Image-Based Rendering of Surfaces from Volume Data

Baoquan Chen * Arie Kaufman† Qingyu Tang †

Abstract

We present an image-based rendering technique to accelerate surfaces rendering from volume data. We cache the fully volume rendered image (called *keyview*) and use it to generate novel view without ray-casting every pixel. This is achieved by first constructing an underlying surface model of the volume and then texture mapping the keyview onto the geometry. When the novel view moves slightly away from the keyview, most of the original visible regions in the keyview are still visible in the novel view. Therefore, we only need to cast rays for pixels in the newly visible regions, which usually occupy only a small portion of the whole image, resulting in a substantial speedup. We have applied our technique to a virtual colonoscopy system and have obtained an interactive navigation through a 512^3 size patient colon. Our experiments demonstrate an average of an order of magnitude speedup over that of traditional volume rendering, compromising very little on image quality.

1 INTRODUCTION

Major efforts have been dedicated to accelerating volume rendering, in both a software and hardware approach. One hardware approach is to leverage existing hardware, especially 3D texture mapping hardware, to accelerate volume rendering [3, 8, 29]. However, 3D texture mapping hardware is not scalable and does not support interactive classification. Another hardware approach is to utilize special-purpose hardware, such as, VolumePro [24], for volume rendering. VolumePro can render a 256^3 volume at 30 frames per second. However, the performance of VolumePro is still challenged by large data set. For example, for a 512^3 data, theoretically only 3-4 frames per second rendering performance can be obtained. Furthermore, the current VolumePro supports only parallel projection. Many applications, for example, the virtual colonoscopy system in biomedical application, require perspective projection. Parallel projection of a straight tube appears as a ring, which prevents us from examining the interior walls of the colon. With some software design, VolumePro could be utilized to support perspective projection, except at great cost of the rendering speed as well as image quality [16]. A typical data size in a virtual conlonoscopy system is 512^3; a straightforward implementation using VolumePro delivers a much lower frame rate than the interaction requirement when perspective rendering is supported.

The software approach includes early ray termination [19], distance encoding, presence acceleration [1, 6], ray coherence [14], and shear warping [17]. These methods still cannot meet the interactive speed requirement unless they are parallelized on powerful machines [17, 23].

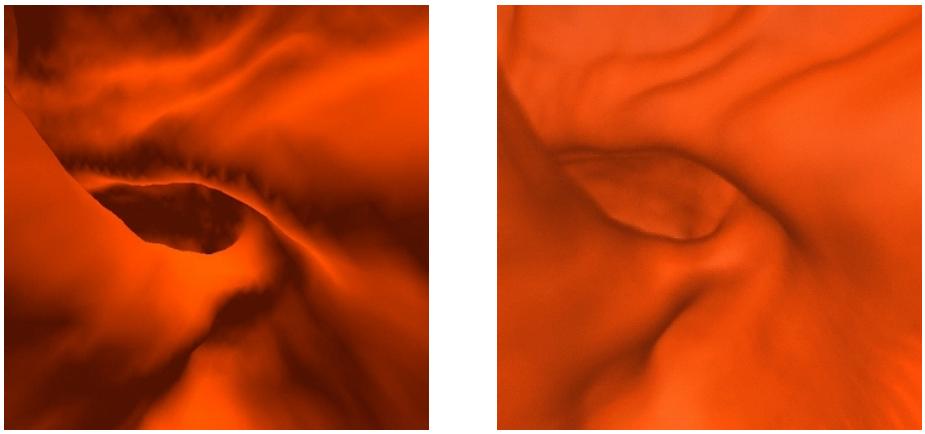
Recently, image-based rendering techniques have been developed to model complex scenes using images or sprites instead of complex geometries, and to render directly from images. One can pre-compute a group of *keyviews* (or *reference images*) so that any novel view can be generated from one (or a subset) of these keyviews. This is similar to sampling the full plenoptic function [21], resulting in 3D, 4D or even 5D image sprites [12, 20]. These methods usually need a very dense view sampling (thousands of images), which requires a large amount of memory storage. Dally et al. [9] use a delta tree to represent all the keyviews.

Even though most of these methods were developed for accelerating polygon rendering, their concepts can be applied to accelerating volume rendering. Choi and Shin [5] have applied Dally et al.'s method to render the fully opaque surface of the volume using a quadtree structure instead. Their method supports only parallel projection. Even though tens of thousands of reference images are precomputed, holes may still appear in the generated image. The nearest neighbor (zero-order) interpolation has been used for efficiency, but at the cost of degrading the image quality. Generally, a traditional IBR method is inappropriate for volume rendering because whenever the transfer function changes, it is required to regenerate the whole set of reference images, resulting in a huge amount of computation and storage. Gudmundsson and Randen [13] have proposed to utilize the coherence between neighboring views and to incrementally generate the novel view. This incremental approach avoids storing thousands of images. Similar to the above technique, it supports only parallel projection and the rendering of the fully opaque surface. In addition, it delivers a degraded image quality as only a zero-order interpolation is used; similarly, holes may appear in the generated image. These disadvantages have prevented these methods from a broad usage in volume rendering. Yagel et al. [30] have extended Gudmundsson and Randen's technique to support flexible volume rendering. Rather than using the incremental method to directly generate the final image, they only use it to generate the "C-buffer," storing the object-space coordinates of the first non-empty voxel visible from every pixel. They further use thus generated "C-buffer" to accelerate the ray-casting operation by "space-leaping" (skipping the empty space). Full rays are still required to cast into the data. Currently, space-leaping is considered as a standard optimization technique for ray-casting, together with several other techniques, such as early ray termination. The technique to be discussed in this paper will be compared against the existing optimized ray-casting.

Most recently, more research has been conducted to further accelerate volume rendering using the image-based approach. Brady et al. [2] have proposed a two-phase approach. This method divides the viewing frustum into regions (or slabs) based on the Euclidean distance from the eyepoint. The algorithm then ray-casts each region separately. The images created from the regions are subsequently texture mapped onto polygons (or billboards) and composited together using conventional graphics hardware. The images of some back regions are reused for compositing to speedup the rendering of subsequent frames. A problem with billboards is that when the viewpoint changes, the pre-generated billboard images will be shifted with each other. This leads to two unpleasant situations: (1) holes may appear at the edges of the billboards or (2) the integrated object may be pulled apart. More recently, Mueller et al. [22] have reported a scheme to enhance the billboards with depth information. There, a billboard (or a slab image) is subdivided into a grid of small tiles and a z-value is approximated for each tile to form a coarse scale "z-buffer." The z-value represents the average of the nearest and farthest z-values of the object. A *polymesh* for each slab image is then generated from the "z-buffer," which represents a

*4-192 EE/CSci Building, 200 Union St. SE, Minneapolis, MN 55455. Email:{baoquan}@cs.umn.edu

†Center for Visual Computing and Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794-4400. Email:{ari}@cs.sunysb.edu



(a) (b)

Figure 1: *Comparison of the images rendered by (a) polygon rendering and (b) volume rendering.*

spatial approximation of the underlying object surface. Now, when the slab images are mapped onto the polymeshes rather than the planes, the viewpoint is allowed to move further before holes and artifacts become visible. Nevertheless, this method does not eliminate the holes and artifacts completely. While for the translucent objects, these artifacts tend to be disguised by their fuzzy nature; they can show up easily for more opaque objects with clear structures. The main source of these artifacts is the way that the polymeshes are constructed and used. First of all, because the polymesh for each slab stays in the center of the slab, the neighboring meshes do not attach to each other, which makes it inevitable that holes will show up when the novel viewpoint moves to the side. For example, if a box is cut into two slabs, the two polymeshes constructed will simply be two separate planes in the center of each slab. When the novel viewpoint moves slightly to the side, the side faces of the box will be immediately pulled apart. This is immediately visible for a non-transparent box. However, when the box is transparent and filled with gel, the artifact may be tolerated longer. Second, since the meshes are view-dependent, they do not reflect the real 3D topology of the scene. For example, when two separate objects in 3D space overlap on screen, the polymesh generated will be a connected mesh. Therefore, when the novel viewpoint moves to a position where we should see the gap between the two objects, instead, we will still see a connected object. This situation is especially problematic for a colonic navigation where holes persist. Finally, while Mueller et al. have experimented the manipulation of an object by rotating it back and forth around the keyview, no continuous navigation has been demonstrated. However, in a continuous navigation system, the smoothness of the image quality is desired; the artifacts issue becomes pressingly critical, but more challenging to solve.

