An Incremental Algorithm for Fast Rotation of Volumetric Data

Balázs Csébfalvi
Department of Process Control, Technical University of Budapest,
Budapest, Műegyetem rkp. 11, H-1111, HUNGARY
cseb@fsz.bme.hu

Abstract:

In this paper a new volume rendering method is presented which provides real-time rotation on SISD architectures without using any acceleration hardware. The algorithm is based on the efficient handling of empty voxels, where according to the opacity values a 3D binary mask is generated indicating the locations of the non-transparent voxels. Ray casting is performed on this binary mask which is rotated rapidly by shear-warp factorization and an incremental alignment technique. The location of a non-zero bit hit by a ray is mapped onto the corresponding voxel location in the original classified volume, in order to get the opacity and color values in the given sample point. The technique proposed for rotating and sampling the binary mask exploits only the standard ALU operations providing interactive frame rates even on low cost machines.

Keywords: Volume Rendering, Ray Casting, Shear-Warp Factorization.

1 Introduction

Ray casting is a flexible but computationally expensive technique for rendering volumetric data, so several algorithms have been proposed for speeding up the resampling process. Interactive rendering rates can be achieved using large multi-processor systems [SS92][VFR92][COF95][HMK+95], but they are not widely used because of their high costs. Instead of applying acceleration hardware, many researchers have proposed algorithms which exploit the coherence in the data set. These methods are based on spatial data structures built in an appropriate preprocessing of the volume. These data structures, like octrees, pyramids or k-d trees [SF90] [DH92][LH91] can be used for efficient handling of homogeneous regions. In a typical classified volume a significant number of voxels are transparent, so along a ray to be evaluated these empty segments can be skipped.

There are two categories of direct volume rendering methods: the image-order and the object-order algorithms. The image-order algorithms like the classical ray casting compute the image pixel by pixel, where the volume is resampled along rays emitted from each pixel [Lev88][NFML88][DCH88]. The opacity and color values in the evenly located sample points are interpolated from the eight closest voxels using trilinear interpolation. Having these samples evaluated, the pixel colors are computed according to the alpha-blending technique, approximating the light transport equation. Although the evaluation can be terminated after the accumulated opacity has reached a predefined threshold [DH92], this approach is rather expensive computationally. The object-order algorithms process the volume voxel by voxel projecting them onto the image plane [LH91][Wes90]. This approach does not avoid redundant computation either, since the invisible voxels are also processed.

The new method proposed in this paper combines the advantages of image-order and object-order algorithms. In a preprocessing step a 3D binary mask is generated as a result of a segmentation, where the value of a given bit is one if the corresponding voxel in the classified volume is non-transparent, and zero otherwise. In the rendering process rays are cast into this binary mask rather then the original volume and whenever a ray hits a non-zero bit, its location is mapped onto a voxel location in the classified volume, where the color and opacity values are stored. The color of the rays are evaluated as a weighted sum according to the alpha-blending technique. The binary mask can be rotated efficiently using shear-warp factorization, where instead of real rotation the 3D array is sheared parallel to the image plane and an intermediate image is computed by parallel ray casting. The final image is produced from this temporary image by a relatively cheap 2D warp operation. Lacroute and Levoy proposed this technique for fast volume rendering, exploiting that scanlines of the volume data and scanlines of the intermediate image are aligned [LL94]. The algorithm presented in this paper does not exploit this property of shear-warp factorization, but it is based on another advantage of this technique discussed in the next section.

The shear operation is performed with an incremental alignment method proposed by Cohen and Fleishman [COF95]. They used this technique to reduce the communication overhead in a multi-processor architecture supporting shearing of volumes. Applying this incremental algorithm for shearing the binary mask, the new binary array is computed from the previous one using bitwise operators.

After shearing the mask, the non-zero bits representing the non-transparent voxels can be determined efficiently along the parallel rays, using binary search and an appropriate lookup-table technique presented in the third section.

