IMM
DEPARTMENT OF MATHEMATICAL MODELLING

Technical University of Denmark
DK-2800 Lyngby – Denmark

J. No.
31-10-2000
JAB

ON THE IMPLEMENTATION OF FAST MARCHING METHODS FOR 3D LATTICES

J. Andreas Bærentzen

TECHNICAL REPORT

IMM-REP-2001-13

IMM

Abstract

This technical report discusses Sethian's Fast Marching Method and its higher accuracy variant. Both methods may be used to compute the arrival times at the points of a discrete lattice of a front which is monotonously expanding. Applications of the method include arrival time computation and the construction of distance fields for 2D or 3D objects.

The main aim of this report is to supplement the available papers with a practical guide to the implementation of the method. Through a simple example the Fast Marching Method and its high accuracy variant are compared with regard to speed and precision.

1 Introduction

This report is about the *Fast Marching Method* (FMM) and its high accuracy variant. The FMM is a technique for tracking the evolution of an expanding front. In this context, a front is a closed surface in 3D (or a closed curve in 2D) which separates an interior and an exterior region. The FMM is simply a technique for computing the arrival time of a front at the points of a discrete lattice. If the front is simply a closed curve in 2D, and the lattice is a pixel raster, the FMM assigns to each pixel the time at which the expanding curve hits the pixel (see Figure 1). The method applies only to cases where the front is uniformly expanding since the arrival time of the front is uniquely defined only in these cases. In this report, I focus on the 3D case where we are dealing with expanding surfaces in a voxel lattice. However, everything applies to 2D, and many of the illustrations are in 2D as well.

One application of the method is the computation of distance fields. Given a surface in 3D, the FMM can be used to compute its distance field, since, if the front evolves at unit speed, the arrival time corresponds to the distance. 3D distance fields have a number of applications in volume graphics, for instance morphing [3] and hyper-texturing [7], and there are several methods for computing distance fields:

The Chamfer distance transforms [1] is a class of algorithms for computing distance transforms – including (pseudo) Euclidean distance transforms. The algorithms visit each voxel twice – once for each of two loops over the entire volume. At each voxel the distance is propagated from neighbouring

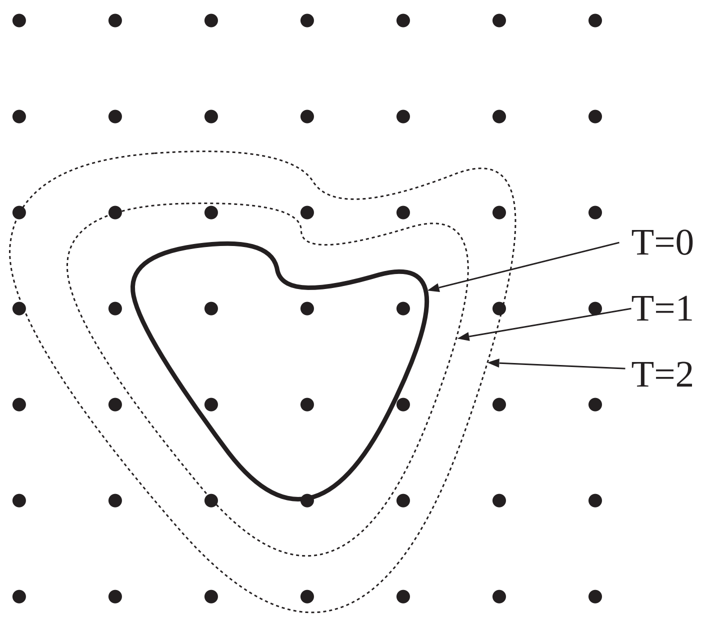

Figure 1: Expanding 2D front (i.e. closed curve) at times $T = 0$, $T = 1$, and $T = 2$.

voxels according to a distance matrix. The VCVDT algorithm [8] is similar but based on propagating closest point vectors rather than distances. Sethian's Fast Marching Methods [10, 12, 11] builds the distance field from a boundary condition by solving the Eikonal equation for all voxels in a systematic way. A variation of the FMM by Breen and Mauch [2] builds the distance volume in the same way but computes the distances differently. These alternatives are mentioned mostly for completeness, since in this report we shall concern ourselves only with the FMM.

There is a (mostly) second order implementation of the Fast Marching Method which is known as the High Accuracy Fast Marching Method [13] (FMMHA). The high accuracy method is far better than the normal version

and just as simple to implement. In fact, there is little difference between the implementations. This is also why the emphasis in this report is on the normal version of the Fast Marching Method. Once the hurdles in implementing this scheme are overcome, it it easy to proceed to the high accuracy method.