The goal of this paper is to design a novel method solving all above problems, except we focus on accelerating Levoy's surface rendering from volumes [18]. We eliminate dense view sampling by constructing a 3D model from the volume and utilizing spatial coherence during the navigation, very similar to [22]. Here, we first render the volume using the conventional volume rendering and cache the rendered image as a *keyview*. For subsequent novel views, we texture-map the keyview image onto the underlying geometry and view it from a novel camera position. Although the high level philosophy here is similar to Brady et al.'s and Mueller et al.'s methods, the differences are distinguishable, which mainly come from the mesh we use: (1) We use a view independent full geometry, which faithfully represents the underlying object surface. It is an enclosed and integrated mesh, therefore no holes will show up in the rendering. In addition, the mesh does not require to be constructed for every keyview (it does need to be reconstructed whenever the transfer function changes). (2) We utilize this view-independent full geometry to detect newly emerging scenes on the fly and send new rays to render these scenes. Furthermore, we utilize the mesh to detect which part of the object needs to be re-rendered to avoid poor quality of texture mapping. These operations allow us to control the quality of the generated image, rather than passively let the image quality degrades. (3) We employ an adaptive mesh by using more polygons for the surface with a higher curvature, while less polygons for a relatively flat surface. This is superior to the uniform polymesh, which does not adapt to the underlying surface topology. An adaptive mesh promises better image quality than a uniform mesh if both use the same number of polygons. This is very important when we have a limited budget to render polygons, which is the usual condition. (4) In addition, we have built up continuous Levels-Of-Detail (LOD) meshes of the 3D geometry so that we can select a typical level for a practical navigation to fine tune the performance and image quality. In general, using the model with more polygons delivers a higher quality image, yet, lowers the rendering performance (but there is a limit on the image quality gain by increasing the level of the model).

We strive to apply our technique to a real navigation system to demonstrate the effectiveness of the technique. We have chosen the virtual colon navigation system as our major application, even though our technique can be applied to other volume rendering systems. Previously, a polygon-based interactive virtual colon navigation system has been developed [15], where the colon was represented as a polygonal surface to seek graphics hardware support. Yet, polygon rendering not only removes small features of the object but also adds some artifacts caused by the sharp edges and silhouettes of the polygon surface [28] (See Figure 1a). A direct volume rendering technique can meet physicians' demand by directly mapping certain ranges of sample values of the original volume data to different colors and opacities (See Figure 1b). However, direct volume rendering, even rendering only thin surfaces, is very expensive, especially for large size colon data, such as 512^3. As we have discussed earlier, it is a paramount challenge even with the VolumePro support. Mueller et al's method appears inappropriate for such an application for the aforementioned reasons. Further efforts have been dedicated to accelerating the volume rendering using a depth buffer [31] or distance field [28], to skip the empty space inside the colon, both with a parallel implementation. The performance achieved still did not meet the interactive speed. Generating a frame of a patient's colon of $512 \times 512 \times 361$ needs about

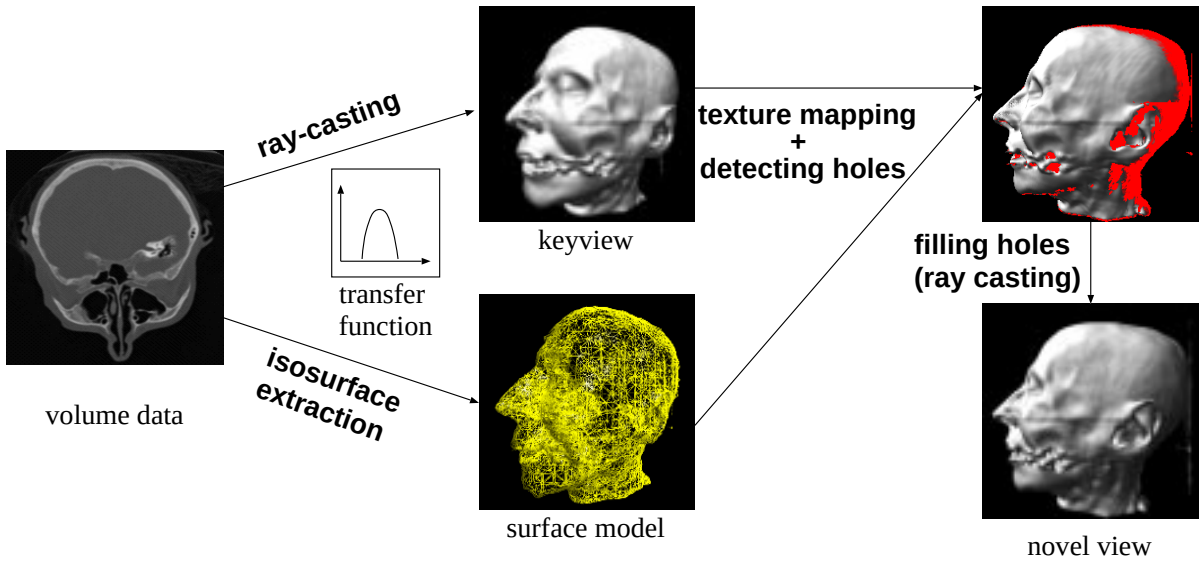


Figure 2: *Overview of our algorithm.*

one second when fully casting 512×512 rays on an SGI Challenge using 9 R10000 processors. Our goal is to make this system interactive on a commodity machine (with only a single processor) while maintaining the features of direct volume rendering.

In the next section (Sec. 2), we give an overview of our algorithm. After that, we discuss issues involved in the algorithm, including geometry construction (Sec. 3), texture mapping and polygon visibility detection (Sec. 4) and strategies of rendering keyviews (Sec. 5). Finally, implementations and results are demonstrated in Sec. 6, followed by conclusions in Sec. 7.

