Using 3-Dimensional Seed Filling in View Space to Accelerate Volume Rendering

Jarkko Oikarinen

Department of Diagnostic Radiology, University of Oulu, Finland

Abstract

This paper presents and evaluates a novel method for projecting a three-dimensional volumetric 6-connected object onto a 2D view using seed filling in the view space. Resampling a 6-connected volume by ray casting produces a new 6-connected volume, provided that the sampling rate is high enough. This new volume can be projected by a 3-dimensional seed filling algorithm. We implement and evaluate a volume rendering algorithm which performs the resampling and seed filling in one pass. With this algorithm the projection can be calculated by accessing only part of the voxels within the data set. The method was implemented on a general purpose computer architecture and tested with several artificial and medical volumetric data sets. The frame rate achieved by the algorithm allows nearly interactive (4-5 frames per second) rotation of typical medical volumetric images (MR data 256x256x128 was used) on a common single-processor personal computer with no specialized hardware. The algorithm requires a preprocessing step to find the initial seed voxels. Our current implementation supports only parallel projection.

Introduction

The process of visualizing three-dimensional volumetric data can be conceptually divided into two steps. The first step is the projection, which determines which voxels of the original data set contribute to each pixel in the final image. The second step is the shading, which uses these voxels to calculate the shade for each pixel in the three-dimensional projection. In this paper we concentrate on the projection step, in particular the projection of 6-connected volumetric objects.

The known algorithms performing the volumetric projection can be divided into two groups; algorithms which operate in *object-space* and algorithms which operate in *image-space*. We will concentrate on image-space algorithms within this paper.

Image-space algorithms operate by casting a ray from each pixel on the screen through the volume. The ray is terminated if it hits an opaque object or accumulates an opaque value. This is called ray casting. If reflection, shadows and refraction are taken into account, the process is called ray tracing.

The empty space around the object does not contribute to the final image. Therefore one way to speed up the ray casting algorithm is to leap over the empty space which is around the object to be displayed [8]. This can be accomplished by using octrees [4], [5]. Another way to skip the processing of the empty space is to calculate a convex polygonal model around the object and start the ray traversal from the point where the ray intersects the polygon the first time and stop the ray traversal where the ray intersects a polygon the next time [1]. The proximity-clouds method stores the distance to the nearest object voxel into all non-object voxels. When a ray hits a non-object voxel, the value of the voxel can be used to calculate the safe distance to leap over [2], [9]. Over 30% speed-up was gained in the total time for ray casting volumes of size 128x128x128 [2], and five-fold speed-ups were gained for medical volumes of sizes 256x256x124 and 256x256x149 [9]. The different results are explained by the data set size and differences in implementations.

The similarity between rays is utilized in the template method. The actual traversal of the ray is sped up by a look-up table, which contains the steps taken by one ray. This look-up table is then used to faster generate the path traversed by each ray. Only the starting point within the look-up table varies, depending on the starting location of the ray. This look-up table can be calculated in advance and thus the arithmetic required in the innermost loop of the ray casting algorithm is reduced [7].

Seed filling is a basic primitive in two-dimensional computer graphics. Seed filling requires a seed pixel, P. An interior-defined region is the largest connected region of points whose value is the same as that of the P [3]. The efficient seed filling algorithms work with spans, which are defined by the starting and the ending point within a row. The algorithms usually require one read-writable bit for each pixel. This bit is used to indicate that the pixel has already been handled by the algorithm. The seed filling algorithms generalize quite easily into three dimensions.

Our aim is to combine the ray casting and seed filling steps together into one compact algorithm. In plain ray casting algorithm a significant amount of time is spent resampling the empty voxels outside the object and the completely opaque voxels inside the object, which do not contribute to the final image. If this time exceeds the overhead time needed for seed filling, then the algorithm will be faster than plain ray casting.

Definitions

The *volume* to be visualized is given as a 3D array of scalars. Each element of the array is called a *voxel*. Each voxel in this volume has been preprocessed by a binarizing function (segmentation) to have a value of 0 or 1. These voxels are referred to as non-empty, i.e. opaque or partially transparent and empty, i.e. transparent, respectively. We performed the segmentation algorithmically for artificial data sets and via thresholding and morphological opening for medical data sets.

