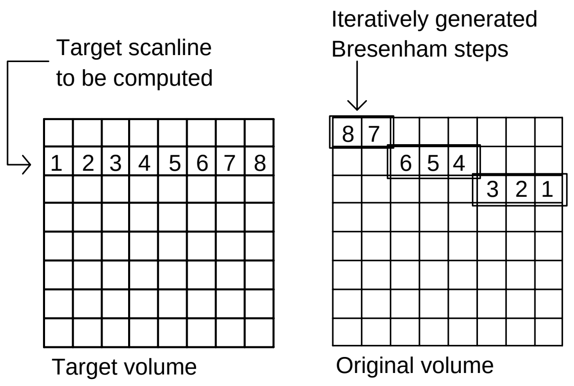


Figure 5: *The target scanline and its rasterized backprojection are processed simultaneously.*

The central idea of our algorithm is depicted in Figure 5. The backprojected line is iteratively generated using the Bresenham algorithm. Meanwhile the corresponding scanline in the target volume is traversed simultaneously. For every line step generated by the Bresenham rasterizer in the original volume, one voxel in the scanline is accessed. The voxel intensity at the integer position generated by the line rasterizer is directly copied to the currently considered target voxel.

By repeating the process for all scanlines in the target volume, the entire volume dataset is transformed according to matrix A. For each scanline, the backprojected positions of its endpoints have to be computed. These backprojected endpoints are then used for initializing the line rasterization algorithm.

3.1. Implementation of the Algorithm

The complete algorithm is shown as pseudocode in Figure 7. It requires an input volume dataset $\{v_{ijk}\}$ and a rigid transformation A. After performing the transformation, the result volume $\{w_{ijk}\}$ is returned. For the sake of simplicity, we suppose that the input and output volume datasets have the same dimensions. It is also assumed that in both volumes the lower left voxel of the front slice is at the coordinate origin (0,0,0). This corresponds to the assumptions made in Section 2.

In practice it might be reasonable to place the output volume's boundaries in a different way. The axis-aligned bounding box of a rotated volume normally has a different size because its corner voxels lie outside of the original boundary. When the volume is translated, it would be useful to also translate the position of the output volume in world coordinates. These changes could be easily incorporated into the algorithm by adding appropriate coordinate offsets during the computations.

The `initializeBresenham()` and `nextBresenhamStep()` procedures are assumed to contain the respective elements of the line rasterization algorithm. An explanation of the Bresenham method can be found in [Fv82] as well as in the original work [Bre65]. An example of an implementation in 3D is available at [Pen92].

It is possible that a backprojected scanline lies partly or entirely outside of the original volume. Thus the validity of each generated voxel position is checked within the innermost loop. The necessary coordinate boundary tests account for a big portion of the method's computational complexity.

3.2. Line Error Correction

The new algorithm described in Section 3.1 and shown in Figure 7 works well for most transformations. However when the volume is rotated by a steep angle, considerable distortion is visible in the result. The reason for this effect is the way the Bresenham algorithm generates discretized line steps.

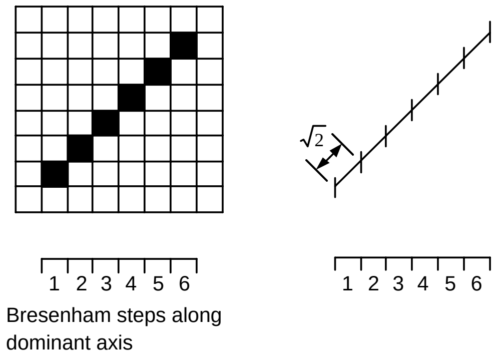


Figure 6: *Steps generated by the line algorithm can correspond to line segments that are longer than 1.0.*

The Bresenham algorithm selects as dominant axis the coordinate axis along which the line endpoints have their largest distance. The line generation method then iterates along this dominant axis. In each step a new integer voxel position with an incremented dom-

```
(* Define constant background voxel intensity *)
const DEFAULT_INT := ...

function transformVolume( $\{v_{ijk}\}$ :Volume$_{(\mathrm{IJK})}$, $A$ :Matrix4x4 )
var
  $\{w_{ijk}\}$   : Volume$_{(\mathrm{IJK})}$ (* Result volume *)
  i,j,k     : Integer    (* Coordinate loop counters *)
  start     : Point4d    (* Double datatype 4d-vectors *)
  end       : Point4d
  i',j',k': Integer    (* Current backprojected point *)


begin

  (* i, j and k iterate through the target volume *)
  for k := 1 to K do
    for j := 1 to J do

      (* Project start and end of scanline back into original volume *)
      start := $A^{-1}$ · ( 0  j-1 k-1 1)$^T$
      end   := $A^{-1}$ · (I-1 j-1 k-1 1)$^T$

      (* Initialize line generation using computed line ends *)
      (i',j',k') := initializeBresenham(start, end)

      for i := 1 to I do
        if (i',j',k') *is within volume boundaries* then
           $w_{ijk}$ := $v_{i'j'k'}$
        else
           $w_{ijk}$ := DEFAULT_INT
        endif

        (i',j',k') := nextBresenhamStep()
      done

    done
  done

  return $\{w_{ijk}\}$
end
```

Figure 7: *Pseudocode formulation of the new algorithm.*

inant axis coordinate is generated. As a consequence, each Bresenham step usually covers a line segment longer than 1.0. This is illustrated in Figure 6 using a twodimensional example. In 3D, depending on the line slope, a single discretization step can advance the position as much as $\sqrt{3}$ along the line.

