Volume Graphics (2005)
E. Gröller, I. Fujishiro (Editors)

iSBVR: Isosurface-aided Hardware Acceleration Techniques for Slice-Based Volume Rendering

Daqing Xue, Caixia Zhang, Roger Crawfis†

Department of Computer Science and Engineering

The Ohio State University, Columbus, OH 43210, USA

Abstract

In this paper, we examine the performance of the early z-culling feature on current high-end commodity graphics cards and present an isosurface-aided hardware acceleration algorithm for slice-based volume rendering (iSBVR) to maximize its utilization. We analyze the computational models for early z-culling of the texture based volume rendering. We demonstrate that the performance improves with two to four times speedup against an original straightforward SBVR on an ATI 9800 pro display board. As volumetric shaders become increasingly complex, the advantages of fast z-culling will become even more pronounced.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation – Display Algorithms, Viewing Algorithms; I.3.6 [Computer Graphics]: Methodology and Techniques – Interaction techniques.

Keywords: slice-based volume rendering, hardware acceleration, isosurface, early z-culling.

1. Introduction

Real-time direct volume rendering (DVR) of fairly large volumetric dataset (256^3 or more), due to its intrinsic huge number of sampling points, is still a challenge in computer graphics community. Using graphics hardware to accelerate volume rendering is continuously exploited by researchers with the advancing of new hardware techniques. Cullip and Neumann [2] first addressed the capability to render a volume on the 3D texture hardware. Akeley [1] and Cabral et al. [3] described a slice-based volume rendering (SBVR). SBVR is a direct mimic of ray-casting, but samples the volume for all rays at once. The original SBVR slices the whole volume. Engel et al. [4] developed a pre-integrated volume rendering technique for high quality images using multi-texturing. This improves the quality, but not the performance, unless a lower sampling rate can be applied to the volume integration. To improve rendering performance for fairly large dataset, Li et al. [5] split the volume into small bricks. The bricks in empty space are removed and only the non-empty bricks are rendered with SBVR. With the powerful programmability of graphical processing units (GPU) today, many software-based acceleration techniques like empty space skipping and early ray termination [8, 9, 10, 11] can be implemented on the GPU directly. Krüger and Westermann [6] and Roettger et al. [7] develop their algorithms to perform ray-casting using a pixel shader 2.0 program [12] on the GPU with early ray termination and space-leaping. Krüger and Westermann propose an ingenious encoding of the ray direction and length into floating point render targets. These textures are then used to

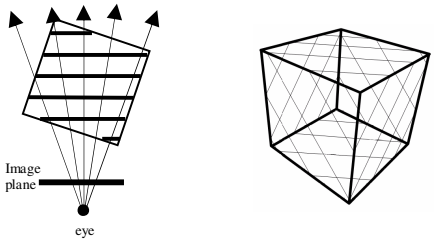


Figure 1: *The proxy geometries of image-aligned slicing planes. (a) 2D diagram of slice planes. (b) The slicing planes intersecting with the volume box.*

†Email: {xue | zhangc | crawfis}@cse.ohio-state.edu

determine where to sample the 3D texture (volume). The early z-culling feature on the latest graphics hardware with pixel shader2.0 makes early ray termination possible in their algorithms.

In a typical slice-based volume rendering, the volume is sliced by the object-aligned or image-aligned planes (see in figure 1). These planes are rendered in a back-to-front or front-to-back order, textured by the 3D texture (volumetric dataset) during rasterization, and finally composited into the frame buffer to generate the final image. A main drawback of SBVR is that, for each slice during rasterization, all fragments are sampled from the 3D texture even though some fragments do not contribute to the final image at all. This greatly reduces the rendering speed, especially when a complex fragment shader including lighting or high-order gradient computation is employed. This is very inefficient since the empty space usually occupies more than one-third of many volumetric datasets. In this paper we present an isosurface-aided hardware acceleration technique for slice-based volume rendering (iSBVR). The acceleration is based on the early z-culling feature provided by the latest consumer level graphics hardware. Given a transfer function, we can analyze it to determine values where the resulting opacity is completely opaque. Extracting iso-contours corresponding to these values provides a blocking surface, where any samples of the volume along the ray that are behind (or within) this surface are not visible. Isosurfaces can also be extracted corresponding to any minimal thresholds in the specified transfer function (i.e., where the transfer function goes to zero opacity). These surfaces do not block the rays as in early ray termination, but can provide a simple space-leaping as we will show later in the paper. More importantly, the minimal isosurfaces can be used to flag areas on the screen where the ray passes entirely through volume without hitting any values that would contribute to the volume integral. We call these rays, empty rays and our algorithm provides an efficient solution for *empty ray skipping*. It should be noted, that these isosurfaces are rendered only to initialize the z-buffer. Nothing is ever skipped, but with early z-culling enabled, the hardware quickly processes these areas resulting in a substantial performance improvement.

