Light Weight Space Leaping using Ray Coherence

Sarang Lakare and Arie Kaufman*
Center for Visual Computing (CVC) and Department of Computer Science
Stony Brook University
Stony Brook, NY 11790-4400, USA

ABSTRACT

We present a space leaping technique for accelerating volume rendering with very low space and run-time complexity. Our technique exploits the ray coherence during ray casting by using the distance a ray traverses in empty space to leap its neighboring rays. Our technique works with parallel as well as perspective volume rendering, does not require any pre-processing or 3D data structures, and is independent of the transfer function. Being an image-space technique, it is independent of the complexity of the data being rendered. It can be used to accelerate both time-coherent and non coherent animation sequences.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.4 [Computer Graphics]: Graphics Utilities; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism;

Keywords: Direct Volume Rendering, Space Leaping, Empty Space Skipping, Ray Coherence, Volume Rendering Acceleration

1 INTRODUCTION

In this work our focus is on accelerating volume rendering by efficiently skipping empty space in the volumetric data. The main idea of empty space skipping, or space leaping, is to skip empty voxels while traversing a ray. Empty voxels are those that have zero opacity and do not contribute to the volume rendering integral. Skipping empty voxels is beneficial because it avoids sampling data along the ray in the empty region, saving a substantial number of interpolations.

Two approaches to space leaping have been well studied for ray casting volumetric data. One uses data structures to mark empty space so that during ray casting this empty space can be skipped [1, 5, 6, 10, 12]. The other exploits temporal coherence when rendering a sequence of images using ray casting [13]. There are also other techniques that combine the two approaches [11]. We analyze these techniques briefly in this section, and survey existing techniques in the next section.

The advantage of the data structure approach is that ray casting for every frame is accelerated. Almost all object space techniques use some kind of a data structure to store the transparency status of a voxel or a group of voxels. However, all such techniques require a large amount of memory that is proportional to the size of the dataset being rendered. The contents of the data structure are also dependent on the transfer function. A change in the transfer function requires rebuilding the data structure. This prohibits interactive transfer function changes, which are key to data exploration. Accessing a large data structure during ray casting may also be time consuming and therefore reduce the benefits of acceleration. Another disadvantage of this approach is the pre-processing needed to compute the data structure. Depending on the data structure used, the complexity and time required for the construction will vary.

The main disadvantage of temporal coherence techniques is that the acceleration achieved depends on the coherence between frames. Hence, this technique cannot be used to accelerate images generated from different view points and view directions or for generation of images out of order. For example, construction of a mosaic may not get accelerated by this method [8].

In this paper, we present an ideal space leaping acceleration technique that assumes the best properties from all the previous techniques. In particular, our technique does not need any data structure or pre-processing. It has very small computational cost per frame. It accelerates each frame and does not use any temporal coherence. Our technique allows interactive transfer function changes. It also works with orthographic as well as perspective projection. Finally, our technique is extremely easy to implement and understand. The memory requirement and run-time complexity for our technique is only proportional to the image size.

The basic idea of our technique is to use ray coherence to skip over empty voxels. Often a group of rays from the image plane traverse the same distance before they intersect the object. Ray coherence has been used earlier for accelerating ray tracing of traditional surface models [3]. However, use of ray coherence for accelerating ray casting on volumetric datasets is not well studied.

The paper is organized as follows. In the next section we survey the existing space leaping techniques. In Sections 3 and 4 we describe our method and its accuracy, respectively. In Section 5 we show the results of our method and discuss the implications. In Section 6 we conclude the work with some pointers for future work.

2 EXISTING SPACE LEAPING TECHNIQUES

In this section we discuss some of the existing space leaping techniques. We limit our discussion to techniques proposed recently and to those that do not make use of a special data structure for space leaping.

Yagel and Shi [13] have used inter-frame coherence to accelerate generation of consecutive frames using volume rendering. The basic idea is to use a C-buffer (Coordinates buffer) to store the object-space coordinate of the first non-empty voxel encountered by each ray. For the next frame, the C-buffer is transformed according to the change in the viewing parameters. The rays for the region visible in both frames are space leaped, whereas rays for new regions that become visible in the new frame cannot use space leaping. Depending on the amount of transformation, the acceleration varies.

Wan et. al. [12] have proposed a distance from boundary (DFB) based technique to skip empty space. The idea is to store the distance from the nearest boundary for every voxel, and then use this information to skip the empty spaces along the ray. Using DFB for space leaping has been shown to be very effective. However, storing the DFB information for each voxel may require substantial memory. For accurate floating point DFB, the storage requirement maybe as high as 4 bytes per voxel. For a 100MB dataset, this could mean an additional 400MB of data just for space leaping. In addition, a DFB-based technique does not allow interactive transfer

*e-mail:{lsarang,ari}@cs.sunysb.edu

IEEE Visualization 2004
October 10-15, Austin, Texas, USA
0-7803-8788-0/04/$20.00 ©2004 IEEE

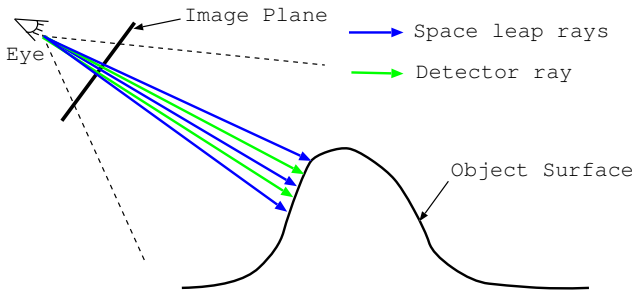


Figure 1: The detector rays that determine the depth of the object are interspread among the rays that are space leaped.