The result of this discrepancy between line generation steps and actually processed line length are visible distortions in the target volume. In fact it could be said that due to the change of sampling direction when rotating a volume, there is not enough data to fill the corresponding areas in the result volume. We solve this problem by optionally repeating single voxels from the original volume. In order to decide whether this is necessary, we have introduced an additional error variable. This decision variable is called ”line error”. The principle is demonstrated in the following pseudocode fragment, which is in the context of the code in Figure 7.

```
(i',j',k') := initializeBresenham(start, end)

$\delta$ := end - start
$\delta_{dom}$ := max(x$_\delta$,y$_\delta$,z$_\delta$)
$\{\delta_{x_1},\delta_{x_2}\}$ := $\{x_\delta,y_\delta,z_\delta\} \setminus \{\delta_{dom}\}$
lineSlope := $\sqrt{(\delta_{x_1}/\delta_{dom})^2+(\delta_{x_2}/\delta_{dom})^2+1}$
lineError := lineSlope - 1.0

for i := 1 to I do
  $w_{ijk}$ := $v_{i'j'k'}$

  (* Repeat last voxel if lineError > 1 *)
  if lineError > 1.0 then
     i := i + 1
     $w_{ijk}$ := $v_{i'j'k'}$
     lineError := lineError - 1.0
  endif

  (i',j',k') := nextBresenhamStep()
  lineError := lineError + lineSlope
done
```

In this sourcecode, δ is the difference vector between the line endpoints. Its maximum coordinate is denoted as δ_{dom}, whereas the other two vector components are called δ_{x_1} and δ_{x_2}. Using this information, the length of the line segment skipped by each Bresenham step (`lineSlope`) is computed.

Within the inner loop the line error variable is accumulated using the line slope. Whenever the line error is greater than one after a Bresenham step, the current original voxel is copied twice into the target volume instead of only once. Note that we left the necessary volume boundary checks out of this code fragment for the sake of clarity.

After adding line error correction to the new volume transformation algorithm, it produces undistorted results for all kinds of rigid transformations.

4. EXPERIMENTAL RESULTS

We have implemented a software framework for comparing various volume transformation algorithms. In addition to the new algorithm, we have also developed own implementations of several variations of the standard approach. Moreover we included capabilities for comparing our own results with transformation results obtained using the popular volume software package VTK described in [SML98].

Using the benchmarking framework, both the algorithm runtimes and the quality of the produced results can be examined. In order to measure the quality, one volume transformation algorithm is selected as reference. The result of each of the other approaches is then compared on a voxel-to-voxel basis to the reference volume.

Algorithm	**Minimum**	**Average**	**Maximum**
Runtimes [msecs]			
Nearest N.	984	1003	1046
Trilinear	1141	1543	2203
New method	156	178	188
N.N. (vtk)	140	392	578
Cubic (vtk)	125	2117	3079
Relative differences [%]			
Nearest N.	0.0	0.34	0.49
Trilinear	0.0	0.17	0.24
New method	0.0	0.34	0.49
N.N. (vtk)	0.0	0.34	0.49

Table 1: *Benchmark results with input volume* CThead *(256x256x113 voxels). Rotations around x-axis with angles from 0° to 360° and a stepping of 10° (36 transformations).*

Tables 1 and 2 contain the benchmarking results of two test series. In both experiments a number of transformations was generated automatically, each of which was then used as input parameter for all algorithms. We have recorded minimum, average and maximum runtimes and relative variance for all test runs. The input volumes used for the benchmarks have been taken from the Stanford volume data archive [Lev02]. They were converted to 8-bit voxel depth because of limitations in our current implementation.

We compared the following algorithms in the experiments: **Nearest N.** is the standard approach with nearest-neighbour interpolation, but without the optimized matrix-vector product. **Trilinear** is the same with trilinear interpolation. **New method** is our new algorithm, still without a number of possible optimizations. **N.N. (vtk)** is the Visualization Toolkit's standard approach with nearest-neighbour interpolation, which includes many optimizations (see [SML98]). **Cubic (vtk)** is the Toolkit's algorithm with cubic interpolation, which provides a considerably bigger interpolation kernel. This algorithm was used as the reference algorithm for quality comparisons in all experiments.