2 ALGORITHM OVERVIEW

The central idea of our algorithm is to texture-map the volume rendered image, or *keyview*, onto an underlying geometry constructed from the volume to speedup the generation of novel view. Figure 2 illustrates our algorithm. During the preprocessing, we construct a surface model from the input volume. Intuitively, this surface model represents the outmost visible surface of the volume determined by the input transfer function — a mapping from a voxel's density to opacity value. Therefore, the iso-value of the surface model is chosen as the minimum voxel value with non-zero opacity. Once we have the surface geometry of the volume and the volume rendered keyview(s), we can use them to speedup the generation of the novel view(s). The general guideline is that we can texture map the keyview image onto the geometry and project the geometry to the novel view position. A critical issue that we strive to address is to maintain the image quality. When performing the texture mapping, holes may appear. We consider two situations. The first situation is that when the viewpoint changes, the originally occluded or viewing frustum culled parts in the keyview could become visible. For these newly visible parts, no textures are available from the keyview, hence, we have to cast new rays (the red rays in Figure 3). To achieve this, we detect each polygon's visibility in the keyview when performing the texture mapping and render invisible polygons with a user specified special color (red (or lightly shaded on black-and-white printing) in Figure 2). The second situation is that the projection area of a visible polygon may increase greatly because of the view position change, which leads to a great texture magnification, resulting in a fuzzy image. To guarantee the rendering quality of each visible polygon, we calculate its *texture fidelity*, a parameter factoring the texture magnification. If the calculated fidelity value is less than a given threshold, we render it in the special color as well. Therefore, after the texture mapping, all pixels having the special color indicate holes. As the final operation, we scan the frame buffer and cast new rays to fill these holes.

Errors are introduced in this procedure. Depicted in Figure 3, black rays of keyview (v_1) are used to approximate blue rays in novel view (v_2). Apparently, two rays of two views hitting the same point traverse through different directions and with different depths, leading to errors. However, if (1) two viewpoints are close and (2) all rays do not traverse too far, the error caused by this approximation will be invisible. The first condition is usually the case during a smooth navigation; the second condition generally holds for Levoy's surface volume rendering, where rays traverse only a short distance before they reach the full opacity. For more general volume rendering case, the second condition is not valid anymore, and we could break the volume into slabs as in Mueller et al.'s and Brady et al.'s methods to ensure the second condition holds within each slab. In this paper, we will only emphasize on surface volume rendering.

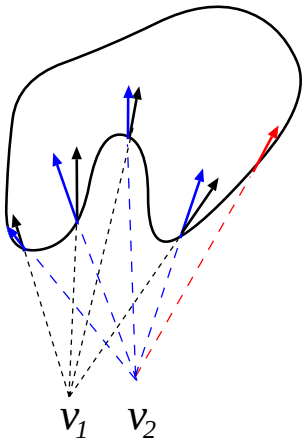
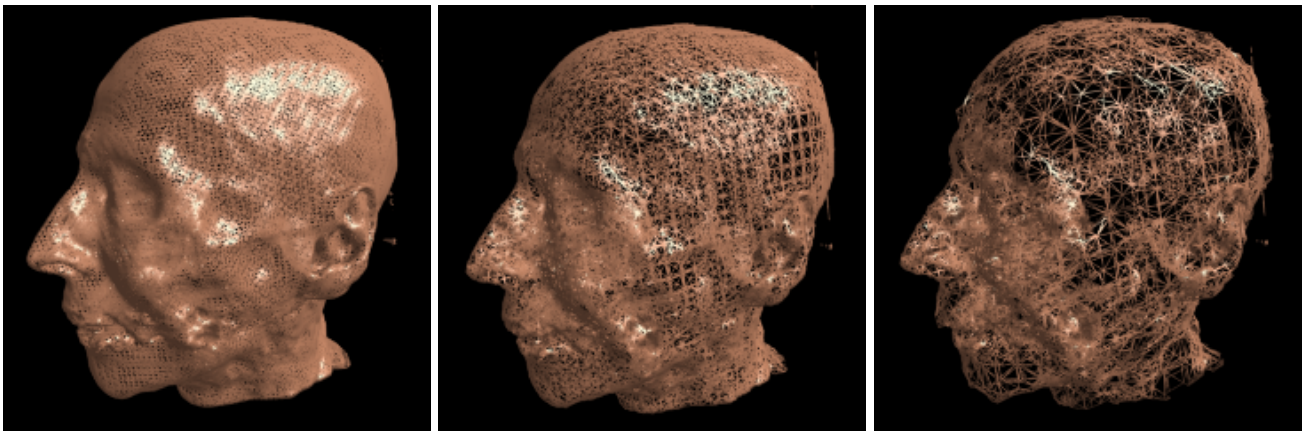


Figure 3: *Black rays (dotted) in keyview (v_1) are used to approximate blue rays (dashed) in novel view (v_2). The red ray (rightmost) is the new ray shot in the novel view.*



(a) (b) (c)

Figure 4: Different levels of an iso-surface of a CT head ($128 \times 128 \times 113$): (a) high level (original) mesh (361,528 triangles), (b) middle level mesh (168,708 triangles), (c) low level mesh (81,246 triangles).

3 GEOMETRY MODEL

One issue of constructing a geometry model is its accuracy, on which the texture mapping quality depends. We have some opposing considerations. A fine detailed geometry will maintain the object topology, especially the silhouette, but is slow to render; while a simplified model enhances the speed of polygon rendering, but is less accurate. The conclusion is that we need to construct a geometry using certain level-of-detail to balance between these issues. Our technique is open to any volumetric surface extraction and simplification method, except that a crack-free method is desired, such as methods in [25, 32]. If a crack appears in a polygonal model, it may end up a hole in the final image. Here, we have extended the method by Zhou et al. [32] to create a crack-free LOD mesh. First, we represent a volume using an implicit tetrahedra subdivision and then fuse the adjacent tetrahedra together. This fusion is based on a user specified error threshold, which defines the maximum deviation of the resulting iso-point after the fusion. Care is taken in the fusion stage so that the generated tetrahedra representation maintains a smooth transition between levels. After that, an iso-surface is extracted from the tetrahedra approximation, which is automatically simplified, smoothened and crack-free. Figure 4 shows three different levles of detail meshes (controled by different error thresholds). As we can see, the number of triangles varies greatly (from 361K to 81K) but the topology is preserved. The middle level mesh is the one that we have used in our implementation in Section 6. It gives us an appealing balance between overall performance and the generated image quality.

4 TEXTURE MAPPING NOVEL VIEWS

Texture mapping of a keyview could be achieved using hardware supported projective texture mapping in OpenGL. However, we discard this approach because its current hardware implementation does not test for the visibility of polygons. Occluded polygons in the keyview are mapped with wrong textures of the front occluding polygons when they become visible in the novel view. Instead, we seek to detect the polygon visibility on our own. On the other hand, to perform texture mapping we will have to compute the 2D texture coordinates for each vertex. Since this computation involves projection of each vertex onto the screen, we can automatically obtain the depth value, hence the visibility test for each vertex becomes free. We further propose a scheme to perform visibility test of a triangle based only on the visibility of the triangle's three vertices. As we will see from the following, our calculation of the visibility and the texture fidelity of a triangle is independent of the novel view, we therefore calculate them only once and store the result together with each keyview.

In Figure 5, all three vertices $1, 2$ and 3 of the shaded triangle are first projected to the keyview v_1 using its projection matrix M_1 to obtain their screen coordinates t_1, t_2 and t_3, which are used as texture coordinates. The triangle is then re-projected using the novel view projection matrix M_2 during the texture mapping. To ensure projective-correct texture mapping, we use homogeneous coordinates for texture coordinates. This has been discussed in detail in [10].

