Fast And Reliable Space Leaping For Interactive Volume Rendering

Ming Wan [1]

The Boeing Company
P.O. Box 3707, MC 7L-40, Seattle, WA 98124-2207

Aamir Sadiq [2] and Arie Kaufman [2]

Department of Computer Science
State University of New York at Stony Brook, Stony Brook, NY 11794-4400

Abstract

We present a fast and reliable space-leaping scheme to accelerate ray casting during interactive navigation in a complex volumetric scene, where we combine innovative space-leaping techniques in a number of ways. First, we derive most of the pixel depths at the current frame by exploiting the temporal coherence during navigation, where we employ a novel fast cell-based reprojection scheme that is more reliable than the traditional intersection-point based reprojection. Next, we exploit the object space coherence to quickly detect the remaining pixel depths, by using a pre-computed accurate distance field that stores the Euclidean distance from each empty (background) voxel toward its nearest object boundary. In addition, we propose an effective solution to the challenging new-incoming-objects problem during navigation. Our algorithm has been implemented on a 16-processor SGI Power Challenge and reached interactive rendering rates at more than 10 Hz during the navigation inside 512^3 volume data sets acquired from both a simulation phantom and actual patients.

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

Additional Keywords: Virtual navigation, volume visualization, ray-casting optimization, and space leaping.

1. INTRODUCTION

Various fast rendering techniques are available now to generate on-the-fly navigation frames of 3D volumetric environments in real time, which includes both geometric rendering and direct volume rendering. The former first converts the volume data set into a set of polygonal isosurfaces and subsequently renders the polygons by using graphics hardware accelerators, while the later directly renders the volume data set without explicit extraction of geometric primitives. Because of the high speed obtained from graphics hardware, geometric rendering has been widely employed [9]. However, compared to direct volume rendering, it has shown several weaknesses in visualizing volumetric objects. First, extra effort is needed to convert voxel-based object representation to polygon-based representation in a preprocessing stage. Second, it only displays 2D iso-surfaces explicitly extracted from the object boundary, taking the risk of losing the 3D information beyond 2D surfaces. Third, when the camera moves close to the object boundary, aliasing (e.g., the silhouettes of polygons) may appear. Moreover, the complexity of the extracted polygonal mesh can overwhelm the capabilities of the graphics hardware. Therefore, a great deal of research has also been conducted on high-performance direct volume rendering.

Popular direct volume-rendering techniques include pure software volume rendering (such as volumetric ray casting, volume splatting [15, 27], and shear-warp factorization approach [11]), texture-based volume rendering [4, 10, 7], special-purpose real-time volume rendering hardware [17], and image-based volume rendering [18]. Among these techniques ray casting has seen the largest body of publications. Although it was originally a high-accuracy but time-consuming method [12], ray casting has been highly optimized to be among the fastest volume rendering techniques [16, 23]. Our goal in this paper is to improve the performance and quality of an effective ray-casting optimization, called *space leaping*, by fully exploiting spatiotemporal coherence between adjacent frames during interactive navigation.

The concept of *space leaping* was introduced by Yagel and Shi [28], which exploits the *temporal coherence* when rendering a sequence of ray-casting images in order to rapidly traverse or skip over the background voxels that have no contribution to the rendered image. Their basic idea is to obtain the depth value of each image pixel at the current frame by reprojecting the first intersection point along each ray in the previous frame. Because of the noninteger reprojection on the image plane, a reprojected point rarely coincides with the integer pixel grid and thus its depth value is assigned to the nearest pixel. If a pixel is not covered by any reprojected point, a normal full-range ray casting has to be performed at this pixel (we call it a *hole pixel*).

Note that the idea of exploiting the temporal coherence between neighboring frames has also been explored in other rendering approaches. In traditional geometric rendering, for example, Adelson and Hodges [2] employed a similar strategy to that in [28], to obtain the depth information of a new image from the previous image using reprojection. However, their method was designed for ray tracing on solid geometric models and worked for convex objects only. The space-leaping-accelerated ray casting [28] is also different from image-based rendering (IBR) [6, 14, 20], IBR-assisted ray casting [5, 18], or stereo-rendering [1, 25] approaches. The former only reprojects intersection points to build a depth buffer rather than directly generating the final image. Therefore, it can conveniently render high-quality realistic images of a complex volumetric environment with transparent views and view-dependent shading.

[1] ming.wan@boeing.com

[2] {aamir, ari}@cs.sunysb.edu

IEEE Visualization 2002 Oct. 27 - Nov. 1, 2002, Boston, MA, USA
0-7803-7498-3/02/$17.00 © 2002 IEEE

However, we found several problems when we employed Yagel and Shi's space leaping method to accelerate the ray-casting algorithm in practical navigation systems. First, we found that because the noninteger reprojections were rounded onto the image grid, the reliability of the depth value at each pixel was affected and also the number of hole pixels increased. The former might affect the image quality, while the later might increase the rendering time. Second, because of the lack of depth values at hole pixels, the time-consuming full-range ray casting had to be performed from the beginning of those rays. Third, in our large-scale virtual navigation systems, usually only part of the entire volume scene was visible at each frame. When a previously invisible part of an exposed object or a new-incoming object appeared in the current frame, the renderer might skip (part of) it.

To provide a fast yet more reliable space leaping for volumetric ray casting in practical virtual navigation systems, we present a new ray-casting optimization by improving and extending the existing space-leaping ideas in a number of ways.