The next section describes how to generate and store the binary mask, and how to rotate it using a shear-warp factorization. The third section introduces the new ray casting technique, the fourth one discusses the extension of the algorithm to arbitrary viewing directions and finally the simulation results are presented.

2 Shear-Warp Factorization

For the sake of clarity, assume that we have a classified volume with size 256^3 with a shaded color and an opacity value in each voxel location and let the viewing direction be the direction of the axis y. According to the opacity values a 3D binary mask array is built storing 256^3 bits, where the value of a bit is zero if the corresponding voxel in the original volume is fully transparent, and one otherwise.

Assume that our architecture can handle 32 bit integers. In this case the 32 bit long segments of the binary mask can be stored in integers, where inside a segment the coordinates x and z are the same. This 3D mask can be rotated efficiently applying shear-warp factorization. For example, to rotate the mask around the axis z between angle $-45°$ and $+45°$ the mask is sheared parallel to the axis x. In this shearing operation some bits are shifted out of the mask array, so to avoid it, the size of the mask is extended with 128 bits along the axis x in both directions (Figure 1). The extended binary $mask$ is an integer array, where the size in directions z, y and x are 256, 8, 512 respectively.

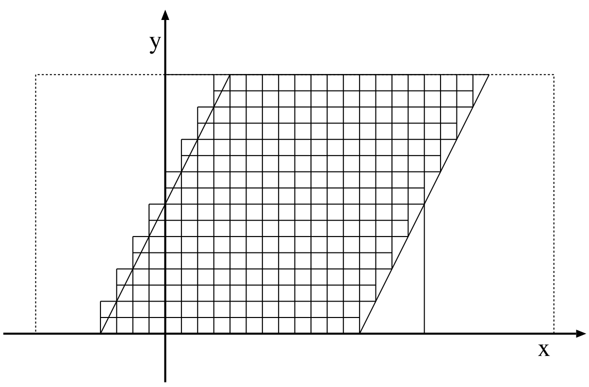

Figure 1: Shearing of the binary mask.

The shearing operation is executed incrementally, where in one step there is no bit shifted with more then one position. The current offset values of different y positions are stored in the $offset$ vector with size 256. Initially the $offset$ vector is the null vector. In each step first the new $offset$ vector is computed, with a symmetric DDA line-drawing algorithm, like Bresenham. Because of the incremental property the maximum distance between the old and the new offset values cannot be greater than one. Let's store the binary distance vector in an integer array with size 8, denoted by $shift$ (Figure 2). One shearing step is executed according to the following algorithm:

```
void Shear() {
  int x, y, z;
  for(z = 0; z < 256; z++) {
    for(y = 0; y < 4; y++) {
      for(x = 0; x < 511; x++)
        mask[z][y][x] =
        mask[z][y][x] & shift[y] |
        mask[z][y][x+1] & ~shift[y];
      mask[z][y][511] &= ~shift[y];
    }
    for(y = 4; y < 8; y++) {
      for(x = 511; x > 0; x--)
        mask[z][y][x] =
        mask[z][y][x] & shift[y] |
        mask[z][y][x-1] & ~shift[y];
      mask[z][y][0] &= ~shift[y];
    }
  }
}
```

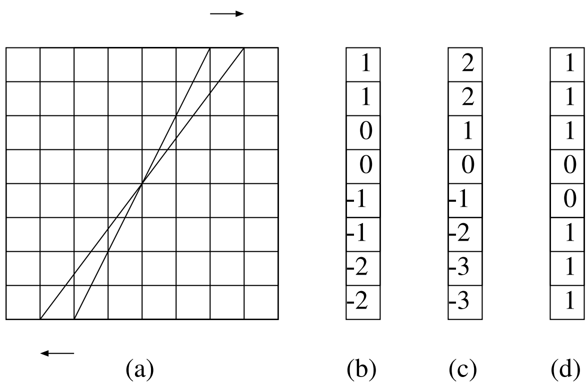

Figure 2: Computation of the new offset values with line drawing (a), b: the old $offset$ vector, c: the new $offset$ vector, d: the binary $shift$ vector.