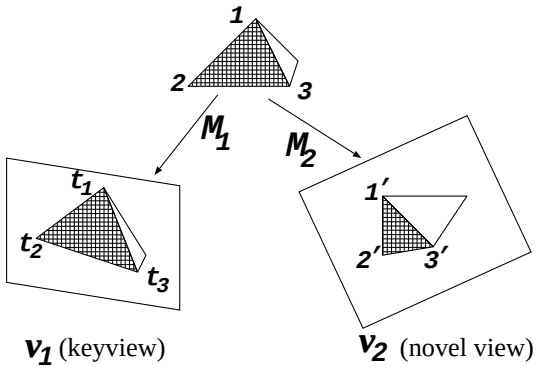


Figure 5: *Texture mapping.*

When assigning texture coordinates for each triangle, we have to determine whether the keyview is a valid texture source. First, we test whether the triangle is visible in the keyview. If yes, we further evaluate its texture fidelity and test whether it is greater than a certain threshold. Only succeeding these two tests, will the triangle be texture mapped with the keyview image; otherwise, it is drawn in the special color indicating a potential hole. Notice that most of the previously invisible triangles remain invisible in the novel view; therefore, holes only occupy a small portion of the novel frame.

In OpenGL, drawing a triangle using a uniform color requires flat shading. Intuitively, we need to switch the shading mode between flat shading and texture mapping according to the visibility of the triangle. This mode switching reduces the efficiency of triangle drawing. One candidate solution is to organize the whole trian-

gle list into a visible and an invisible list. We render these two lists successively and switch the rendering mode only once. However, this is not an appealing approach because whenever the keyview changes, we have to reorganize the lists. Instead, we adopt a simpler approach. Before we load a keyview into the texture memory, we overwrite the texel at $(0, 0)$ (the origin of the image) with the special color. For an invisible triangle, we specify all its three vertices with the same texture coordinates $(0, 0)$. The triangle is thus uniformly filled with the special color.

The visibility of a polygon in the keyview falls into four categories: (1) fully visible; (2) fully occluded or viewing frustum culled; (3) partially occluded; (4) partially viewing frustum clipped. During the texture mapping, a partially occluded triangle is simply drawn in the special color without using the texture corresponding to the visible part, similar to Chen et al. [4]. This is because achieving that would require visibility clipping on the triangle — subdividing the triangle into visible and invisible children triangles — as described by Debevec et al. [11]. Performing this visibility clipping on triangles is not only expensive, but also generates too many small triangles and has to be performed for every keyview. Nevertheless, if a triangle is partially clipped by the viewing frustum, we desire to make use of the partially visible texture. This becomes very important for virtual colon navigation, where the camera is very close to the colon wall, such that these clipped triangles are usually the closest to the camera, having large projections on the screen. Merely invalidating texture mapping for the whole triangle leads to big holes. Our solution is to overwrite the border of the texture image as the special color and specify the texture mapping parameter as CLAMP in the calling function *glTexParameter()*. The viewing frustum clipped part of a triangle has texture coordinates outside the range [0,1], which is clamped to the border color (now special color); while the non-clipped part is still correctly texture mapped.

To accurately detect the visibility, we need to scan-convert the whole triangle, and for each pixel inside, we determine its visibility, the same as z-buffer testing. This is obviously too expensive to perform. Instead, we only detect its three vertices. Only when all three vertices are visible is the triangle considered visible. See Figure 6, where triangles 3, 4, and 5 both have a vertex occluded and are treated as invisible. But this detection fails for triangle 2, where it is partially occluded but none of its vertices is, such that it is erroneously determined as visible. Fortunately, our experiments show that this is very rare in a real data set, with only a few occurrences during a long navigation.

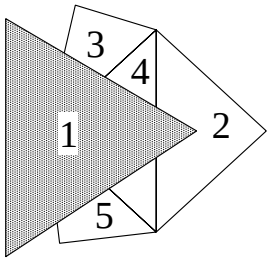


Figure 6: *Visibility detection.*

For each keyview, we generate and store its *z*-buffer for visibility detection. A vertex is projected onto the keyview projection plane and its depth is compared with the corresponding *z* value in the *z*-buffer. The *z*-buffer for a keyview can be generated in two ways: as a by-product of ray casting by detecting the iso-point when sampling along the ray, or by rendering the underlying geometry at the keyview camera position and reading out the *z*-buffer. We adopt the latter approach since the depth buffer can be utilized to accelerate ray casting by quickly skipping empty space (space leaping).

One issue for using the *z*-buffer is that a vertex is usually projected to a non-grid point on the keyview screen. Using a *z*-buffer value at the closest grid position does not always give us correct visibility because that *z* value can represent the neighboring triangle. Interpolating among neighboring *z* values is also inappropriate because they can represent disconnected objects. To better solve this problem, we evaluate the following equation:

$$|Z_{vertex} - Z_{buffer}| < \epsilon \quad (1)$$

where Z_{vertex} is the calculated *z* value of the vertex, Z_{buffer} is the *z* buffer value at the closet grid point, and ϵ is the specified "thickness" of the visible surface. As long as Equation 1 is true, the vertex is visible. Yet, this evaluation may appear too conservative for certain situations. For example, when a visible plane is almost perpendicular to the viewing plane, two distant points on the plane projecting to a similar screen location may have a great z difference. The above evaluation may mistakenly detect one vertex as invisible. We argue that in this situation planes project to a very small area, such that even we falsely take the area as a hole the extra ray-casting effort is little.

To detect whether a triangle is clipped by the viewing frustum in the keyview, we check its 2D vertex projections on the keyview plane. If any of them is beyond the clipping window, it indicates that the triangle is partially clipped.

When a triangle is detected as visible in the keyview, we need to further evaluate its texture fidelity. Here, our metric is the texture magnification, or scaling factor as in [7]. While we could use the same scheme as in [7] to compute the scaling factor for each triangle, instead, we use a static and faster, but more conservative approximation. We calculate the angle between the keyview direction and the triangle normal, which can be approximated by a simple dot product. The larger the angle, the lower the texture fidelity.

5 RENDERING KEYVIEWS

Keyviews are fully volume rendered, either on-the-fly or during preprocessing. Because keyviews are used to generate novel views, we can not support a view dependent specular lighting model in keyview rendering. Instead, we employ a purely diffuse lighting model. This is a common choice for most of the IBR methods. Further, during the texture mapping, we use DECAL mode in calling the *glTexEnvf()* function to avoid superimposing any lighting effect.

To render the keyview on-the-fly, one issue is when to render it. When the novel view moves away from the keyview, more invisible polygons get exposed, resulting in larger holes; therefore more efforts is required for ray-casting. For every novel view, we evaluate the *hole area percentage*; that is, the ratio of the number of new rays to the area of object occupancy on screen. When the hole area percentage of a novel view is beyond a given threshold, we decide to render a new keyview.

Alternatively, we can pre-render a group of keyviews during the preprocessing so that any novel view can be generated from one (or a subset) of them. Thus the system never has to ray-cast the whole volume during the navigation. In contrast to previous image-based rendering methods [12, 20], the number of keyviews can be greatly reduced, since we take advantage of a full underlying 3D geometry (in Lumigraph, only a coarse 3D model is used). The major issue in pre-rendering is where to place keyview cameras. It remains an open problem to determine for a given 3D model, the minimal set of keyframe images, such that for all novel viewpoints, a correct image can be obtained by combining the keyframe images [27]. This statement also applies to volume representation. However, full coverage of the model by keyviews is not our goal, since we can detect holes and fill them by casting new rays on the fly.

