MAXIMUM INTENSITY PROJECTION BY 3-DIMENSIONAL SEED FILLING IN VIEW LATTICE

Jarkko OIKARINEN[†] **Lasse JYRKINEN**[†‡]

[†]Department of Diagnostic Radiology
Oulu University Hospital
PO Box 22, 90221 Oulu
FINLAND

[‡]InfoTech Oulu and Department of Electrical Engineering
University of Oulu
PO Box 444, 90571 Oulu
FINLAND

ABSTRACT

In this paper we evaluate the performance of a seed filling algorithm operating in view lattice when rendering maximum intensity projections (MIP). We evaluate the combination of the seed filling algorithm and the template algorithm. We show that the template algorithm is a particularly attractive companion to the seed filling algorithm, because the template algorithm allows stepping in all six directions within the resampled data set with very few instructions and without conditional branches. The results also indicate that given a reasonable low-level threshold for empty voxels the MIP projection can be calculated in real time with standard desktop workstations. In addition we observe that the performance of our seed filling algorithm stays relatively constant with different viewing angles, as opposed to the template algorithm alone.

Key Words: Volume Visualization, Three-dimensional (3D) Imaging, Ray Casting, Volume Rendering, Medical Applications.

INTRODUCTION

In this paper we present and evaluate a new approach to accelerate a volumetric rendering called maximum intensity projection (MIP), which is commonly used in medical workstations to visualize arteries or veins from magnetic resonance angiography (MRA) or computer tomography angiography (CTA) data sets. The MIP projection algorithm displays onto view the maximum value encountered along each ray cast through the volume. The applications of MIP in magnetic resonance imaging are increasing. The MIP projection is used to present images generated both from TOF (time-of-flight) and phase contrast magnetic resonance angiography. The need for fast MIP algorithms has recently significantly increased when high-quality bolus contrast and fast 3D imaging sequences are being used in routine diagnostic imaging. At the present, MRA is used in numerous applications, replacing conventional invasive angiographies in the imaging of even the smallest vascular changes in the head, body and extremeties, caused either by atherosclerosis, neoplastic tissues or other diseases.

The process of generating three-dimensional views from volumetric data sets is highly CPU-intensive due to the high amount of data that needs to be processed. Interactive applications require three-dimensional visualization algorithms and computer architectures capable of high rendering rates with commonly used medical data sets. Several new methods for accelerating the rendering of volumetric data sets have been presented in the literature during the last ten years.

The existing methods for projecting volumetric data sets can be divided into two groups; algorithms which operate in object-space and algorithms which operate in screen-space. These groups differ by the manner in which the volume is sampled to produce the final image.

The object-space algorithms operate by going through the data set, voxel by voxel, and projecting each voxel to the pixels on the screen. The object-space algorithms can further be divided in two groups; back-to-front (BTF) and front-to-back (FTB). BTF algorithms go through the voxels in the data set in order of decreasing distance from the observer. The benefit of this is that the algorithm can avoid the hidden volume testing [Farrell et al. 84, Frieder et al. 85]. FTB algorithms, on the other hand, go through the data set in the order of increasing distance from the observer, and can take advantage of the pixels which already have accumulated an opaque value and thus skip processing those parts of the volume known to be hidden behind closer voxels [Reynolds et al. 87].

The screen-space algorithms operate by going through the view, pixel by pixel, casting a ray from each pixel and compositing the final pixel value from sample points accessed by the ray. The most common way to optimize the ray traversal in screen-space algorithms is to leap over the empty space which is around the object to be displayed. Several methods have been proposed to achieve this. Octrees were used by Levoy [Levoy90] and Samet and Webber [Samet-Webber88]. The proximity clouds have been used to calculate the distance to nearest object voxel for each empty voxel [Zuiderveld et al. 92, Cohen-Sheffer93]. When an empty voxel is encountered by the ray traversal algorithm, this distance can be used to find out the size of the empty neighborhood that can be safely skipped.

The sample points along a ray are found by interpolation. With higher-order interpolations the time spent for resampling starts to dominate in the total rendering time. One method to reduce the number of trilinear or higher order interpolations is to have a two-phase algorithm which adaptively decides to interpolate only those sample points which have at least one neighbor having a value above certain threshold [Yagel-Shi93, Yagel et al. 91].

The traversal along a ray can be stored in a template, when all rays are cast from the same base plane and the neighboring rays sample the voxels at same locations [Yagel-Kaufman92]. The template is simply a look-up table which contains the steps taken when traversing along a ray. The use of the look-up table avoids the tests which otherwise would have to be done in a three-dimensional Bresenham line drawing algorithm. The use of templates imposes the restriction that only parallel projections are allowed.

We presented a way for avoiding the empty voxels in [Oikarinen97] by using a three-dimensional seed filling algorithm to traverse the view lattice and simultaneously search for the surface of an object of binary opacity. In this study we improve our previous approach by combining the seed filling algorithm with the template algorithm. We use the algorithm to calculate MIP projections from MRA data. This differs from the previous approach in that all voxels within the object are processed, instead of the mere surface voxels of the earlier research.