The remainder of this paper is organized as follows. Section 2 examines the computational model for early z-culling in volume rendering. Section 3 describes our algorithm for an isosurface-aided acceleration technique and the implementation details of the rendering process. In section 4, we address the isosurface extraction for our volume rendering. We present our results in section 5. In section 6, we draw conclusions from our study and propose future work.

2. Early Z-culling for Volume Rendering

A key observation of brute-force texture-based volume rendering is probably the sheer number of fragment and pixel operations which do not contribute to the final image. This problem becomes more serious with a complex fragment shader, which includes texture accesses for volume sampling, transfer function lookups, gradient and lighting calculations, and blending operations.

Effective utilization of early z-culling feature on graphics hardware is the impetus for our isosurface-aided acceleration technique. The key criterion here is that the z-buffer must be set up properly such that only fragments on the slicing plane that contribute to the final image can pass the depth test. By means of early z-culling, the fragments that do not contribute to the final image will exit from the graphics processing pipeline immediately. As pointed out by Krüger and Westermann [6] and our own experiments, this early termination greatly reduces the rendering time, particularly when complex shaders are desired.

Assume that for each pixel a z-value can be determined such that further samples will be occluded. Ideally, we would like to set our z-buffer to these values. Furthermore, if the ray passes entirely through empty space or air, then the processed fragments can be skipped. Our goal is to set the z-value to the front of the volume for these rays. By setting the z-buffer as such, the rendering speed can be benefited from the early z-culling feature of modern graphics hardware.

A two pass rendering process is used in most games containing complex shaders. In the first pass, a simple shader is performed to set up the z-buffer. If the color buffer is not being changed, newer hardware can actually render this pass twice as fast. In the second pass, a final complex shader is performed. Theoretically, this shader is performed for all fragments. Any fragments which fail the depth test, are then simply discarded. The early z-culling feature of the hardware performs the depth test first, and only if it passes does the resulting complex shader get processed. Hence, only the visible fragments are rendered in the second pass (note, the hardware is a little more complicated than this). The remaining *non-effective* fragments are occluded and the shader on them is skipped.

For direct volume rendering, things are much more complex, as opaque surfaces (or positions) are not clearly defined. Kruger and Westermann [6], developed a ray-caster in the graphics hardware. An early ray termination was implemented using the early z-culling feature, by processing the rays in slabs. After each slab, a rendered texture from the opacity buffer would be examined in a fragment shader, and pixels which were fully opaque would have their z-values set to the current slab position. Roettger, *et al* [7], do a similar thing, with a slab width of 4 samples and an occlusion query test for region of entire image termination. Newer hardware, such as nVidia's 6800 allows for better looping and branching [15] and a true implementation of ray-casting with early ray termination. This does not need, nor use the early z-culling feature of the hardware.

Our first step in solving this problem for slice-based volume rendering is to analyze the performance characteristics of the graphics hardware with respect to the early z-culling. We classify all fragments, F, into either affecting the volume integral or not. Those affecting the integral will need to execute their corresponding fragment shader. Thus, we have F_a fragments for which a complete and potentially complex shader needs to execute, taking on average T_c time per fragment. Our goal is not to remove or ignore any superfluous fragments, but to reduce their shader time to the minimal execution time T_z by the early z-culling. Essentially, there are three main computational parts for a two pass volume rendering:

- Time to set up the z-buffer in the first pass: F_s*T_s;
- Rendering fragments that are discarded by early z-culling in the second pass: $(F-F_c)*T_z$;
- Fragments rendered with the complex shader in the second pass: F_c*T_c.

The above three parts lead to Equation 1 as a computational model for the 2-pass volume rendering time, $T_{2\text{-}pass}$, with a maximal potential speedup, δ, given in equation 2. This is provided we can control the hardware to only execute the complex shader on the affective fragments.

$$T_{2-pass} = F_s * T_s + (F - F_c) * T_z + F_c * T_c \qquad (1)$$

$$\delta = \frac{F * T_c}{F_s * T_s + (F - F_c) * T_z + F_c * T_c} \qquad (2)$$

Where,

- F: the total number of fragments generated from the volume;
- F_s: the number of fragments to set up the z-buffer in the first pass;
- F_c: the number of fragments fed into the complex shader in the second pass;
- T_s: the operation time of a simple shader to set up the z-buffer in the first pass;
- T_z: the operation time for a fragment discarded by the early z-culling (with no fragment program at all) in the second pass;
- T_c: the operation time of a complex shader to render the final image in the second pass.