The upper routine can be optimized by introducing local pointer variables in order to avoid unnecessary array addressing. On the other hand, only that part of the 3D integer array needs to be sheared which contains the non-zero bits representing the non-transparent voxels.

The inner loop in which the running variable is y is broken into two parts, because the directions of shifting are opposites in the upper and lower half of the mask as it is demonstrated in Figure 1.

Shearing the mask along the axis x in the other direction is similar but the two parts of the broken loop have to be exchanged. Note that, shifting can be executed by bitwise logical operators between the neighbor elements of the integer array $mask$ due to the incremental property of the algorithm. The binary shift vector is determined in advance during the computation of the next offset vector:

```
void NewOffset() {
  int x0 = offset[0] - 1;
  int x1 = offset[255] + 1;
  int dx = x1 - x0, dy = 255, x = x0, y;
  int E = -dy;
  int dEp = 2 * (dx - dy), dEn = 2 * dx;
  int bit = 0x80000000, i = 0;
  for(y = 0; y < 256; y++) {
    if(E <= 0) E += dEn;
    else { E+= dEp; x++; }
    if(offset[y] == x) shift[i] &= ~bit;
    else shift[i] |= bit;
    if(!(bit >>= 1)) {
      bit = 0x80000000; i++;
    }
    offset[y] = x;
  }
}
```

The endpoints of the old offset vector are shifted with one position in opposite directions and the new endpoints are connected with the Bresenham line-drawing algorithm[SKe95]. Since the distance in the direction x between the points of the old and the new line is the greatest in the endpoints the distance in the intermediate points must not be greater than one. If there is no difference in the direction x in a given y position then the corresponding bit is set to zero in the binary $shift$ vector and otherwise it is set to one.

3 Ray Casting

Ray casting is performed by sampling the binary mask with parallel rays emitted from the regular grid points of the x, z plane. Considering the rows parallel to the axis y as rays to be traced the first intersection point can be determined by finding the first non-zero bit in the corresponding row. Since the 32 bit long segments of these rows are stored in integers the following binary search algorithm can be used to find the position of the first non-zero bit in the segment:

```
int Trace(int segment) {
  int e, pos = 0, threshold;
  for(e = 16; e > 0; e /= 2) {
    threshold = 2^(pos + e);
    if(segment >= threshold) pos += e;
  }
  return 31 - pos;
}
```

The discrete power 2^{pos+e} is just a symbolic notation, since these constants can be stored in a lookup table. Of course, the optimal solution would be the direct addressing of a lookup table with the segment value, which stores the position of the first non-zero bit for all possible combination. It would require allocating an array with size 2^{32} byte, so instead of this, the first non-zero byte can be found by binary search, and the offset position inside this byte is read from a lookup table. In this case, just a 256 byte array has to be stored. Thus the modified "ray tracing" algorithm is the following:

```
int Trace(int segment) {
  if(segment < 0x00010000) {
    if(segment < 0x00000100)
      return 24 + look_up[segment];
    else
      return 16 + look_up[segment >> 8];
  } else {
    if(segment < 0x01000000)
      return 8 + look_up[segment >> 16];
    else
      return look_up[segment >> 24];
  }
}
```

This $Trace$ routine works for only one non-zero segment, so the segments along a "ray" have to be checked sequentially:

```
int RayTrace(int x, int z) {
  int y = 0, segment;
  for(y = 0; y < 8; y++)
    if(segment = mask[z][y][x])
      return y * 32 + Trace(segment);
  return 255;
}
```

Having the first intersection point for each ray the x, z projection of the volume can be rendered directly by the depth cueing technique. Note that, this image is scaled differently in the direction x and z due to the shearing so it has to be corrected by a warp operation. The greatest advantage of the application of the depth cueing rendering is that it is not necessary to store the original volume in the main memory.