At this point we assume, for the sake of simplicity, that the set of non-empty voxels all belong to only one 6-connected *object*. We can also assume that the object does not meet space boundaries (if it does, we just expand the data set with empty voxels until there is at least one extra layer of empty voxels on each side of the object). We also assume that the viewing window is large enough to completely show the visible surface of the 6-connected object.

A set of rays parallel to the viewing direction are cast through this space, one ray for each pixel in the view. The binary segment volume is resampled by taking steps along each ray at even intervals. The new volume in the view lattice is also 6-connected, provided that the sampling rate is high enough.

A preprocessing step finds a *seed voxel* from the volume. The *seed voxel* is any voxel within the 6-connected object. A *seed point* is the projection of the seed voxel into the view space. The x and y coordinates of the seed point are rounded to the nearest integer value, identifying the ray intersecting the seed voxel. Thus the seed point is specified by view x and y coordinates and the position of the seed voxel along the ray.

Jarkko Oikarinen is with the Department of Diagnostic Radiology, Oulu University Hospital, PL 22, 90221 OULU, Finland.
E-Mail: jto@iki.fi URL: http://www.iki.fi/~jto

Algorithm Overview

All sample points within the object can be accessed by a 3D seed filling algorithm operating in view space and starting from the initial seed point (see Fig. 1). Our algorithm will, instead of using the resampled view space stored in the memory, perform the resampling on demand as the values are needed to composite the view. The benefit of this is that it will be necessary to sample only those points which are accessed by the seed filling algorithm. These points consist of, in addition to the non-empty sample points, only those empty sample points, which have a non-empty 6-neighbor. If the seed filling algorithm stores the coordinates of the first non-empty sample point for each ray into a buffer, this buffer becomes a *z-buffer.* When each pixel within the z-buffer is modified to have the shading of the first non-empty sample point, a shaded view is created. A maximum intensity projection (MIP) view is created, if the seed filling algorithm stores the maximum valued sample point for each ray into a buffer. In this short paper we present and evaluate the performance of a 3D seed filling algorithm which determines the first non-empty sample point intersected by a ray. In this case the seed filling algorithm needs to access only the surface voxels of the object. The resampled grid is not stored anywhere and is shown in the figures for illustration purposes only.

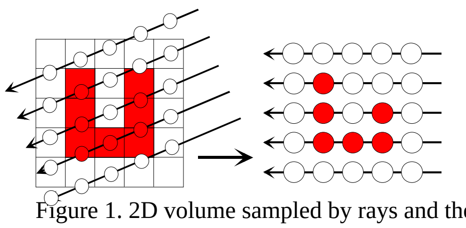

Figure 1. 2D volume sampled by rays and the resampled grid, where the object is also 6-connected

Seed Filling

The seed filling algorithm requires a set of simple operations which move the current location within the volume. Usually these are implemented by pointer increments or decrements. In our case we define them as macros, because after we have an algorithm which uses these basic operations to traverse within any regular grid, it is easy to replace these operations to work in the resampled view space. These operations consist of:

- A step forward and backward on a ray. This routine is called STEP_RAY(direction). If the direction is -1, the step is backward, and if it is 1, the step is forward.
- A jump to an arbitrary integer depth on the current ray. This step is called JUMP_TO(location).
- A step to the neighboring rays, preserving the current depth within the ray. These routines are named STEP_RIGHT, STEP_LEFT, STEP_UP and STEP_DOWN

A set of subsequent voxels within a ray is named a span, which is identified by inclusive starting and ending coordinates. The seed fill algorithms use spans as a common speed-up technique. These algorithms usually require a read-writable bit for each point in the view space. That is not available in our resampled grid unless we allocate a separate memory area. To avoid this, the roughly equivalent information was stored within the spans in the z-buffer. The result is an algorithm, which always finds the first entry point and the last exit point to the object for all rays. The algorithm avoids a significant portion of the sample points and accesses each sample point at most once. If there are more than one 6-connected object within the data set, the algorithm always fills *at least* the given object, but it may leak to other objects as well. This is not a problem in our case, because we just want to optimize the algorithm to avoid accessing empty voxels unnecessarily and simultaneusly minimize the algorithm complexity overhead. Our method requires three spans for each ray (see Fig. 2):

- The *unknown span* is used to tell which part of the ray should be searched for possible new non-empty voxels.
- The *known span* indicates the voxels which have already been searched during the filling. The known span is always inside the unknown span.
- The *object span* indicates the first and the last non-empty voxel within the known span. The object span is always inside the known span.