For slice-based volume rendering, each slice is rendered twice. In the first pass, a simple shader is applied to modify the z-buffer if a pixel reaches opaque in the opacity buffer. This slice is rendered again by a complex shader with early z-culling enabled. In this case, the number of fragments, F_s, in the first pass to set up the z-buffer equals to the total number of fragments, F. In general, the simple shader time, T_s, is close to T_z and we will use T_s to approximate T_z in our later discussion. Equation 1 and 2 can thus be approximated by:

$$T_{2-pass} = F * T_s + (F - F_c) * T_s + F_c * T_c \qquad (3)$$

$$\delta = \frac{F * T_c}{F * T_s + (F - F_c) * T_s + F_c * T_c} \qquad (4)$$

The simple shader time T_s is fixed for a given graphics hardware, and the complex shader time T_c varies upon different shaders. Let the fragment culling rate be $\alpha = (F - F_c)/F$ and the simple shader speed-up be $\gamma = T_s / T_c$. Equation 4 can then be simplified as:

$$\delta = \frac{1}{\alpha\gamma + 1 - \alpha + \gamma} \qquad (5)$$

Two pass rendering is beneficial when the speedup, δ, is greater than 1. Substituting $\delta > 1$ into equation 5, we achieve our desired property:

$$(1-\gamma)(\alpha+1) > 1 \qquad (6)$$

Inequality 6 describes for a given shader (γ is fixed), how many fragments must be occluded to gain a speedup in any two pass SBVR of the volume. For example, if a complex shader has $\gamma = 0.2$, the fragment culling rate, α, must be greater than 25% to gain a speedup. The goal of the next section is to provide a fast and efficient scheme for setting the z-buffer such that F_c is as close to the number of affecting fragments, F_a, as possible.

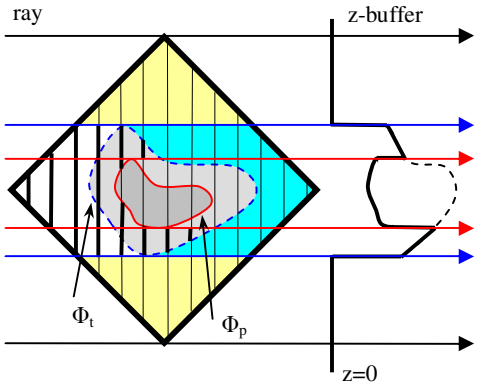


Figure 2: *The back faces of isosurfaces Φ_t and the front faces of isosurface Φ_p are rendered with parallel projection and their corresponding z-buffer (right). Only the slices in bold pass the depth test and contribute to the final image.*

3. Isosurface-aided Hardware Acceleration

While our algorithm will work with any opacity-based volume shader, it relies on the mapping from function values to opacities (i.e., the transfer function), to have certain characteristics. Early ray-termination will only occur when the transfer function reaches a maximum opacity of one. Space-leaping and empty ray removal, provide greater benefits when the transfer function contains regions with zero opacity. In other words, if there is no any empty space, we can not remove it. If the transfer function does not have either of these properties, then it should be noted that there is no overhead associated in the volume rendering due to this technique.

For simplicity in the discussion, we will assume we have only two isosurfaces, Φ_t and Φ_p, given by a boundary threshold where the opacity goes from zero to a non-zero

value and an opaque threshold where the opacity reaches one. A very simple example is given in Figure 9e. In general, several iso-values can be used, albeit at a potential rendering cost. The resulting isosurfaces are extracted in either a pre-processing stage, or whenever the transfer function is changed. Figure 2 illustrates a cross-section of the volume rendering process containing an opaque iso-contour in red, and the boundary iso-contour in blue (dashed line). For discussion, we will also assume that all isosurfaces are closed for now, and that the opaque iso-value surfaces are contained within the minimal isosurfaces. This former assumption will be discussed and removed in later sections. If the later assumption is violated, then only an opaque surface will be visible. Note, these assumptions are on the opacity values, not the actual function values.

3.1 *Pseudo* Early Ray-Termination

Clearly, any fragments which lie behind another fragment which is opaque, will not contribute to the volume integral. A fragment which is opaque will be contained with the isosurface, Φ_p. If we set the z-buffer to the front-faces of this isosurface, we will enable early z-culling on the remaining fragments. This is not true early ray-termination, in that the ray could reach maximal opacity long before reaching an opaque isosurface. This region is depicted by the depth buffer between the two red rays in Figure 2. The main steps to initialize the z-buffer for early ray-termination are thus:

- **z-Buffer Initialization:** for early ray termination.

  1. Disable the output to the color buffer;
  2. Set the depth function to GL_LESS (the default);
  3. Render the **front** faces of Φ_p.

This simple process provides a speed-up from 30% to 50% in our tests.

