

# Three Dimensional Fast Exact Euclidean Distance (3D-FEED) Maps

Theo E. Schouten<sup>a</sup>, Harco C. Kuppens<sup>a</sup> and Egon L. van den Broek<sup>b</sup>

<sup>a</sup>Institute for Computing and Information Science, Radboud University Nijmegen  
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

{T.Schouten, H.Kuppens}@cs.ru.nl

<http://www.cs.ru.nl/~{ths, harcok}/>

<sup>b</sup>Department of Artificial Intelligence, Vrije Universiteit Amsterdam  
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

egon@few.vu.nl

<http://www.few.vu.nl/~egon/>

## ABSTRACT

In image and video analysis, distance maps are frequently used. They provide the (Euclidean) distance (ED) of background pixels to the nearest object pixel. Recently, the Fast Exact Euclidean Distance (FEED) transformation was launched. In this paper, we present the three dimensional (3D) version of FEED. 3D-FEED is compared with four other methods for a wide range of 3D test images. 3D-FEED proved to be twice as fast as the fastest algorithm available. Moreover, it provides true exact EDs, where other algorithms only approximate the ED. This unique algorithm makes the difference, especially there where time and precision are of importance.

## 1. INTRODUCTION

More and more, three dimensional (3D) images are used in many applications of computer vision. Various methods that have proven to be useful in two dimensional (2D) image processing, were adapted to 3D images. One of these methods is the distance transformation<sup>1</sup>, which generates an image in which the value of each pixel is its distance to a given set of pixels  $O$  in the original binary image:

$$D(p) = \min\{dist(p, q), q \in O\} \quad (1)$$

The set pixels is called  $O$  because often it consists of object pixels. Such a distance map can then for example be used for a (robot) navigation task<sup>2</sup>. In the case, non-object pixels are present, the distance map can be used to describe the internal structure of objects.

From the definition in Equation 1, a straight forward implementation can be derived directly. However, its execution time will be so large that it is of no practical use. Rosenfeld and Pfaltz<sup>1</sup> introduced the first fast algorithm for the city-block and chessboard distance measures. These distance maps are build up during two raster scans over the image using only local information; i.e., distances to neighboring pixels. These algorithms were originally developed for 2D images, but they can be adapted to 3D images.

Borgefors<sup>3</sup> introduced the idea of applying different weights to the different kind of neighboring pixels during the scans. Using different sets of weights and neighborhood sizes, a family of so called chamfer distance maps can be constructed, providing better approximations to the Euclidean Distance (ED) than the city-block and chessboard distances. These chamfer distances have also been developed for 3D images<sup>4,5</sup>.

Exact ED maps can not be obtained using local distances alone. This is due to the fact<sup>6</sup> that the tiles of the Voronoi diagram are not always connected sets on a discrete lattice. What can be obtained (as in<sup>7</sup>), is a semi-exact ED, where for most pixels the obtained distance is correct but sometimes a distance slightly to high is assigned to pixels. A few methods have been developed to find and correct the wrong pixels, obtaining the exact ED<sup>8</sup>. Such semi-exact ED transforms have also been developed for 3D images<sup>9,10</sup>. Several other (parallel) implementations of ED transforms have been proposed<sup>2,11</sup>. However, even among the parallel implementations, mostly the EDs were approximated.

Schouten and Van den Broek<sup>8</sup> developed the Fast Exact Euclidean Distance (FEED) transformation. This algorithm starts directly from the definition (see Equation 1), or rather its inverse: each object pixel *feeds* its ED to

all pixels in the image, which in turn calculate the minimum of all received EDs. Three approaches are then taken to speedup this naive algorithm to obtain an exact ED map in a computationally cheap way. Recently, FEED was extended by Schouten, Kuppens and Van den Broek<sup>2</sup> for fast processing of image sequences. However, this timed FEED (tFEED) was also developed for 2D images.

This paper describes the development of FEED for 3D images (3D-FEED). In the next Section, the principles of FEED are restated independently from the dimension of the image. In Section 3, a short description of the 3D implementation is given. The performance of 3D-FEED is compared with other algorithms, as described in Section 4, followed by a discussion in Section 5.

## 2. PRINCIPLES OF THE FAST EXACT EUCLIDEAN DISTANCE (FEED)

The FEED algorithm<sup>2, 8, 12</sup> calculates the ED transform starting from the inverse of the definition (see Equation 1): each object pixel *feeds* its ED to all pixels in the image, which in turn calculate the minimum of all received EDs. The naive algorithm then becomes:

- (1) initialize  $D(p) = \text{if } (p \in O) \text{ then } 0, \text{ else } \infty$
  - (2) foreach  $q \in O$
  - (3)    foreach  $p$
  - (4)       update :  $D(p) = \min\{D(p), ED(q, p)\}$
- (2)

The direct implementation of this naive algorithm is extremely time consuming, but can be easily proven to be correct using classical methods. Therefore, we have used it to generate correct ED maps for the images (see Section 4) used for testing the correctness of our implemented speedups, applied to the naive algorithm. These speedups are based on the following principles: (i) restricting the number of object pixels  $q$  that have to be considered, (ii) restricting the number of pixels  $p$  that have to be updated for each considered object pixel  $q$ , and (iii) suitable handling of the calculation of  $ED(q, p)$ .

In line (2) of Algorithm 2, only the “border” pixels  $B$  of  $O$  have to be considered because the minimal ED from any background pixel to the set  $O$ , is the distance from that background pixel to a border pixel  $B$  of  $O$ . For a  $n$ -dimensional ( $nD$ ) image, a border pixel  $B$  is defined as an object pixel with at least one of its neighbors with a  $(n-1)$  hyperplane in common in the background. So, in 2D a border pixel  $B$  is defined as an object pixel with at least one of its four 4-connected pixels in the background. In 3D this becomes at least one of the six 6-connected voxels (see also Figure 3a).

Moreover, the number of pixels that have to be considered in line (3) can be limited to only those that have an equal or smaller distance to the current  $B$  than to any object pixel  $q$  (see Figure 1a). By taking all bisection hyperplanes into account, it can be assured that each background pixel is updated only once (see Figure 1b). For that, background pixels on a bisection hyperplane are only updated when  $B$  is on a chosen side of the hyperplane. In 2D the bisection hyperplane is a line, in 3D a plane.

