A Vertex Program for Efficient Box-Plane Intersection

Christof Rezk Salama and Andreas Kolb

Computer Graphics and Multimedia Systems Group
University of Siegen, Germany
Email: {rezk,kolb}@fb12.uni-siegen.de

Abstract

Object-order texture-based volume rendering decomposes the volume data set into stacks of textured polygons. The performance of such hardware-based volume rendering techniques is clearly dominated by the fill-rate and the memory bandwidth, while only very little workload is assigned to the vertex processor. In this paper we discuss a vertex program which efficiently computes the slicing for texture-based volume rendering. This novel technique enables us to balance the workload between vertex processor, fragment processor and memory bus. As a result we demonstrate that the performance of texture-based volume rendering can be efficiently enhanced by trading an increased vertex load for a reduced fragment count. As an application we suggest a novel approach for empty space skipping for object-order volume rendering.

1 Introduction

In computer graphics, algorithms for the synthesis of virtual images can be categorized into object-order or image-order approaches. Image-order approaches, such as raytracing, decompose the *target* image into elements (usually pixels[1]) and traverse the virtual scene to compute its contribution to the final color of each element. Object-order approaches, such as rasterization, on the other hand split the virtual scene (the *source*) into geometric primitives (points, lines, triangles) and then compute how these primitives contribute to the final image, e.g. by "splatting" the primitive onto multiple pixels in screen space at once.

Direct volume rendering can be performed with both approaches, efficiently accelerated by modern graphics processing units (GPUs). The traditional texture-based volume rendering algorithm is an object-order approach. It decomposes the volume data into one or more stacks of polygonal slices which are usually composited in back-to-front order onto the image plane. Modern graphics hardware also allows direct volume rendering to be implemented as an image-order approach, basically by performing ray-casting within a large loop that samples the volume successively along the viewing ray in the fragment shader.

If we compare both implementations, we find that slice-based volume rendering might be considered the "brute-force"- approach, that relies solely on the fill-rate and the high fragment throughput of the rasterization unit. Ray-casting on the other hand employs optimization techniques such as empty-space skipping and early-ray termination. At the bottom line, however, the brute-force approach is still advantageous in terms of performance, while GPU-based ray-casting has several clear advantages when rendering iso-surfaces or sparse volumes.

The performance limit for both approaches, however, is the same. It is either the fill-rate or the memory bandwidth. Neither of these approaches is geometry-limited. The major work is done in the fragment processor and only a negligible computational load is assigned to the vertex processor. In this paper, we demonstrate that the performance of object-order volume rendering can be improved by moving the necessary slice decomposition, which is usually done on the CPU, onto the vertex processor. This gives us the flexibility which is necessary to load-balance the rendering process for maximum performance. We demonstrate that this is effective for optimizing bandwidth and texture cache coherency. Finally, we introduce an implementation of empty-space skipping for slice-based volume rendering using an octree-decomposition of the volume data.

In the following section, we will point the reader

[1] elements smaller than pixels may be used to account for sub-pixel accuracy and anti-aliasing

to important related work in the field of real-time volume graphics. Section 3 describes the original 3D-texture based approach to volume rendering and examines its limitations. In Section 4 our algorithm for plane-box intersection is explained. Section 5 describes the implementation of this algorithm in a vertex program. This implementation is the basis of our straight-forward object-order algorithm for empty-space skipping using an octree decomposition. This is described in Section 6. The results of our implementation are analyzed in Section 7. Eventually, Section 8 concludes our work.

2 Related Work

Real-time volume graphics has a long history in computer graphics, starting with the first straightforward CPU-based image-order approaches in 1984 [**?**]. Maybe the most important purely CPU-based method was the shear-warp-algorithm introduced by Lacroute and Levoy in 1994 [**?**]. This approach is also the basis for efficient GPU-based implementations, such as 2D-multitexture based volume rendering [**?**]. With the increasing availability of hardware accelerated 3D textures, many volume rendering techniques have utilized this feature for different purposes. For an overview of state-of-the-art object- and image-order volume rendering techniques, we suggest reading the SIGGRAPH 2004 course notes on *Real-Time Volume Graphics* [**?**], which include many implementation details that are usually omitted from scientific papers. The idea of slicing simplices has been examined in different contexts before by Reck et al. [**?**] for tetrahedra and by Lensch et al. [**?**] for prisms.