3.2 Space-Leaping

A typical volume will have many pockets of empty space, some between the eye and the volume material, some within the volume and some between the volume and the background. Culling away all of these fragments is a challenging research question. Space-leaping typically concentrates on removing the material between the eye and volume. This corresponds to the first crossing of the ray with the minimal iso-contour value or Φ_t surface. We can set the z-buffer to these crossing, by rendering the front faces of the isosurface. Early z-culling can then be achieved by using a GL_GREATER depth test on the fragments. The main steps to initialize the z-buffer for space-leaping are thus:

- **z-Buffer Initialization:** for space-leaping.

  1. Disable the output to the color buffer;
  2. Render the **front** faces of Φ_t.
  3. Set the depth function to GL_GREATER;

3.3 Combined Space-Leaping and Early Ray-Termination

Early ray-termination requires a GL_LESS test, while space-leaping, a GL_GREATER test, seeming to preclude the use of both accelerations in the same rendering. Space-leaping is usually associated with setting the initial sample location for a ray. We can reverse the ray direction, and test if the current sample location is the last contributing sample along the ray. Here, we remove the material between the volume and the background, and call this exit-based space-leaping. This corresponds to the last crossing of the ray with the minimal iso-contour value or Φ_t surface. This is the region between the two blue rays in Figure 2. The main steps to initialize the z-buffer for exit-based space-leaping are thus:

- **z-Buffer Initialization:** for exit-based space-leaping.

  1. Disable the output to the color buffer;
  2. Render the front faces of the volume's bounding box.
  3. Set the depth function to GL_GREATER;
  4. Render the **back** faces of Φ_t.
  5. Set the depth function back to GL_LESS;

Now, to combine this with the early ray-termination, we simply need to perform the initialization for exit-based space-leaping before the initialization for early ray-termination. After the exit-based space-leaping initialization, the z-buffer will either have values corresponding to the front faces of the bounding box, or the last surface of the minimal iso-value. For our assumptions with closed iso-contours, the early ray-termination surfaces, Φ_p, will project only to areas already covered by the isosurface, Φ_t. Since the z-buffer was pushed away from the cube faces in these regions, the early ray-termination initialization will pull these back towards the viewer.

3.4 Empty Ray Removal

For sparse values, many rays do not intersect any meaningful data in the volume. The rays end up being set to the background color. This implies that a ray never crosses through the minimal isosurface, Φ_t, (and by the closed assumption the opaque isosurface, Φ_p, as well). Early z-culling for the fragments in these areas will work if the z-buffer is set to a minimal value (zero or the front faces of the volume). The exit-based space-leaping algorithm above actually accomplishes this already. The region in yellow in Figure 2 represents the empty rays, and the resulting z-buffer is set to zero in this case. Combining the empty ray removal and the exit-based space-leaping provides a substantial speed-up between 200%-300%.

3.5 Culling Efficiency

Our final, and significant, result is that only the fragments on the bold portion of the slices in Figure 2 will pass the depth test and execute any complex shader associated with them. The performance improvements from both empty

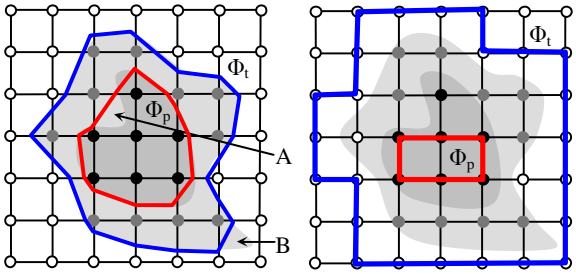


Figure 3: *Isosurfaces and their reduced form in cube faces. Left: isocontouring, Φ_t and Φ_p. Right: Φ_t is inflated to the outer faces of the cubes containing it. Φ_p is shrunk to the outer faces of the inter cubes.*

space skipping and early ray termination, achievable with most software-based ray-casting algorithms [8, 9, 10, 11], are now accomplished in the context of slice-based volume rendering by leveraging the early z-culling feature of modern graphics hardware. We still have some fragments which do not contribute to the final image, but pass through the z-cull operation. Hopefully this set is greatly reduced. The actual results will be data set and transfer function dependent. To render the volume, we simply need to turn on depth testing as usual and process the volume slices. The algorithm works equally well using a back-to-front or a front-to-back slicing order. The volume is rendered as usual:

- **Volume pass:** render the proxy geometries.

1. Enable the output to color buffer;
2. Set the depth function to GL_LESS;
3. Enable the fragment or volume shader;
4. Render the proxy geometry.

Figure 9 shows the isosurfaces and the resulting z-buffer after the initialization passes. A black or darker value indicates $z = 0$ while a white value indicates $z = 1$. Darker values are closer to the eye.