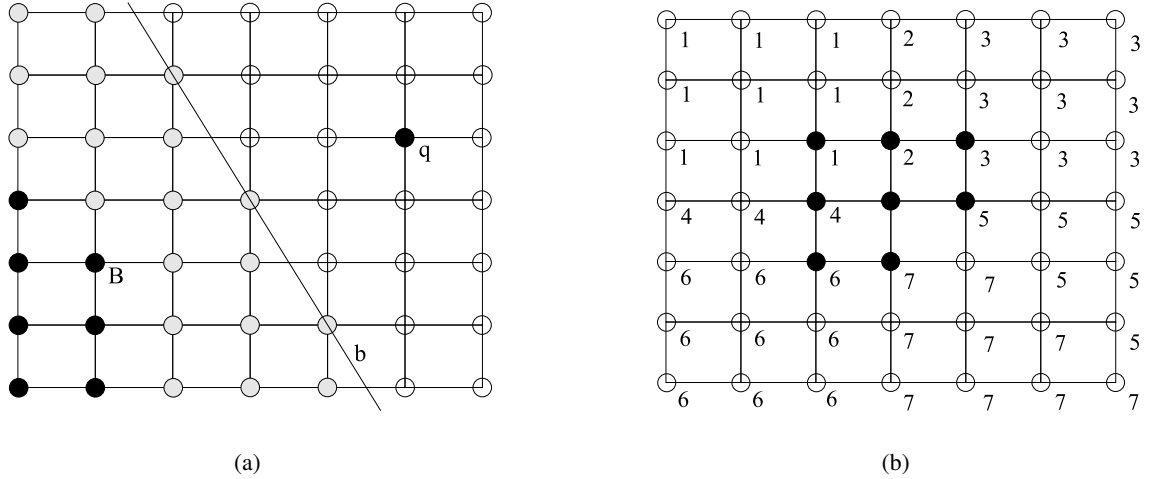
However, searching for and bookkeeping of bisection hyperplanes takes time and that time should be smaller than the time gained by updating less pixels, otherwise no speedup would be obtained but the algorithm would become slower.

The search for object pixels  $q$  to define hyperplanes is done best along a set of lines under certain angles, starting from the current border pixel  $B$ . Placing a local coordinate system at  $B$ , when point  $\vec{n}$  is an object pixel, it defines a bisection hyperplane as:

$$\begin{aligned} \|\vec{x} - \vec{0}\| &= \|\vec{x} - \vec{n}\| \\ \vec{x} \cdot \vec{x} &= \vec{x} \cdot \vec{x} + \vec{n} \cdot \vec{n} - 2(\vec{x} \cdot \vec{n}) \\ \vec{x} \cdot \vec{n} &= \frac{1}{2}\vec{n} \cdot \vec{n} \end{aligned}$$

When point  $\vec{n}$  is the  $n$ 'th point on a line in a direction determined by angle  $\vec{m}$ , thus  $\vec{n} = n\vec{m}$ , the equation of the hyperplane becomes:

$$\begin{aligned} \vec{x} \cdot n\vec{m} &= \frac{1}{2}n\vec{m} \cdot n\vec{m} \\ \vec{x} \cdot \vec{m} &= \frac{n}{2}\vec{m} \cdot \vec{m} \end{aligned}$$



**Figure 1.** Principle of limiting the number of background pixels to update in 2D. (a) Only pixels on and to the left of the bisection line  $b$  between a border pixel  $B$  and an object pixel  $q$  have to be updated. (b) An example showing that each background pixel has to be updated only once. Each background pixel is labeled by the border pixel which updates it.

Thus, only one integer  $n$ , indicates the presence or absence of an object point along a search-line  $\vec{m}$  and, subsequently, completely defines the bisection hyperplane. As soon as an object point on a search-line is found, further searching along the line can be stopped.

To keep track of the pixels  $p$  that have to be updated, a bounding box around  $B$  (see Figure 2) is defined, which is initially set to the whole image. Each bisection hyperplane found might then reduce the size of the bounding box or not. The size of the bounding box is also used to define the maximum distance to search along each search-line because a new bisection hyperplane should partially be inside the bounding box to have any effect. Bisection lines closer to the origin  $B$  give a larger reduction of the bounding box than lines further away. Since in general a number of object points from the same object are close to  $B$ , they are located first by searching a small area around  $B$ . Then the defined search-lines are searched further out.

When the search process is finished, a scan over the bounding box is made. For each scan-line, the bisection hyperplanes are then used to define a minimum and maximum value on the scan-line such that only pixels in that range have to be updated according line (4) of Algorithm 2. Exceptions are the bisection hyperplanes parallel to the main axis of the image, they reduce the bounding box and have no further effect on each scan-line.

Besides a suitable selection of the search-lines and of the size of the small area around each  $B$  to handle first, several other methods can be used to speed up FEED:

- Stopping the search process as soon as the remaining bounding box is small enough.
- Taking larger than 1 steps along the search-lines.
- Separate handling of special cases, like the reduction of the bounding box to a hyperplane.
- Saving certain information when moving from border pixel to border pixel.

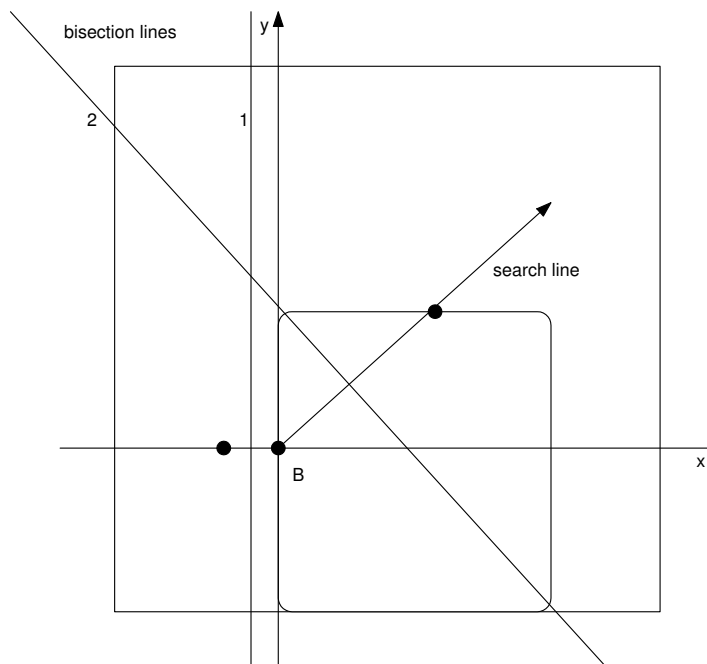
The  $ED$  in line (4) can be retrieved from a pre-computed matrix  $M$  with a size equal to the image size, for 2D:

$$ED(q, p) = M(|x_q - x_p|, |y_q - y_p|)$$

Due to the definition of  $D(p)$ , the matrix  $M$  can be filled with any non-decreasing function  $f$  of  $ED$ :

$$f(D(p)) = \min(f(D(p)), f(ED(q, p))).$$

For instance, the square of  $ED$  that provides an exact representation of the  $ED$  and allows the use of integers in the calculation, which makes it faster. Alternately, one can use floating point or truncated integer values when the final  $D(p)$  is used in such format in the further processing of the image.