First, we present a novel fast cell-based reprojection scheme to replace the previous point-based schemes [28], so that we avoid the trouble caused by the noninteger reprojection and reduce the number of uncovered pixels as well. One should be aware, however, that the noninteger reprojection problem in [28] is not unique to volumetric space leaping but could potentially occur for any point-based reprojection [2] or 3D warping [14]. Some researchers proposed to solve this problem by using expensive volume splatting [27] or A-buffer techniques [18, 25]. Our cell-based reprojection scheme is provable to be cheaper and therefore faster compared to those techniques, without skipping over any object voxels that contribute to the rendered image.

Second, by exploiting the object space coherence in the volumetric scene, we devise a simple but fast scheme, called *Distance-From-Boundary (DFB) jumping*, to rapidly determine the depth values of the hole pixels. Our rapid DFB-jumping is based on a pre-computed accurate distance field that stores the Euclidean distance from each empty (background) voxel toward its nearest object boundary. Unlike other kinds of presence acceleration techniques that traverse a hierarchical data structure, such as octrees [13, 16] and K-d trees [22] to skip over empty regions, our DFB-jumping directly and hence more quickly traverses the original regular grid. An early version of our fast DFB-jumping method [24] has not considered the combination with the temporal-coherence based space leaping.

Third, we further propose a novel efficient and reliable detection scheme to solve the new-incoming-objects problem during navigation in a complex volumetric scene. This problem has challenged all fast-rendering algorithms that exploit the temporal coherence using either reprojection [2, 28] or 3D warping [14]. Our scheme automatically updates the depth values of those pixels where the new incoming objects are exposed so that these objects are not skipped over during ray casting.

Although we have conducted our experiments on medical related data, our algorithm is generic. It can apply to a wide range of volume visualization applications, especially those high-fidelity ones in the life science and industry, wherever the volumetric data model is available and a *navigable region* exists inside the virtual environment, where we can navigate by manipulating the virtual camera. However, our algorithm would show its best performance in those volumetric scenes with a large amount of navigable (empty) space and high-opacity (near opaque) objects. By the way, because our algorithm essentially accelerates ray casting by reducing samplings along each ray, it is, unfortunately, not applicable to non-volume rendering (e.g., geometric rendering) where exact ray-object intersection is required.

This paper is organized as follows. Firstly, in Section 2, we define some general concepts and ideas to be used in this paper. Then, in Section 3, we present our fast and reliable cell-based reprojection scheme. Next, in Section 4, we describe our accelerated ray casting that exploits spatiotemporal coherence by combining cell-reprojection-based spacing leaping and fast DFB-jumping. Moreover, in Section 5, we propose our novel efficient detection scheme for new incoming objects during interactive navigation. Section 6 reports our primary experimental results on a simulation data set and more than 40 actual patient data sets.

2. GENERAL DEFINITIONS

The volumetric data sets rendered by our accelerated ray-casting algorithm with space leaping are typical 3D grids of voxels defined in volume visualization. We define a *cell* as the 3D volumetric region bounded by eight neighboring grid vertices (voxels), which is different from the *intersection point* (a point in object space) used in other publications (e.g., [2, 28]). Three different grids are defined according to the spacing between samples. If the grid has a uniform distance between adjacent grid vertices, it is a *Cartesian* or *cubic* grid. If it has a uniform spacing along the same axis but different ones among different axes, it is a *regular* grid. Otherwise, if the spacing changes along the same axis, it is a *rectilinear* grid. Our algorithm works on *all* these grids.

Our space leaping works on two adjacent navigation frames at a time — a previous frame and a current frame. Two different buffers, a *cell buffer* and a *depth buffer*, are defined during rendering. Both have the same size as the rendered image, and both are updated at each frame. Each element of the cell buffer stores the object cell (i.e., the cell of a visible object) firstly encountered by the ray emitted at each pixel. Note that this cell buffer is different from the *C-buffer* used by Yagel and Shi [28], because its elements are volumetric cells rather than intersection points. We further employ a depth buffer to store the depth of each pixel. In our algorithm, the cell buffer from the previous frame (in short, the previous cell buffer) is used to build the depth buffer in the current frame, and this depth buffer is then used to accelerate ray traversal in the current frame. The cell buffer is updated at each frame so that it can be used to build the depth buffer in the next frame. Although the previous cell buffer can be derived from its corresponding depth buffer during the rendering of the current image, we prefer to pre-store the cell information in a separate cell buffer to avoid such a computational cost.

As mentioned before, our new space-leaping scheme is aimed at building a reliable depth value for each pixel so that the empty regions along each ray can be rapidly skipped over according to its depth value. Accordingly, the accelerated ray-casting algorithm consists of four steps: (1) derive most of the pixel depths from the previous frame by using our cell-based reprojection (if the previous frame is not applicable, skip this step), (2) perform DFB-jumping at each hole pixel to rapidly detect its depth, (3) detect the correct depth values derived from Step 1 for those pixels where new incoming objects appear, and (4) skip over the empty

region along each ray during ray casting according to its depth value. In fact, DFB-jumping is also employed in Steps 3 and 4 for further speedups, as described in Sections 4 and 5, whenever empty regions appear during the ray traversal.

3. CELL-BASED REPROJECTION

To quickly determine the pixels occupied by each reprojected cell, we define a *bounding sphere* for each cell. It is centered at the cell center and its radius is half the length of the diagonal of the cell, which ensures that the entire cell is enclosed within the sphere. Note that such a bounding sphere works on all kinds of cubic, regular, and rectilinear grids, but in the case of a rectilinear grid, the size of the bounding sphere may differ from cell to cell.