There are different strategies for placing keyview cameras, depending on the navigation style. Figure 7 illustrates the keyview

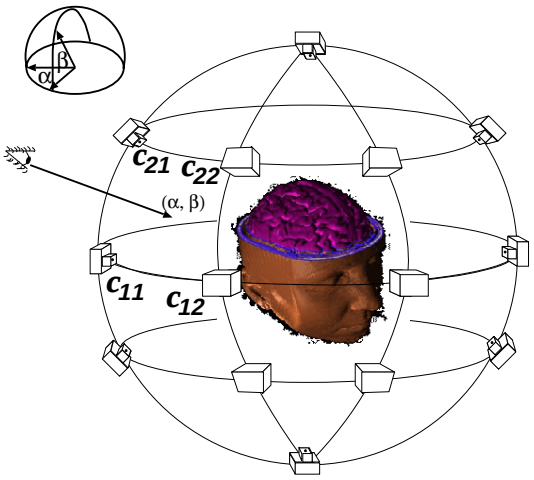


Figure 7: *Distribution of keyviews around the object.*

Figure 8: *Placing keyview cameras along the skeleton. The line is the smoothened skeleton line, and the dots represent the camera locations.*

camera configuration when examining an object from the outside. Here we place a sphere around the volume, and the cameras evenly on the sphere with cameras all pointing at the center of the object. Given the novel view (a camera position and view direction) shown in Figure 7, the view vector intersects the sphere at point (α, β). There are several ways of utilizing the surrounding keyviews, c_{11}, c_{12}, c_{21} or c_{22} to generate the novel view (α, β). One option is to render the object multiple times using each keyview in turn and blending the results. The blending method not only creates too fuzzy an image, but also is very expensive to perform. We determine, instead, to render only once by choosing the closest keyview for each polygon. We determine the closest keyview for each triangle by comparing the angle between the polygon normal and the view vector of each surrounding keyview. In our implementation, we only test against the four neighboring keyviews for efficiency.

The virtual colon fly-through has a different navigation style. A critical issue is where to place keyviews so that we can most appropriately capture the entire colon. There are different ways to specify keyviews along the center line: either interactively specify them during the initial navigation, or automatically assign keyviews based on the curvature of the center line. The first approach is very tedious for the user. We have adopted the second approach. We place more cameras where the curvature of the center line is high. In addition, we limit the maximum distance between two neighboring cameras to guarantee better coverage for a long straight path. One issue is that the provided center line is defined on the grid point, so it has too much local noise. It is inaccurate to calculate a curvature from the local values. Our solution is to first apply a low pass filter to smoothen the curve and then calculate the curvature from the local points. For low-pass filtering, we convolve the center line twice with a box filter of width 3. Figure 8 shows the result of camera placement. Because of the 2D projections, it does do not reflect the overall curvature in 3D.

6 IMPLEMENTATIONS AND RESULTS

We have implemented our framework on an SGI Challenge equipped with an Infinite Reality engine, 3GB memory, and 16 R10000 processors. However, in our experiments, only one R10000 processor is used. We have applied our algorithm to different data sets: a CT head ($128 \times 128 \times 113$), a CT scan of a plastic pipe ($512 \times 512 \times 107$), and a patient's colon ($512 \times 512 \times 361$). The plastic pipe has a radius of 20mm. To simulate colonic polyps, three small rubber objects of size 7mm, 5mm, and 3mm are attached to the inner surface of the pipe.

We have experimented with accelerating two kinds of volume navigations: examining the CT head from the outside, and flying through the pipe/colon. Renderings on a CT head create images of 256×256 resolution, while 512×512 resolution images are created for pipe and colon data sets. The ray-casting method we are to compare against is the optimized version as in [28]. The same ray-casting routine is used in our hole-filling procedure to guarantee a fair comparison. We have used perspective projection for all the rendering.

Our results have shown that our method can deliver high speedup while maintaining image quality. Because our image-based volume rendering is divided into two steps, polygon rendering and ray-casting, the rendering speed has to do with two facts: the complexity of the polygon model and the transfer function ramp, which determines how far a ray traverses the data before it reaches full opacity. We have used a moderately simplified surface model as in Figure 4b for the CT head data (Figure 9); but a non-simplified surface model for both colon (Figure 10) and pipe (Figure 11) data. We used a linear ramp between 45 and 65 as the transfer function for the CT head, which means that the opacity is zero below 45, changes from zero to one (indicating full opacity) from 45 to 65 and remains one beyond 65 (maximum voxel value 255). For the plastic pipe data, we used a linear ramp between 30 and 80, while a linear ramp between 30 and 100 is used for the colon data. The iso-value of the surface is set to 45, 30 and 30 respectively for the three data sets for constructing iso-surfaces. Figure 9 – 11 shows that when rendering a novel view we only need to cast rays for a very small portion of the image. For all these data, we have obtained an average of a magnitude of speedup over the optimized volume rendering (all the timings listed in the captions are average timings). In the mean time, the filled pixels match well with the surrounding texture mapped pixels. Figure 9e justifies this by showing an intensity scaled-up difference image between a volume rendered novel view and our image-base rendered view. The calculated *Mean Square Errors* (MSE) of the image is 0.012.

We have successfully applied our algorithm to the virtual colonoscopy system. Our experiments were conducted on top of the original polygon-based prototype system. In virtual colonoscopy, image fidelity is considered a top priority. Therefore, the underlying geometry has not been simplified. Since both view-frustum culling

and occlusion culling, by taking advantage of the twisted colon feature, have been applied to the surface model during the rendering, the model can be rendered very efficiently. The navigation is performed in the same style as in the previous surface colonoscopy system [15]. Without clicking the mouse, the camera will automatically fly through the colon along the center line. A mouse clicking will impose a force to influence the moving speed and direction of the camera. The speedup obtained from our algorithm is substantial. Figure 12 illustrates the timing of an interactive pipe navigation when keyviews are rendered on the fly. The hole area percentage is set to 1.5%. 32 out of 467 frames are fully volume rendered and used as keyviews, yielding a keyview update ratio of 7.4%. The average rendering time of the keyviews is 1.9 seconds per frame, which includes the time for calculating the visibility map. For clarity, the keyview rendering time is clamped on the graph. The average rendering time for novel views is 0.17 seconds, indicating more than an order of magnitude speedup. The keyview update ratio depends on the user's navigation path and the underlying object structure. If the user makes a sudden turn, or more small features of the object emerge, it will demand the keyview be updated more frequently. Figure 12 also shows the timing of the two rendering steps: texture mapping and filling holes. The texture mapping time is dependent on the complexity of the underlying geometry within the current view frustum, with an average of 0.08 seconds, which includes the frame-buffer loading time of 0.02 seconds; time for filling holes is also view dependent, with an average of 0.11 seconds. The total time of each frame is the sum of two components. For the keyview, the texture mapping time component represents the total time of the polygon rendering (used for accelerating ray-casting) and calculation of the triangle visibility map. The average time for calculating a visibility map is 0.18 seconds per frame. This justifies the benefits of pre-calculating the visibility.

A delay in keyframe rendering causes jerky navigation. Therefore, we pre-render keyviews and calculate their visibility maps in advance. The keyviews are positioned according to the method described in Section 5. Figure 13 illustrates the rendering time of an interactive pipe navigation using 100 pre-rendered keyviews. The rendering is smoother than rendering a keyview on-the-fly. Nevertheless, the rendering time depends on the navigation path. Between frame 220 and 260, the rendering time increases abruptly; this is because the user has made a sudden maneuver in turning the camera to the colon wall. Since currently all keyviews are rendered with the camera pointing along the center line, a view facing the wall benefits little from the keyview, resulting in big holes to fill. After the user changes the camera back to following the center line direction, the rendering time drops again. Another time increase around frame 390 is caused by a similar camera turn. Similar experiments have been done on a patient's colon data set. Because of the complex colon wall, we have used 200 keyframes to capture most of the inside structure. The rendering time is captured in Figure 14. The average rendering time of our method is 0.33 seconds per frame, while the average rendering time for volume rendering is 3.6 seconds.