**Figure 2.** Principle of the bounding box and search line. Originally the bounding box around the current boundary pixel  $B$  is equal to the whole image. The immediate left neighbor pixel of  $B$  generates bisection line 1. This reduces the bounding box from the left to the vertical line through  $B$ . The object point on the search line generates bisection line 2 that reduces the bounding box to the small box indicated with rounded corners. In the update process only the pixels in the small box that are on or to the left of bisection line 2 have to be considered.

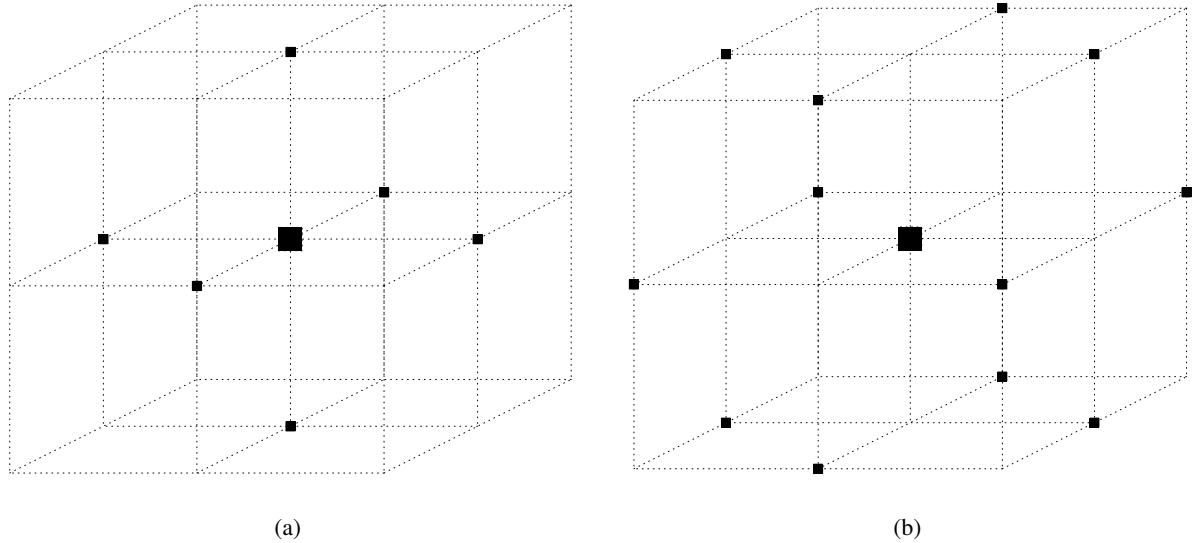
### 3. 3D-FEED IMPLEMENTATION ISSUES

For the 2D implementation,<sup>2,8</sup> the use of pre-computed EDs with floating point or truncated integer values stored in a matrix  $M$  provided a considerable speedup. This was not the case for 3D where it is about 30% faster to recalculate the square of  $ED$  in line (4) of Algorithm 2 each time and then at the end convert  $D(p)$  to its final floating point or integer format. This is due to the large number of caches that are present in current computer systems, which makes access time to memory locations effectively not uniform. Matrices and images are stored in a “non-local” way: if voxel  $v(x,y,z)$  is stored at memory location  $m(l)$ ,  $v(x+1,y,z)$  is stored at  $m(l+1)$ ,  $v(x,y+1,z)$  at  $m(l+width)$ , and  $v(x,y,z+1)$  at  $m(l+width*height)$ . The use of the extra matrix  $M$  leads to too many reloads of cache lines, making it slower than recalculating the  $ED$ .

To keep track of the update area in the 2D implementation, the maximum  $x$  and  $y$  values of each quadrant around the current  $B$  were used. This scheme was extended in 3D to using the maximum  $x$ ,  $y$  and  $z$  values of the 8 octants around  $B$ . However, for the update of each octant a separate run in the  $z$  direction has to be made. Therefore, we tried using a single bounding box around  $B$  and keeping track of its minimum and maximum values instead of using maximum values per octant. This allows the use of only one run in the  $z$  direction in the update process. The latter approach showed to be more than 10% faster than using the octant scheme since it generates less cache misses.

In 3D, a border voxel  $B$  is an object voxel, which has one of its six 6-connected voxels in the background (see Figure 3a). Searching for them proceeds along scan-lines in the  $x$  direction. When a border pixel  $B$  is found, its 6-connected voxels immediately provide a large cut on the bounding box, unless  $B$  is an isolated voxel. As it often occurs that this reduces the bounding box to a single line along one of the main axes, this is detected and handled separately, using only the relevant parts of the following procedure.

If a 6-connected voxel is in the background, a search for an object voxel is made in that direction. However, for the negative  $x$ ,  $y$ , or  $z$  direction it was faster to save the position of the closest object voxel from previous border voxels in respectively a variable, a (one dimensional) vector, and a 2D matrix. In many cases, the bounding box is



**Figure 3.** (a) A voxel (in the center) and its six 6-connected voxels. (b) A voxel (in the center) and its twelve 12-connected voxels.

reduced to a plane, handling this as a separate case provides a speedup. In other cases, the volume of the bounding box is less than a number of voxels (currently set to 25); then, directly updating it completely is fastest.