Each coordinate within a span is stored in two bytes, thus requiring four bytes for each span. In addition to these, each pixel in the Z-buffer contains one byte for flag information and another byte for storing the pixel shade. Thus the total amount of memory required for each pixel in the z-buffer is 14 bytes.

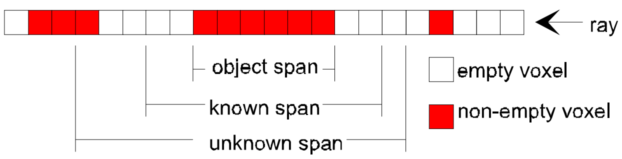

Figure 2. The object, known and unknown spans

The *surface search* algorithm (see Fig. 3) processes one span and examines the part of the unknown span not covered by the known span, starting from the end points. The search for the surface proceeds towards center, if the starting point is empty, and away from the center, if the starting point is non-empty. If new object voxels are found within processed voxels, the object span and the shade in the z-buffer is updated as well.

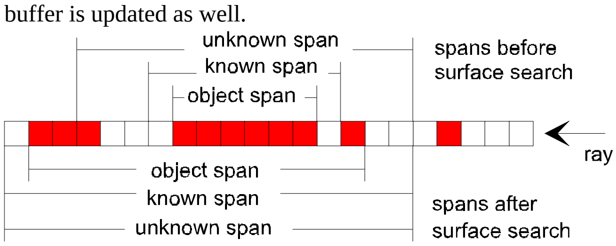

Figure 3. Surface search algorithm

If the surface search algorithm modifies the object span, the *seed fill* algorithm is called. The seed fill algorithm compares the updated object span with the known span of all four neighboring rays. If the object span goes past either end of the the known span of the neighboring ray (see Fig. 4), the unknown span of the neighbor is updated and the surface search algorithm is called recursively for each such neighbor.

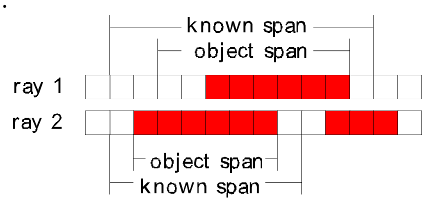

Figure 4. The object span of ray 1 goes past the known span of ray 2 on the right side

Volume Traversal

This section describes the traversal within the resampled lattice. $\vec{t}$ is the vector along the viewing direction, away from the viewer. $\vec{u}$ is the vector right in view space. $\vec{v}$ is the vector down in view space. $\vec{t}$, $\vec{u}$ and $\vec{v}$ form a right-handed coordinate system, but the vectors do not necessarily form 90° angles, unless the voxels are cubic. $\vec{t}$ is always normalized so that its largest absolute component has an absolute value of 1. The length of the vectors $\vec{u}$ and $\vec{v}$ depends on image scaling, but the sampling rate must be enough for the seed fill.

Here we assume, without loss of generality, that the volume coordinate system has been rotated so that the volume z axis is the axis most parallel to $\vec{t}$. Thus, when a step is taken along $\vec{t}$, the volume z coordinate is incremented by one and the x and y coordinates are incremented by t_x and t_y, respectively. The view steps are sheared so that the volume z coordinate is always an integer. When stepping right (left), the volume x and y coordinates are incremented (decremented) by Δx_u and Δy_u, respectively, but the z coordinate does not change. When stepping down (up), the volume x and y coordinates are incremented (decremented) by Δx_v and Δy_v, respectively, but the z coordinate does not change. These steps are calculated from equations (1), (2), (3) and (4), below, and illustrated in figure 5.