During the reprojection, we project these bounding spheres rather than the exact cells so that we avoid the time-consuming 3D scan-conversion of the cell from its three front-facing facets [3]. Our projection scheme reprojects only the cell center and then rapidly estimates the reprojected region during a *perspective* projection (required by most virtual navigation systems).

Specifically, our cell-based reprojection from the previous cell buffer can be divided into three steps: (1) reproject each cell center onto the current image plane, (2) determine those pixels in the current image occupied by the projected bounding sphere of the cell, and (3) determine the depth values of the occupied pixels and update the corresponding entries in the current depth buffer, when necessary.

3.1 Reproject the Cell Center

Assume that E is the viewpoint at the current frame and f is the distance from E to the image plane, as illustrated in Figure 1a. Given cell C and its center point V, we can calculate the exact projected position P of V on the image plane as the intersection between the image plane and the line L passing through V and E. P may be located at a subpixel location in the current image, but we do not round it to the nearest pixel. We can also calculate the depth d of V as the distance between V and E.

3.2 Determine the Reprojected Region

We determine the projected region of the bounding sphere of cell C on the image plane in two different situations, as shown in Figures 1a and 1b. First, we consider a specific situation where line L is perpendicular to the image plane, as shown in Figure 1a. Obviously, the projected region of the bounding sphere of cell C is a *circle* centered at P with radius s. To calculate the radius s, we assume that there is a tangent plane T of the bounding sphere of cell C that is passing through viewpoint E, A is the tangent point, and B is the projection of point A on the image plane. Therefore, the distance between B and P is s, and the distance between A and V is the radius r of the bounding sphere, i.e., half the length of the diagonal of the cell in question. Evidently, E, P, and B form one right triangle, and E, A, and V form another right triangle. Because these two right triangles are similar triangles, we have

$$s = r \cdot f / \sqrt{d^2 - r^2} \tag{1}$$

To speed up the computation of s, we replace $(d^2 - r^2)$ in Equation 1 with $(d - \varepsilon)^2$, where ε is a small constant and $\varepsilon < d$. Therefore, we get a new radius s'

$$s' = r \cdot f / (d - \varepsilon) \tag{2}$$

In order to have $s' \geq s$, we require that $(d^2 - r^2) \geq (d - \varepsilon)^2$, i.e., $d \geq (r^2 + \varepsilon^2)/(2\varepsilon)$. Therefore, if cell C is cubic, then $r^2 = 3$. If we further set ε to 1.0, then d should be no less than 2.0. In other words, the distance from the cell center to the viewpoint should be no less than twice the voxel width. This requirement is acceptable during most of the navigation time when the camera is not very close to the object boundary. That is why ε is normally set to 1.0 in our algorithm. In extreme cases when some object voxels are closer than a two-voxel width to the viewpoint, we would rather use Equation 1 to calculate the radius. Because such extreme cases are very rare, using Equation 1 in these cases does not really affect the performance of our algorithm.

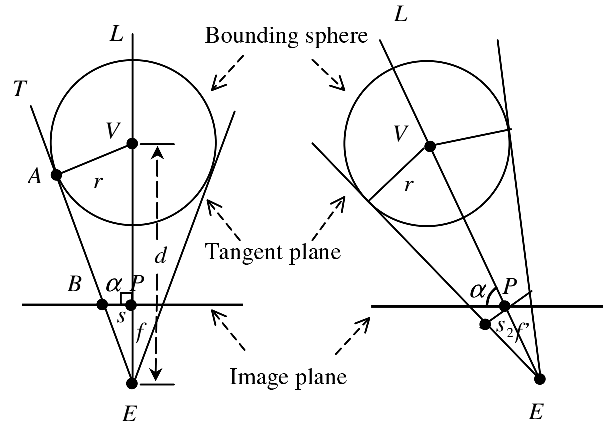


Figure 1. Bounding-sphere projection of a cell.

Next, we consider a more general situation where line L is *not* perpendicular to the image plane, as shown in Figure 1b. If we assume that α is the angle between line L and the image plane, then α is less than 90 degrees. Now, the projected region of cell C on the image plane becomes an *ellipsoid* rather than a circle. To simplify the computation of such an ellipsoidal region, we consider a *bounding circle* that is centered at P and covers the ellipsoid.

To calculate the radius of this bounding circle in the image plane, we assume that there is a plane perpendicular to L and passing through P. Obviously, the projection region of the cell bounding sphere on this plane is a circle with a radius, say, s_2. As proved in the appendix at the end of this paper, we can use $2s_2$ as a conservative estimation of the radius of the bounding circle. The calculation of s_2 is similar to that of s or s' in Equation 1 or 2, but the distance f from E to the image plane in those two equations should be replaced by the distance f' from E to P, as shown in Figure 1b.

To further accelerate the determination of those pixels inside the bounding circle, we approximate in our implementation the bounding circle with radius $2s_2$ as a square of size $4s_2 \times 4s_2$, which is centered at P and aligned with the primary axes of the image plane. As a result, our solution beats the speed of the fast template-assisted projection [23] that works only for parallel

projection. Although extra pixels that are not covered by the actual reprojection of the cell are included as a result of these overestimations, they do not affect the accuracy of the rendered image, as discussed later.

3.3 Update the Depth Buffer

Once we have determined which pixels the reprojected cell may occupy, we store or update the corresponding depth buffer entries at the current frame. Specifically, if the depth value of an occupied pixel is available in the depth buffer, we compare it with the new distance l from the reprojection of the cell in question. If the depth value in the depth buffer is greater than that distance l, then this depth value is replaced by l.