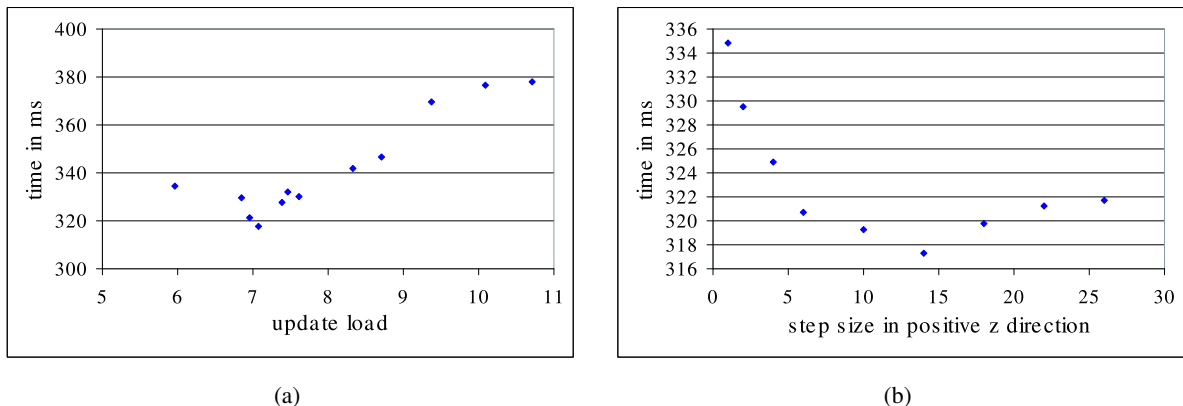
The twelve 12-connected neighbors of  $B$  (see Figure 3b) that have a line in common with  $B$  are checked for being an object voxel. If this is the case, that voxel defines a bisection plane and it is immediately checked whether or not the bounding box can be reduced in size. In addition, the combination with other bisection planes, not being parallel to the main axes, is taken into account, because this can provide an extra reduction of the size of the bounding box. Then the remaining 8-neighbors of  $B$  are tested, followed by a test on the volume of the remaining bounding box. Next, if the direct neighbor was a background pixel, a search for object pixels is done along lines, defined by the 12-connected and 8-connected neighbors. The search is not done if the volume of the bounding box is less than a number of voxels (currently set to 120).

For further reduction of the size of the bounding box, we have tried other search-lines, such as in the direction  $(1,1,2)$  and rotations thereof. In contrast to 2D images, this did not provide a reduction in execution time. Instead, we concentrated on trying to reduce the size of a bounding box when it is too large in the  $z$  direction, because that will have the largest effect on the executing time due to the effectively, non-uniform access time to the memory. Indeed, searches for object pixels along search-lines parallel to the  $x$  and  $y$  axes provided sufficiently large reductions in the  $z$  direction that the total execution time is reduced. These searches are only performed when the size in the positive or negative  $z$  direction is larger than a number of voxels (currently set to 12). A similar method is used for bounding boxes that are larger than 25 voxels in the positive or negative  $y$  direction.

This balancing of search time and update time is illustrated in Figure 4(a). The update load is the average number of times that an object voxel is updated. As, starting from the right of the figure, more search-lines directions are added, the update load becomes smaller. The time gained by updating less voxels, is larger than the extra search and bookkeeping time, thus the total time also reduces. At a certain point the update load still reduces but the added search and bookkeeping time is larger than the time gained by the reduction in the update load. Therefore the total time increases again.

During the update process, as described in Section 2, it makes no sense to calculate minimum and maximum values for each scan-line in the  $x$  direction when the scan-line is already small. This parameter was set at 10 voxels. Note that certain bisection planes define cuts on the range of  $y$  values while others define cuts on the  $x$  values, both of which are only dependent on the  $z$  value. These cuts are always applied.

A reduction in time was further obtained by taking larger than 1 steps along the search directions. These steps were set differently for the different directions and were larger for the  $z$  direction than for the  $y$  and  $x$  directions.



**Figure 4.** Balancing search time and update time. (a) Execution time in ms versus the update load for various search strategies. The update load is the average number of times that an object voxel is updated. (b) Execution time in ms versus the step size in the positive z (0,0,1) direction.

This is shown in Figure 4(b), a step size of 12 in the (0,0,1) direction gives the minimum execution time. Not shown is the update load, which is minimal for a step size of 1 and which increases with the step size.

Consequently, there are more parameters to adjust in the 3D than in the 2D implementation. Adjusting them was not difficult because there appeared to be no large correlations between them. Also the effect of a setting is not critical, a sequence of 3 to 5 trials is sufficient to find the optimal setting. After the parameters were set for a subset of the  $128 \times 128 \times 128$  images on the AMD computer (see Section 4), this parameter set was compared with optimal sets for the other combinations of image size and execution platform. The obtained execution times were within 5% of the optimal execution times. Therefore, we used this particular parameter set for all the combinations.

In conclusion, we can say that the main difference between the 2D and 3D implementations is, that in the latter the jumping through memory has a larger effect. To take this into account different search and bookkeeping strategies were introduced. The optimal setting of the larger parameter set appeared not to be difficult.

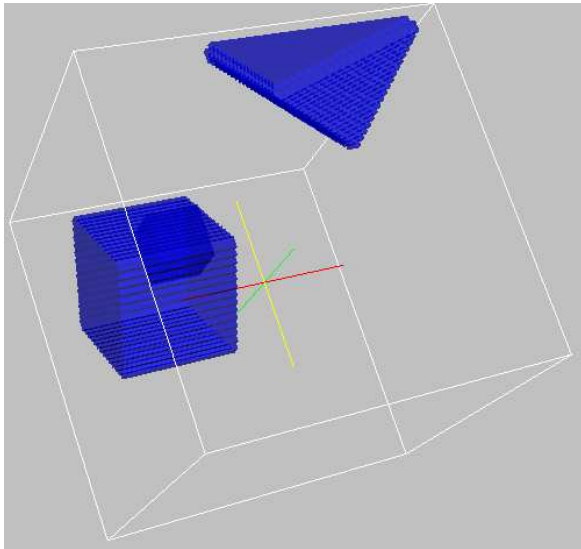
## 4. BENCHMARK

### 4.1. Test images

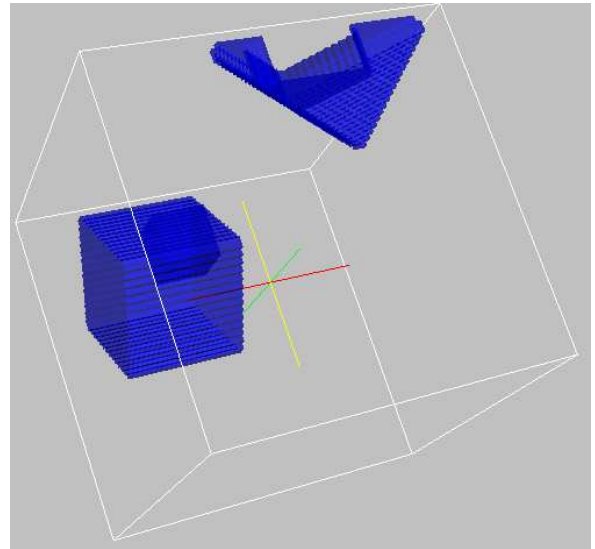
To aid the development of 3D-FEED and to compare 3D-FEED with other distance transformations a large number of test images were generated. For each iteration step of the generation process the number of objects in the image was randomly chosen between a minimum and maximum value.