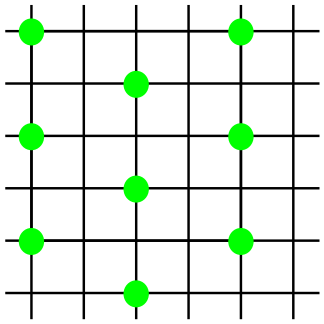


Figure 2: A portion of the image plane showing the location of the detector pixels (green dots). The rest of the grid points are non-detector pixels.

function changes. Any change in the transfer function would need re-computation of the DFB values. Also, computing the DFB is a pre-process that may be time consuming [7].

Another recent technique by Wan et. al. [11] has used temporal coherence in addition to the spatial coherence used by the DFB technique. For exploiting temporal coherence, they project cells visible from the previous frame onto the image plane of the current frame and update the depth buffer to be used for the current frame. For the pixels in the depth buffer that have no values (i.e., no cell projected on them), the earlier DFB based technique [12] is used. Due to the use of DFB, this technique also has high memory requirements. In addition, this technique is also susceptible to errors when an object that is absent in a frame, suddenly appears in the next (due to large changes in viewing angle, changes in transfer functions, etc.). This is not desirable for applications that demand high accuracy (e.g., medical imaging applications).

Sharghi and Ricketts [9] have also proposed a method similar to Wan et al [11]. Their method differs from Wan's in the way the cells to be projected are detected. They detect the cells to be projected at run-time by first shooting a ray and finding the cell that the ray would intersect. Then, the cell is projected onto the image plane and the pixels on which the cell gets projected are space leaped. However, their method has accuracy problems as mentioned by the authors themselves. These inaccuracies make the technique impractical for applications that demand accuracy.

3 Our Space Leaping Method

We now present our light weight space leaping acceleration technique. The main idea of our technique is to exploit the ray coherence. It is observed that a group of rays often traverses the same distance before intersecting an object. Knowing how far one of the rays in the group travels, we can determine how to space leap the other rays in the group (Figure 1).

For a projection, we divide the rays that are cast into two groups. The first group of rays is responsible for detecting the depth of the object while also performing the usual ray casting tasks: sampling, compositing and shading. We call these rays the *detector rays*. The rest of the rays are leaped ahead before performing the usual ray casting tasks. We call these rays *space leap rays*. The image plane pixels from which the detector rays emanate are called the *detector pixels*, and the rest of the pixels from which the space leap rays emanate are called the *space leap pixels*.

Before we describe our algorithm, we give some additional definitions. We define a *cell* as a cubical region with voxels on all of its eight corners and no voxels inside. A cell is called empty or transparent when the opacity of all the voxels on its corners is zero. This results in zero opacity for all sample points inside the cell. This is true when using an interpolation technique similar to the commonly used tri-linear interpolation. In case of higher order interpolation, all neighboring voxels that would contribute to the opacity inside the cell should be considered when deciding if a cell is empty or not. For this discussion, we assume the use of tri-linear or similar interpolation.

We now describe our algorithm. We start with the detector rays, then explain the leap buffer before describing how we accelerate the rest of the rays.

3.1 The Detector Rays

The detector rays return the distance to the first non-empty cell location as well as the composited color along the ray. The distance information is used to space leap the neighboring rays.

We increase the efficiency of detector rays by not performing interpolation, shading or compositing when the detector ray passes through empty region. We efficiently detect if a cell is empty or not by using simple OR operation instead of expensive interpolation. In case of tri-linear interpolation, we simply OR the opacity values of all the voxels of the cell in which the sample point occurs. If the result is non-zero, then the cell is non empty. Else, the cell is empty. In case of a higher order interpolation, the OR should include all the voxels that will be considered when performing the interpolation at the given sample point.

The detector pixels are interspersed among the space leaped pixels. In Figure 2 we show an example scheme for the location of detector pixels in an image plane. The configuration shown in the image uses 1 detector ray for every 4 pixels in the image. That is, one detector ray is cast for every group of 4 rays. We found the zig-zag scheme shown here to be the optimal scheme for positioning the detector pixels. Compared to other possible schemes, this scheme can detect the smallest objects. We refer the reader to Section 4 where we derive the size of the smallest object that can be guaranteed to be detected.

3.2 The Leap Buffer

We construct a buffer called the *leap buffer* to store the distance information obtained from the detector rays and to store the leap information for the space leap rays. This information can be stored in various ways. One way is to store the absolute distance traversed by the detector ray before hitting a non-empty cell. This would need more memory as we have to store them as float or double. Instead, we choose to store the number of samples the detector ray makes before hitting a non-empty cell. We use an 8 bit or 16 bit integer per pixel to store the sample count. The distance traversed by the ray can be easily computed by multiplying the number of samples by the sample distance. In the following discussions we assume the use of such an integer buffer. The technique can be very easily adjusted to use a float buffer with actual distances if needed.