GPU-based image-order techniques have been developed only recently with the introduction of graphics boards that support true conditional branches and loops of variable length in the fragment processor. Raycasting has been implemented both in a combined CPU/GPU solution [**?**] and a purely GPU-based solution [**?**]. Another important GPU-based raycasting algorithm for isosurface rendering has been introduced recently by Hadwiger et al [**?**]. It includes many advanced concepts such as empty space skipping, and non-photorealistic drawstyles in a unified framework.

Although, the raycasting technique seems to be a more straight-forward way of implementing volume graphics on a GPU, such approaches still suffer from a of series of problems, most of them arising from the fact, that graphics hardware has not been designed for image-order approaches in the first place. However, it is very likely that these limitations will be overcome in the near future.

Nevertheless, also object-order approaches still suffer from several limitations in practise. If 3D textures are used, one popular problem is the limited amount of graphics memory which often forces the programmer to split the volume data set into several smaller chunks (usually called *bricks*), each of which fitting entirely into local video memory. The 3D texture based approach and its current limitations will be explained in detail in the following section.

3 3D Texture-Based Volume Graphics

Current graphics hardware does not support truly three-dimensional rendering primitives. If you want to leverage such hardware for direct volume rendering, your volumetric object must be decomposed into polygonal primitives. However, most graphics boards do have support for three-dimensional texture objects, which allows you to "cut" the texture image out of a 3D texture block using trilinear filtering.

With 3D textures, the volume is usually split into viewport aligned slices. These slices are computed by intersecting the bounding box of the volume with a stack of planes parallel to the current viewport. In consequence, viewport-aligned slices must be recomputed whenever the camera position changes. Figure 1 illustrates the rendering procedure. During rasterization, the transformed polygons are tex-

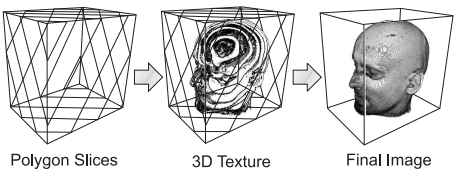

Figure 1: Polygons are computed by intersection the bounding box with a stack of planes parallel to the viewport. The textured polygons are blended in back-to-front order into screen space to create the final rendering.

tured with the image data obtained from a solid texture block by trilinear interpolation. During fragment processing the resulting polygon fragments are blended semi-transparently into the frame buffer and finally displayed on screen.

The intersection calculation for generating the slice polygons is done on the CPU and the resulting polygons are transferred to the graphics processor for each frame. In this case, the vertex processor is only used to transform each incoming vertex with one affine matrix, which comprises the modeling-, viewing-, and perspective transformation. If the volume data set is small enough to fit into local video memory, the overall rendering process is clearly fill-rate limited. This means that the major workload is performed by the fragment processor which splits each polygon into fragments and obtains the trilinearly filtered texel values from local video memory. If the arithmetic computation in the fragment program is not too complex, the fill-rate is mainly determined by texture cache coherency and the memory bandwidth between the fragment processor and local video memory.

3.1 Bricking

Difficulties arise when the volume data set does not fit entirely into local video memory. The texture block must be divided into smaller *bricks*, each of which is rendered separately. In this case the rendering process is limited by the memory bandwidth between the GPU and the host memory, while the GPU is stalled until the required texture data is fetched from host memory. To make matters worse, bricking increases the overall memory for storing the volume data set. This is because correct interpolation across brick boundaries requires one plane of voxels to be duplicated at the boundary between any two bricks [**?**].

The size of the bricks has significant influence on the overall performance. In order to optimize cache coherency, the bricks should be kept small enough to fit into the texture cache. On the other side, however, the bricks should not be too small, otherwise the duplicated voxels at the brick boundaries would significantly increase the memory required for storing the volume. To make matters worse, a large number of bricks result in a higher number of intersection calculations for the CPU, a higher number of vertices which must be transferred to the GPU for each frame and thus a deterioration of the bandwidth problem.