$$\Delta x_u = u_x - t_x u_z \quad (|\Delta x_u| \leq 1) \qquad (1)$$

$$\Delta y_u = u_y - t_y u_z \quad (|\Delta y_u| \leq 1) \qquad (2)$$

$$\Delta x_v = v_x - t_x v_z \quad (|\Delta x_v| \leq 1) \qquad (3)$$

$$\Delta y_v = v_y - t_y v_z \quad (|\Delta y_v| \leq 1) \qquad (4)$$

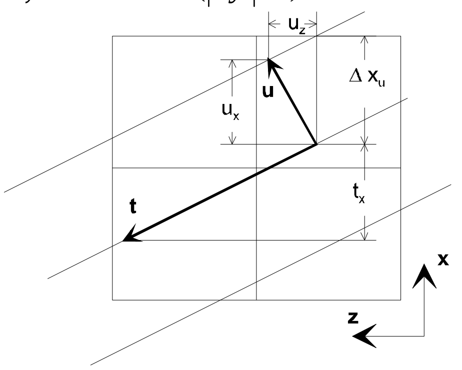

Figure 5. The view coordinate system within the volume

All STEP_ routines were implemented with a fixed point implementation of the Bresenham algorithm, the fixed point factor being 2^{16}. The Bresenham algorithm requires two decision variables, x_f and y_f, which simply represent the fractional x and y coordinates of the current position within the current voxel. When x_f is decremented below zero, the volume pointer is moved one step to the negative x direction and x_f is incremented by one. When x_f is incremented above one, the pointer is moved one step to positive x direction and x_f is decremented by one.

The JUMP_TO routine calculates the difference δ between the current volume z coordinate and the desired volume z coordinate and multiplies t_x and t_y with δ. The result is added to x_f and y_f, respectively, and the current pointer is moved δ , $\lfloor x_f \rfloor$ and $\lfloor y_f \rfloor$ steps in z, x and y directions, respectively.

Optimization

The algorithm presented so far is recursive, thus imposing the heavy load of a function call for each step to neighboring ray in view space. In order to avoid this, the recursion information is stored in the z-buffer. This information consists of steps left, right, up and down within the z-buffer, as well as a termination code. These five values can be stored in three bits, which are reserved within the z-buffer. There is one difference from the recursive algorithm that should be considered when storing the recursion information into the z-buffer. With the recursive routine it may happen that, for a given ray, there are several routines in the call stack, each newer call updating the same ray. This results in self-intersecting trail. It is not possible very efficiently to support this with the unrolled z-buffer implementation, because there is no space in the z-buffer to store the environment for more than one call into one pixel. Thus, whenever the seed fill would step into a ray which is already in the call stack, the unknown span of that ray is widened and a bit is set in the flag word. Everytime a return step is taken on the trail, this bit is tested. If the bit has been set, the ray needs to be processed again. Thus the information needed totals to 4 bits, which are reserved from the flag character within the z-buffer. The rest of the flag character is unused.

In reality the complete algorithm was implemented within one single C procedure. The various algorithms listed here were implemented with C macros. This solution was selected to avoid any overhead caused by the function calls.

Additional Features

The algorithm can also be used for projecting any number of 6-connected objects. During preprocessing a seed point is searched for each object and the seed filling algorithm is called separately for each seed. If the amount of seed points stays relatively low, the overhead to the algorithm is negligible.

The objects can be quickly toggled on or off without new preprocessing or slow down of the algorithm, because the objects can be specified to the algorithm in a 8-bit field for each voxel. When a voxel belongs to an object, the bit for that object is set; otherwise that bit is zero. Therefore a simple binary AND operation can be used to check if the voxel belongs to the list of visible objects. This limits the number of objects that can be separately toggled on or off to 8.

The viewing window may not be large enough to contain the full volume. In this case it will be necessary, in addition to processing the seed voxel, to perform ray casting along the pixels where the data set overlaps the view boundaries and start the seed filling if these rays intersect the volume to be displayed. This is not yet implemented in our algorithm.