To quickly determine distance l, we use the following simple and uniform formula for all those pixels covered by the same reprojected cell:

$$l = d - r / 2 \quad (3)$$

where d and r are the same as those defined in Equation 1. The reason we do not use the distance from the viewpoint to the center of the cell or to the exact intersection point is that the ray may pass the nearest boundary of the cell at that distance.

As we noted before, using our bounding sphere, bounding circle, and bounding square to estimate the reprojection of each cell do not affect the image quality, even though some detected pixels may fall outside the regions of the actual reprojection. When updating the depth buffer, we use the distance value that is closest to the viewpoint so as not to jump beyond the cell. In the case of those pixels that are not covered by the true reprojection, the worst case is that we do not jump far enough. However, we continue to traverse the ray from that depth on, through the volume. In our experimentation, we found that the impact of this short jump on performance was negligible because most of the overestimated depths were soon to be updated by subsequent reprojections of other cells that were densely overlapped with each other at the current viewing direction.

It is also worth mentioning that, although the depth buffer is updated by the distance values that are closest to the viewpoint, it does not completely eliminate the danger of jumping beyond some object voxel during the subsequent ray traversal. Yagel and Shi [28] pointed out that the “discretization artifacts” caused this problem. In other words, this problem is caused by the discrete sampling on the object contour among different rays at the previous frame. Hence, an artificial “hole” may appear between two adjacent pixels after they are reprojected onto the current image, even though there is no hole in the true object. Then, voxels from other objects that are actually hidden by this object may be projected onto the depth buffer through this hole, resulting in an altogether skipping over of this object at those rays cast through the hole region. To solve this artificial “hole” problem, we employ the same strategy used in [8, 28]. The basic idea is to check the relative position between two adjacent pixels from the previous frame. If it changes after these two pixels are reprojected onto the current image, then discard the reprojection that is farther away from the current viewpoint. In our algorithm, because we reproject cells rather than intersection points, we check the relative position between two cell centers instead.

Compared to the previous point-based reprojection schemes [2, 28], our cell-based reprojection has shown several advantages. First, our algorithm does not round or shift the reprojected voxels on the image plane, so it provides a safe estimation of a jumping distance without missing any part of object voxels. Second, it does not significantly increase the cost during the reprojection of cells because it only reprojects the cell center and rapidly estimates the reprojected region. In fact, it could be faster than the traditional point reprojection because when adjacent rays intersect with a common cell in the previous frame, we need to reproject this cell only once. Third, because it reprojects 3D cells rather than points, each reprojection covers a larger area on the image plane. Therefore, there are fewer uncovered pixels, resulting in more effective jumps during the subsequent ray traversal procedure. This is somewhat similar to the effect of splatting each reprojected point on the image plane [27]. However, our solution is much less expensive than the volume splatting operation, especially in the situation of perspective projection where different cells have different projection shapes and sizes as a result of their different distances and orientations to the viewpoint.

4. ACCELERATED RAY CASTING

For each pixel whose depth value is available in the depth buffer, its ray-casting procedure is accelerated by directly jumping to the nearest object voxel. Once the ray approaches the nearest object boundary, we conduct regular ray sampling, shading, and compositing, using a uniform sampling interval. Other ray-casting optimizations, such as early ray termination [13], can then be conveniently incorporated for further speedup.

For other pixels whose depth values are not available in the depth buffer, we perform a simple but efficient DFB-jumping scheme to rapidly skip over the empty ray segment by exploiting the object space coherence. This idea is derived from the observation that during an interactive virtual navigation, the camera is usually located within an empty region; therefore, at each ray sample in the empty region, we can safely jump forward along the ray according to the *DFB* value that is the distance from the current sample position to the nearest object boundary. Because the DFB value of each voxel is view-independent, we compute it only once in a preprocessing. We use the exact real Euclidean distances in our algorithm, computed by a fast four-path algorithm proposed by Saito et al. [19]. By using such an exact Euclidean distance, we achieve a more accurate and hence faster space leaping, especially in a large-scale volumetric scene, than the previous methods (e.g., [21, 29]).

In fact, our ray casting through those “lucky” pixels whose depth values are available in the depth buffer can also benefit from the above DFB-jumping scheme. Note that these rays may reenter the empty region by only grazing the object boundary or passing through a low-opacity object, or the ray may even not encounter any object voxel because of the conservative estimation for each cell reprojection region discussed in Section 3. Therefore, whenever a ray sampling point is located in the empty region, we apply the DFB-jumping at that position. Once the first object voxel is encountered at each ray, we update the corresponding entry in the cell buffer so that it can be used at the next frame.

5. NEW-INCOMING-OBJECT DETECTION

When we skip over the empty regions according to the depth values derived from the cell reprojection discussed in Section 3, we implicitly assume that there is no object within such depths.

However, this assumption holds only when all the objects within the field of view of the current frame are also within that of the previous frame. In most of the practical virtual navigation systems, unfortunately, the volume scene is so large that the virtual camera is usually immersed in the virtual environment and captures only part of the volume scene at each frame. In these systems, the new incoming objects that just enter the current frame would be skipped over if we conducted space leaping based on the depth buffer.

Figure 2 gives an example of a current view that includes a new incoming object. *E* and *E'* are the current viewpoint and previous viewpoint, respectively. The upper blue square indicates a normal object that has been exposed at the preview frame, and the lower right red ellipsoid indicates a new incoming object that just emerges at the current frame. Obviously, the depth of pixel *P* saved in the depth buffer is the distance from *E* to *A* rather than that from *E* to *B*. Therefore, the new incoming object would be skipped over if we used the depth buffer for space leaping.