Both the FMM and the FMMHA are discussed with a strong emphasis on the practical implementation. This is motivated by the concern that certain details make the method hard to implement, and that a report with a very practical outlook would be a good supplement to the existing literature. On the other hand, the theoretical niceties are largely ignored, and this report should not be read alone but together with the publications by Sethian et al. [10, 12, 11, 13].

In the next section, Section 2, I discuss the structure of the algorithm known as the Fast Marching Method. In Section 3 details are provided on how to compute the distance values. In Section 4 we shall see why the high accuracy variant is needed and how the computation of distances is changed. Section 5 contains a brief discussion, and pseudo–code is provided in Appendix A.

2 The Fast Marching Method

As mentioned, the FMM can be described as a family of schemes for computing the evolution of fronts. Things become interesting when the front evolves over time. In the context of the Fast Marching Method, we assume that the front evolves by motion in the normal direction as illustrated in Figure 1. The speed does not have to be the same everywhere, but the speed must always be non–negative. At a given point, the motion of the front is described by the equation known as the Eikonal equation

$$||\nabla T(\mathbf{x})||F(\mathbf{x}) = 1 \tag{1}$$

where T is the arrival time of the front at point $\mathbf{x}$ and $F \geq 0$ is the speed of the front at point $\mathbf{x}$. Because the front can only expand, the arrival time T is single valued.

The scheme is illustrated in Figure 2 where a front emanates from a single point with speed $F = 1$ everywhere. Since the front has equal speed in

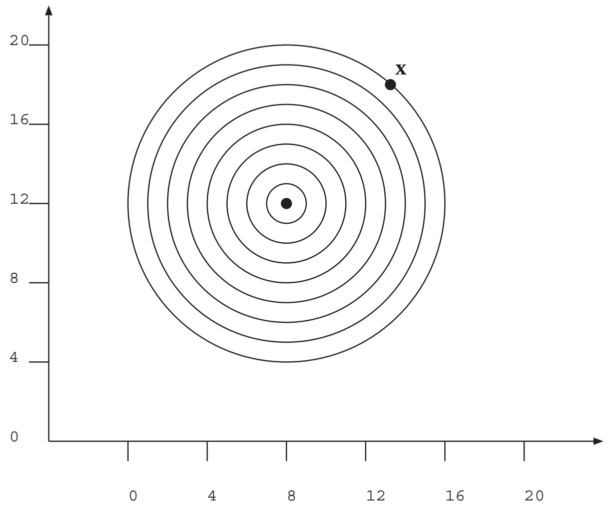

Figure 2: Front crossing point **x** at time $T = 8$

all directions it becomes circular. The front traverses the point **x** at time $T = 8$.

In the figure, the front evolves from a point, but, of course, this need not be the case. In general, the front may evolve from any sort of boundary.

The Fast Marching Method operates on a lattice. Although the method is more general, for simplicity, we will restrict our attention to voxel lattices of the usual sort, i.e. isotropic, rectangular 3D lattices. We will generally assume that $F = 1$ everywhere. In this case, the fast marching method simply propagates the shortest distance to the boundary to all other points in the lattice. In the context of volume graphics this is most frequently what we need.

2.1 The Algorithm

The philosophy of the method is to work outwards from an initial condition. Say we are computing the distance field from a point. In that case, we would typically use a single voxel as the initial condition. However, in general, many voxels can form the initial condition. The voxels that form the initial condition are *frozen*, and we compute distances at their neighbours. Voxels that have computed distances but are not yet frozen are said to be *narrow band* voxels. For each iteration of the central loop of the algorithm, the narrow band voxel having the smallest distance value is frozen, and distances are computed for its neighbours. Frozen voxels are used to compute the values of other voxels but are never computed again. Thus, we can see the method itself as a front of narrow band voxels that propagates from the initial condition, freezing voxels as it moves along.

An important data structure used by the algorithm is a binary heap. The key to the binary heap is distance value and each element in the binary heap is a pointer to a voxel. We recall that in a binary heap [9, 4], the element having the smallest (or largest) key is always on top. This makes the binary heap a good choice, because in the following, we need to be able to find the narrow band voxel having the smallest distance value.

I now describe the algorithm in more detail:

Initialization Before the loop, we tag the initial condition voxels as being *frozen*. For each frozen voxel, we visit all the neighbours in a six–connected neighbourhood (see Figure 5) and at each of these neighbours, the distance is computed using only information from frozen voxels. The precise method for computing distances is discussed in the next section. The neighbouring voxel is now tagged as a *narrow band* voxel and it is inserted into the binary heap.