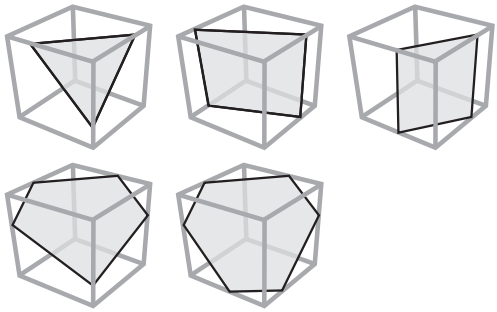

Figure 2: Intersecting a box with a plane. The resulting polygon has between 3 and 6 vertices. Symmetric cases are omitted.

To these ends we present a vertex program which shifts all the intersection calculation between the bounding box of the volume and a stack of slice planes in the vertex processor. We will see that such a vertex program minimizes the amount of data that must be transferred from host memory to the GPU. This allows us to render significantly smaller bricks and thus increases overall performance.

4 Cube-Slice Intersection

The intersection between a box and a plane results in a polygon with 3 to 6 vertices (assuming that the plane actually intersects the box). The different cases are illustrated in Figure 2. Our approach is to compute such intersection polygons directly in the vertex processor. A vertex program, however, can only modify existing vertices. It can neither insert new vertices into the stream nor remove vertices from the stream. As a consequence, we design a vertex program that always receives 6 vertices and outputs 6 vertices. If the intersection polygon consists of less than 6 vertices, the vertex program will generate one or more duplicate vertices (i.e two identical vertices with an edge of length zero inbetween).

Intersecting an edge of the box with the slice plane is easy, if the plane is given in Hessian normal form,

$$\langle \vec{n}_P \circ \vec{x} \rangle = d \tag{1}$$

with $\vec{n}_P$ denoting the normal vector of the plane and d the distance to the origin. For viewport-aligned

slicing the normal vector $\vec{n}_P$ is simply the viewing direction. An edge between two vertices V_i and V_j of the bounding box can be described as

$$\begin{aligned} E_{i\to j} : X(\lambda) &= V_i + \lambda\,(V_j - V_i) \\ &= V_i + \lambda\,\vec{e}_{i\to j} \quad \text{with } \lambda \in [0,1] \end{aligned} \tag{2}$$

Note that the vector $\vec{e}_{i\to j}$ does not have unit length in general. The intersection between the plane and the straight line spanned by $E_{i\to j}$ is simply calculated by

$$\lambda = \frac{d - \langle \vec{n}_P \circ V_i \rangle}{\langle \vec{n}_P \circ \vec{e}_{i\to j} \rangle}\,. \tag{3}$$

The denominator becomes zero only if the edge is coplanar with the plane. In this case, we simply ignore the intersection. We have found a valid intersection only if λ is in the range $[0,1]$, otherwise the plane does not intersect the edge.

The main difficulty in performing the intersection calculation in the vertex processor is to maintain a valid ordering of the intersection points, so that the result forms a valid polygon. To understand the slicing algorithm, let us assume for now, that we have one vertex V_0 that is closer to the camera than all other vertices, as displayed in Figure 3 (left) Vertex V_7 is then identified as the vertex lying on the opposite corner across the cube's diagonal. In the following we will refer to the vertex indices given in Figure 3 (left).

If V_0 is the front vertex and V_7 is the back vertex, there are exactly three independent paths from V_0 to V_7 as marked in Figure 3 (left) by the solid lines in different shades of gray. In this context, *independent* means that these paths do not share

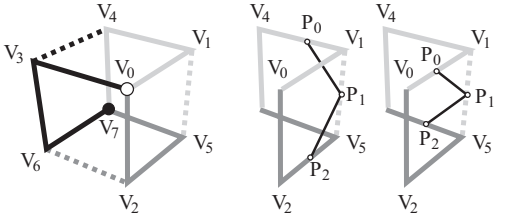

Figure 3: Left: The vertices are numbered sequentially. There always exist three independent paths from the front vertex V_0 to the back vertex V_7 as marked by the solid lines. Right: The intersection point of the dotted line must be inserted between the intersection points from the solid lines.

any vertices other than the start and the end vertex. Each path consists of a sequence of three edges $\{E_1, E_2, E_3\}$, e.g. $E_1 = E_{0\to1}$, $E_2 = E_{1\to4}$ and $E_3 = E_{4\to7}$ for the light gray path. For a given front vertex, we can construct these three paths uniquely by forcing that the vectors corresponding to E_1, E_2 and E_3 for each path form a right handed system.