DEFINITIONS

Medical tomographic images, such as CT and MR, have the form of a regular three-dimensional grid. Each element within this grid is called a voxel, each of which is typically represented by a 12 bit intensity value. The arteries and veins in MRA and CTA are represented by voxels with high intensity. The optimizing capability of the algorithm we are about to evaluate is based on completely avoiding the processing of those voxels which are below given intensity threshold and hence unlikely to contribute to the final MIP image. Therefore the voxels need to be classified into empty and non-empty voxels; all voxels falling below the given threshold are labeled empty. An object is composed from a set of 6-connected non-empty voxels. A non-empty voxel for each object is selected and named as a seed voxel.

A parallel projection from view direction is performed by casting a set of parallel rays through the volume, one ray from each pixel on the screen. This is illustrated in Figure 1. This set of rays forms the view lattice to which the proposed algorithm is applied. The color for each ray, or pixel, is calculated from the sample points along the ray. The most common sampling methods used are nearest neighbor and trilinear interpolation. The nearest neighbor sampling requires one memory access for each sample point, while trilinear interpolation requires accessing 8 voxel values and several additions and multiplications. Trilinear interpolation produces images of higher quality and is therefore the preferred method. Image supersampling is performed when more rays are cast from each pixel and the pixel value is composited from several rays. Ray supersampling occurs when the samples along the ray are taken more often than once for each voxel. These more CPU intensive supersampling methods can be used to reduce aliasing effects in the final views. The final image is hence a tradeoff between rendering speed and quality.

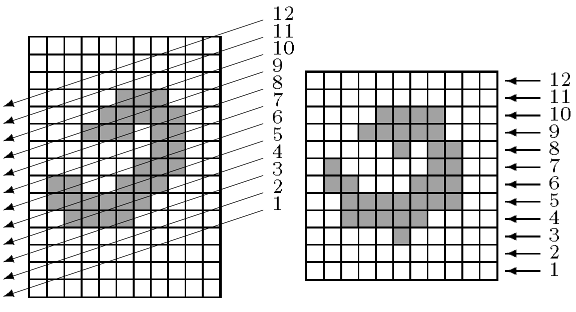

Figure 1: A set of parallel rays is cast through the volume to form parallel projection

MIP projection is a simple but computationally heavy projection, which simply assigns the sample point with the maximum value to the pixel color for each ray.

When a regular volumetric grid is sampled at regular intervals by ray casting, the sample points also form a regular volumetric grid, the view lattice. When the data to be projected consists of a set of 6-connected objects, the resampled data will contain the same set of 6-connected objects, provided that the sampling rate satisfies the Nyquist criteria.

We define a *perfect seed fill algorithm* to be an algorithm which, given a seed point within a 6-connected object, fills exactly the given object and does not leak to other objects. A *non-perfect seed fill algorithm* is an algorithm, which fills at least the 6-connected object, but may leak to other objects as well. Figure 2 gives an example of pixels covered with the type of non-perfect seed fill algorithm we will be using.

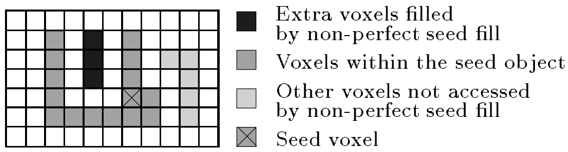

Figure 2: A non-perfect seed fill accesses **at least** all voxels belonging to the object identified by the seed point

All points within a set of 6-connected 3-dimensional discrete objects will be accessed when a seed point for each object is passed on to a three-dimensional seed fill algorithm. Hence the seed fill algorithm will also access all points that are required to compose the MIP view. The next section introduces an algorithm which performs the seed filling and the MIP view calculation at the same time.

BACKGROUND OF THE ALGORITHM

There is no need to perform the resampling and store the resulting view lattice in memory beforehand, if the seed fill algorithm is able to traverse in the original volume data and sample the points as they are needed. In the following paragraphs we describe and implement a seed fill algorithm, which searches the voxels with the maximum intensity value for each ray while traversing the virtual view lattice. In principle the benefit of this kind of seed fill algorithm is that it accesses and samples, besides all non-empty sample points, only those empty sample points which have a non-empty neighbor. Those empty sample points not accessed by the seed fill algorithm are avoided altogether, which results in time savings. However, the seed fill algorithm itself causes some overhead, and the time saving should be higher than the extra overhead if we want to accelerate the overall image generation process.