In order to characterize our algorithm, we need to consider the rendering time, T_a, from the two initialization passes (equation 7) and the rendering time, T_m, for the volume shader (equation 8). Any resulting speedup is characterized by equation 9.

$$T_a = F_t * T_s + F_p * T_s \tag{7}$$

$$T_m = (F - F_c') * T_s + F_c' * T_c \tag{8}$$

$$\delta = \frac{F * T_c}{T_a + T_m} = \frac{F * T_c}{F_t * T_s + F_p * T_s + (F - F_c') * T_s + F_c' * T_c} \tag{9}$$

Where,

F_t: the total fragments generated from the back faces of Φ_t ;

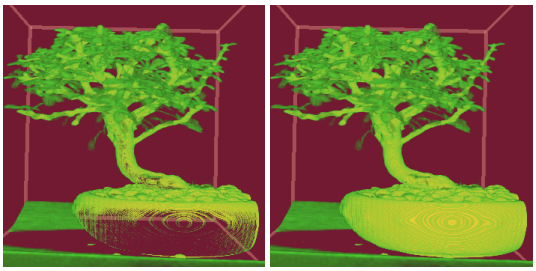


Figure 4: *(a) Generated from the original isosurface. There are holes in the image due to the incorrect occlustion. (b) Generated from the reduced isosurface with holes removed.*

F_p: the total fragments generated from the front faces of Φ_p;
F: the total fragments generated from the volume;
F'_c: the number of fragments fed into the complex shader in iSBVR.

Obviously, if the time it takes to render the isosurfaces approaches the volume rendering time, and potential speed-ups in the volume rendering are lost. The next examines the issues associated in generating and rendering the isosurfaces.

4. Isosurface Extraction

For a typical dataset of 256^3, there could be more than a million triangles on the isosurface (see figure 6a) for a reasonable transfer function. The rendering time for this large number of triangles offsets any speedup from early z-culling. To reduce the isosurface rendering time, an octree is generated from the underlying volume to extract the isosurface. Each octree node contains a min-max value pair representing the minimal and maximal voxel values it includes.

However, when generating the isosurface from the octree using the maximal value in the octree node, the iso-suface, Φ_t, may occlude some voxels even though their values are greater than the iso-value since the voxel with the maximal value is not necessarily the vertex in the node for iso-contouring. The voxel labelled B in figure 3(left) shows this case. Similarly, the isosurface, Φ_p, may contain the non-opaque area as A in figure 3(left). The holes in Φ_p will produce serious aliases since the rays will stop at the Φ_p due to early z-culling. To solve these problems, the isosurface, Φ_t, is inflated to fill the outmost cubes that containing it and the Φ_p is shrunk to the maximal set of the cubes completely included inside Φ_p. Figure 3(right) shows the reduced isosurfaces for Φ_t and Φ_p. Figure 4 show the bonsai dataset rendered by the original isosurface and its reduced isosurface. The holes in the original one (figure 4a) have been removed in figure 4b.

4.1 Isosurface for Empty Space Skipping

As shown in figure 2, only the back faces of Φ_t are used to set up the z-buffer for empty space skipping. Thus, we can generate a set of cubes which contain the manifold of the isosurface, and render these cubes with back faces and with GL_GREATER for depth testing.

The isosurface must be closed to correctly set up the z-buffer in the two initialization rendering passes in section 3.1 and 3.3. Otherwise, there are the undesired z-values from the front faces for the open area in the z-buffer that will incorrectly occlude the fragments in the final image. However, if the value of the voxel on the volume boundary is greater than the input isovalue, the final output surface will be open around such voxels. To create the close isosurface, the cube on the volume boundary is also added to the cube set if it is inside of the isosurface. The algorithm to create the cube set is listed in figure 5.

```
Input:  the cubes of the octree
Output: the cube set S containing Φt.
1)  Set the cube set S = ∅;
2)  For each cube d in the octree of the volume
3)     If d contains isosurface
4)        S = S ∪ {d};
5)     Else if d is inside the isosurface and on the volume
          boundary
6)        S = S ∪ {d};
7)     Endif
8)  Endfor
9)  Return S.
```

Figure 5: *algorithm to generate the cube set for Φ_t.*

4.2 Isosurface for Early Ray-Termination

To create isosurface, Φ_p, for the opaque values, we use the minimal value in each octree node. When the octree node size is big (accordingly low resolution with respect to the original volume), some parts of the isosurface could be missing. This is not desired since we want more fragments can be rendered to set up the z-buffer for early ray termination. This problem becomes more serious, especially for medical datasets in which the skull or thin bones cannot contain a complete cube from the octree node. Figure 6 shows a 2D diagram where the iso-contour shrinks when iso-contouring using the minimal value from the min-max pair in the octree node. Figure 7 shows the isosurface of Φ_p from a Siemens' head dataset. It shrinks drastically when octree node size increases from 1 to 8. There are almost no pixels to be set with the z-values for early ray termination in the initialization pass if using octree node of 8x8x8. In our experiments, the octree node of 2x2x2 for Φ_p provides the good balance between the overhead to rendering the triangles on Φ_p and the benefit from the early ray termination.