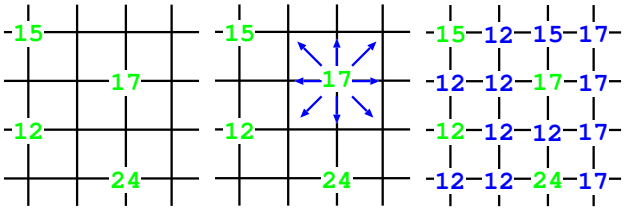


Figure 3: Filling of the leap buffer: (a) shows the leap buffer after casting the detector rays (green); (b) shows how one detector pixel value is spread around; and (c) shows the leap buffer after the value at every detector ray is spread around. The values in blue are used for space leaping.

The resolution of the leap buffer is the same as that of the image plane. Every pixel on the image plane has a corresponding location in the leap buffer. For the detector pixels, the leap buffer stores the number of samples the ray encountered before reaching a non-empty cell. For the space leap pixels, the leap buffer stores the minimum number of samples during which the rays would encounter empty space. This information will be used later to space leap these rays by moving the start point of the rays ahead by those many samples. Initially, the leap buffer is empty. When a detector ray is cast, the leap buffer location corresponding to the detector pixel is marked indicating it to be a detector ray. The number of samples before hitting a non-empty space, as returned by the detector ray, is stored at the location in the leap buffer.

In the first phase of our algorithm, all the detector rays are cast and the returned value is stored in the leap buffer. As the detector rays also serve as regular ray-casting rays, the composition along these rays is done according to the rendering technique used and the resultant color is stored in the image buffer.

After casting all detector rays, all non-detector pixels (i.e., the space leap pixels) in the leap buffer are empty. The empty regions are filled using the leap information from the detector rays. The algorithm for filling up is as follows. From every detector pixel, we take the leap information (Figure 3a) and spread it around to the neighboring non-detector pixels that are in the influence area of the detector pixels (Figure 3b). The influence area includes all the pixels around the detector pixel that should use detector ray information for space leaping. The selection of the influence area is discussed in Section 4.2.

While spreading the leap information, we check if the value at the neighboring location was already set. If it had been set, we compare it with the new value and store the lower of the two (Figure 3). The lower value gives us a conservative estimate for space leaping making sure that the ray will not overshoot a non-empty cell in between. If the value was not set, then we simply store the leap information.

After the leap information from all the rays is spread around, we get a completely filled leap buffer (Figure 3c). For all non-detector pixels, the leap buffer has the space leaping information stored in them. In the next step we cast the space leap rays using this information.

3.3 Space Leaping Rays

In the second phase of the algorithm, the space leap rays are cast. The leap buffer is swept from top to bottom looking for locations that are not detector pixels. Whenever a non-detector location is found, a ray is cast from that location. The ray is space leaped using the information stored in the leap buffer.

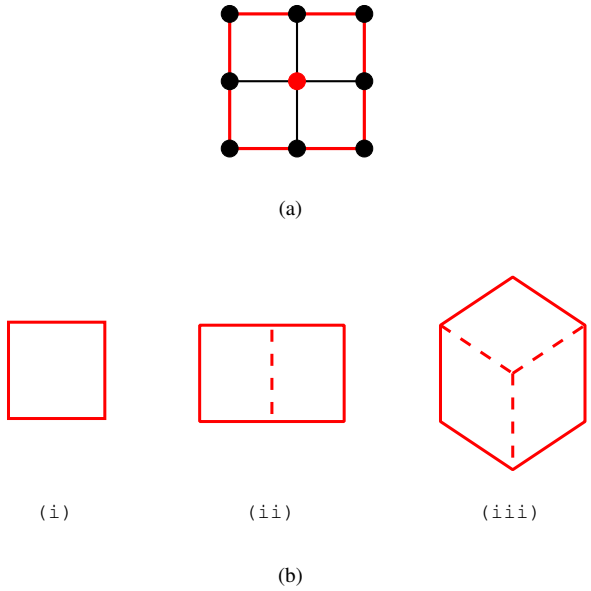


Figure 4: (a) A voxel (shown in red) is part of 8 cells (only 4 cells are shown in 2D for simplicity). Collectively, we call these cells the *cell group* of the voxel (marked by the red boundary). (b) Projection of a cell group on the image plane can have several shapes. The projection size is smallest when only one face is visible (i). When two or three faces are visible (ii and iii, respectively), the projection size is larger.

The space leaping is performed by moving the ray ahead by a distance of leap buffer value times sample distance along the ray direction. After this, the ray continues on its path of regular ray sampling, compositing and shading. Other ray acceleration techniques, such as early ray termination [4], can then be applied to the ray to achieve further speedup.

4 Accuracy

An ideal space leaping technique would not have any change in the rendered images compared to non-space leaped rendering. This is because space leaping should do exactly that: leap over empty space. As empty space does not contribute to the final rendered image, there should be no difference in the final image with and without space leaping.

We now show that our technique is accurate and that there is no change in the final image after using our space leaping technique. We guarantee that the detector rays will detect even a single non-empty voxel in an otherwise empty region and not miss it. We also prove that no region is accidentally leaped over by the space leap rays.

4.1 Single Voxel Regions

We guarantee that our technique will not miss any part of the object. Due to discrete sampling, the smallest object will be represented by a single voxel. If we can guarantee that even a single voxel object will be detected by our detector rays, then we guarantee that nothing will be missed during rendering.