In this paper we try to find the circumstances under which the seed filling algorithm can be used to speed up plain ray casting by combining the seed fill algorithm with the template traversal algorithm and evaluating the performance by calculating MIP projections. To objectively measure the effectiveness of the algorithm we compare it with the plain template algorithm. To calculate a MIP projection the algorithm needs to access all sample points within the objects. The relative amount of non-empty sample points (arteries or veins) is typically fairly small compared to the amounts of bone and soft tissue in CT or MR images. Figure 3 shows the cumulative histogram of an MRA data set. Figure 4 presents the effect on the final MIP view when the values of the voxels falling under the threshold were set to zero. From these images we can see that the total number of voxels actually contributing to the perception of the final MIP view is low in an MRA data set. The effect of chosen threshold can be seen only barely in the images with highest threshold. From this we can conclude that ignoring the sample points below certain threshold has a negligible effect in image quality. This property of MIP views is also used in the proximity cloud optimization in [Zuiderveld et al. 92]. The amount of voxels in our data set making a visible contribution to the final image is in the order of 1% to 2%.

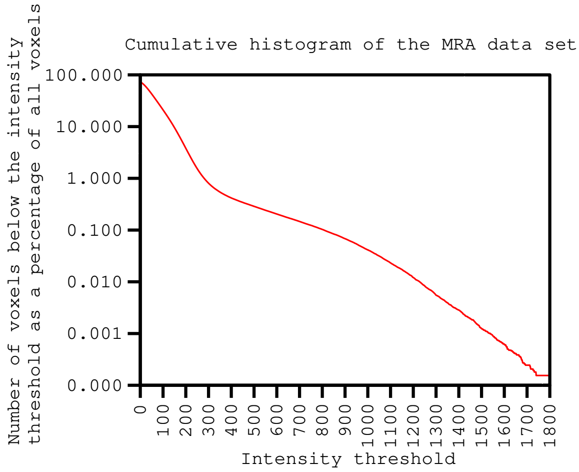

Figure 3: The decreasing cumulative histogram of the MRA data set

The studied algorithm consists of four closely intertwined sections which all have to be implemented efficiently. These parts are:

1. *The seed fill* is the main algorithm which performs a non-perfect seed fill within a regular grid.
2. *The span composition* is the portion of the algorithm which is used to process a span and search for the maximum intensity within that span.
3. *The traversal primitives for view lattice* are used to step in all six directions within the virtual view lattice.
4. *The sampling* determines how a value for a sample point is found. Here we cover only nearest neighbor and trilinear interpolation.

In the following we take a closer look at each of these parts.

SEED FILL

The seed fill algorithms presented in the literature have mostly covered 2-dimensional seed filling, but they generalize quite easily to 3 dimensions. These algorithms usually take advantage of shadows or spans, which describe a section of row, or in our case ray, which may need to be filled [Heckbert90]. A span is processed by performing a 1-dimensional seed fill within the span. After that the span casts a new span to the neighboring rows at those locations where non-empty pixels were found. The algorithm is recursive, but the recursion is typically unrolled by keeping track

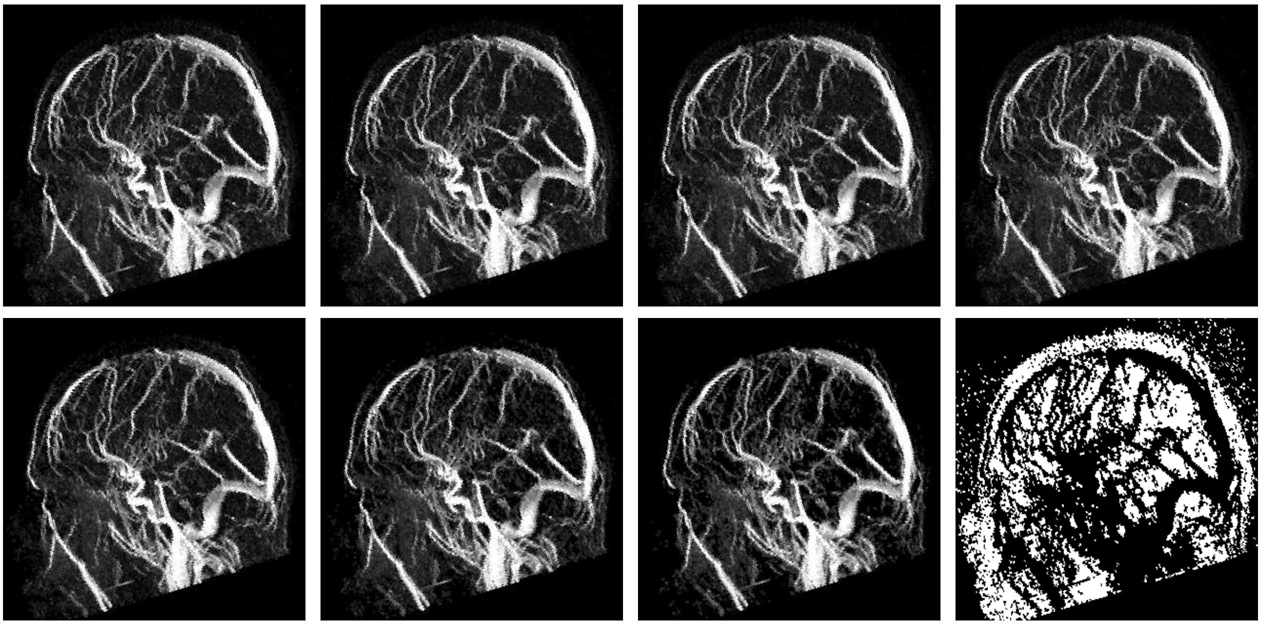

Figure 4: MIP views generated with different thresholds settings (108, 151, 194, 216, 259, 302 and 345 from left to right, top to bottom). The bottom right view shows the binarized difference between 345 and 108 thresholded images