To shrink the isosurface as shown in the figure 3(right), we shrink the cube set containing isosurface, Φ_p, until it only includes the cubes which are completely inside Φ_p. The algorithm for creating such cube set, *T*, is in figure 8.

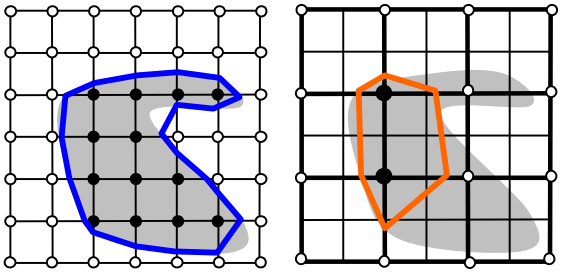
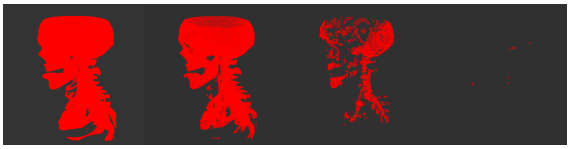


Figure 6: *Left: iso-contouring for 7x7 grid. Right: the grid is generated from left with quad-tree node of 2x2. The vertex value is determined by the minimal value of each 2x2 node from the left grid.*



(a)	(b)	(c)	(d)
original	*2x2x2 node*	*4x4x4 node*	*8x8x8 node*
1,059,856 Tri	*213,416 Tri*	*16,792 Tri*	*80 Tri*

Figure 7: *The isosurface shrinks drastically when using the minimal value to perform contour on different octree levels.*

```
Input: the cube set E containing Φp
Output: the shrinking cube set T
1)  Set the cube set T = ∅;
2)  Repeat each cube d in E
3)     E = E - {d};
4)     For all d's neighbour nj along shrinking direction
5)        If nj is completely inside Φp and nj∉ T
6)           T = T ∪ { nj };
7)        Else
8)           E = E ∪ { nj };
9)        Endif
10)    Endfor
11) Until E = ∅
12) Return T.
```

Figure 8: *algorithm to generate the cube set for Φ_p.*

5. Results and Discussion

All performance data were obtained on a PC equipped with an ATI 9800 pro graphics card with 128 MB video memory. The slice spacing was set as same as the voxel interval. The results are listed in table 1. The resulting imagery from a gradient-based shader is shown in figure 10. Our results show that we obtain on average a 2 to 4 times speedup. The performance of empty space skipping (ESS) and early ray termination (ERT) for goldenlady dataset is greatly reduced due to the large fuzzy area in the volume (figure 10c).

The slices in front of any front face of Φ_t are still fully rasterized and executed by the fragment program (see in figure 2) since their depths are always less than the pre-rendered depth value in the z-buffer. This problem can be solved by rendering each slice with one more pass as in [4], in which a simple shader is performed to modify the z-buffer to occlude pixels reaching opaque. On the other hand, considering the large number of slices, the overhead of the additional rendering passes for all the slices partly offsets the performance improvement. If OpenGL would support a depth band-test with dual z-buffers, this would further improve the current frame-rates by culling all fragments outside of the two z-buffers without additional overhead (assuming the additional depth test is free). The newly introduced GL_DEPTH_BOUNDS_TEST_EXT provides such a similar but much simplified function, in which a user specified depth range test between [0..1] is applied to fragments in addition to the normal depth test. This extension can help the performance improvement of the iSBVR if all the affecting voxels of the underlying volume fall in a small range.

The iso-contouring is performed in either a pre-processing stage, or whenever the transfer function is changed. Since we only apply iso-contouring to volume octrees (8x8x8 and 2x2x2 nodes for Φ_t and Φ_p, respectively), this greatly reduces the number of cells for iso-contouring. We can still obtain interactive rendering speed when changing transfer functions. In our experiments, the timings for isosurface extraction are 10 ms and 210 ms for Φ_t and Φ_p, respectively. The other well-studied accelerated isosurface extraction techniques [16, 17] can be used to further enhance the performance.

6. Conclusion and Future Work

By means of the early z-culling feature, we have developed an isosurface-aided hardware acceleration technique for slice-based volume rendering to gain the improved frame-rates of three to four times. The advantages of early z-culling become more pronounced for hardware accelerated volume rendering.

This isosurface-aided acceleration can be easily fit into the other existing GPU volume rendering pipeline like Krüger and Westermann's GPU-based ray caster and the pre-integrated volume rendering [4].