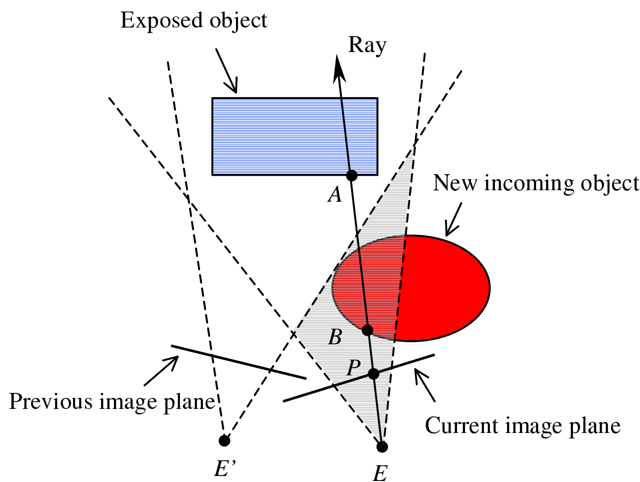


Figure 2. Emergence of new incoming objects.

In this section, we provide a simple but practical solution to detect the new incoming objects, based on an assumption that the view changes slightly between adjacent frames, so that the new incoming objects always emerge from the periphery of the reprojected region on the current image plane before they move to the central area. Our solution is illustrated in Figure 3, which shows the current image plane. We use a lower right small red circle to indicate a new incoming object and use a large ellipsoid to indicate a simplified reprojection region from all cells in the previous cell buffer. The new incoming object enters the current image from the periphery of the reprojected region, according to our assumption. In practice, there may be more than one new incoming object, and they may occupy several disjointed regions on the image. Also, the reprojection region may have an arbitrary shape, and it may include some holes that are not covered by the reprojection. Our task is to update the depth buffer for those pixels that are occupied by the new incoming objects, i.e., recalculate the depth values of those pixels that are located within the overlapped region of the large ellipsoid and the small circle shown in Figure 3.

Our solution consists of four steps:

(1) Find those pixels located at the periphery of the reprojection region in the current image. We employ a simple strategy to rapidly find out these pixels by scanning the image twice, first in a horizontal scanline order and then in a vertical scanline order. At each scanline, we look for the first and last pixels whose depth values are available in the depth buffer. For example, pixels *A* and *B* are found on a horizontal scanline in Figure 3. Although we may not find out all the pixels at the actual periphery, this fast method is accurate enough to detect the new incoming objects.

(2) Recalculate the depth of each peripheral pixel by casting a ray through that pixel and then conduct the fast DFB-jumping along the ray until the first object voxel is encountered. If the new depth value is less than the original depth in the depth buffer, then *mark* this pixel to indicate that a new incoming object is detected. Finally, update the depth buffer with the recalculated accurate depth value. In Figure 3, for example, we mark pixel *B*, but not *A*, because the new coming object appears at *B*.

(3) Further check each marked pixel's eight neighboring pixels and find those that have depth values in the depth buffer but have not been accessed yet. Then, recalculate the depth of each of those pixels, update the depth buffer with the new depth value, and mark this pixel if a new incoming object is detected, as done to the peripheral pixels in Step 2.

(4) Repeat Step 3 until no more pixels can be marked.

Obviously, when the view changes too much between adjacent frames, the new incoming objects may directly jump into the center region of the current image. Then, our solution will fail to detect them. However, in such cases, the coherence between these frames may become so weak that we doubt whether it is capable of accelerating the rendering compared to its reprojection overhead. Thus, it may be more efficient to use our DFB-jumping optimization directly.

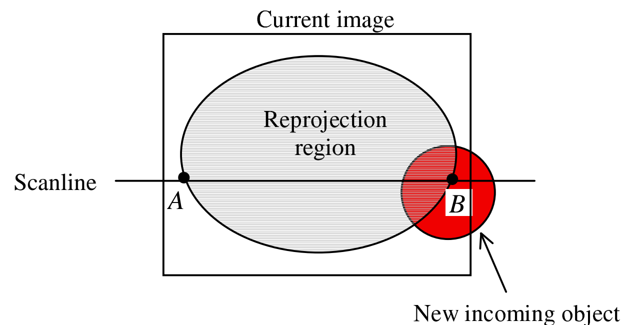
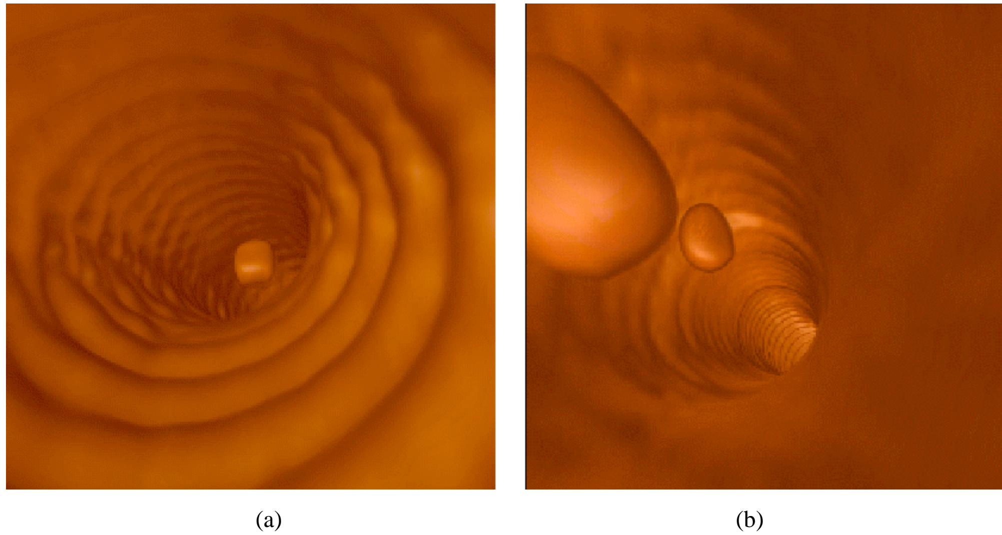


Figure 3. Detection of the new incoming objects.