We also evaluated how rendering benefits from increasing the number of keyframes for the same object. The rendering time of a CT head navigation with a pre-defined path using a different number of pre-rendered keyviews (6, 14 and 26) is plotted in Figure 15. Keyviews are placed according to Section 5. For a 6 keyview configuration, two keyviews are placed on the north and south poles of the sphere, and the remaining 4 are evenly distributed on a 0 degree latitude. For a 14 keyview configuration, we add another two latitudes, and again another two for 26 keyviews. The navigation rotates the CT head around x, y and z axes consecutively. The average rendering time for 6, 14, and 26 keyviews is 0.16, 0.11 and 0.09 seconds, respectively. When the number of keyviews increases, the rendering becomes not only faster, but also smoother. There is a limit in speed benefit when increasing the number of keyviews. For 26 keyviews, the average percentage of new rays to the whole image is 1.0%. A further increase in the number of keyviews does not improve the performance significantly.

7 CONCLUSIONS AND FUTURE WORKS

We have presented an image-based rendering framework for accelerating surface volume rendering. We have provided alternative implementation designs and applied them to different volume data sets and different types of navigation. Its successful application to the virtual colonoscopy system has clearly proven the effectiveness of our approach. Our experiments have demonstrated an order of magnitude speedup over the existing optimized volume rendering while delivering visually indistinguishable image quality. The rendering speed and the image quality can be tuned by employing different levels of detail of the underlying surface geometry. Even though we have done our experiments on workstations, a modern PC could be sufficient for delivering similar performance using our method. Today's PC equipped with the commodity graphics card can meet our computation requirement.

When rendering keyview on-the-fly, our most important future work is to address the variable latency caused by rendering the keyframes. One on-going work is to predict where the next keyview position will be and to amortize the keyview rendering among the successive novel view rendering. Another future experiment is to cache the previous keyviews for reuse. This caching will especially benefit the situation when the user moves back and forth to examine a particular part of the object. To alleviate the time increase when the camera turns toward the colon wall, one candidate solution is to generate multiple keyviews at each selected keyview position. Besides generating keyviews pointing along the center line, we also generate keyviews pointing to the wall; for example, four views looking left, right, up and down. We are currently experimenting with this approach.

In the current implementation, we have emphasized on applying our technique to accelerate volume rendering with narrow transfer function ramp. As has been briefly mentioned in Section 2, when the transfer function has a wider ramp (more translucent volume), we should cut the volume into slabs parallel to the viewing plane and then composite the slab images. An immediate issue to be addressed is how to efficiently perform this view dependent cutting on a surface model. We would also like to know: (1) how thick should the slabs be so that we can have a balance on the performance and the image quality, and (2) do we always have to render the whole slabs or can we still perform early ray termination, and how. Finally, a general issue worth addressing is to design more sophisticated error metrics for the on-the-fly evaluation.

One limitation of our method is that whenever the transfer function changes, we have to reconstruct the mesh. This may become a bottleneck of our technique. It becomes critical to design a more efficient surface construction method. Another issue related to the surface model is its silhouette, which is critical to the image quality [26]. We are currently investigating techniques to improve the silhouette of the model while conducting a more aggressive simplification for the inside.

8 ACKNOWLEDGMENTS

This work has been supported by ONR grant N00014011034, NIH grant #CA82402, and Viatronix Inc. The first author would also like to acknowledge Grant-in-Aid of Research, Artistry and Scholarship from the Office of the Vice President for Research and Dean of the Graduate School of the University of Minnesota. We wish to thank Ming Wan for providing us the ray-casting codes and

Huamin Qu for his valuable help on the implementation of the virtual colonoscopy system. Thanks to Dongrong Xu for his evaluation work and Klaus Mueller and Andreas Baerentzen for their review and comments. The pipe and patient data sets were provided by Stony Brook University Hospital.

References

[1] R. Avila, L. Sobierajski, and A. Kaufman. Towards a comprehensive volume visualization system. *Proc. of IEEE Visualization'92*, pages 13–20, Oct. 1992.

[2] M. Brady, K. Jung, H. Nguyen, and T. Nguyen. Two-Phase Perspective Ray Casting for Interactive Volume Navigation. In *Proceedings of Visualization '97*, pages 183–189, Oct. 1997.

[3] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *1994 Symposium on Volume Visualization*, pages 91–98, Oct. 1994.

[4] B. Chen, J. E. Swan, E. Kuo, and A. Kaufman. LOD-Sprite technique for accelerated terrain rendering. *Proc. of IEEE Visualization'99*, pages 291–298, Oct. 1999.

[5] J. J. Choi and Y. G. Shin. Efficient image-based rendering of volume data. In *Proceedings of Pacific Graphics'98*, pages 70–78, Oct. 1998.

[6] D. Cohen and Z. Shefer. Proximity clouds - an acceleration technique for 3D grid traversal. Technical Report FC 93-01, Department of Mathematics and Computer Science, Ben Gurion University of the Negev, Feb. 1993.

[7] D. Cohen-Or, Y. Mann, and S. Fleishman. Deep compression for streaming texture intensive animations. *Computer Graphics*, 33(Annual Conference Series):261–268, 1999.

[8] F. Dachille, K. Kreeger, B. Chen, I. Bitter, and A. Kaufman. High-quality volume rendering using texture mapping hardware. *SIGGRAPH/Eurographics Workshop on Graphics Hardware Proceedings*, pages 69–76, Aug. 1998.

[9] W. J. Dally, L. McMillan, G. Bishop, and H. Fuchs. The delta tree: An object-centered approach to image-based rendering. Technical memo 1604, MIT AI Lab, May 1996.

[10] L. Darsa, B. C. Silva, and A. Varshney. Navigating static environments using image-space simplification and morphing. *1997 Symposium on Interactive 3D Graphics*, pages 25–34, April 1997.

[11] P. E. Debevec, Y. Yu, and G. Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. *Rendering Techniques'98*, pages 105–116, 1998.

[12] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. *Computer Graphics Proceedings*, pages 43–54, 1996.

[13] B. Gudmundsson and M. Randen. Incremental generation of projections of ct-volumes. In *the first conference on biomedical computing*, pages 27–34, May 1990.

[14] T. He and A. Kaufman. Fast stereo volume rendering. *Proc. of IEEE Visualization'96*, pages 49–56, Oct. 1996.

[15] L. Hong, S. Muraki, A. Kaufman, D. Bartz, and T. He. Virtual voyage: Interactive navigation in the human colon. *Computer Graphics Proceedings*, pages 27–34, Aug. 1997.

[16] A. Kaufman, F. Dachille, B. Chen, I. Bitter, K. Kreeger, N. Zhang, and Q. Tang. Real-time volume rendering. *International Journal of Imaging Systems and Technology*, 2000.

[17] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transform. *Computer Graphics Proceedings*, pages 451–457, July 1994.

[18] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, 8(5):29–37, May 1988.