The 26-connected ray traversal algorithm does not have high enough sampling rate along the view z-axis for all 6-connected objects to be projected correctly from directions near 45 degrees, thus requiring higher than 6-connected objects. Our images satisfy this connectivity requirement, because of the morphological opening during the segmentation.

Experiments

Two algorithms, the described 3-dimensional seed fill algorithm and an early-ray terminating ray casting algorithm with the same Bresenham line traversal algorithm in the innermost loop, were applied to 5 different data sets:

- A sphere of radius 60 centered in a volume of 128x128x128.
- A torus with radii 64 and 60, in a volume of 256x256x128.
- A wireframe box with 3x3 thick lines, in a volume of 200x200x200.
- A MRI head data set, with the skin segmented, of size 256x256x128.
- A CT data set, with the bone segmented, of size 512x512x33.

A simple shading step was performed during the data preprocessing. Several techniques, such as supersampling and better shading, to literally polish the images and improve the shading quality can be found in computer graphics textbooks [3] and are not covered in this paper. The gradient was calculated at each voxel from the 6-neighbors of the voxel. The ambient and diffuse components were calculated from the gradient and added together. The specular component was not used in the shading. A new volume was created, where each voxel had two bytes of storage. The first byte was set to contain the shade for the voxel. The shade value was scaled to be within the range [128-255]. Voxel value 0 was reserved for empty voxels. The values [1-127] were not used. The second byte was set to contain the segment bits of the voxel. The test, whether a voxel is empty or not, was performed with a binary AND operation with the segment mask. During the projection the value of the first non-empty voxel is directly copied into the view buffer, thus practically eliminating the time needed for shading during rendering, but at a small cost in image quality. The colormap had linear grey-scale range from 128 (black) to 255 (white).

During rendering the preprocessed data sets were fed into the two projection algorithms. 1000 projections of each object from different angles at 1 degree steps were computed and the time was measured. This time was divided by 1000 to get the average.

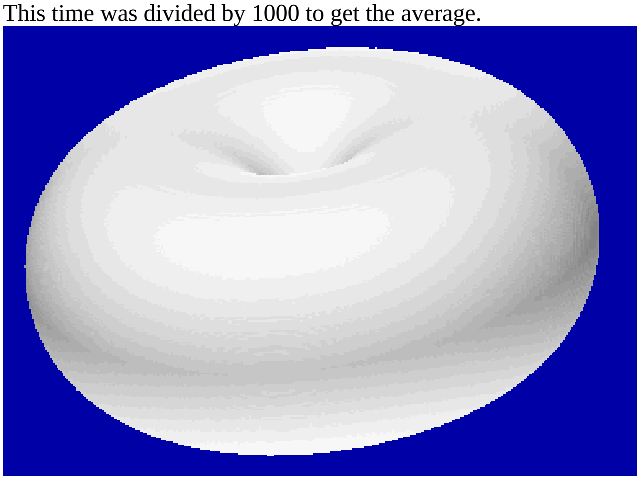

Figure 6 256x256x128 torus

Results

The Table 1 summarizes the projection times of the data sets, executed on a 200MHz Pentium Pro with Linux 2.0.0. The algorithm was written in the C programming language and compiled with the GNU CC compiler. The measurements listed here include times spent in rendering as well as displaying the images onto screen. Our non-optimized preprocessing step, including the seed search and the shade calculation, took 42 and 61 seconds for the MRI and CT, respectively.

Volume	Ray casting	Seed filling	Speedup
Sphere (view 512x512)	172 ms	121 ms	1.4
Torus (view 512x512)	540 ms	207 ms	2.6
Box (view 512x512)	1456 ms	109 ms	13.4
MRI (view 512x512)	1145 ms	212 ms	5.4
CT (view 1000x1000)	1980 ms	572 ms	3.5

Table 1. Speed comparison with plain ray casting algorithm

Discussion

The method described and evaluated in this paper reduces the inherent memory bottleneck problem in volume rendering algorithms by leaping over the empty space before and after the object, therefore falling into the category of space-leaping optimizations for image space. The algorithm can operate on the original data, provided that the segmentation is based on voxel gray scale values. The only other information required is the list of seed voxels. Hence the speed optimization does not come at the cost of specialized memory-consuming data structures. The preconditions of the algorithm require that the objects to be displayed consist of a fairly low number of objects and that the sampling rate is high enough.