of the stack of spans in a separate buffer. The algorithm records the direction from where the span was cast and can use that information to avoid testing some already processed pixels. The implementation of a perfect 3D seed filling algorithm usually requires a bit for each sample point or the algorithm efficiency suffers significantly. These seed fill algorithms also usually access a sample point more than once, mostly just to check if the bit has been set. We want to avoid allocating the extra space to store this bit and minimize the multiple accesses to the same sample point. Both of these can be achieved with a non-perfect seed fill algorithm, which does not push spans into a span buffer, but stores the roughly equivalent information directly into the z-buffer.

We impose a limit of one pushed span for each ray in the z-buffer to keep the memory requirements constant regardless of volume complexity. This span is called the *unknown span* and it describes those areas of the ray which are shadowed by the presence of non-empty sample points in the neighboring rays.

The z-buffer contains also the *known span* for each ray. The known span describes those sample points which have already been explored. The algorithm may omit those sample points which are within the known span while stepping through the unknown span. This will speed up the algorithm. The use of known span also avoids the need to store a bit for each pixel, and it ensures that each sample point will be accessed at most once.

The third span stored in the z-buffer is the *object span*. It describes the first and last sample point within the current ray known to contain non-empty voxels. The z-buffer structure as we implemented it is:

```
struct Zbuffer {
  unsigned short objectSpan[2];
  unsigned short knownSpan[2];
  unsigned short unknownSpan[2];
  unsigned char flags;
  unsigned char shade;
};
```

Figure 5 shows an example situation where a portion of a ray has been searched and known and object spans cover that area. The unknown span is pointing to an area which has not yet been explored.

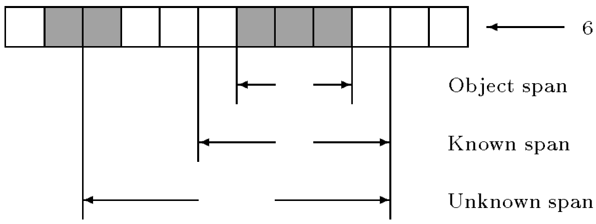

Figure 5: The z-buffer contains three spans for each ray

The object span is updated during the processing to contain the range between the first and last non-empty sample point. When a span is cast from one ray to another, the object span of the first ray is compared with the known span of the second ray (Figure 6). If the object span contains sample points which are outside the known span, the unknown span of the second ray is extended with the object span.

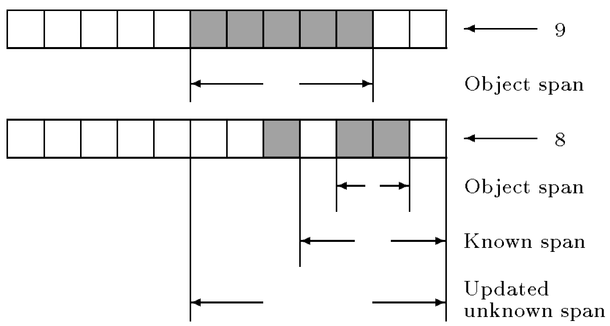

Figure 6: The object span of the upper ray is compared with the known span of the lower ray to determine if the seed fill should step into lower ray. The unknown span of lower ray is updated.

A direction code is stored in the z-buffer when the algorithm moves to process a neighboring ray after the span was cast. The direction code is stored in the flags field of the target pixel of the z-buffer. Hence it can later be used to step back the trail within the z-buffer taken by the seed filling algorithm. When the span casting does not extend the unknown spans of the neighboring rays, the algorithm examines the direction code and returns to the previous ray. Figure 7 gives an example of seed filling a small object of size $4 \times 6 \times 7$. The view is 4×6 and the ray length is 7. The filling occurs in 16 steps which are explained below figure 7. The syntax for setting a span is $span_{ray} \Leftarrow [first - last]$. For example, the known span of ray D3 from c to f would be set by $K_{D3} \Leftarrow [c - f]$. The syntax for describing a direction code is DC_{ray}. For example the direction code at ray D3 would be DC_{D3}.