Every voxel in the dataset is part of 8 cells (Figure 4a). We refer to this as the *cell group* of the voxel. For a non-transparent voxel, this means that all the eight cells around it are considered

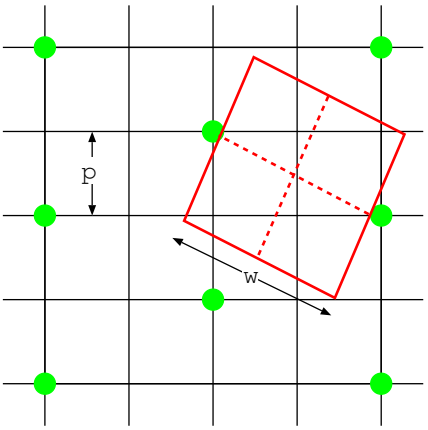


Figure 5: Projection of the smallest face of a group of 8 cells onto the image plane. The maximum width of the projection without intersecting a detector ray is w. Whenever the projection width is greater than w, at least one detector ray will intersect the cell group.

non-empty. This means that if we have at least one detector ray that goes through this cell group and detects it, then no single voxel can go un-detected.

We determine the maximum size of a cell group that can miss detector rays by analyzing the projection of the cell group onto the image plane. The maximum projection that can fit between detector rays gives the maximum size of a cell group that can miss detection by the detector rays. The minimum projection of a cell group is the smallest face of the cell group (Figure 4b). The maximum size of this face that can fit between the detector pixels gives us the maximum size of the cell group that can be missed by the detector rays.

In Figure 5, we show the maximum projection of the smallest face of the cell group such that no detector ray intersects it. The projection of the cell group is marked by red lines. The detector pixels are shown by green dots and the rest of the pixels are not shown for simplicity. It can be observed that an edge of the projection of length equal to the maximum distance between two detector pixels (given by w) stretches between the two detector pixels. For any projection edge larger than or equal to w, there will be at least one detector ray that intersects the projection.

The minimum length of an edge of the projection that is acceptable is thus w. The minimum w can be computed from Figure 5 in terms of the distance p between the pixels on the image plane. We make a practical assumption here that the image plane is isotropic with an inter-pixel distance of p in the world coordinates. However, it is not too difficult to extend the following calculations for anisotropic image planes.

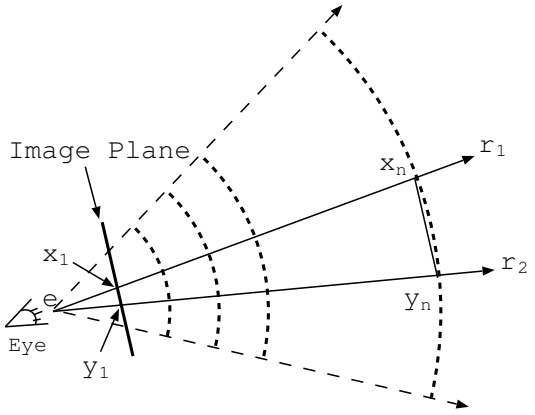
The minimum value for w in world coordinates can be calculated to be:

$$w_{min} = \sqrt{p^2 + (2p)^2} = p\sqrt{5} \tag{1}$$

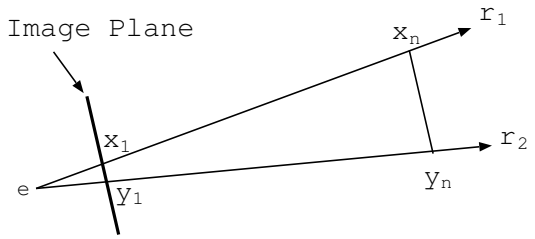
The value of w_{min} gives the minimum projection width of the cell group's smallest face. The smallest face will be of size $2\mu_{min} \times 2\mu_{min2}$, where μ_{min} and μ_{min2} are the lengths of the two smallest edges of the cell (voxel unit sizes), with $\mu_{min} \leq \mu_{min2}$. For simplicity, we substitute μ_{min} for μ_{min2}. The smallest face is then $2\mu_{min} \times 2\mu_{min}$. To guarantee an intersection with the detector rays, each side of the cell group should be at least w_{min}. In other words,

$$2\mu_{min} \geq w_{min}$$

or



(a)



(b)

Figure 6: The case of perspective projection. The distance between the rays increases as the rays move away from the image plane.

$$\mu_{min} \geq \frac{w_{min}}{2} \tag{2}$$

From Equations 1 and 2, we can say that for a given volumetric data with the lowest voxel unit distance μ_{min}, the maximum inter-pixel distance has to be:

$$p_{max} = \frac{2}{\sqrt{5}} \times \mu_{min} \tag{3}$$

In other words, the inter-pixel distance in order to guarantee a detection by detector rays has to be:

$$p \leq \frac{2}{\sqrt{5}} \times \mu_{min} \tag{4}$$

For any projection with the pixel distance less than p_{max}, we can guarantee that at least one detector ray will hit a cell group, making sure that no non-transparent voxel is missed. It has to be noted however, that this condition on pixel width is only for parallel projection. In parallel projection, the distance between the pixels on the image plane is exactly the same as the distance between the rays cast from those pixels. In case of perspective projection, this is not true. We now discuss the accuracy in case of perspective projection.

For perspective projection, the distance between the rays increases as they move away from the eye. In Figure 6 we show an example of perspective projection. We consider two neighboring rays r_1 and r_2 as shown in the figure. The two rays originate from the eye position e. At the image plane, the distance between the two

rays is $\overline{x_1y_1}$ and after travelling a certain distance, the rays are $\overline{x_ny_n}$ apart.