Now imagine we are sweeping a viewport-parallel plane from front to back through the box in Figure 3 (left). The first vertex that the plane touches is V_0. Before this happens, we do not have any valid intersection with the box. The last vertex that the plane touches, if we proceed from front to back, is vertex V_7. After that, we will not have any valid intersection anymore. As a consequence, any viewport-aligned plane that intersects the box will have exactly one unique intersection point along each of the three paths, respectively. In the case that our intersection polygon has only three vertices, they will be exactly those intersection points with the three paths. As a result, we can compute three of the possible six intersection points P_i by checking intersections with a sequences of edges, respectively.

$P_0 =$ Intersection with $E_{0\to1}$ or $E_{1\to4}$ or $E_{4\to7}$

$P_2 =$ Intersection with $E_{0\to2}$ or $E_{2\to5}$ or $E_{5\to7}$

$P_4 =$ Intersection with $E_{0\to3}$ or $E_{3\to6}$ or $E_{6\to7}$

Now, let us consider where the remaining intersection points must lie if our polygon has more than three vertices. We will first examine the light gray dotted edge $E_{1\to5}$ in Figure 3. If there exists a valid intersection with this edge, then it must be inserted between the intersection points with the light gray path and the medium gray path as can be easily seen in Figure 3 (right). If an intersection with the dotted edge does not exist, we simply set the point equal to P_0, which is the intersection point with the light gray path. The other dotted edges can be treated analogously, resulting in the remaining three intersection points:

$P_1 =$ Intersection with $E_{1\to5}$, otherwise P_0

$P_3 =$ Intersection with $E_{2\to6}$, otherwise P_2

$P_5 =$ Intersection with $E_{3\to4}$, otherwise P_4

We have now determined all six intersection points of the plane with the box in a sequence that forms a valid polygon. Now it is easy to check, that

```
01  void main(
02    int2             Vin          : POSITION,
03
04    // updated per cube
05    uniform float3   vecTranslate,
06    uniform float    dPlaneStart,
07
08    // updated per frame
09    uniform float4x4 matModelViewProj,
10    uniform float3   vecView,
11    uniform int      frontIndex,
12
13    // const:  never updated
14    uniform float    dPlaneIncr,
15    uniform int      nSequence[64],
16    uniform float3   vecVertices[8],
17    uniform int      v1[24],
18    uniform int      v2[24],
19
20    // output variables
21    out float4       VertexOut    :  POSITION,
22    out half3        TexCoordOut  :  TEXCOORD0
23  )
24  {
25
26    float dPlaneDist = dPlaneStart + Vin.y * dPlaneIncr;
27
28    float3 Position;
29
30    for(int e = 0; e < 4; ++e) {
31
32      int vidx1 = nSequence[int(frontIndex * 8 + v1[Vin.x*4+e])];
33      int vidx2 = nSequence[int(frontIndex * 8 + v2[Vin.x*4+e])];
34
35      float3 vecV1 = vecVertices[vidx1];
36      float3 vecV2 = vecVertices[vidx2];
37
38      float3 vecStart = vecV1+vecTranslate;
39      float3 vecDir = vecV2-vecV1;
40
41      float denom = dot(vecDir,vecView);
42      float lambda =
43          (denom!=0.0)? (dPlaneDist-dot(vecStart,vecView))/denom:-1.0;
44
45      if((lambda >= 0.0) && (lambda <= 1.0)) {
46        Position = vecStart + lambda * vecDir;
47        break;
48      } // if(...
49
50    } // for(...
51
52    VertexOut = mul(matModelViewProj, float4(Position,1.0));
53    TexCoordOut = 0.5 * (Position) + 0.5.xxx;
54    return;
55  }
```

Listing 1: Cg vertex program for box-plane intersection.

the same sequence works fine if the front edge or the front face of the box is coplanar with the viewing plane. We simply select one of the front vertices as V_0 and set V_7 to the opposite corner. Remember that we ignore any intersections with an edge that is coplanar with the plane.

5 Implementation

The algorithm for computing the correct sequence of intersection points as described in the previous section has been implemented as a vertex program. The Cg code for the program is given in Listing 1. The program has been designed for slicing a high number of equally-sized and equally-oriented boxes with a stack of equidistant planes. Care has been taken to minimize the number of state changes and the amount of data transferred to the graphics board for each frame.