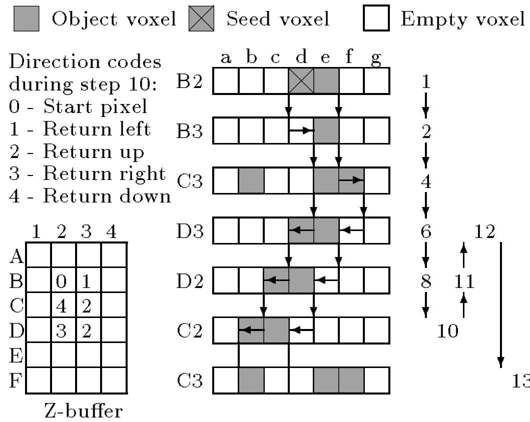

Figure 7: Taking steps in the z-buffer

1. The seed fill is started from seed point in ray B2. The span traversal sets $DC_{B2} \Leftarrow start$, $O_{B2} \Leftarrow [d - e]$ and $K_{B2} \Leftarrow [c - f]$. After that the algorithm casts O_{B2} to U_{B3} and sets $DC_{B3} \Leftarrow left$. The neighboring rays are always explored in clockwise order, starting from the right neighbor.
2. The span traversal sets $O_{B3} \Leftarrow [e - e]$ and $K_{B3} \Leftarrow [d - f]$. After that the algorithm casts O_{B3} to U_{B4} and sets $DC_{B4} \Leftarrow left$.
3. The span traversal in B4 does not find object voxels. Therefore the spans are set to $O_{B4} \Leftarrow []$ and $K_{B4} \Leftarrow [e - e]$. Because no object voxels were found, the algorithm sets $DC_{B4} \Leftarrow clear$, uses the DC value to return to ray B3. Next O_{B3} is cast to U_{C3} and $DC_{C3} \Leftarrow up$.
4. The span traversal in C3 sets $O_{C3} \Leftarrow [e - f]$ and $K_{C3} \Leftarrow [d - g]$. The voxel b is not yet found by the span traversal. The O_{C3} is cast to U_{C4} and $DC_{C4} \Leftarrow left$.
5. The span traversal in C4 does not find object voxels. Therefore the spans are set to $O_{C4} \Leftarrow []$ and $K_{C4} \Leftarrow [e - f]$. Because no object voxels were found, the algorithm will set $DC_{B4} \Leftarrow clear$ and use it to return to ray C3, cast O_{C3} to U_{D3} and set $DC_{D3} \Leftarrow up$.

6. The span traversal in D3 sets $O_{D3} \Leftarrow [d-e]$ and $K_{D3} \Leftarrow [c-f]$. The O_{D3} is cast to U_{D4} and $DC_{B4} \Leftarrow left$.
7. The span traversal in D4 does not find object voxels. Therefore the spans are set to $O_{D4} \Leftarrow []$ and $K_{D4} \Leftarrow [d-e]$. Because no object voxels were found, the algorithm sets $DC_{D4} \Leftarrow clear$ and uses it to return to ray D3 and cast O_{D3} to U_{E3} and set $DC_{E3} \Leftarrow up$. E3 does not contain object voxels, so the spans are $O_{E3} \Leftarrow []$ and $K_{E3} \Leftarrow [d-e]$. The algorithm returns again to D3. After this the O_{D3} will be cast to U_{D2} and $DC_{D2} \Leftarrow right$.
8. The span traversal in D2 sets $O_{D2} \Leftarrow [c-d]$ and $K_{D2} \Leftarrow [b-e]$. The algorithm will attempt to cast O_{D2} to U_{D3}, but it will notice that K_{D3} already contains O_{D2}. Next the algorithm will cast O_{D2} to U_{E2}.
9. The span traversal in E2 does not find object voxels. Therefore the spans are set to $O_{E2} \Leftarrow []$ and $K_{E2} \Leftarrow [c-d]$. Because no object voxels were found, the algorithm will return to ray D2 and casts O_{D2} to U_{D1}. D1 does not contain object voxels either, so the spans are set to $O_{D1} \Leftarrow []$ and $K_{D1} \Leftarrow [c-d]$ and the algorithm returns to D2. After this O_{D2} will be cast to U_{C2}.
10. The span traversal in C2 will set $O_{C2} \Leftarrow [b-c]$ and $K_{C2} \Leftarrow [a-d]$. After that the algorithm will cast O_{C2} to U_{C3}. Because $DC_{C3} \neq clear$, the algorithm will not step into C3, as it will be processed when the algorithm returns back the trail using the direction codes. Next the algorithm in C2 will try rays D2 and C1 with no new object voxels. Finally the O_{C2} is cast to U_{B2}, but as $DC_{B2} \neq clear$, the algorithm will not step into that pixel.
11. The algorithm will set $DC_{C2} \Leftarrow clear$ and return to D2.
12. The algorithm will set $DC_{D2} \Leftarrow clear$ and return to D3. The algorithm will attempt to cast O_{D3} to U_{C3}, but U_{C3} is not modified.
13. The algorithm will set $DC_{D3} \Leftarrow clear$ and return to C3. Because U_{C3} has been modified, a new span traversal is performed and the voxel b is found. The spans are set to $O_{C3} \Leftarrow [b-f]$ and $K_{C3} \Leftarrow [a-g]$. After this the algorithm will cast O_{C3} to O_{C4}, O_{D3} and O_{C2} and perform the span traversal in rays C4, D3 and C2 with no new object voxels found. The algorithm will cast O_{C3} to U_{B3}, but does not step into B3 as $DC_{B3} \neq clear$.
14. The algorithm will set $DC_{C3} \Leftarrow clear$ and return to B3. Because U_{B3} has been modified, a new span traversal is performed. The O_{B3} is not modified but $K_{B3} \Leftarrow [b-f]$. Because A3 has not yet been searched, the O_{B3} will be cast to U_{A3} and the span traversal is performed. No new object voxels are found.
15. The algorithm will set $DC_{B3} \Leftarrow clear$ and return to B2. Because U_{B2} has been modified, a new span traversal is performed. The O_{B2} is not modified but $K_{B2} \Leftarrow [b-f]$. The O_{B2} is cast to C2, B1 and A2 with no new object voxels found.
16. The seed fill will terminate because it returned back to starting point and all directions from the starting point have been explored.