[19] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[20] M. Levoy and P. Hanrahan. Light field rendering. In *Computer Graphics Proceedings*, pages 31–42, 1996.

[21] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. *Computer Graphics Proceedings*, pages 39–46, Aug. 1995.

[22] K. Mueller, N. Shareef, J. Huang, and R. Crawfis. Ibr-assisted volume rendering. In *Late Breaking Hot Topics of Visualization'99*, pages 5–9, Oct. 1999.

[23] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P. Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings of Visualization '98*, pages 233–238, Oct. 1998.

[24] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro real-time ray-casting system. *Computer Graphics Proceedings*, pages 251–260, 1999.

[25] T. Roxborough and G. M. Nielson. Tetrahedron based, least squares, progressive volume models with application to free-hand ultrasound data. *Proc. of IEEE Visualization 2000*, pages 93–100, Oct. 2000.

[26] P. V. Sander, X. Gu, S. J. Gortler, H. Hoppe, and J. Snyder. Silhouette clipping. In K. Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 327–334. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[27] S. Teller and D. Dobkin. *CRC Handbook of Discrete and Computational Geometry*, chapter Computer Graphics. CRC Press, 1997.

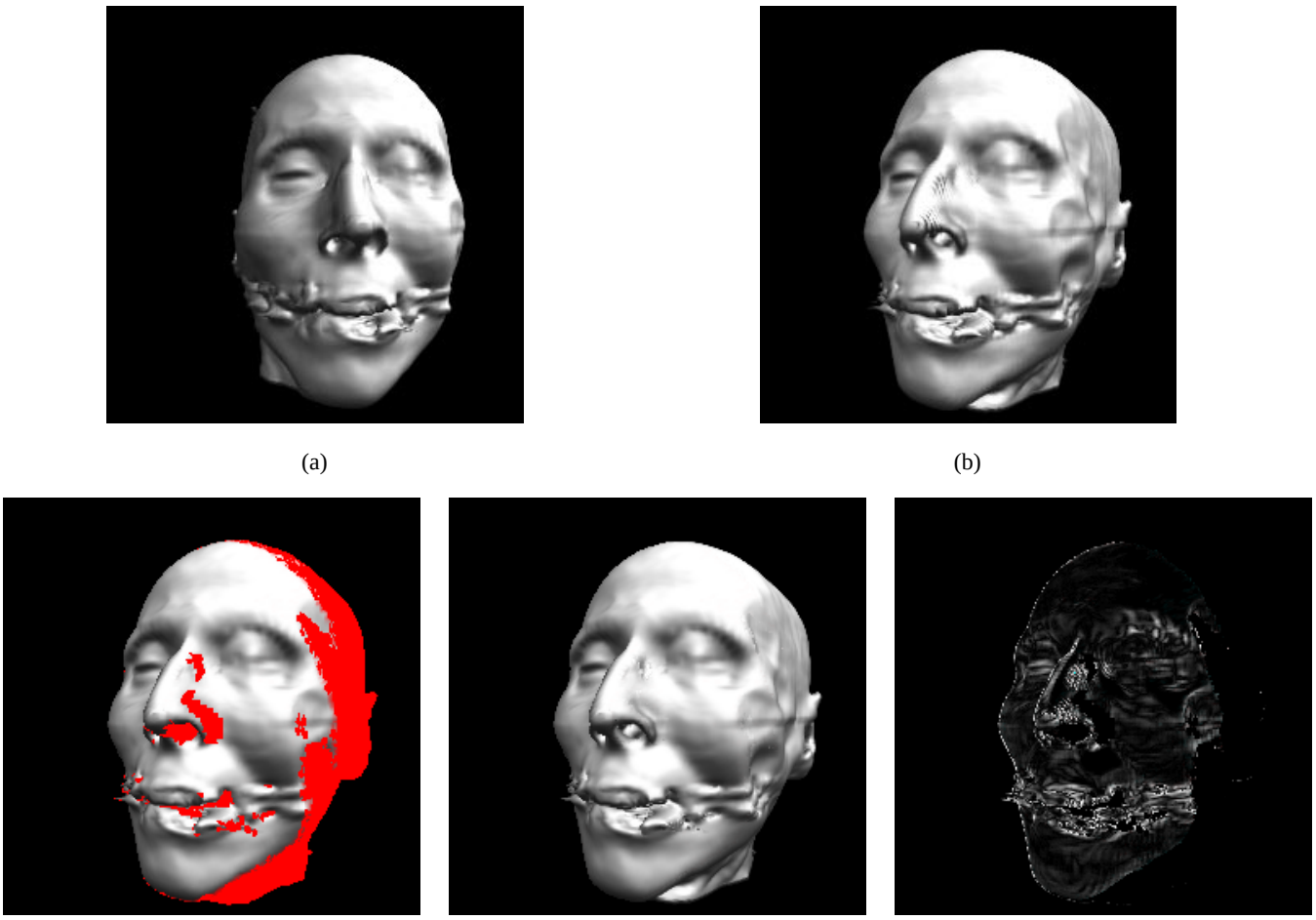
[28] M. Wan, Q. Tang, A. Kaufman, Z. Liang, and M. Wax. Volume rendering based interactive navigation within the human colon. *Proc. of Visualization'99*, pages 397–400, Oct. 1999.

[29] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH Conference proceedings*, pages 169–178, July 1998.

[30] R. Yagel and Z. Shi. Accelerating volume animation by space-leaping. In *Proceedings of Visualization '93*, pages 62–69, Oct. 1993.

[31] S. You, L. Hong, M. Wan, K. Junyaprasert, A. Kaufman, S. Muraki, Y. Zhou, M. Wax, and Z. Liang. Interactive volume rendering for virtual colonoscopy. *Proc. of IEEE Visualization'97*, pages 433–346, Nov. 1997.

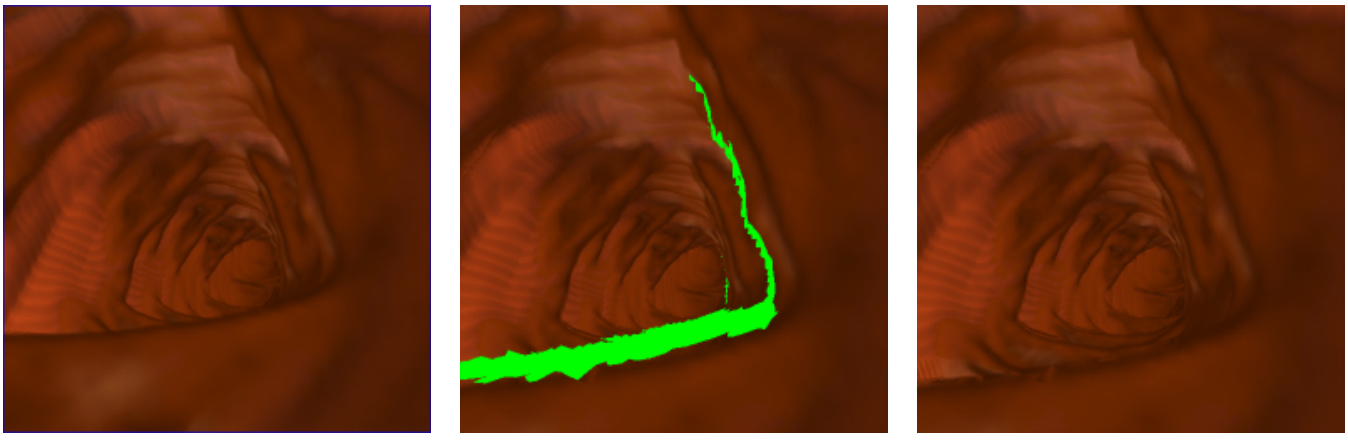
[32] Y. Zhou, B. Chen, and A. Kaufman. Multiresolution tetrahedral framework for visualizing regular volume data. *Proc. of IEEE Visualization 97*, pages 135–142, Oct. 1997.