6. EXPERIMENTAL RESULTS

Our space-leaping-accelerated ray-casting algorithm has been implemented on a 16-processor SGI Power Challenge in a bus-based symmetric shared-memory configuration. We parallelized our algorithm by employing a simple task-partitioning scheme, which assigned cell buffer entries and pixels to each processor in an interleaved scanline order in both the reprojection and the subsequent DFB-jumping procedures. Because the detection time of the new incoming objects was very short compared to those of the reprojection and ray traversal procedures, we preferred not to parallelize this procedure so that we could avoid the extra synchronization overhead.



(a) (b)

Figure 4. Two navigation frames inside the pipe data set.

We conducted experiments on one simulation phantom pipe data and more than 40 actual patients' CT data sets. During the reprojection, we set the radius of the bounding circle to *2s'*, and *s'* was calculated using Equation 2 instead of Equation 1, where constant ε was set to 1.0. We never encountered a colon boundary voxel that was closer than two voxel widths to the viewpoint because, to avoid collision, our physically based camera control model prevented the camera from being too close to the colon wall [9, 26].

Figure 4 shows two navigation frames inside a phantom pipe data set. The image sizes are 256×256. This simulation is based on a CT scan of a plastic pipe of 20 mm radius forming a cubic volume of $512\times512\times107$. To simulate colonic polyps, we attached small-rounded rubber objects to the inner surface of the pipe, which were clearly depicted in the rendering (see Figures 4a and 4b). The rendering time of Figure 4a with different number of processors is reported in Table 1.

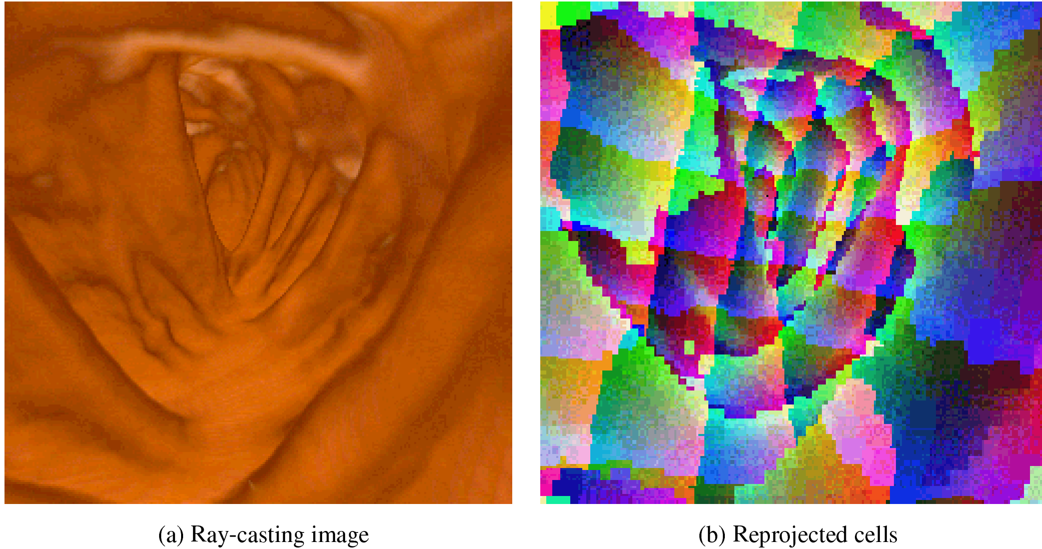
Table 1. Rendering times (in seconds) on a multiprocessor. (NP: number of processors; RC: ray-casting methods; PRC: pure ray casting [12, 13]; DFB: DFB-jumping optimization [24])

Data sets	Pipe			Colon		
NP/RC	PRC	DFB	Ours	PRC	DFB	Ours
1	2.10	1.45	0.87	1.75	1.00	0.66
4	0.52	0.36	0.24	0.53	0.33	0.19
9	0.23	0.17	0.11	0.24	0.15	0.09
16	0.15	0.10	0.07	0.15	0.09	0.06

Table 1 also lists the rendering times of another navigation frame (as shown in Figure 5a) inside a patient's colon by using our algorithm. The size of this colon data is $512\times512\times411$. The spacing ratio among the three axes is 0.703:0.703:1.000. In Figure 5b, there were only 8407 cells instead of 256×256 intersection points reprojected because of many duplicated cells in the cell buffer. Also, only 4390 pixels were hole pixels in a 256×256 image. Each tiny square in Figure 5b was the reprojection of a different cell, whose color was assigned according to its index so that adjacent cells had different colors. Although the number of reprojected cells and hole pixels changed at different frames, for example, when the camera zooms in or out, our reprojection of cells is provable to cover more pixels than the traditional reprojection of intersection points.