RAY COMPOSITING

This portion of the algorithm takes a ray with three spans, performs the 1-dimensional seed fill along the unknown span (avoiding the known span) while simultaneously composing the final pixel value. In our case the final pixel value is simply the maximum of all sample points along the ray; hence a simple conditional assignment is performed for each step.

TRAVERSAL PRIMITIVES

Our 3-dimensional seed fill algorithm can operate in any regular lattice, given a set of simple primitives which are able to traverse the lattice points. Hence with proper basic primitive operations the algorithm should be able to traverse the virtual lattice created by the viewing transformation, and it is not necessary to have a precalculated resampled volume. Our seed fill algorithm requires 7 primitives, of which 3 move along the current ray (along the z-axis of the view) and 4 move into the neighboring ray without changing the depth (along the x and y axes of the view). We implemented these primitives to use the resampled lattice of the template algorithm [Yagel-Kaufman92], instead of the less efficient Bresenham approach in [Oikarinen97].

When the volume is traversed by rays which are cast from the same base plane, the steps for each ray can be stored in a common template. This assumes one ray for each pixel in the base plane. These steps can be precalculated in an array. The use of this array to adjust the pointer to the data set avoids the typical tests used in three-dimensional Bresenham algorithm. The steps to neighboring rays have a trivial implementation. The following structure was used for implementing the template traversal:

```
struct Template {
  long step;          // The pointer
                      // increment needed
                      // to go to the next
                      // sample point
  long cumulative;    // The cumulative
                      // increment from
                      // base plane
  long fractions[8]; // Fractional
                      // parameters exist
                      // only in trilinear
                      // interpolation
};
```

Within the seed fill routine we have the following variables defined:

unsigned short *volumePtr:
: the current sample point within the volume.

int depth:
: the current location within the template.

Template *templateArray:
: the template array (templateArray[depth] indicates the current position within the array).

Zbuffer *zBufferPtr:
: pointer to the current pixel within the z buffer.

int width:
: width of the z buffer. The number of steps required to move to the previous or next row within the z buffer.

int xJump:
: The increment/decrement of the volume pointer when moving horizontally to the next/previous ray in the view.

int yJump:
: The increment/decrement of the volume pointer when moving vertically to the next/previous ray in the view.

Hence, the 7 traversal primitives can be implemented using the variables which were defined above. The following list describes the traversal implementation in detail:

STEP_UP():
: moves to the previous ray above (up in view):

```
zBufferPtr = zBufferPtr - width;
volumePtr = volumePtr - yJump;
```

STEP_DOWN():
: moves to the next ray below (down in view):

```
zBufferPtr = zBufferPtr + width;
volumePtr = volumePtr + yJump;
```

STEP_LEFT():
: moves to the previous ray left (left in view):

```
zBufferPtr = zBufferPtr - 1;
volumePtr = volumePtr - xJump;
```

STEP_RIGHT():
: moves to the next ray right (right in view):

```
zBufferPtr = zBufferPtr + 1;
volumePtr = volumePtr + xJump;
```

STEP_IN():
: moves one step along the ray into the view:

```
volumePtr += templateArray[depth].step;
depth = depth + 1;
```

STEP_OUT():
: moves one step along the ray towards the viewer:

```
depth = depth - 1;
volumePtr -= templateArray[depth].step;
```

STEP_DEPTH(newDepth):
: steps onto a given depth within the ray.

```
/* First move back to the base plane */
volumePtr -=
        templateArray[depth].cumulative;
/* Then move to the new depth */
depth = newDepth;
volumePtr +=
        templateArray[depth].cumulative;
```

As can be seen, the primitives 1-4 can be implemented with two additions. The stepwise traversal along current ray (primitives 5 and 6) can be implemented with one array reference and two additions. The primitive 7 requires two array references, two additions and an assignment. If we compare these steps to a ray casting routine using templates we notice that the ray casting uses exactly the same primitives 5 and 6 to move along the ray. However, as shown in this study, the template algorithm can be efficiently used to implement the other primitives as well. The attractive property of the template primitives is that they contain no conditional branches.