In our earlier discussion on parallel projection, we derived that as long as the parallel rays are at most p_{max} apart (Equation 3), we can guarantee that no voxels are missed. We can extend this to perspective projection rays too and say that as long as the distance between the perspective rays does not exceed p_{max}, we can guarantee that no voxels are missed. In other words, our guarantee stays if:

$$\overline{x_ny_n} \leq p_{max} = \frac{2}{\sqrt{5}} \times \mu_{min} \tag{5}$$

From Figure 6b we observe that the two triangles ex_1y_1 and ex_ny_n are similar triangles. We can thus write:

$$\frac{ex_1}{ex_n} = \frac{\overline{x_1y_1}}{\overline{x_ny_n}} \tag{6}$$

Our goal is to compute the distance the rays traverse before they move so far apart that we cannot guarantee accuracy anymore. That is, the condition in Equation 5 is not satisfied. Substituting the condition in Equation 6 we get:

$$\frac{ex_1}{ex_1 + x_1x_n} = \frac{p}{\frac{2}{\sqrt{5}} \times \mu_{min}} \tag{7}$$

where p is the distance between pixels on the image plane. Solving for $\overline{x_1x_n}$ we get:

$$\overline{x_1x_n} = (\frac{\frac{2}{\sqrt{5}} \times \mu_{min}}{p} - 1) \times ex_1 \tag{8}$$

In Equation 8, p, μ_{min} and ex_1 are all known for a given implementation of ray casting and the input dataset. Using these values, we get the maximum length of a ray up to which we can guarantee the accuracy of our space leaping. If δ is the sampling distance, then the number of sample points a ray traverses before the accuracy cannot be guaranteed is:

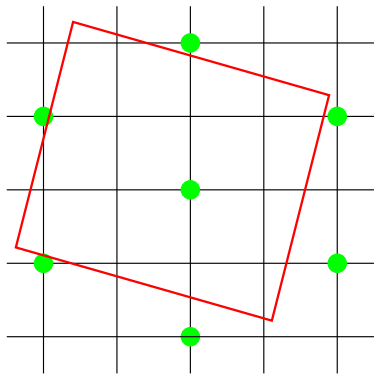
$$N = \frac{\overline{x_1x_n}}{\delta}$$

In our implementation, we use N as the upper limit to the leaping for a space leap ray. Whenever the leap distance is greater than $\overline{x_1x_n}$, we use $\overline{x_1x_n}$ as the leaping distance. The rays continue on their regular path after that.

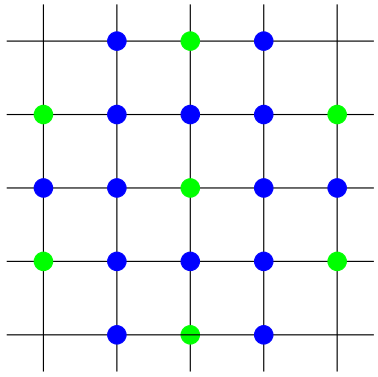
4.2 Influence of a Detector Pixel

The leap buffer is filled after the detector rays are cast. During the filling of the leap buffer, the distance returned by the detector ray is spread to the neighboring pixels for space leaping the neighboring rays later in the algorithm. The neighboring pixels to which the information is transfered are said to be in the influence area of the detector pixel. The selection of the pixels that belong to the influence area is crucial for the accuracy of the algorithm.

The idea behind spreading the distance information is that if a detector ray detects a non-empty cell group at a certain distance, then it is very likely that the rays around this detector ray will also hit the same cell group at approximately the same distance. As we never explicitly project the cells on to the image plane, we do not know the exact pixels around the detector pixel whose rays would hit the same cell group. However, we can determine an upper limit on the pixels around the detector pixel whose rays could possibly hit the same cell group as the detector ray, without the cell group intersecting any other detector ray. In other words, for a projection of a cell group on the image plane that includes only one detector pixel, we want to determine all non-detector pixels that can also be part of the projection, for the various shapes of the projection.



(a)



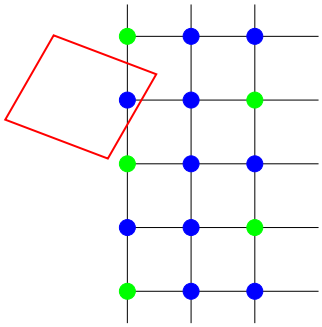
(b)

Figure 7: (a) A projection of a cell group on the image plane such that only one detector pixel intersects it. It can be observed that the cell group can never intersect any non-blue pixel (from (b)) without also intersecting another detector pixel. (b) Influence of a detector pixel (green in the center) on the neighboring non-detector pixels (blue). The rest of the non-detector pixels are not influenced by the central detector pixel.

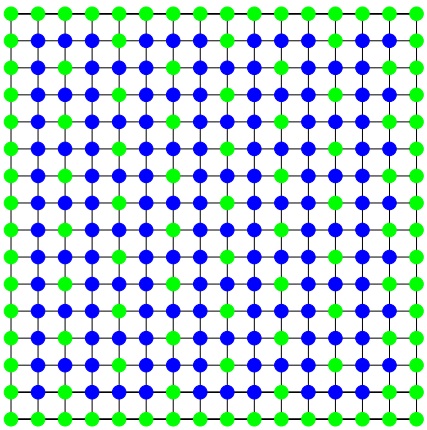
In Figure 7a we show an example projection of a cell group onto the image plane such that only one detector ray hits the cell group. To determine all the possible neighboring space leap pixels that the cell group could project to, without projecting on another detector pixel, we consider all possible shapes for the projection (Figure 4b). All the possible space leap pixels that the cell group could project to are shown in Figure 7b by blue dots. These pixels make up the influence area of the detector pixel. If a cell group projects onto any other space leap pixel on the image plane, then it is guaranteed to also project onto another detector pixel. In addition, the other space leap pixel will be in the influence area of the other detector pixel whose ray intersects the cell group.