Higher image quality can be achieved by shading the original volume in a preprocessing step. The shaded colors are computed for each voxel according to the view independent Lambert shading model:

$$c_{\text{shaded}}(x, y, z) = c(x, y, z) \cdot \frac{\nabla f(x, y, z)}{|\nabla f(x, y, z)|} \cdot L,$$

where L is the direction of the light source, $c(x, y, z)$ is the own color of the voxel in the x, y, z position and $\nabla f(x, y, z)$ is the approximated gradient vector, which can be considered as the surface normal in the given voxel location:

$$\nabla f(x, y, z) = \frac{1}{2} \cdot \begin{bmatrix} (f(x+1, y, z) - f(x-1, y, z)) \\ (f(x, y+1, z) - f(x, y-1, z)) \\ (f(x, y, z+1) - f(x, y, z-1)) \end{bmatrix}.$$

For each bit of the binary mask the corresponding voxel location in the original volume can be computed according to the offset value in the given y position.

Having a binary opacity function only the color of the first opaque voxel intersected by the ray must be determined. In this case the view dependent Phong model can be applied, since only one voxel per ray has to be shaded so it can be done during the rendering.

If the opacity function is not binary then rays are evaluated according to the alpha-blending technique:

```
rgb AlphaBlending(int x, int z) {
  int y, i, segment;
  rgb color = BLACK;
  voxel v;
  double trans = 1.0;
  for(y = 0; y < 8; y++) {
    segment = mask[z][y][x]);
    while(segment) {
      i = Trace(segment);
      v = volume[z][y][x-offset[y*32+i];
      trans *= 1.0 - v.opacity;
      if(trans  < threshold) return color;
      color += v.color * v.opacity * trans;
      segment &= ~(0x80000000 >> i);
    }
  }
  return color;
}
```

The upper algorithm skips the fully transparent voxels from the evaluation, since the routine $Trace$ jumps into the position of the next opaque voxel. The accumulated transparency is stored in the variable $trans$ and when it falls under a predefined threshold the front to back evaluation is terminated. In the variable $segment$ each bit is erased after the appropriate voxel has been processed, in order to find the next non-zero bit efficiently by calling $Trace$.

The "ray casting" method presented in this section approximates the rays with discrete lines, but it can be extended to exact sample point computation. In this case before shearing the binary mask, the continuous offset vector needs to be determined which stores the real offset values in different y positions. Let us denote this vector by $realoffset$. In order to execute the shearing transformation, the discrete $offset$ vector is computed from the $realoffset$ by making the ceiling of each component, instead of using the Bresenham line drawing algorithm.

Finding a non-transparent voxel in the binary mask the discrete voxel location is mapped onto a real sample point location in the original classified volume taken into account the value of the $realoffset$ vector in the given y position (Figure 3). Since the binary mask is sheared perpendicular to the axis y the coordinates y and z are mapped onto the same values but the coordinate x is mapped onto $x - realoffset[y]$. In this intermediate x position the opacity and color values are computed by linear interpolation. Since, for the neighbor sample points the difference in coordinates y is one, the Euclidean distance between them is not the same in different viewing angles. Note that opacity values are assigned to the samples assuming that the distance between the sample points is one so these values must be scaled with a correction factor depending on the viewing direction.

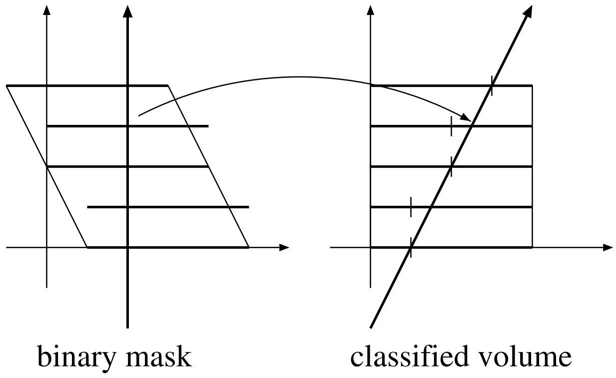

Figure 3: Mapping the binary mask location onto a voxel location in the original classified volume.