(a) (b)

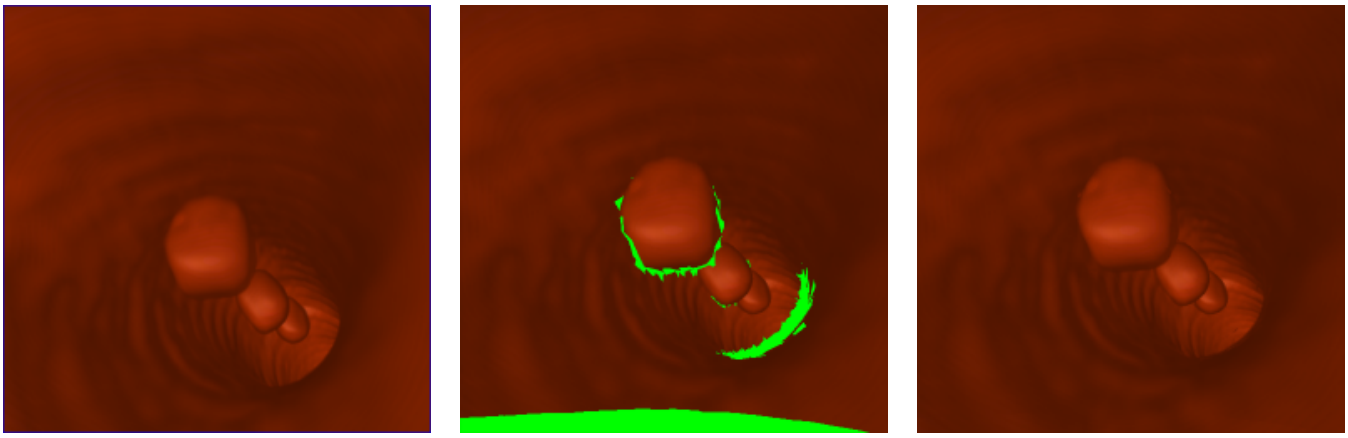
(c) (d) (e)

Figure 9: Two views during an interactive CT head navigation (image resolution: 256×256): (a) fully volume rendered keyview, (b) fully volume rendered novel view (rotated 20 degrees, 3.5 secs), (c) the same novel view after texture mapping the image (a) onto the surface model in Figure 4b with unfilled holes (colored red – see color in color plate, 0.07 secs), (d) after filling the holes in (c) (c+d: 0.16 secs), (e) intensity-scaled-up difference image between (b) and (d) (scale factor: 5, MSE: 0.012).



(a) (b) (c)

Figure 10: Two views during an interactive colon navigation (image resolution: 512×512): (a) fully volume rendered keyview (3.6 secs), (b) texture mapped novel view with unfilled holes (colored green — lightly shaded on black-and-white printing, 0.09 secs), (c) after filling the holes in (b) (b+c: 0.3 secs).



(a) (b) (c)

Figure 11: Two views during an interactive pipe navigation (image resolution: 512×512): (a) fully volume rendered keyview (1.9 secs), (b) texture mapped novel view with unfilled holes (colored green — lightly shaded on black-and-white printing, 0.08 seconds), (c) after filling the holes in (b) (b+c: 0.17 secs).

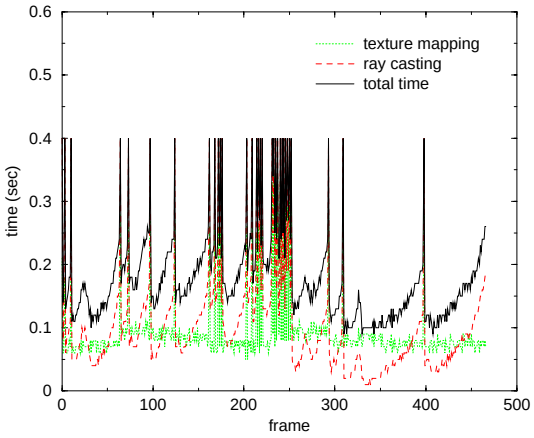


Figure 12: *The rendering time of an interactive pipe navigation using on-the-fly keyview rendering (also in color plate).*

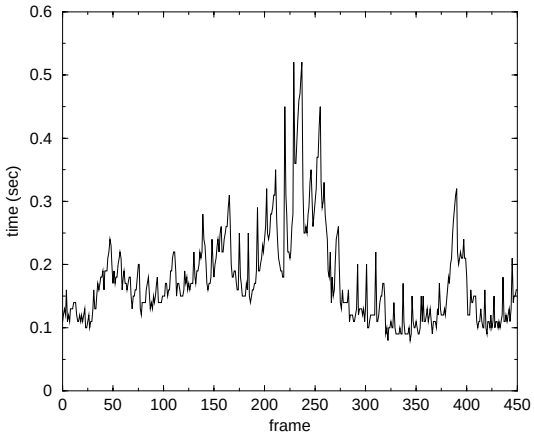


Figure 13: *The rendering time of an interactive pipe navigation using 100 pre-rendered keyviews.*

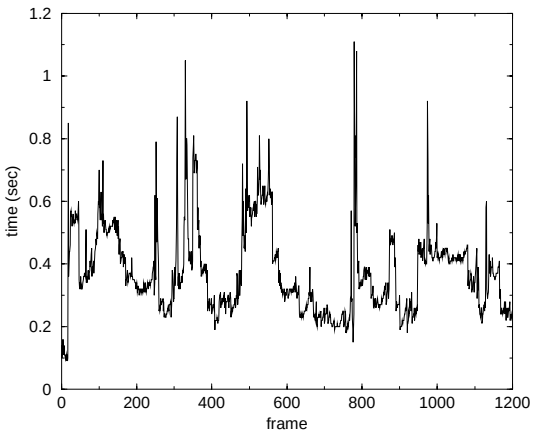


Figure 14: *The rendering time of an interactive colon navigation using 200 pre-rendered keyviews.*

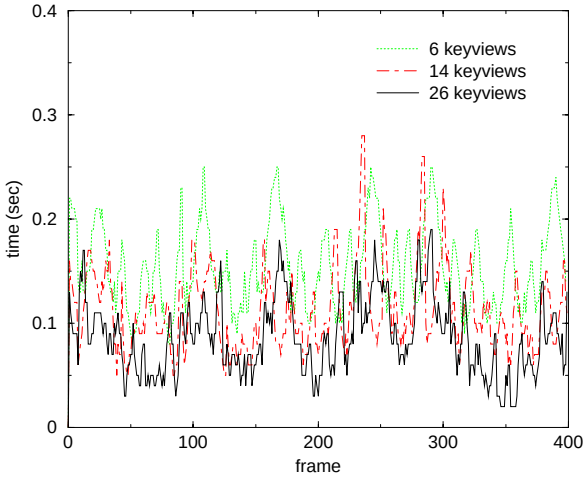


Figure 15: *The rendering time of an interactive CT head navigation using different number of pre-rendered keyviews (also in color plate).*