4.3 Objects on the Boundary

In Section 4.1 we proved that even a single voxel object will not be missed by our space leaping technique. However, the proof breaks down at the boundary of the image plane. We show an example in Figure 8a where a ray on the image plane intersects the cell group but the ray is not a detector ray. This breaks our algorithm because we now have a cell group that is not intersected by any detector ray.



(a)



(b)

Figure 8: An example leap buffer (b) showing the detector pixels (green) and non-detector pixels (blue). The boundary pixels of the image buffer are all used as detector pixels in order to make sure that objects that only project on the boundary pixels are not missed, as in (a).

This cell group will be leaped over and missed if the detector rays around this cell group projection intersect another cell group further away.

In order to avoid this, we consider all the boundary pixels as detector pixels (Figure 8b). That is, detector rays are shot from all the boundary pixels. This guarantees that no object on the boundary is missed.

4.4 Backtracking Space Leap Rays

We have proven that our technique can never miss a cell group. Now we want to prove that a space leap ray will never leap beyond an object boundary inside a cell group. It is equivalent to saying that the space leap ray will never miss a non-transparent sample point inside a cell group. We apply a rule to all space leap rays: the first sample of a space leap ray should always be in an empty cell. If the first sample is not in an empty cell, then it is possible that the space leap ray has missed a non-transparent sample point (Figure 9). However, if the first sample point is in an empty cell, then the ray is definitely before the cell group.

In order to satisfy the above rule, we perform two operations. First, when we leap the space leap rays, we leap them one sample less than the number stored in the leap buffer. This is because the leap buffer stores the number of samples the detector ray took to reach a non-empty cell. Thus, the sample before that is the last sample in an empty cell. Care should be taken to check for a boundary condition of leap buffer value of zero. In that case the space leap ray should not be leaped at all. Second, we backtrack a space leap ray if the first sample point is inside a non-empty cell (Figure 9). The ray is backtracked until the sample point is inside an empty cell. Consideration has to be given to the leap buffer value for the leap ray. The backtracking should not be performed beyond the leap buffer value else the ray will go behind the image plane.

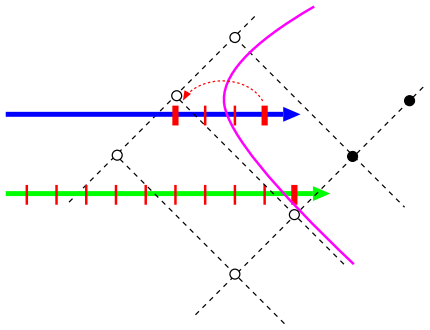


Figure 9: An example showing a space leap ray (blue) getting leaped beyond an object boundary (pink) due to the detector ray (green) finding the first sample inside the cell (thick red). The location on the space leap ray is moved back until a transparent cell is found.

It is important to note here that the number of samples a ray needs to backtrack depends on the sampling distance. The upper limit for the number of samples can be calculated by dividing the longest edge of a cell group by the sample distance:

$$M_{max} = 2 \times \frac{MAX(\mu_x, \mu_y, \mu_z)}{\delta} \tag{9}$$

In practice, we found M_{max} to be between 1 – 3. Backtracking a ray does not cause any additional overhead because the ray should have been sampled at the missed points anyway. While backtracking, we simply store the sample values in a temporary buffer and use them for compositing once the backtracking is completed to avoid sampling at the same locations again.

5 Results and Discussion

We have incorporated our space leaping acceleration technique into our *Vikon* system that provides perspective direct volume rendering using ray casting. The system allows the user to navigate inside a 3D dataset, which is especially useful for endoscopy applications, such as Virtual Colonoscopy [2]. Interactive rendering is essential for such a system, so as to allow the user to interactively navigate inside the organ of interest. Space leaping acceleration technique is appropriate for such a system because often there is plenty of empty space in front of the camera.

In Figure 10, we show images generated using our system with our space leaping acceleration enabled. Table 1 shows the effectiveness of our acceleration technique for rendering the images. The timings were recorded on a Pentium-IV, 2.6GHz PC with 1GB RAM and Linux OS. For the patient's CT Colon dataset (Figures 10a-c), we compute the acceleration at three different locations inside the dataset. The three locations were selected such that the camera is at varying distances from the surface and the ray traversal distance varies. This distance is directly proportional to the amount of computation a ray would spend on going through the transparent region. It can be observed from the table that the location where