7. Acknowledgement

The Authors would like to thank the anonymous reviewers for their valuable comments. The datasets of CT head and golden lady are courtesy to Siemens Medical Solutions, the bonsai datasets courtesy to S. Roettger at University of Stuttgart, and the aneurism dataset courtesy to Philips Research, Germany.

Dataset	Basic	ESS	ERT	ESS+ERT	Speedup
Aneurism (256^3)	7.3	21.8	9.2	24.8	3.4
Head (256^3)	7.3	16.0	11.6	24.3	3.3
Golden lady (256^3)	7.3	14.2	8.6	16.3	2.2
Bonsai (256^3)	7.3	15.8	9.1	27.8	3.8

Table 1: *The rendering FPS for SBVR. ESS: empty space skipping; ERT: early ray termination.*

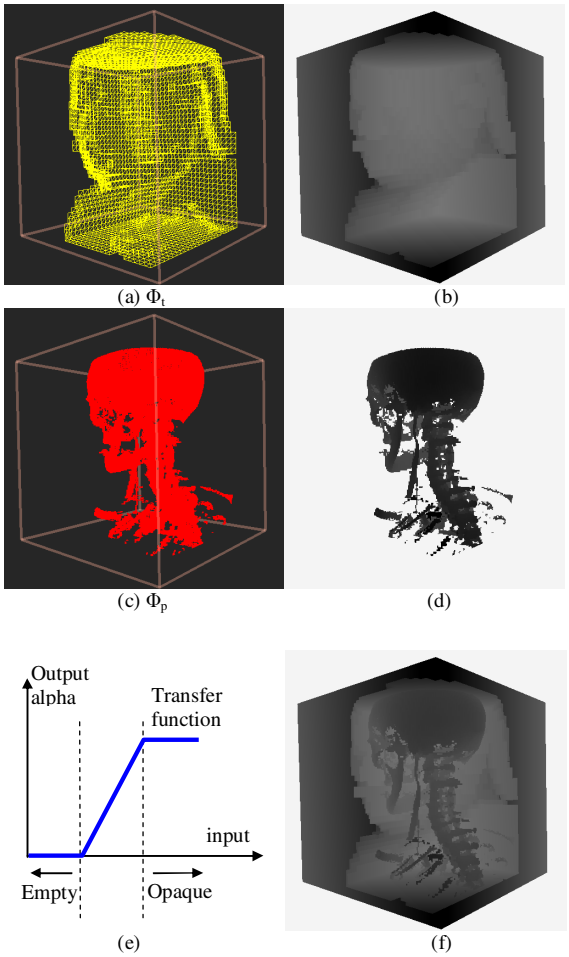


Figure 9: *(a) The back faces of isosurface Φ_t; (b) The z-buffer after rendering the isofurace in (a); (c) The front faces of isosurface Φ_p; (d) The z-buffer after rendering the isosurface in (c); (e) the transfer function for two isosurfaces; (f) The z-buffer is rendering after the two initialization passes. Note: the values in the z-buffer images (right column) are rescaled to highlight the difference.*

References

[1] Kurt Akeley. RealityEngine Graphics. *Computer Graphics (SIGGRAPH '93 Proceedings)*, 27:109–116, 1993.

[2] T. Cullip, and U. Neumann. Accelerating volume reconstruction with 3D texture hardware. Tech. Rep. TR93-027, University of North Carolina, Chapel Hill N.C.

[3] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings ACM Symposium on Volume Visualization 94*, 91–98.

[4] K. Engel, M. Kraus, and Th. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Eurographics Workshop on Graphics Hardware '01*, pages 9–16. ACM SIGGRAPH, 2001.

[5] W. Li, K. Mueller, and A. Kaufman. Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering. In *IEEE Visualization*, Seattle, WA, 2003.

[6] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *IEEE Visualization*, Seattle, WA, 2003.

[7] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart Hardware-Accelerated Volume Rendering. In *Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization*, 2003.

[8] M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics 9,* 3(July), 245-261.

[9] K. Danskin, and P. Hanrahan. Fast Algorithms for Volume Ray Tracing. In *ACM Workshop on Volume Visualization '92*, 91-98.

[10] R. Yagel, and Z. Shi. Accelearted Volume Animation by Space-Leaping. In P*roceedings IEEE Visualization '93*, 62-69.

[11] J. Freund, and K. Sloan. Accelerated Volume Rendering Using Homogeneous Region Encoding. In *Proceedings IEEE Visualization '97*, 191-197.

[12] Microsoft. DirectX9 SDK. http://www.microsoft.com/DirectX. 2002.

[13] T.Y. Lee and C.H. Lin. Growing-cube isosurface extraction algorithm for medical volume data. Comput Med Imaging Graph. 2001 Sep-Oct;25(5):405-15.