For comparison purposes, we collected the rendering times by using the *pure ray casting* (PRC) [12] or our previous DFB-jumping optimization (DFB) [23] at the same views, both with early ray termination optimization [13], but without space leaping. The comparison shows that our new algorithm is faster than both of them and reaches interactive rendering rates at more than 10 Hz on both data sets. We further ported our algorithm onto low-cost PCs and reached more than 10 Hz frame rates on a single Pentium 900 processor and more than 20 Hz frame rates on the same PC with dual processors.

We also detected new incoming objects during our rendering. For example, when we stepped back from the further rubber block shown at the center of Figure 4b, we detected the closer rubber block on the left to be a new incoming object as it enters the field of view. However, when we flew through the pipe and encountered the first rubber block, as shown in Figure 5a, this block was not detected as a new incoming object in our algorithm. This is because this block was already located within the field of view of the previous frame, although the curved pipe wall occluded it. In the patients' colon data, although there were no such floating blocks in the colon interior, we found that the new-incoming-object detection was also needed to render a correct image. This is because sometimes the new incoming colon wall appeared from the region very close to the eye, occluding the farther colon surface otherwise visible to the eye.



(a) Ray-casting image (b) Reprojected cells

Figure 5. A navigation frame inside a patient's colon.

7. CONCLUSIONS

We have presented a fast and reliable space-leaping method to accelerate ray casting by combining both the temporal coherence and the object space coherence during interactive navigation in a large-scale volumetric scene. Our method improves the existing space-leaping technique in a number of ways, including a reliable cell-based reprojection, fast DFB-jumping, and efficient detection of the new incoming objects. It is applicable to all kinds of Cartesian, regular, and rectilinear volume data. The high performance and robustness of our algorithm has been verified on both simulated and actual patient CT data sets.

Acknowledgements

This work has been supported by grants from ONR N000140110034, NIH CA82402, the NYS Center for Biotechnology, the Strategic Partnership for Industrial Resurgence (SPIR) program, and Viatronix Inc. The patients' data sets were provided by the University Hospital of the State University of New York at Stony Brook. Special thanks to Erin C. Allender for proofreading this paper.

Appendix

Assume that points M_1 and M_2 are the two ends of the long axis of the ellipsoid that is the projection of the cell bounding sphere on the image plane (cf., Figure 6). Evidently, there exist two tangent planes T_1 and T_2 from viewpoint E and passing through M_1 and M_2, respectively. We assume that A_1 and A_2 are the tangent points on T_1 and T_2, respectively. Without losing generality, we assume that A_1 is closer to the image plane than A_2. Evidently, M_1 and M_2 are respectively the projections of A_1 and A_2 on the image plane. Because line L is not perpendicular to the image plane, P is no longer at the center of the long axis of the ellipsoid. Instead, P is located closer to one end of the axis that is closer to E, e.g., M_2, as shown in Figure 6. Therefore, if we use the distance n between M_1 and P as a radius to draw a bounding circle centered at P, this circle will enclose the ellipsoid.

If ω is the angle of the current viewing cone and α is the angle between L and the image plane, then α must be greater than $\omega/2$; otherwise, cell C will not be viable in the current view. In fact, if we detect that $\alpha \le \omega/2$, we immediately discard the cell in question C. Therefore, the length n is within a limited range.

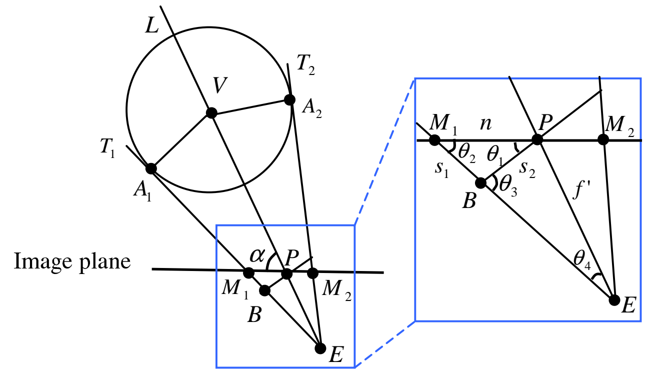


Figure 6. An example of bounding sphere projection.

In order to quickly determine length n, we consider the plane passing through P and perpendicular to line L. We assume that point B is the projection of point A_1 on this plane, θ_1 is angle $\angle BPM_1$, θ_2 is $\angle BM_1P$, θ_3 is $\angle EBP$, and θ_4 is $\angle BEP$. Let us consider the relationship between θ_1 and θ_2. From $\theta_1 + \theta_2 = \theta_3$ and $\theta_3 + \theta_4 = 90°$, we have $\theta_1 + \theta_2 = 90° - \theta_4$. Because $\cos(\theta_4) = r/d$ and r is normally very small compared to d, θ_4 is usually a very small angle. Therefore, the sum of θ_1 and θ_2 is very close to 90 degrees. On the other hand, the range of θ_1 is from 0 to $\omega/2$. Normally, ω is specified to be no larger than 90 degrees in a rendering system and, therefore, θ_1 is

no larger than 45 degrees. Accordingly, θ_2 is very close to or larger than θ_1. That is to say, the distance s_2 between B and P is very close to or larger than the distance s_1 between B and M_1. Thus, we use $2s_2$ as an overestimation of $s_1 + s_2$. Because $n < s_1 + s_2$, we further use $2s_2$ as an overestimation of n or the radius of the bounding circle.

In fact, when angle α decreases from 90 degrees (as illustrated in Figure 1a) to $\omega / 2$, the radius of the bounding circle of the projected bounding sphere increases from s or s' in Equation 1 or 2, but it does not exceed $2s_2$. Therefore, although the bounding circle with radius $2s_2$ may not be the optimal estimation of the projected bounding sphere, it is a safe and computationally economic estimation.