The type of an object was chosen randomly out of 8 possibilities: 4 rotations of a cube, a ball, and three ellipsoids (e.g.,  $2x^2 + y^2 + z^2 \leq r^2$ ). In addition, the radius of each object was randomly chosen from a range. The voxels were combined both using an “or” and a “xor”, the latter produced images with possible holes in the objects. By roughening the surfaces of the objects in a random way, two further images were generated in each step. For this, each border pixel was with a probability of 25% changed into a background pixel. In addition, each border pixel had with a probability of 25% a randomly chosen neighboring background pixel changed into an object pixel. Finally, isolated object and background pixels were removed from the generated images.

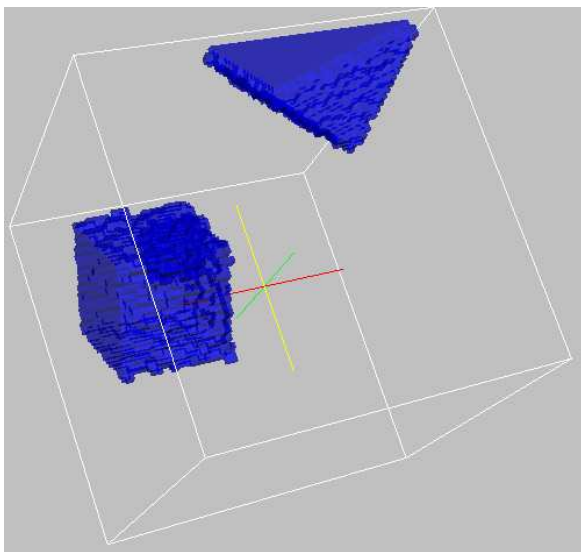
Two sets of images with size  $64 \times 64 \times 64$  were generated using a number of objects between 4 and 20, resp. 16 and 32, and a radius between 4 and 16, resp. 2 and 14. Also two sets of images with size  $128 \times 128 \times 128$  were generated using a number of objects between 14 and 34, resp. 28 and 68, and a radius between 8 and 24, resp. 7 and 20. All four sets had 64 iterations, thus 256 images each. The generation parameters were chosen such that the number of object pixels varied between 5 and 40% of the total number of pixels in the image. In Figure 5 some examples of the generated images are shown.



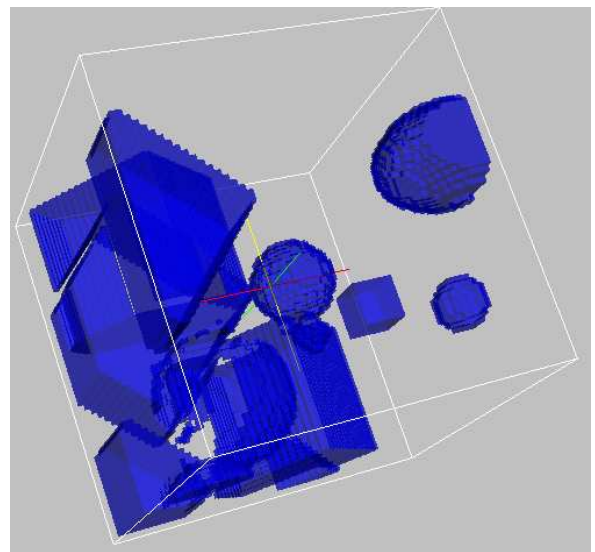
(a)



(b)

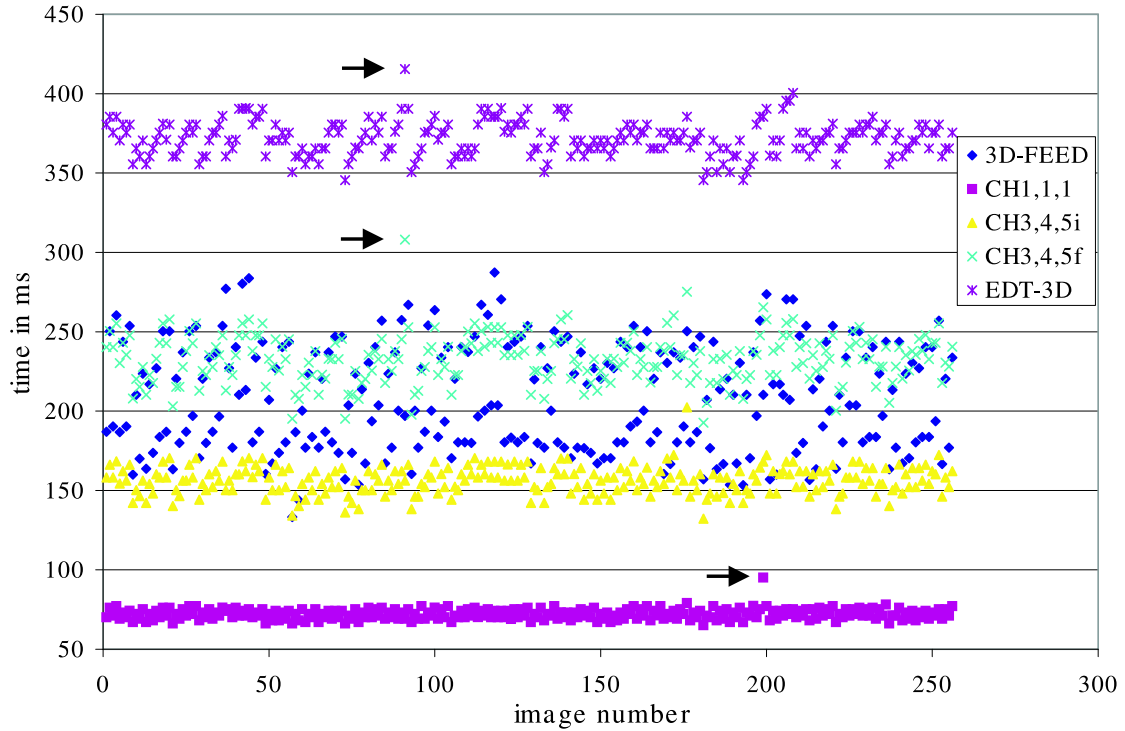


(c)



(d)

**Figure 5.** Example images: (a) Generated with the “or” method. (b) The same generation step but with “xor”, note the hole in the top right object. (c) The roughened image in (a). (d) Another image generated with “xor” but with more objects.