The input stream of vertices to calculate one intersection polygon is specified in Listing 2. The x-coordinate of the vertex is an index that specifies which of the six possible intersection points should be computed. The y-coordinate of the vertex is the index of the plane that is used for intersection. As the plane index is constant for one polygon, it could alternatively be specified as a separate parameter in the vertex stream (e.g. as texture coordinate). However, current hardware implementations do not support vertices that have only one coordinate, so I decided to incorporate the plane index into the y-coordinate of the vertex.

In this implementation we assume that all the boxes have the same size and orientation, although simple modifications to the program will allow arbitrary size and orientation at the cost of a slightly larger number of state changes. In our case each box consists of the same set of vertices and a translation vector `vecTranslate` (line 05 in Listing 1) which is specified once for each box to be rendered. The vertices of one box are kept in a constant uniform vector array `vecVertices[8]` (line 16) and will not be changed at all.

```
glBegin(GL_POLYGON);
  glVertex2i(0,nPlaneIndex);
  glVertex2i(1,nPlaneIndex);
  glVertex2i(2,nPlaneIndex);
  glVertex2i(3,nPlaneIndex);
  glVertex2i(4,nPlaneIndex);
  glVertex2i(5,nPlaneIndex);
glEnd();
```

Listing 2: OpenGL example vertex stream for calculating one intersection polygon simply stores the index of the intersection point to be calculated and the index of the plane.

Besides the usual modelview-projection-matrix (line 08), we specify for each frame the index of the front vertex with respect to the viewing direction in the uniform parameter `frontIndex` (line 10). Since all our boxes are equally oriented, the front index will not change during one frame. Additionally, we set the uniform parameters `vecView` (line 09) to the normal vector $\vec{n}_P$ of the plane and `dPlaneIncr` (line 14) to the distance between two adjacent planes. In line 26, the correct distance d for the plane equation is computed.

The constant uniform index array `nSequence` (line 15) stores the permutation of vertex indices with respect to the given index of the front vertex `frontIndex` (see lines 32 and 33). As described in the previous section, several edges must be checked for intersection in sequence, according to the index of the intersection point.

In order to calculate the intersection points P_1, P_3 and P_5, we must first check for an intersection with the *dotted* edge, and if this intersection does not exist we must check for intersection with the corresponding path (solid line, Figure 3). Hence, the maximum number of edges that must be tested for intersection is four. This is done within the for-loop that starts in line 30. For the intersection points P_0, P_2 or P_4, we have to check only three edges. In this case the program breaks out of the for-loop when the intersection point is found after a maximum of three iterations.

The two constant index arrays `v1` and `v2` store the indices of start and end vertices of the edges that must be tested successively for intersection. They are indexed by the intersection index `Vin.x` from the vertex stream in combination with the current iteration count `e` (line 32 and 33).

Line 32 and 33 compute the correct vertex indices of the edge that must be tested for intersection. The vertices are fetched from the constant uniform array `vecVertices` in lines 35 and 36. Lines 38 and 39 compute the correct start point and the edge vector for the current edge, taking into account the local translation of the box. Line 41 calculates

the denominator for Equation 2. If the denominator is unequal zero (which means that the edge is not coplanar with the plane), the λ value for the edge is computed as in Equation 2. Line 45 tests if we have a valid intersection. If this is true, the program breaks out of the for loop. Finally, line 52 transforms the resulting intersection point into screen space and line 53 calculates the texture coordinate for the vertex. The texture coordinates in this example are obtained by simply scaling the vertex position to the range $[0, 1]$. Alternatively texture coordinates could be specified by another uniform parameter similar to `vecVertices`.

This admittedly intricate implementation allows one box to be intersected with several parallel planes using one single function call that feeds a predefined vertex buffer into the graphics pipeline.

6 Empty-Space-Skipping

As an application of the described vertex program, we have implemented an empty space skipping approach based on an octree decomposition of the volume data, similar to the one proposed in [**?**] for isosurface extraction. At each level, the volume block is subdivided into eight equally sized sub-blocks. Each node of the octree stores the minimum and the maximum scalar value contained in its sub-block. Typical transfer functions for volume visualization map a significant portion of the scalar data range to complete transparency. The octree structure allows us to completely skip sub-blocks that do not contribute to the final image, because all of its voxel values are being transparent. This is the well-known idea of empty-space-skipping borrowed from raycasting approaches.