References

[1] S. Adelson and C. Hansen, "Fast Stereoscopic Images with Ray-Traced Volume Rendering," *Proc. Symposium on Volume Visualization*, October 1994, 3-9.
[2] S. Adelson and L. Hodges, "Generating Exact Ray-Traced Animation Frames by Reprojection," *IEEE Computer Graphics & Application*, May 1995, 15(3): 43-52.
[3] R. Avila, L. Sobierajski, and A. Kaufman, "Towards a Comprehensive Volume Visualization System," *Proc. IEEE Visualization*, October 1992, 150-158.
[4] B. Cabral, N. Cam, and J. Foran, "Accelerated Volume Rendering and Tomographics Reconstruction Using Texture Mapping Hardware", *Proc. Symposium on Volume Visualization*, October 1996, 91-98.
[5] B. Chen, A. Kaufman, and Q. Tang, "Image-Based Rendering of Surfaces from Volume Data," *Proc. Volume Graphics Symposium*, June 2001, 279-295.
[6] S. Chen and L. Williams, "View Interpolation for Image Synthesis," *Proc. SIGGRAPH*, 1993, 279-288.
[7] K. Engel, M. Kraus, and T. Ertl, "High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading," *Siggraph/Eurographics Workshop on Graphics Hardware*, 2001, 9-16.
[8] B. Gudmundsson and M. Randen, "Incremental Generation of Projections of CT-Volumes," *Proc. The First Conference on Visualization in Biomedical Computing*, 1990, 27-34.
[9] L. Hong, S. Muraki, A. Kaufman, D. Bartz, and T. He, "Virtual Voyage: Interactive Navigation in the Human Colon," *Proc. SIGGRAPH*, 1997, 27-34.
[10] J. Kniss, G. Kindlmann, and C. Hansen, "Interactive Volume Rendering Using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets," *Proc. IEEE Visualization*, October 2001, 255-262.
[11] P. Lacroute, "Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization," *Proc. Parallel Rendering Symposium*, 1995, 15-22.
[12] M. Levoy, "Display of Surfaces from Volume Data," *IEEE Computer Graphics & Application*, 1988, 8(5): 29-37.
[13] M. Levoy, "Efficient Ray Tracing of Volume Data," *ACM Transactions on Graphics*, 1990, 9(3): 245-261.
[14] L. McMillan, "An Image-Based Approach to Three-Dimensional Computer Graphics," Ph.D. Dissertation (also UNC Computer Science Technical Report 97-013), University of North Carolina at Chapel Hill, 1997.
[15] K. Mueller, N. Shareef, J. Huang, and R. Crawfis, "High-Quality Splatting on Rectilinear Grids with Efficient Culling of Occluded Voxels," *IEEE Transactions on Visualization and Computer Graphics*, 1999, 5(2): 116-134.
[16] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P. Sloan, "Interactive Ray Tracing for Isosurface Rendering," *Proc. IEEE Visualization*, October 1998, 243-238.
[17] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler, "The VolumePro Real-Time Ray-Casting System," *Proc. SIGGRAPH*, 1999, 251-260.
[18] H. Qu, M. Wan, J. Qin, and A. Kaufman, "Image-Based Rendering with Stable Frame Rates," *Proc. IEEE Visualization*, October 2000, 251-258.
[19] T. Saito, and J. Toriwaki, "New Algorithm for Euclidean Distance Transformation of an N-Dimensional Digitized Picture with Applications," *Pattern Recognition*, 1994, 27(11): 1551-1565.
[20] J. Shade, S. Gortler, L. He, and R. Sze-liski, "Layered Depth Images," *Proc. SIGGRAPH*, 1998, 231-242.
[21] M. Sramek and A. Kaufman, "Fast Ray-Tracing of Rectilinear Volume Data Using Distance Transforms," *IEEE Transactions on Visualization and Computer Graphics*, 2000, 6(3): 236-252.
[22] K. Subramanian and D. Fussel, "Applying Space Subdivision Techniques to Volume Rendering," *Proc. IEEE Visualization*, October 1990, 150-158.
[23] M. Wan, A. Kaufman, and S. Bryson, "High Performance Presence-Accelerated Ray Casting," *Proc. IEEE Visualization*, October 1999, 379-386.
[24] M. Wan, Q. Tang, A. Kaufman, Z. Liang, and M. Wax, "Volume Rendering Based Interactive Navigation within the Human Colon," *Proc. IEEE Visualization*, October 1999, 397-400.
[25] M. Wan, N. Zhang, A. Kaufman, and H. Qu, "Interactive Stereoscopic Rendering of Voxel-Based Terrain," *Proc. IEEE Virtual Reality*, March 2000, 197-206.
[26] M. Wan, F. Dachille, and A. Kaufman, "Distance-Field Based Skeletons for Virtual Navigation," *Proc. IEEE Visualization*, October 2001, 239-245.
[27] L. Westover, "Footprint Evaluation for Volume Rendering," *Proc. SIGGRAPH*, 1990, 367-376.
[28] R. Yagel, and Z. Shi, "Accelerating Volume Animation by Space-Leaping," *Proc. IEEE Visualization*, October 1993, 62-69.
[29] K. Zuiderveld, A. Koning, and M. Viergever, "Acceleration of Ray-Casting Using 3D Distance Transformas," *Visualization in Biomedical Computing II, Proc. SPIE 1808*, 1992, 324-435.