4 Extension to Arbitrary Directions

In order to view the volume from an arbitrary direction three 3D binary masks need to be handled at the same time. The appropriate mask is selected according to the main component of the viewing direction. Let us denote the masks supporting shearing perpendicular to the axes x, y and z with $maskx$, $masky$ and $maskz$ respectively. For example, if the main component is the coordinate y then $masky$ is used in the ray casting process. Since the volume can be sheared parallel to the axis x and axis z as well, the binary mask is extended along the axes x and z with 128 bits in both directions (Figure 4).

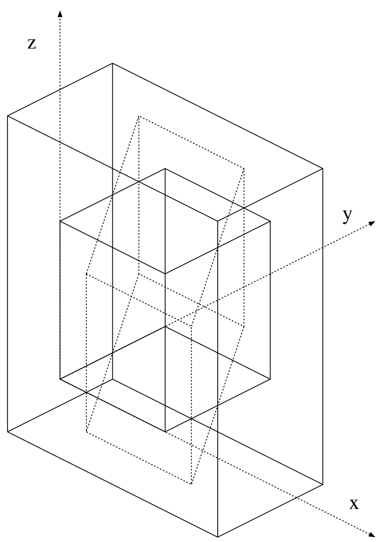

Figure 4: Extension of the binary mask.

When the main component of the viewing direction is changed then another mask is selected. Note that in order to guarantee the continuity the newly selected mask must be in the appropriate shearing phase. Thus the different masks are sheared in a parallel way. For example, when $masky$ is sheared along axis x then $maskz$ is sheared along axis x as well. On the other hand, $maskx$ needs to be sheared also, since when the viewing angle $45°$ (or $-45°$) is reached then $maskx$ must be in the shearing phase corresponding to the given viewing angle (Figure 5).

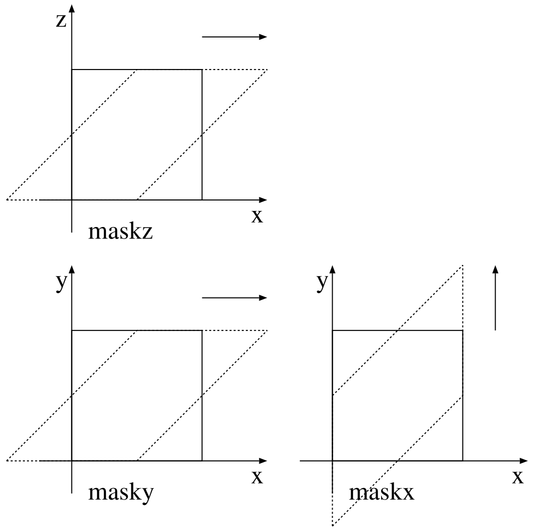

Figure 5: Parallel shearing of the three binary masks.

Assume that the volume is rotated first around the axis z by $45°$ than around the $x = y$ line by $90°$. In this case, the image scanlines and the voxel scanlines are not aligned so not just a 2D warp operation must be executed on the intermediate image but a 2D rotation by $45°$. In this sense, this algorithm is an object-order method since the temporary image is evaluated according to the voxel scanlines rather then the pixel scanlines. These 2D operations are relatively cheap comparing with the 3D shearing. In order to avoid holes in the rotated intermediate image the rotation is performed as three 2D shears demonstrated in Figure 6.

Of course, the upper extension to arbitrary viewing direction reduces the performance because of two more 3D shears and the 2D rotation. In a practical application it could be useful to allow the user to set the viewing direction limitations in order to achieve higher frame rates.