During rendering the octree is traversed topdown by the CPU. Depth sorting of the sub-blocks is necessary to ensure correct back-to-front ordering. This is efficiently achieved during the octree traversal. Each octree nodes traverses its eight children in the correct sequence with respect to the current viewing direction. Before a subblock is traversed, our implementation uses the precomputed minimum and maximum scalar values to check whether the sub-block will contribute to the final image or not. Additionally, we can decide at each node of the octree, whether we store the 3D texture data for this level or store one separate 3D-textures for each sub-level, to optimize the size of 3D textures for texture cache coherency.

7 Results

The presented approach for empty space skipping has been implemented on an NVidia Geforce6 board with 256 MB of local video memory. The dataset used for performance measurement was a 16 bit CTA[2] data set of size $512 \times 512 \times 96$ showing blood vessels in the human brain. In medical practise, 3D angiography data sets are usually visualized by setting the soft tissue to completely transparent. Angiography data is thus ideal for evaluating empty space skipping techniques. Example images are given in Figure 6.

The vertex program described in the paper was supplemented by two different fragment programs:

1. a simple shader for direct volume rendering using dependent texture lookups for postclassification (2 texture lookups per fragment)
2. a fragment shader for direct volume rendering with postclassification and Phong shading. This fragment shader estimates the gradient vector on-the-fly using central differences and multiple texture lookups per fragment (8 texture lookups per fragment).

We have tested different levels of subdivision and measured the overall performance. In order to obtain optimal image quality, a high number of slice images has been used to render the volume. Figure 4 shows the performance using the first fragment shader (direct rendering without shading) for different levels of subdivision. We have tested each configuration with two different transfer functions, one which sets all voxels below one third of the

[2]CTA = computed tomography angiography

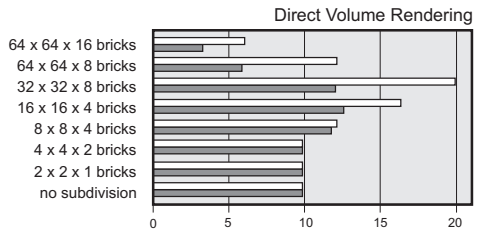

Figure 4: Performance in frames per second for direct volume rendering with postclassification via dependent texture lookup. The white and gray bars show results for different transfer functions.

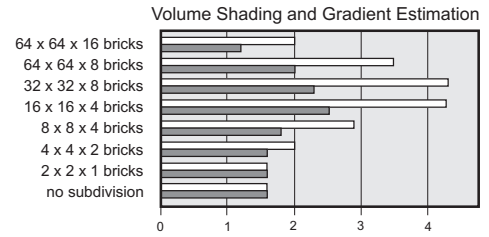

Figure 5: Performance in frames per second for direct volume rendering with on-the-fly gradient estimation and Phong shading. The white and gray bars show results for different transfer functions.

scalar range to completely transparent (the gray bar) and another which sets all voxels below one half of the scalar range to zero (the white bar). The second transfer function allows a higher number of blocks to be skipped. Such transfer functions are typically chosen for visualizing angiography data.

As displayed in Figure 4, our empty-space-skipping technique does not improve the performance for low subdivision levels. Due to the large size of the blocks, it is not likely that one block can be completely skipped during traversal. If we increase the depth of the octree, we reach optimum performance by subdividing into $32 \times 32 \times 8$ bricks. For higher subdivision levels, the rendering process is dominated by the increasing load on the vertex processor, which does not outweigh the reduced number of fragments to be processed. In the worst case, where we have a fully opaque volume, empty space skipping is not applicable at all and the increased vertex load of our algorithm could significantly degrade the performance. However, since we are using an octree decomposition which is independent of the texture storage, we can stop the octree traversal at any time and generate the slice images for the current level.

The second fragment shader significantly increases the load on the fragment processor by computing gradient vectors on the fly using central differences and 8 texture samples per fragment (one for the scalar value, six for the central differences and one for the transfer function). In this case the performance gain of our empty space skipping approach is very high as displayed in Figure 5. We have measured a performance gain of about 150% for the $16 \times 16 \times 8$ and the $32 \times 32 \times 8$ configuration compared to not applying empty space skipping.