**Figure 6.** Execution time in ms for 256  $128 \times 128 \times 128$  images for the five 3D ED algorithms.

## 4.2. Test bed

Two execution platforms were used for testing. The first, further referred to as AMD, was a desktop PC with an AMD Athlon XP<sup>®</sup> 1666 MHz processor having 64kB L1 cache, 256kB L2 cache and 512 MB memory. The second, further named DELL, was a DELL Inspiron<sup>®</sup> 510m laptop. It had an Intel Pentium<sup>®</sup> M 725 processor with a clock of 1600 MHz, no L1 cache, 2MB L2 cache and 512MB memory. On both platforms the Microsoft<sup>®</sup> Visual C++ 6.0 programming environment was used in the standard release setting. Please note that no effort was spent on reducing the execution time by varying compiler optimization parameters, by using pointers instead of indices or by exploiting the particular characteristics of a machine.

Timing measurements were performed by calling the Visual C++ `clock()` routine before and after a repeated number of calls to a distance transformation routine on an image. As the `clock()` routine returns elapsed time with a granularity of  $10ms$  the number of repeats is set such that the total elapsed time is more than  $1s$ . The error on a single elapsed time measurement is then less than 2%, provided that the computer was not busy with other tasks like processing interrupts. That this happens is shown in Figure 6 where the elapsed time is shown for the 256 images of the first  $64 \times 64 \times 64$  set for the five 3D ED algorithms used in Section 4. The arrows show clearly too high times, an indication that the computer was then also serving other tasks. The number of times that this happens, was reduced by disabling local area communication devices. For the elapsed times reported in Section 4, these erroneous measurements were removed by hand. The effect of this procedure on the average times was less than 1%.

During the development of the 3D-FEED implementation, as described in Section 3, a faster measurement strategy was used. The calls to the `clock()` routine are done before and after the 3D-FEED processing of a large number of images. This is done twice; hence, it enabled us to verify whether or not the computer is busy with other tasks. In the case it was busy with other tasks, the measurement is simply repeated.

## 4.3. The algorithms

We compared FEED with four other algorithms:

1. The city-block distance (CH1,1,1), as introduced by Rosenfeld and Pfaltz<sup>1</sup>, but extended to 3D. It provides crude but fast approximations of the ED.



2. The Chamfer 3,4,5 distance (CH3,4,5i), from Svensson and Borgefors<sup>5</sup>, using the integer (i) weights. It provides a more accurate approximation of the ED.
3. The Chamfer 3,4,5 distance (CH3,4,5f), from Svensson and Borgefors<sup>5</sup>, using floating point (f) weights.
4. EDT-3D: a changed version of the two-scan based algorithm of Shih and Wu<sup>10</sup> that produces a semi-exact ED map (see also Section 1). The original algorithm saves during the scans the square of the intermediate EDs for each voxel and decompose them when needed into the sum of three squared integer values, which denote distances along principal axis. If more than one decomposition is possible, neighboring voxels are decomposed to find the right decomposition. Since Shih and Wu<sup>10</sup> did not describe their decomposition method, which is rather time consuming, we explored two alternatives. The three values are initialized and then updated during the scans and saved combined into an integer that is stored into a 3D matrix. The intermediated squared EDs for each voxel are either saved into a 3D matrix and updated as in the original algorithm or when needed recalculated from the three values. The latter alternative clearly proved to be the fastest method; hence, we used this alternative.

#### 4.4. Results

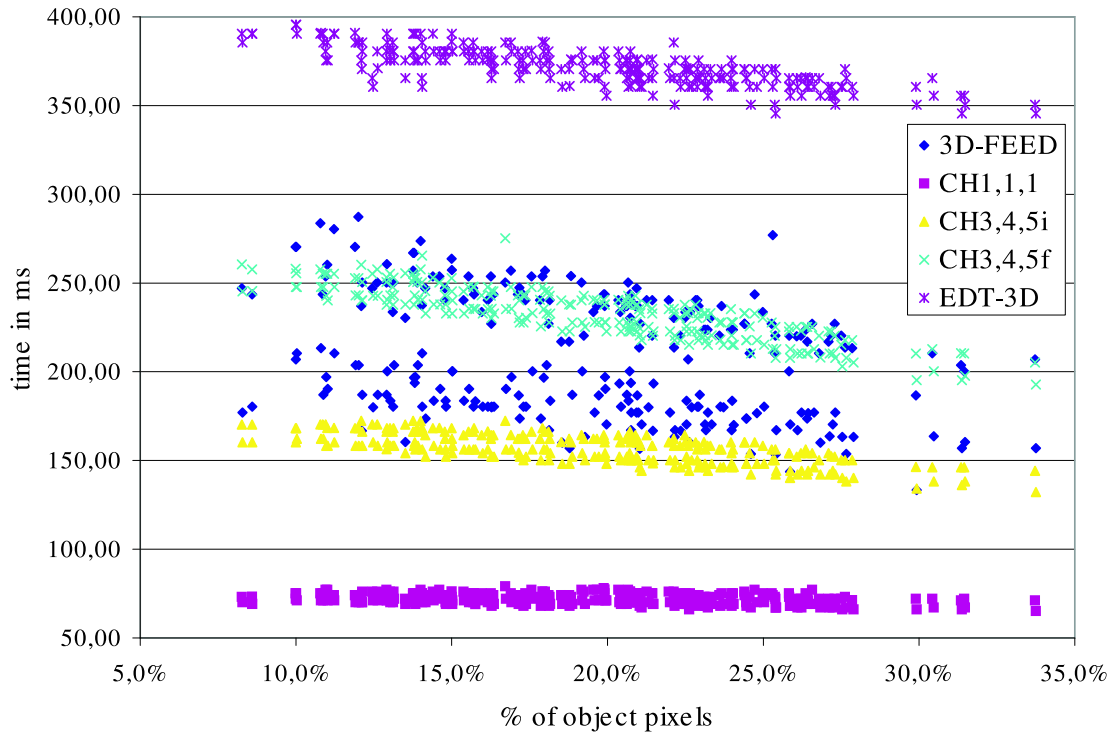
In Table 1, the obtained timing and accuracy results are provided for the different image sizes and execution platforms. It shows that 3D-FEED gives the exact  $ED$  and is around a factor 2 faster than the EDT-3D method adapted from Shih and Wu<sup>10</sup>, which gives a wrong result on about 3 to 4% of the voxels. Regarding the Chamfer methods of Svensson and Borgefors<sup>5</sup>, 3D-FEED is comparable in speed but much more accurate. Moreover, it is only a factor 2 to 3 slower than the crude city-block approximation.