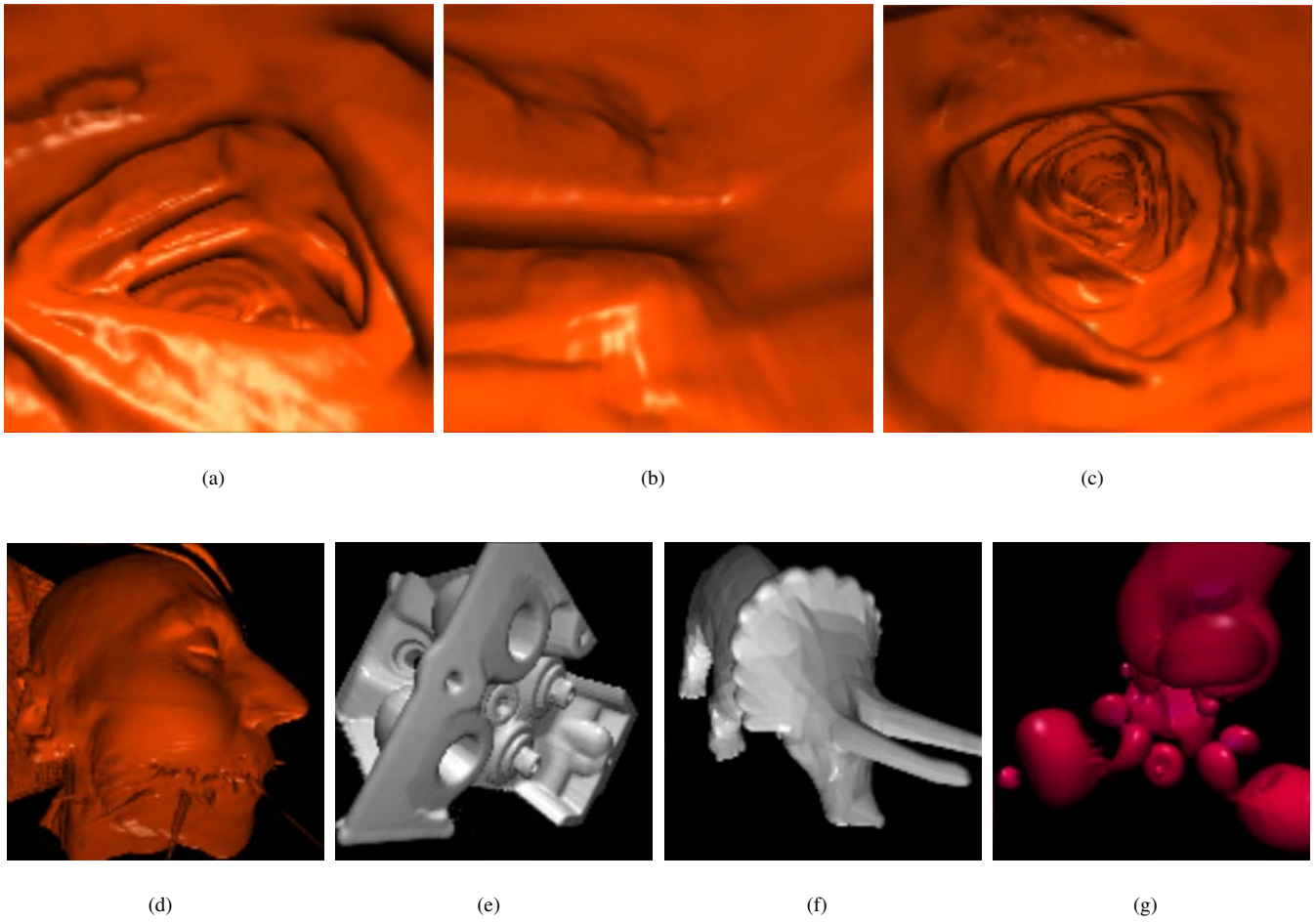


Figure 10: Volume rendered images generated with our space leaping technique. (a) - (c) Three frames from the Colon CT dataset with the camera located at an average distance from the surface, very close to the surface, and very far from the surface, respectively. (d) Head CT dataset. (e) Engine CT dataset. (f) Dino voxel model voxelized from a polygonal surface. (g) Neghip dataset.

the image plane is closest to the object (Figure 10b) shows the least speedup, whereas the location where the object is the furthest away shows the largest speedup (Figure 10c). For the remaining datasets, we compute the acceleration at some random single camera location. Our technique achieves speedups of up to 165% over non-space leap ray casting for the example datasets.

In order to verify the accuracy of the resulting images, we implemented an automatic verification scheme in our system. In the verification mode, the system generates two images simultaneously, The first image uses our acceleration technique, and the second does not. The images are compared pixel by pixel. If a difference is found, the system generates an error. So far we have not found any errors in the datasets that we have rendered.

The memory required by our acceleration technique is $O(N^2)$, where $N \times N$ is the size of the image plane. This is because we only need one buffer - the leap buffer, which has the same size as the image plane. For all the datasets in Table 1, the memory used by our algorithm was 256×256 bytes. The computational complexity of our technique is also $O(N^2)$. The only computation required is filling of the leap buffer (Section 3.2), which scans the leap buffer once.

Our space leaping acceleration technique is ideally suited for the Vikon system. The system requires a substantial amount of memory resources for complex physics based navigation. In particular, it uses a potential field technique to move the user from one end of the organ to another. Such potential fields themselves require memory. Additional memory overhead for the system must be avoided. Due to the low memory requirements of our technique, our Vikon system is able to perform accelerated volume rendering even on large datasets.

Our method does have a few drawbacks when compared to some other space leaping acceleration techniques. It cannot achieve as much empty space skipping as techniques that use data structures to store the empty/non-empty information for the data voxels can. As we space leap one out of every 4 rays that are cast, we get 75% of the possible acceleration. Another drawback of our technique is that the space leaping is only possible between the image plane and the first object hit. If there is empty space beyond that, we cannot leap the rays over this empty space.