The results show speedups of 4 to 5 to plain ray casting with the medical data sets. This shows that seed filling in view space can be used to accelerate volume projection to gain nearly interactive frame rates.

The seed filling algorithm is similar in behaviour as well as in the reduction of the total rendering time when compared to the proximity cloud method evaluated in [2] and [9]. Both algorithms require a preprocessing step, but the storage requirements for preprocessed data are higher in the proximity cloud algorithm when the volume sizes increase. If the distance volume is stored only within the zero-valued voxels, one bit of extra storage for each voxel suffices, otherwise the storage requirements are significantly higher. The seed filling requires extra storage for the z-buffer, which in our case is 14 bytes for each pixel, adding up to 3.5MB for a 512x512 view. If the maximum volume dimension is limited to 256, this can be reduced to 8 bytes for each pixel, adding up to 2MB for a 512x512 view. The seed filling algorithm functions better when objects are toggled on or off within the view, because there is no need to perform the preprocessing again.

The optimization gained with the seed filling algorithm is based on the depth coherence between neighboring rays. Thus the speed of the algorithm depends on the volume surface topology. Noisy volumes have high amounts of seed voxels and thus are not very well suited for the algorithm.

In our implementation the ray traversal was 26-connected, but the basic principle can be extended to other evenly sampled rays as well. The potential problem with the 26-connected traversal is that the algorithm may miss voxels which would be in the path of the ray [6]. However, the visible differences in the results of these algorithms are not significant with common medical images, unless the projection is fed to another algorithm. Shading algorithms requiring detailed information, such as the visible face of the voxel, to perform the post-processing, are one example. The seed point selection has to be done carefully for the 26-connected traversal due to its low sampling rate.

We feel that there still are possibilities to improve the frame rates achieved with the algorithm by optimizing the actual span arithmetic and combining the algorithm with other known optimization methods used in ray casting. The algorithm can also fairly simply be extended to support MIP projection or non-binary opacity values by improving the span traversal to sample all the points required by the shading model to determine the pixel value within preferred accuracy.

References

[1] Avila, R., Sovierajski, L. M., Kaufman, A. E., “Towards a Compherensive Volume Visualization System”, Proceedings Visualization ‘92, Boston, MA, 13-20, October 1992.

[2] Cohen, D., Shefer, Z. “Proximity Clouds - An Acceleration Technique for 3D Grid Traversal”, The Visual Computer, 10(11):27-38, November 1994.

[3] Foley, J., van Dam, A., Feiner, S., Hughes, J., “Computer Graphics; Principles and Practice”, Second Edition, Addison-Wesley Publishing Company, 1993. ISBN 0-201-12110-7.

[4] Levoy, M., “Efficient Ray Tracing of Volume Data”, ACM Transactions on Graphics, 9, 3, 245-261, July 1990.

[5] Samet, H., Webber, R. E., “Hierarchical Data Structures and Algorithms for Computer Graphics, Part II: Applications”, IEEE Computer Graphics and Applications, 8, 7, 59-75, July 1988.

[6] Yagel, R., Cohen, D., Kaufman, A., “Discrete Ray Tracing”, IEEE Computer Graphics & Applications, 12, 9, 19-28, October 1992.

[7] Yagel, R., Kaufman, A., “Template-based Volume Viewing”, Computer Graphics Forum, 11, 3, 153-157, September 1992.

[8] Yagel, R., Shi, Z, “Accelerating Volume Animation by Space-Leaping”, Proceedings of Visualization ‘93, 62-69, San Jose, California, October 1993.

[9] Zuiderveld, K., Koning, A. H. J., Viergever, M. A., “Acceleration of Ray Casting Using 3D Distance Transforms”, Proceedings of Visualization in Biomedical Computing 1992, 1808, 324-335, October 1992.