5 Simulation Results

The proposed algorithm was implemented in C++, and it was tested on a CT and an MRI data sets. The average shearing and rendering times, and the frame rates are summarized in Table 1 and Table 2 for the depth cueing and the shaded rendering respectively. The test

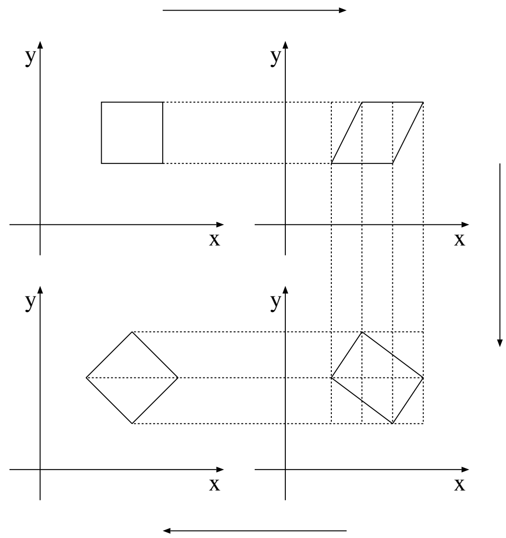

Figure 6: 2D rotation with three shears.

machine was a 166 MHz Pentium PC. These test results show that applying the presented method interactive frame rates can be achieved even on cheap machines, without using any specialized hardware.

file	CT head	MRI brain
volume size	$128 \times 128 \times 113$	$128 \times 128 \times 84$
shearing time	0.02 sec	0.01 sec
rendering time	0.01 sec	0.01 sec
frame rates	33 Hz	50 Hz

Table 1: Running time measurements for the depth cueing rendering.

file	CT head	MRI brain
volume size	$128 \times 128 \times 113$	$128 \times 128 \times 84$
shearing time	0.02 sec	0.01 sec
rendering time	0.04 sec	0.03 sec
frame rates	17 Hz	25 Hz

Table 2: Running time measurements for the shaded rendering.

Figure 7 demonstrates the volume rotation applying the depth cueing rendering technique for the CT head, while Figure 8 and Figure 9 show the same data using bone and soft tissue segmentation respectively and applying the gradient based shading model and finally, in Figure 10 the shaded MRI brain is shown.

6 Conclusion

In this paper a new interactive volume rotation algorithm was presented which is based on shear-warp factorization and an incremental alignment technique. The proposed method does not need large multiprocessor architectures or any acceleration hardware

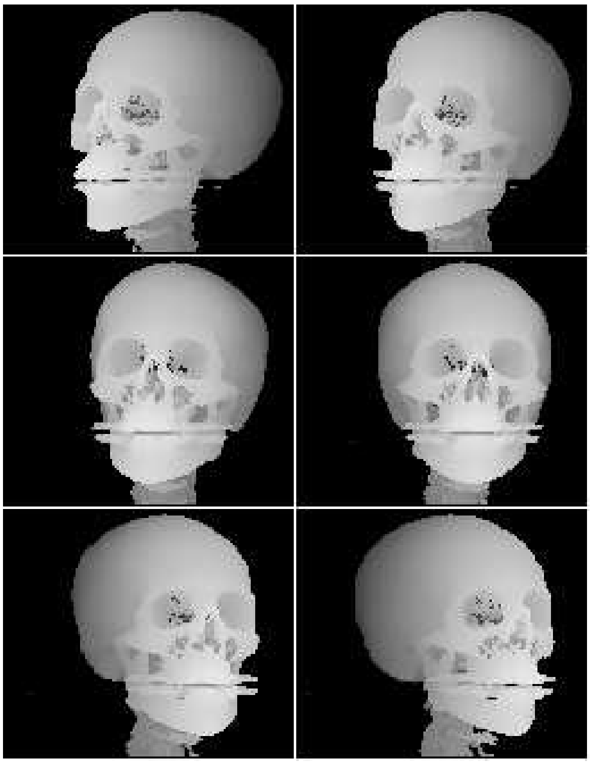

Figure 7: The CT data set rendered with depth cueing.