For the CH methods a large percentage of the voxels is in error. The CH1,1,1 method shows the expected maximum relative error of  $(3 - \sqrt{3})/\sqrt{3} = 73.21\%$ . The errors for the CH3,4,5 methods are much smaller than for the CH1,1,1 methods, the implementation with the floating point weights has a lower error than the version with the integer weights but is about a factor 1.5 slower. For the semi-exact EDT-3D method most of the voxels have the correct ED, the maximum errors are comparable with the CH3,4,5 methods.

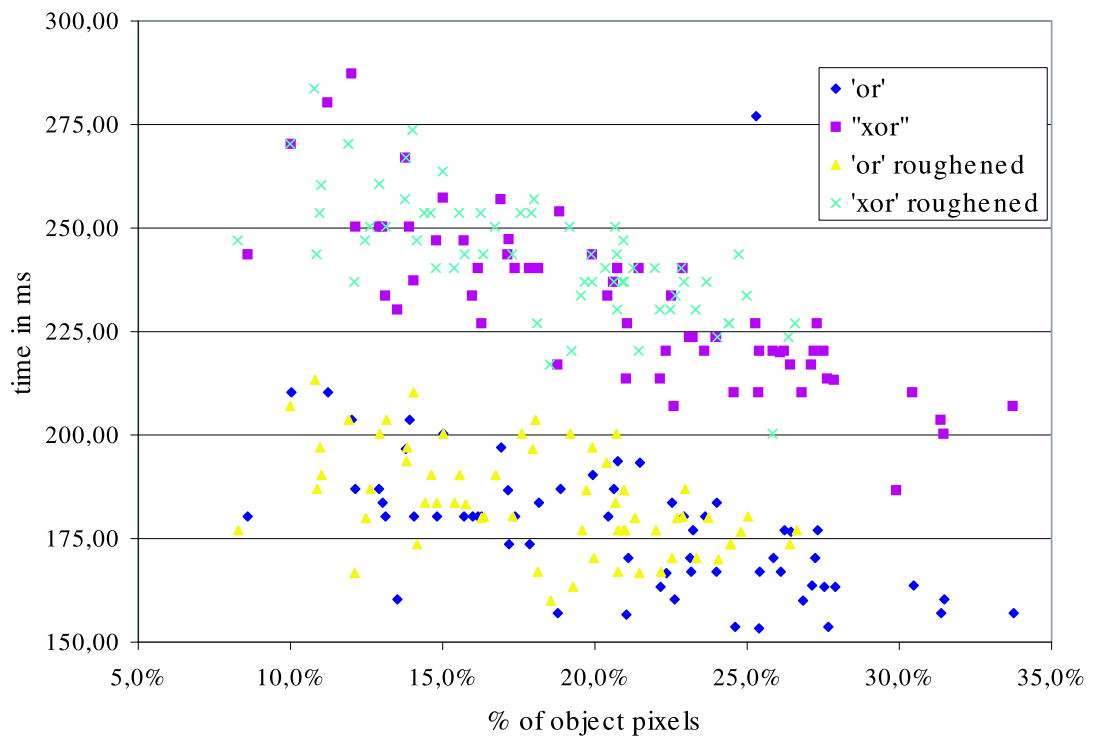
All methods, except 3D-FEED, use a forward and backward raster scan over the image and at each voxel a limited number of neighboring voxels (3 for CH1,1,1 and 13 for the other ones) are considered. The speed of these method is thus proportional to the number of voxels in the image, as is confirmed by our measurements.

**Table 1.** Timing (in ms) and accuracy results. The % of wrong pixels provides the percentage of pixels that have received a different distance than the exact  $ED$ . The average and maximum absolute error are in units of voxels. The relative average and maximum error is defined as the received  $ED$  divided by the true  $ED$  and is given as a percentage. The time ratio is the average time for the large images divided by the average time for the small images.

|                       |                        | 3D-FEED  | CH1,1,1 | CH3,4,5i | CH3,4,5f | EDT-3D   |
|-----------------------|------------------------|----------|---------|----------|----------|----------|
| 64x64x64 images       | ADM average time       | 25.0 ms  | 11.9 ms | 26.4 ms  | 38.2 ms  | 64.4 ms  |
|                       | DELL average time      | 23.4 ms  | 8.5 ms  | 18.9 ms  | 29.4 ms  | 45.4 ms  |
|                       | % of wrong pixels      | 0.00%    | 70.51%  | 68.24%   | 83.65%   | 3.29%    |
|                       | average absolute error | 0.00     | 3.02    | 0.35     | 0.32     | 0.01     |
|                       | maximum absolute error | 0.00     | 34.04   | 5.24     | 4.47     | 2.30     |
|                       | average relative error | 0.00%    | 27.34%  | 3.12%    | 3.11%    | 0.06%    |
|                       | maximum relative error | 0.00%    | 73.21%  | 10.55%   | 8.80%    | 9.38%    |
| 128x128x128 images    | ADM average time       | 320.3 ms | 96.2 ms | 210.2 ms | 299.8 ms | 531.2 ms |
|                       | DELL average time      | 209.5 ms | 72.4 ms | 155.8 ms | 230.9 ms | 371.0 ms |
|                       | % of wrong pixels      | 0.00%    | 69.67%  | 67.88%   | 80.45%   | 4.41%    |
|                       | average absolute error | 0.00     | 3.79    | 0.44     | 0.40     | 0.01     |
|                       | maximum absolute error | 0.00     | 46.46   | 9.31     | 6.88     | 4.72     |
|                       | average relative error | 0.00%    | 26.38%  | 2.93%    | 2.89%    | 0.08%    |
|                       | maximum relative error | 0.00%    | 73.21%  | 10.55%   | 8.80%    | 9.82%    |
| comparing image sizes | AMD time ratio         | 12.8     | 8.1     | 8.0      | 7.9      | 8.3      |
|                       | DELL time ratio        | 8.9      | 8.5     | 8.3      | 7.8      | 8.2      |



**Figure 7.** Execution time in ms as function of the percentage of object pixels in the first set of  $128 \times 128 \times 128$  images for the five 3D ED algorithms.



**Figure 8.** Execution time of 3D-FEED in ms as function of the percentage of object pixels for different ways in which the test images are generated.