It is important to point out that all the considerations for accuracy defined in Section 4 are satisfied in our Vikon system. One of the reasons is that the camera is close to the object surface, which results in an object cell projecting over many pixels on the image. Virtual endoscopy and other applications that require camera placement close to the object surface are prime applications for our space leaping technique. High resolution volume rendering with ortho-

Table 1: Acceleration using our space leaping

Image	Dataset Name	Dataset Size	No Space Leaping (FPS)	Our Space Leaping (FPS)	Speed Up
Figure 10a	Colon	$512 \times 512 \times 361$	3.5	6.3	79%
Figure 10b	Colon	$512 \times 512 \times 361$	5.6	8.9	61%
Figure 10c	Colon	$512 \times 512 \times 361$	4.2	7.8	85%
Figure 10d	Head	$256 \times 256 \times 225$	2.4	5.4	120%
Figure 10e	Engine	$256 \times 256 \times 110$	2.0	5.4	165%
Figure 10f	Dino	$175 \times 229 \times 512$	1.7	4.2	150%
Figure 10g	Neghip	$64 \times 64 \times 64$	5.9	10.8	84%

graphic as well as perspective projections are also a good candidate as the accuracy considerations (Equation 4) are satisfied.

6 Conclusions and Future Work

In this paper, we presented a technique to perform space leaping that has very low memory requirement and very low CPU overhead. Our technique exploits the coherence between adjacent rays. We cast detector rays that detect how far an object is from the image plane, and then use the result to space leap other adjacent rays.

Our technique has various advantages over other space leaping techniques. Being an image space technique, it is independent of the size and complexity of the dataset being rendered. Our technique allows interactive transfer function changes. The space leaping acceleration is done per-frame, allowing acceleration of non-coherent as well as coherent animation sequences. Our technique has very low memory requirement making it ideal for use in memory intensive applications. The low CPU overhead makes our technique suitable for CPU intensive applications. In addition to the above, the technique is very easy to implement and can be easily plugged into an existing ray casting system.

As future work, we would like to exploit more applications for our space leaping acceleration. Being an image space technique, it is a very good candidate for parallelization. Other future work would include combining this technique with cell projection to get an accurate estimate of the projection of the cell that a ray intersects.

Acknowledgments

This work has been supported by grants from NIH #CA82402, NSF grant CCR-0306438, CAT Biotechnology, NYSTAR, and ONR #N000110034. The Colon dataset is courtesy The Stony Brook University Hospital. The Engine dataset is courtesy GE. The authors wish to thank Manjushree Lakare and the visualization lab members for their support. The authors wish to thank Suzanne Yoakum-Stover and Susan Frank for discussions and comments on a draft of this paper.

References

[1] 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.

[2] L. Hong, A. Kaufman, Y. Wei, A. Viswambharn, M. Wax, and Z. Liang. 3D Virtual Colonoscopy. In *Proc. Symposium on Biomedical Visualization*, pages 22–32, 1995.

[3] K. I. Joy and M. N. Bhetanabhotla. Ray Tracing Parametric Surface Patches Utilizing Numerical Techniques and Ray Coherence. In *Proc. SIGGRAPH*, volume 20, pages 279–285, Aug. 1986.

[4] M. Lovoy. Efficient Ray Tracing of Volume Data. *ACM Trans. on Computer Graphics*, 9(3):245–261, 1990.

[5] M. Meißner, M. Doggett, J. Hirche, and U. Kanus. Efficient Space Leaping for Ray Casting Architectures. In *Volume Graphics*, Workshop on Volume Graphics, pages 149–161, June 2001.

[6] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P. Sloan. Interactive Ray Tracing for Isosurface Rendering. In *Proc. IEEE Visualization*, pages 233–238, 1998.

[7] T. Saito and J. Toriwaki. New Algorithm for Euclidean Distance Transformation of an N-Dimensional Digitized Picture with Applications. *Pattern Recognition*, 27(11):1551–1565, 1994.

[8] I. W. O. Serlie, F. M. Vos, R. E. van Gelder, J. Stoker, R. Truyen, F. A. Gerritsen, Y. Nio, and F. H. Post. Improved Visualization in Virtual Colonoscopy using Image-Based Rendering. In *Proc. Joint Eurographics - IEEE TCVG Symposium on Visualization*, pages 137–146, May 2001.

[9] M. Sharghi and I. W. Ricketts. Interactive Visualisation of a Virtual Colonoscopy by Combining Ray Casting with an Acceleration Corridor. In *Proc. 6th Annual Meeting on Medical Image Understanding and Analysis*, pages 133–136, July 2002.

[10] K. Subramaniam and D. Fussel. Applying Space Subdivision Techniques to Volume Rendering. In *Proc. IEEE Visualization*, pages 150–158, Oct. 1990.

[11] M. Wan, A. Sadiq, and A. Kaufman. Fast and Reliable Space Leaping for Interactive Volume Rendering. In *Proc. IEEE Visualization*, pages 195–202, Oct. 2002.

[12] M. Wan, Q. Tang, A. Kaufman, Z. Liang, and M. Wax. Volume Rendering Based Interactive Navigation within the Human Colon. In *Proc. IEEE Visualization*, pages 397–400, 1999.

[13] R. Yagel and Z. Shi. Accelerating Volume Animation by Space-Leaping. In *Proc. IEEE Visualization*, pages 62–69, 1993.