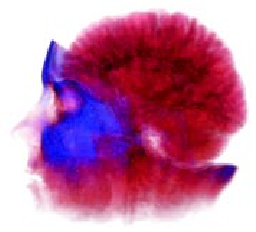
Algorithm	**Minimum**	**Average**	**Maximum**
Runtimes [msecs]			
Nearest N.	12906	15527	19671
Trilinear	7563	19011	28015
New method	1907	5117	9922
N.N. (vtk)	2266	6054	9656
Cubic (vtk)	9734	33277	55547
Relative differences [%]			
Nearest N.	0.41	1.17	1.75
Trilinear	0.16	0.46	0.71
New method	0.41	1.58	3.01
N.N. (vtk)	0.41	1.17	1.75

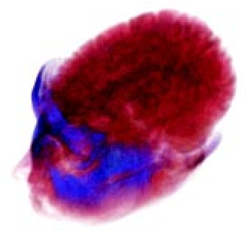
Table 2: *Benchmark results with input volume* bunny-ctscan *(512x512x360 voxels). Rotations around y-axis with angles from 0° to 180° and a stepping of 60°, multiplied with an x-axis rotation of 15° (3 transformations).*

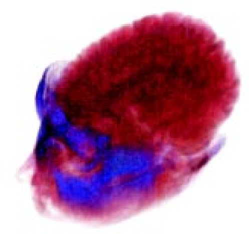
Figure 8 shows example renderings of volume transformation results produced by different algorithms.

5. DISCUSSION AND FUTURE WORK

We have presented a novel approach for applying transformations to volume data. Unlike existing methods, it is aimed at improving runtime performance. Our experiments indicate that the algorithm is considerably faster than the standard approach because it does not have to explicitly transform voxel positions or compute the nearest-neighbour interpolation. Instead integer positions are generated by the modified Bresenham algorithm, which is relatively cheap to compute. This allows for direct copying of voxel intensities.


(a) Unrotated volume


(b) Nearest Neighbour


(c) New method

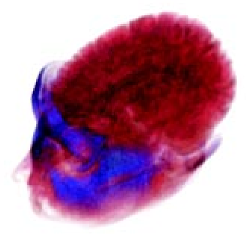

(d) Cubic (vtk)

Figure 8: *Volume transformation results*

Our method produces results comparable in terms of quality to nearest neighbour interpolation in most situations, and only slightly worse for some cases. We have so far limited our considerations to isotropic volume datasets and transformations that do not include scaling. We hope to overcome these limitations in the future by further modifying the line error correction.

Efficient optimization techniques exist that are not yet used in our current implementation of the new approach. At the moment every Bresenham-generated voxel position is tested against the volume boundaries. These tests constitute a big share of the implementation's computational complexity. By determining the intersections of the backprojected scanline with the original volume before discretizing the line, it will be possible to eliminate these boundary checks.

Another important issue when dealing with volume data is the way how memory is accessed. Ideally the data should be processed in a linear, sequential manner. This ensures best utilisation of caching mechanisms existing in the computer hardware. We do already partially exploit cache coherence by traversing the output volume according to its structure in memory. But a sequential access mode is more difficult to achieve when reading from the original volume. Depending on the angle of backprojected lines, consecutive Bresenham steps may read from different scanlines or even slices. This seriously hampers caching efficiency. Thus we intend to investigate ways for improving cache utilisation in our algorithm.

6. ACKNOWLEDGEMENTS

We would like to thank Urs Kanus and Dirk Staneker for interesting and helpful discussions on the topic of volume transformation as well as for reviews of early drafts of this paper.

This work has been supported by the DFG project VIRTUE.

7. REFERENCES

[Bre65] J.E. Bresenham. Algorithms for computer control of a digital plotter. *IBM Systems Journals*, 4(1):25–30, 1965.

[CK00] Baoquan Chen and Arie E. Kaufman. 3d volume rotation using shear transformations. *Graphical Models*, 62(4):308–322, 2000.

[Fv82] J. D. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, 1982.

[KS86] A. Kaufman and E. Shimony. 3d scan-conversion algorithms for voxel-based graphics. In *Proceedings of Workshop on Interactive 3D Graphics*, pages 45–75, October 1986.

[LC99] A. Leu and M. Chen. Modeling and rendering graphics scenes composed of multiple volumetric datasets. *Computer Graphics Forum*, 18(2):159–171, June 1999.

[LCN98] B. Lichtenbelt, R. Crane, and S. Naqvi. *Introduction To Volume Rendering*. Hewlett-Packard Professional Books, 1998.

[Lev88] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

[Lev02] M. Levoy. The Stanford volume data archive. http://www-graphics.stanford.edu/data/voldata/, 2002.

[Pen92] B. Pendleton. line3d - 3D Bresenham's (a 3D line drawing algorithm). ftp://ftp.isc.org/pub/usenet/comp.sources.unix/volume26/line3d, 1992.

[SML98] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit*. Prentice Hall PTR, second edition, 1998.

[TQ97] Tommaso Toffoli and Jason Quick. Three-dimensional rotations by three shears. *Graphical models and image processing: GMIP*, 59(2):89–95, 1997.

[Wol90] G. Wolberg. *Digital Image Warping*. IEEE Computer Society Press, 1990.