SAMPLE GENERATION

The sample points should be calculated during the volume traversal. In practise any type of interpolation can be used with the template algorithm [Yagel-Ciula94], but very often nearest neighbor or trilinear interpolation is used. In case of nearest neighbor interpolation the current volume pointer is directly used to get the current sample point, which is tested against the maximum value encountered so far along the ray. For trilinear interpolation the sample point is calculated from eight neighbors of the current sample point. The current volume pointer points at a corner of a 2x2x2 cube, and a set of weights stored within the template are used to calculate the sample value from these neighbors, as described in [Yagel-Ciula94]. We implemented the above calculations using fixed point integer arithmetic and bitwise shift operations.

PREPROCESSING AND POSTPROCESSING

The seed fill algorithm requires a set of seeds, one for each 6-connected set of non-empty voxels in view lattice. These seeds can be found during the preprocessing stage using a connected component search algorithm. However, Figure 8 shows a situation in which the sampling ratio obtained with the 26-connected template traversal algorithm is not high enough to maintain 6-connectivity from object lattice to view lattice. Therefore we perform an additional morphological dilation to the non-empty voxels after the seed search. This expands the set of non-empty voxels by one voxel in all directions and guarantees that the seed voxels are sufficient to create a correct image.

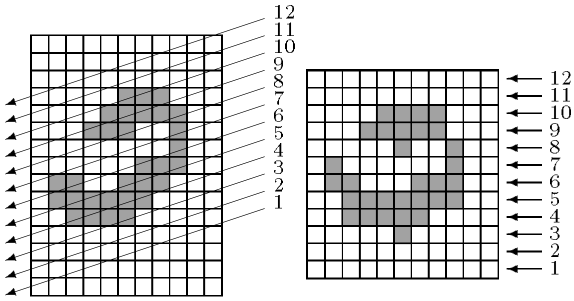

Figure 8: The sampling ratio of the 26-connected template algorithm is not high enough to maintain 6-connectivity from object lattice to view lattice

The image resulting from the template algorithm is warped, hence a postprocessing step as presented in [Yagel-Kaufman92] is required to produce the final image.

EXPERIMENTS

We measured the performance of the plain template and the combined seed fill - template algorithm with a 256x256x118 MRA data set. As the optimizing nature of the seed-filling algorithm is strongly dependent on the amount of empty voxels within the data set, we measured the performance as a function of threshold for non-empty voxels. We also measured the performance as a function of the viewing angle against the major viewing axis. We measured the algorithm performance only with nearest neighbor interpolation, because the amount of additional computation is the same with both algorithms if adaptive sampling is used and hence has a similar effect to both algorithms. The measured times include the volume projection and the postprocessing steps, but not the preprocessing which is performed only once at the beginning. The measurements were performed on a 200 MHz Pentium Pro with 64MB of main memory, under Linux 2.0.30 operating system. GNU CC with full optimizations enabled was used to compile the algorithms.

RESULTS

The results are illustrated in Figure 9. In Table 1 we list the number of seeds that were found for each threshold, ratio of non-empty voxels for each threshold and the average time to render one frame. The average time to render a frame with plain template algorithm was (coincidentally) 1.00 seconds.

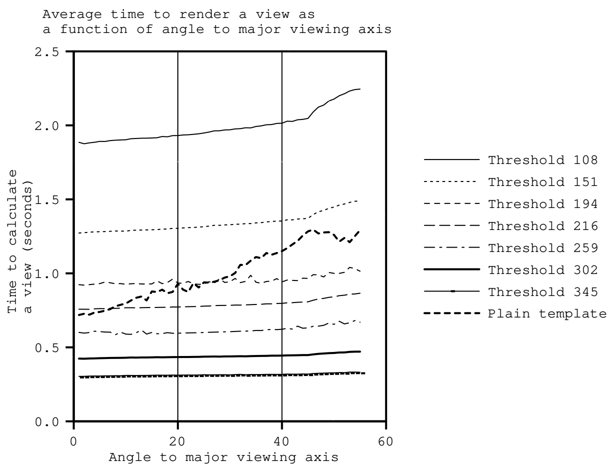

Figure 9: Performance of the seed fill algorithm measured against template algorithm as a function of angle towards main viewing axis

Thres hold	number of seeds	% of voxels above threshold	avg time/ frame
108	96797	18.5%	1.99 s
151	53457	9.3%	1.34 s
194	70782	4.2%	0.95 s
216	55324	2.8%	0.79 s
259	22010	1.3%	0.62 s
302	8839	0.8%	0.44 s
345	4322	0.6%	0.32 s

Table 1: The effect of threshold level to the number of seeds, ratio of empty voxels and average rendering time

DISCUSSION

We have shown that the setting of the gray-level threshold has a significant effect on the performance of our algorithm, because the optimization is based on avoiding the sample points which have a value below the threshold. This is, however, true for all optimizations which are based on avoiding the empty voxels, whether they operate in view lattice or object lattice. The number of seed points can be quite high, because fairly good speed improvement was achieved even with over 22000 seed points. We used a routinely obtained MRA data set and reached roughly twofold speedups with thresholds which do not have a visual impact on the final image. The number of voxels above the given threshold was around 1% with our data set. This number does not, however, include those voxels which were added to the objects during the dilation step in preprocessing. This speedup is in the same magnitude with the proximity cloud algorithm presented in the literature recently [Zuiderveld et al. 92, Cohen-Sheffer93], but a more detailed comparison with equivalent data sets and equivalent traversal primitives is still required.