For 3D-FEED this is not the case, for each border voxel in principle the whole image could be scanned for searching and updating. Thus a simple theoretical limit is that 3D-FEED is proportional to the square of the number of voxels. In practice this is not the case for various reasons. The time per voxel for finding border voxels is much lower than the time per voxel during the update process. Further the more object and border voxels there are the less time is needed for the update scan. Instead of the theoretical maximum factor of 64 between the execution time of 3D-FEED for the two image sizes, a factor of 12.8 for the AMD and 8.9 for the DELL machine was measured. The difference between the two factors can be largely attributed to the different cache systems of the platforms which gives the DELL machine an effectively more uniform memory access.

In Figure 7, the execution time for the five algorithms on the DELL platform is shown as a function of the percentage of object pixels in the  $128 \times 128 \times 128$  images. For all methods the execution time decreases linearly with increasing number of object pixels. This can be explained by the fact that more work is performed on background pixels than on object pixels. 3D-FEED also shows a larger variation in execution time than the other methods and there are clearly two bands visible. The bands are also to a lesser extent visible for the other methods

To study this further in Figure 8 the execution time of 3D-FEED is given as a function of the percentage of object pixels for different ways in which the test images are generated. The images generated with the “or” method (see Section 4.1) have a lower execution time than the images generated with the “xor” method. Roughening of the images does not make a difference. Not shown here is also no difference between the two image sets for each image size. The same effect is analyzed to be present for the other ED methods. It is therefore not something which is particular to the 3D-FEED method, but is intrinsic to the images. The larger variation in the execution time of 3D-FEED is due to the distribution of the object pixels over the image, which influences the speediness of the search strategy employed by 3D-FEED.

## 5. DISCUSSION

In Section 2, we have reformulated the principles of the Fast Exact Euclidean Distance (FEED) transformation, as originally defined by Schouten and Van den Broek<sup>8</sup> and refined by Schouten, Kuppens, and Van den Broek<sup>2</sup>. From this point on, we have developed a 3D implementation, which is described in Section 3. Compared to the 2D implementations<sup>2, 8</sup> certain strategies were chosen differently, in order to reduce the number of cache misses in current computer systems.

In Section 4, we compared the performance of 3D-FEED on a large set of randomly generated, object-like images with four other methods. 3D-FEED is a factor 2 faster than the algorithm of Shih and Wu<sup>7</sup>, even after adapting their method to increase the speed of it, and it provides an exact ED instead of a semi-exact ED. Further, 3D-FEED has about the same speed as the chamfer approximations of the ED from Svensson and Borgefors<sup>5</sup> and is only a factor 3 slower than the crudest and fastest approximation of the ED: the city-block distance.

It is also shown that the execution time of 3D-FEED is more dependent on the content of the image than the execution time of the other methods. A further study of this effect could lead to a more adaptable and faster search strategy for 3D-FEED. Further, the FEED principle can also be adapted to images with “non-square” voxels, a kind of image often used in medical application and produced by tomographic devices or by confocal microscopes<sup>13</sup>.

3D-FEED can be employed in a range of settings; e.g., trajectory planning, neuromorphometry<sup>14</sup>, skeletonization, Bouligand-Minkowsky fractal dimension, Watershed algorithms, robot navigation<sup>2</sup>, video surveillance<sup>15</sup> and Voronoi tessellations<sup>12</sup>. Especially for those applications where both speed and precision of the algorithm are of the utmost importance, 3D-FEED is *the* solution.

## REFERENCES

1. A. Rosenfeld and J. L. Pfaltz, “Distance functions on digital pictures,” *Pattern Recognition* **1**, pp. 33–61, 1968.
2. Th. E. Schouten, H. C. Kuppens, and E. L. van den Broek, “Timed Fast Exact Euclidean Distance (tFEED) maps,” *Proceedings of SPIE (Real Time Imaging IX)* **5671**, pp. 52–63, 2005.
3. G. Borgefors, “Distance transformations in digital images,” *Computer Vision, Graphics, and Image Processing: An International Journal* **34**, pp. 344–371, 1986.
4. G. Borgefors, “On digital distance transforms in three dimensions,” *Computer Vision and Image Understanding* **64**(3), pp. 368–376, 1995.

5. S. Svensson and G. Borgefors, "Distance transforms in 3D using four different weights," *Pattern Recognition Letters* **23**(12), pp. 1407–1418, 2002.
6. O. Cuisenaire and B. Macq, "Fast euclidean transformation by propagation using multiple neighborhoods," *Computer Vision and Image Understanding* **76**(2), pp. 163–172, 1999.
7. F. Y. Shih and Y.-T. Wu, "Fast euclidean distance transformation in two scans using a  $3 \times 3$  neighborhood," *Computer Vision and Image Understanding* **93**(2), pp. 195–205, 2004.
8. Th. E. Schouten and E. L. van den Broek, "Fast Exact Euclidean Distance (FEED) Transformation," in *Proceedings of the 17th IEEE International Conference on Pattern Recognition (ICPR 2004)*, J. Kittler, M. Petrou, and M. Nixon, eds., **3**, pp. 594–597, (Cambridge, United Kingdom), 2004.
9. C. Fouard and G. Malandain, "3-D chamfer distances and norms in anisotropic grids," *Image and Vision Computing* **23**(2), pp. 143–158, 2005.
10. F. Y. Shih and Y.-T. Wu, "Three-dimensional Euclidean distance transformation and its application to shortest path planning," *Pattern Recognition* **37**(1), pp. 79–92, 2004.
11. J. H. Takala and J. O. Viitanen, "Distance transform algorithm for Bit-Serial SIMD architectures," *Computer Vision and Image Understanding* **74**(2), pp. 150–161, 1999.
12. E. L. van den Broek, Th. E. Schouten, P. M. F. Kisters, and H. C. Kuppens, "Weighted Distance Mapping (WDM)," in *Proceedings of the IEE International Conference on Visual Information Engineering (VIE2005)*, N. Canagarajah, A. Chalmers, F. Deravi, S. Gibson, P. Hobson, M. Mirmehdi, and S. Marshall, eds., pp. 157–164, Wrightsons - Earls Barton, Northants, Great Britain, (Glasgow, United Kingdom), 2005.
13. I.-M. Sintorn and G. Borgefors, "Weighted distance transforms for volume images digitized in elongated voxel grids," *Pattern Recognition Letters* **25**(5), pp. 571–580, 2004.
14. Y. Lu, T. Jiang, and Y. Zang, "Region growing method for the analysis of functional MRI data," *NeuroImage* **20**(1), pp. 455–465, 2003.
15. Th. E. Schouten, H. C. Kuppens, and E. L. van den Broek, "Video surveillance using distance maps," *Proceedings of SPIE (Real-Time Image Processing III) [in press]*, 2006.