[14] W. Lorensen and H. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. Computer Graphics, Vol. 21, No. 4, July, 1987.

[15] NVIDIA. http://www.nvidia.com/page/geforce_6800.html.

[16] H.-W. Shen, C. Hansen, Y. Livnat, and C. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). *IEEE Visualization '96*, pages 287-294.

[17] B. von Rymon-Lipinski, N. Hanssen, T. Jansen, L. Ritter, E. Keeve. Efficient Point-Based Isosurface Exploration Using the Span-Triangle, *IEEE Visualization '04*, pages 441-448.

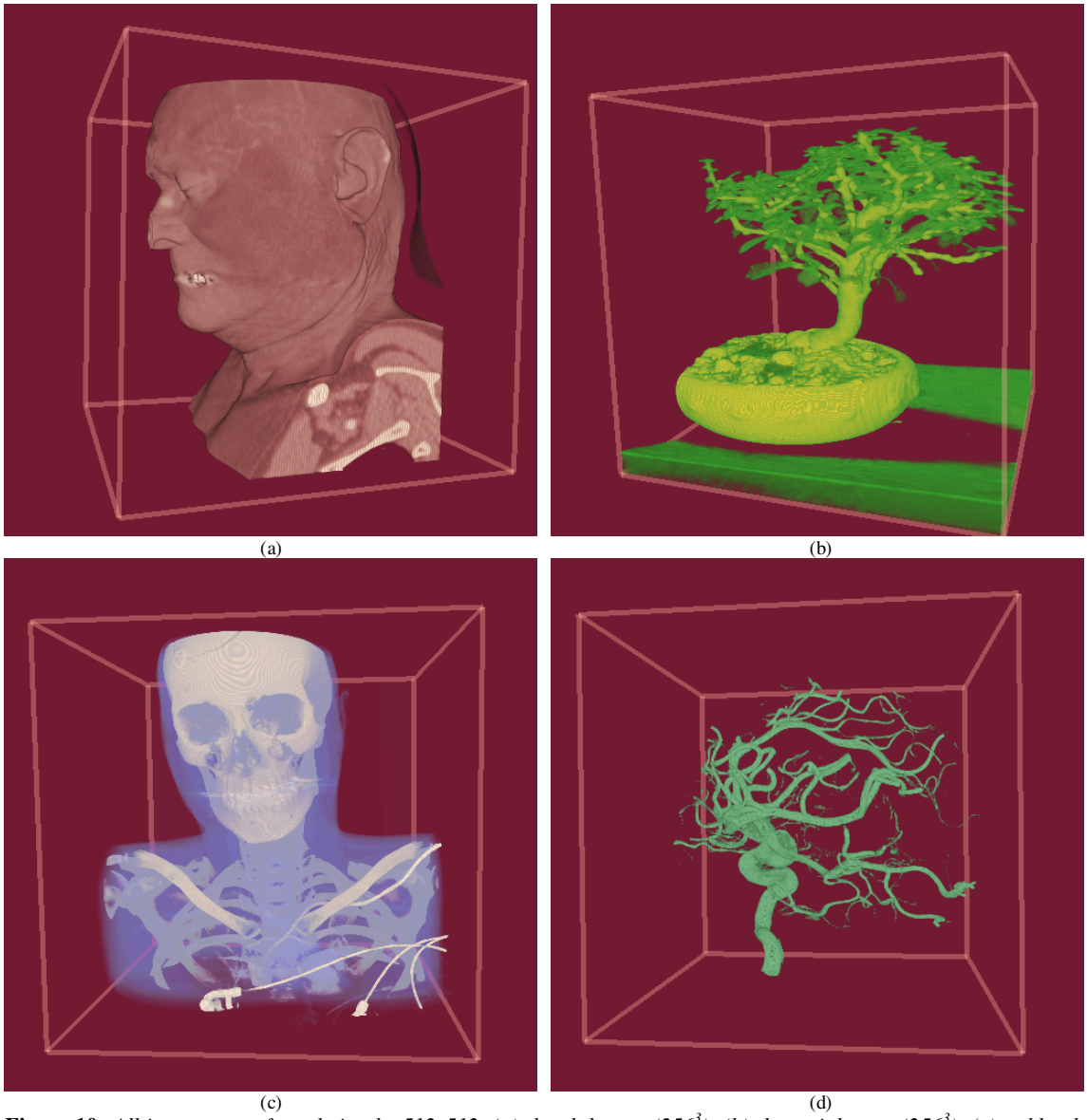


Figure 10: *All images are of resolution by 512x512. (a): head dataset (256^3); (b): bonsai dataset (256^3); (c):golden lady dataset (256^3); (d): aneurism dataset (256^3).*