Our measurements also show that the speed of the combined algorithm is not as sensitive to the viewing angle as template algorithm alone, provided that the threshold is within the area where it is appropriate to use seed filling in the first place. This means that while the plain template algorithm slows down at angles near 45 degrees towards the major viewing axis, the seed filling maintains roughly the same refresh rate, giving a smoother interactive impression.

Additionally we show that the template optimization is an especially convenient volume traversal algorithm for seed filling due to the simplicity of steps in all six directions. The previous research used Bresenham line traversal, which required a comparison and a conditional branch for all steps. The traversal using the templates is straightforward and requires two or three operations for all steps to neighboring voxels.

The compromise of the algorithm is that the pixels in the final image that fall below the given threshold will be set to background color. This has been illustrated in Figure 4. As mentioned before and seen in that figure, this effect is not visible to the observer until the number of non-empty voxels has been reduced near 1% with our data set. This threshold may vary slightly with other MRA data sets.

The dilation step during the preprocessing probably slows down the algorithm, as it increases the number of voxels that need to be accessed more than necessary. Hence by improving the preprocessing it would be possible to get better performance improvement. However, connected component search and dilation are well known algorithms in image processing and hence the algorithms were simple to implement.

In this research we have complemented the previous research in [Oikarinen97] by evaluating the performance of the seed fill algorithm for calculating MIP projections which requires the processing of all voxels within the data set. The previous research covered only surface projection of binary objects and hence the span traversal did not require processing of all non-empty sample points.

ACKNOWLEDGMENTS

The authors would like to express their gratitude for Osmo Tervonen for providing the data set, and Osmo Tervonen, Olli Silvén and Jukka Jauhiainen for proofreading the manuscript. Support of Graduate school in electronics, telecommunications and automation is gratefully acknowledged.

REFERENCES

Cohen, D., Sheffer, Z. 1994 Proximity Clouds, an Acceleration Technique for 3D Grid Traversal, *The Visual Computer*, Vol. 11, no.1, pp. 27-38.

Farrell, E. J., Zapulla, R., Yang, W. C. 1984 Color 3D Imaging of Normal and Pathological Intracranial Structures, *IEEE Computer Graphics and Applications*, Vol. 4, no. 9, September, pp. 5-17.

Frieder, G., Gordon, D., Reynolds, R. A. 1985 Back-to-front Display of Voxel-Based Objects, *IEEE Computer Graphics and Applications*, Vol. 5, no. 1, January, pp. 52-60.

Heckbert, P. 1990 A Seed Fill Algoritm, *Graphics Gems*/edited by Andrew Glassner, Academic Press, Inc, ISBN 0-12-286166-3, pp. 275-277.

Levoy, M. 1990 Efficient Ray Tracing of Volume Data, *ACM Transactions on Graphics*, Vol. 9, no. 3, July, pp. 245-261.

Oikarinen, J. 1997 Using 3-Dimensional Seed Filling in View Space to Accelerate Volume Rendering, *IEEE Visualization '97 Late Breaking Hot Topics*, pp. 17-20.

Reynolds, R. A., Gordon D., Chen L.-S. 1987 A dynamic screen technique for shaded graphics display of slice-represented objects, *CVGIP*, Vol. 38, pp. 275-298.

Samet, H. Webber, R. E. 1988 Hierarchical Data Structures and Algorithms for Computer Graphics, Part II: Applications, *IEEE Computer Graphics and Applications*, Vol. 8, no. 7, July, pp. 59-75.

Yagel, R., Cohen, D., Kaufman, A. Zhang, Q. 1991 Volumetric Ray Tracing, TR 91.01.09, Computer Science, SUNY at Stony Brook, January.

Yagel, R., Ciula, K. 1994 High-Quality Template-Based Volume Viewing, TR17, 3/94, Ohio State University.

Yagel, R., Kaufman, A. 1992 Template-based Volume Viewing, Computer Graphics Forum, Vol. 11, no. 3, *Proceedings of EUROGRAPHICS92*, Cambridge, England, September, pp. 153-167.

Yagel, R., Shi, Z. 1993 Accelerating Volume Animation by Space-Leaping, *Proceedings of Visualization'93*, pp. 62-69.

Zuiderveld, K., Koning, A. H. J., Viergever, M. A. 1992 Acceleration of Ray Casting Using 3D Distance Transforms, *Proceedings of Visualization in Biomedical Computing 1992*, October, pp. 324-335.