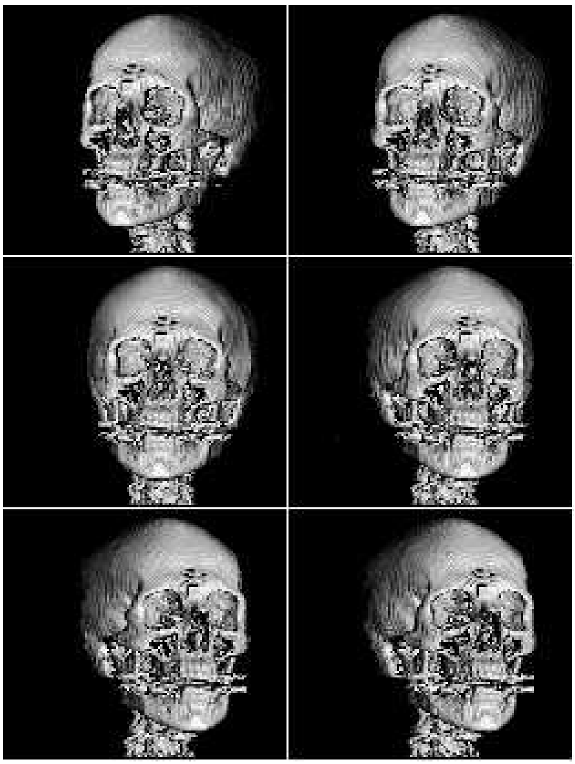

Figure 8: Bone segmentation of the CT data set rendered with gradient based shading.

since it exploits only the standard ALU operations. The locations of non-transparent voxels are stored in a 3D binary mask which is rotated by bitwise operators. Practically, having an architecture handling 32 bit integers, in one step 32 voxels can be processed. Since in the binary mask the bits corresponding to the voxels intersected by a ray are stored in integers, the problem of finding the first non-transparent voxel along a ray can be reduced to the problem of finding the first non-zero bit in a 32 bit integer. This can be done by binary search and an appropriate lookup table technique. After having the first non-zero bit, its location is mapped onto a voxel location in the original classified volume, where the opacity and the color values are stored. Although, this algorithm significantly speeds up the traditional alpha-blending evaluation of rays, it is particularly efficient for binary segmented volumes, where the opacity function is binary or near binary (the non-zero opacities are near one). The performance and the memory requirements can be effected by setting the viewing direction limitations and selecting the rendering algorithm (depth cueing or gradient based shading).

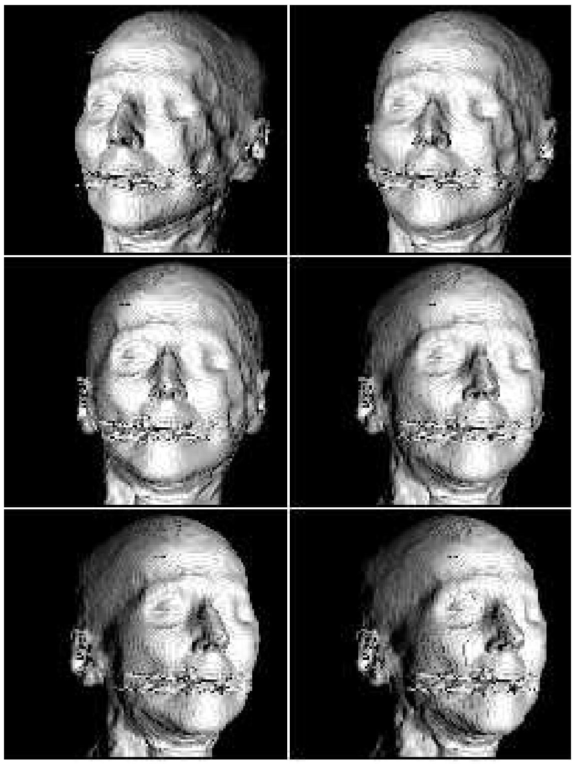

Figure 9: Soft tissue segmentation of the CT data set rendered with gradient based shading.