8 Conclusion

We have presented a vertex program which efficiently calculates intersections between a box and a stack of equidistant parallel planes. The approach has been used to implement empty-space-skipping for object-order texture-based volume rendering. We have demonstrated that applying our empty-space-skipping techniques significantly increases the performance by balancing the workload between the vertex and the fragment processor.

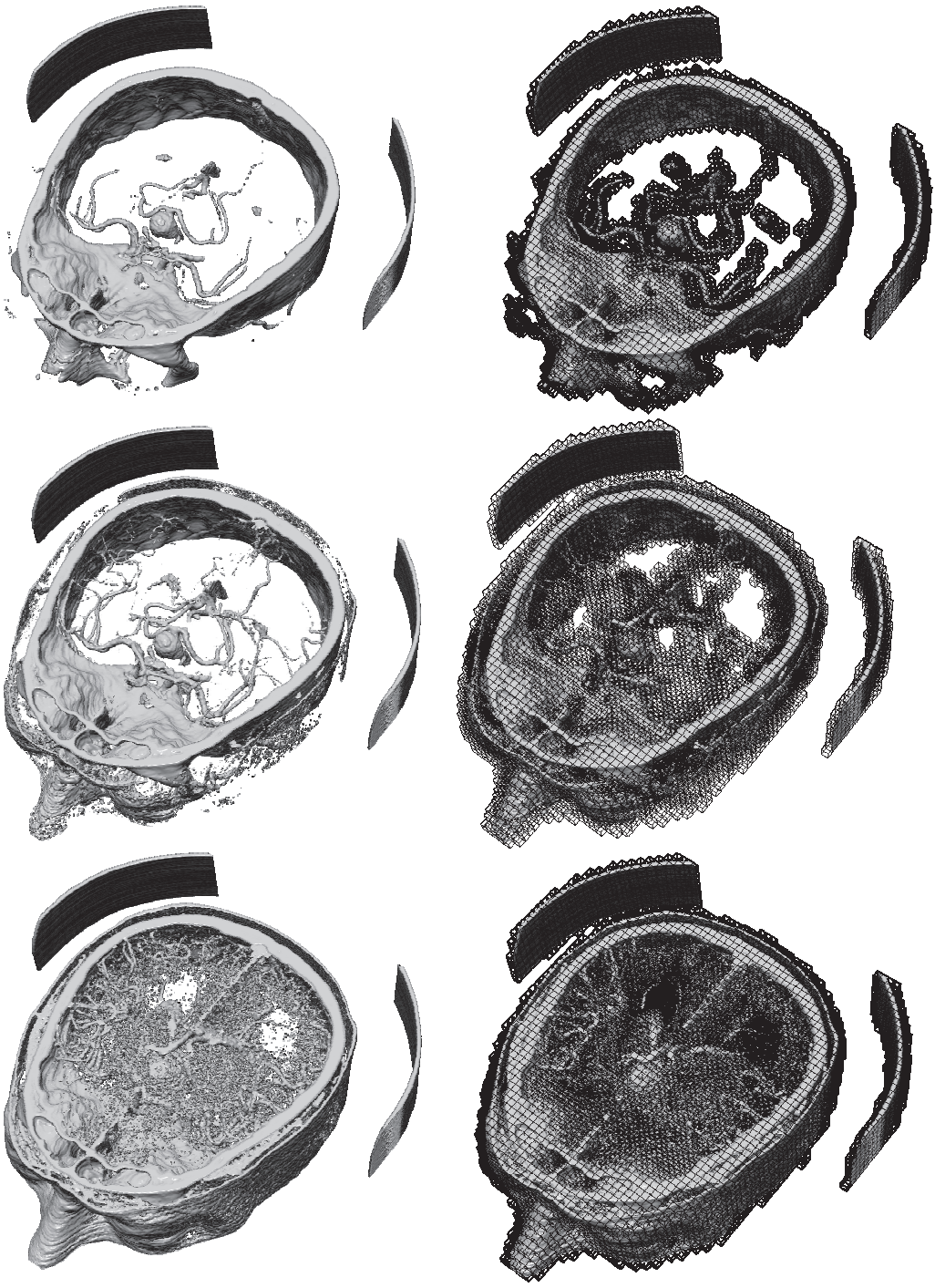

Figure 6: Resulting images and octree decomposition ($32 \times 32 \times 8$) of the example angiography data set for different transfer function settings.