7 Acknowledgements

This work has been supported by the National Scientific Research Fund (OTKA ref.No.: F 015884) and the Austrian-Hungarian Action Fund (ref.No.: 29ö4

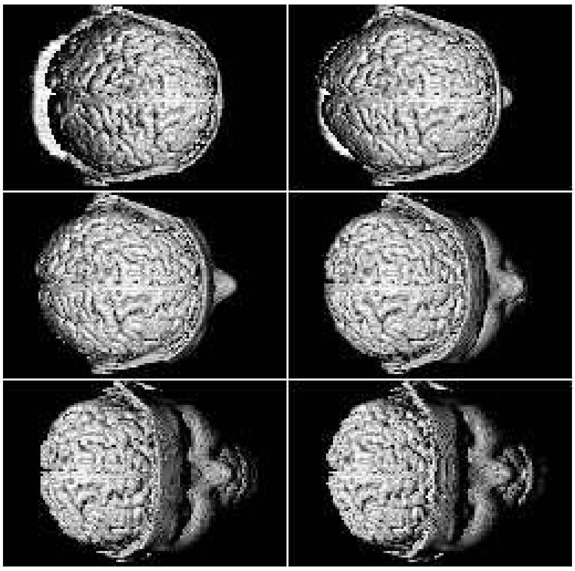

Figure 10: The shaded rendering of the MRI data set.

and 32öu9). The CT and MRI scans are from the Chapel Hill Volume Rendering Test Dataset (available from omicron.cs.unc.edu [128.109.136.159] in pub/softlab/CHVRTD). The CT data was taken on the General Electric CT scanner and provided courtesy of North Carolina Memorial Hospital. The MRI data was taken on the Siemens Magnetom and provided courtesy of Siemens Medical Systems, Inc., Iselin, NJ.

References

[COF95] Daniel Cohen-Or and Shachar Fleishman. An incremental alignment algorithm for paralell volume rendering. *EUROGRAPHICS'95*, pages 123–133, 1995.

[DCH88] R.A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics*, 22:65–74, 1988.

[DH92] John Denskin and Pat Hanrahan. Fast algorithms for volume ray tracing. *Workshop on Volume Visualization*, pages 91–98, 1992.

[Gro95] E. Groeller. *Scientific Visualisation*. Technical University of Vienna, Institute of Computer Graphics, Vienna, 1995. in German.

[HMK+95] Jürgen Hesser, Reinhard Männer, Günter Knittel, Wolfgang Strasser, Hanspeter Pfister, and Arie Kaufman. Three architectures for volume rendering. *EUROGRAPHICS'95*, pages 111–122, 1995.

[Lev88] Marc Levoy. Display of surfaces from ct data. *IEEE Computer Graphics and Application*, 8:29–37, 1988.

[Lev90] Marc Levoy. Efficient ray tracing of volume data. *ATG*, 9(3):245–261, 1990.

[LH91] David Laur and Pat Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *SIGGRAPH'91*, pages 285–288, 1991.

[LL94] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *SIGGRAPH'94*, pages 451–457, 1994.

[NFML88] Derek R. Ney, Elliot K. Fishman, Donna Magid, and Marc Levoy. Computed tomography data: Principles and techniques. *IEEE Computer Graphics and Application*, 8, 1988.

[SF90] K.R. Subramanian and Donald S. Fussell. Applying space subdivision techniques to volume rendering. *Visualization'90*, pages 150–159, 1990.

[SKe95] L. Szirmay-Kalos (editor). *Theory of Three Dimensional Computer Graphics*. Akadémia Kiadó, Budapest, 1995.

[SKM94] L. Szirmay-Kalos and G. Márton. On hardware implementation of scan-conversion algorithms. In *8th Symp. on Microcomputer Appl.*, Budapest, Hungary, 1994.

[SS92] Peter Schröder and Gordon Stoll. Data paralell volume rendering as line drawing. *Workshop on Volume Visualization*, pages 25–32, 1992.

[VFR92] Guy Vézina, Peter A. Fletcher, and Philip K. Robertson. Volume rendering on the maspar mp-1. *Workshop on Volume Visualization*, pages 3–8, 1992.

[Wes90] Lee Westover. Footprint evaluation for volume rendering. *SIGGRAPH'90*, pages 367–376, 1990.