Loop The first step of each iteration is to extract from the top of the heap the narrow band voxel that has the smallest distance. The smallest distance voxel is then tagged as being frozen, i.e. we consider its distance value to be computed, and for each neighbour that is not frozen, we compute the distance, tag the neighbour as being a narrow band voxel and insert it into the heap. Of course, the neighbour may already be in the narrow band. In this case, we merely recompute the value and change its position in the heap to reflect the new value. Finally, we loop back and extract the new smallest distance narrow band voxel. (The loop is illustrated in figure 4.)

In Figure 3, the initialization and loop steps are shown in relatively high level pseudo–code.

2.1.1 Implementation Details

We need to be able to find the heap elements that correspond to voxels whenever the distance value of a voxel changes. In order to find the corresponding heap element, Sethian suggests [10] that each narrow band voxel in the lattice should contain a pointer to the corresponding heap element. However, that is not in itself enough, since elements in the heap might change their positions when they have been recomputed. This means that the pointers in the voxel lattice which point to elements in the heap must be updated whenever the heap is changed. This entails that the heap and the lattice cannot be entirely separate data structures which is unfortunate from a software engineering perspective. Hence, it is a good idea to use a heap consisting of a list of values and a list which is a permutation of their ordering. The permutation list contains pointers to the value list and vice versa. This heap implementation is suggested in [9]. When a value is inserted into the heap its position in the value list is never changed – only the permutations. Hence, the heap element pointers in the lattice never have to be changed and the heap does not require access to the lattice.

```
Initialization()
{
        for each voxel v in I
            {
                Freeze v;
                for each neighbour vn of v
                    {
                        compute distance d at vn;
                        if vn is not in narrow band
                            {
                                tag vn as narrow band;
                                insert (d,vn) in H;
                            }
                        else
                            decrease key of vn in H to d;
                    }
            }
}

Loop()
{
    while H is not empty
        {
            Extract v from top of H;
            Freeze v;
            for each neighbour vn of v
                if vn is not frozen
                    {
                        compute distance d at vn;
                        if vn is not in narrow band
                            {
                                tag vn as narrow band;
                                insert (d,vn) in H;
                            }
                        else
                            decrease key of vn in H to d;
                    }
        }
}
```

Figure 3: Pseudo–code illustrating the initialization and loop of the Fast Marching Method. The pseudo–code assumes the existence of a list of voxels I containing voxels whose distances are known and thus form the initial condition, and a binary heap H that is initially empty.

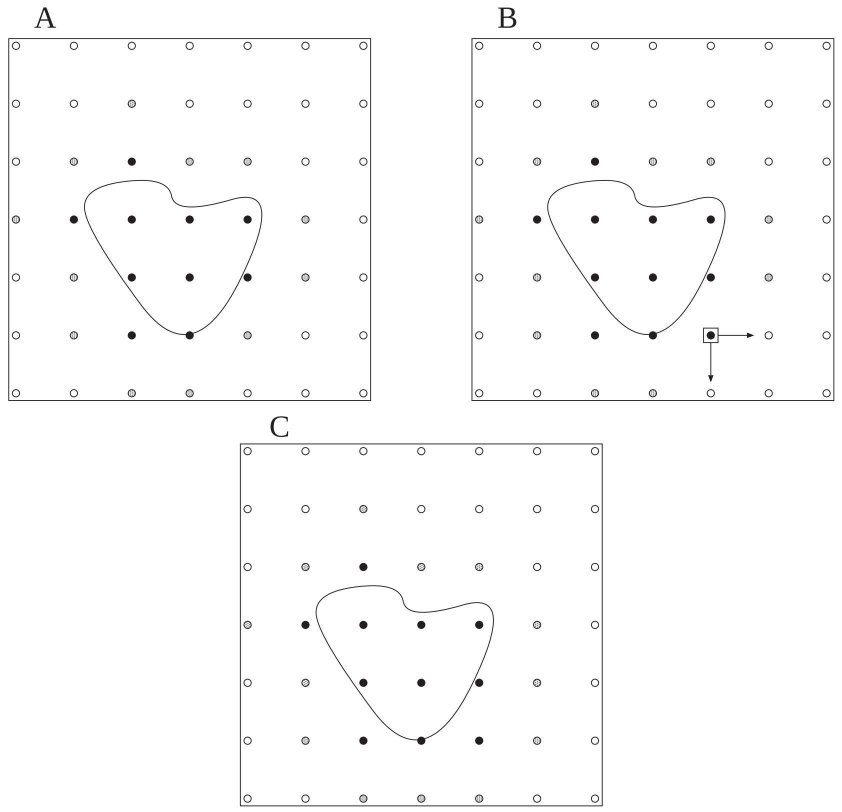

Figure 4: A shows the state after initialization. The voxels whose distances are known initially (I) are frozen (shown as black) while their neighbours are tagged as being in the narrow band (shown as grey). B shows the state at the beginning of an iteration of the loop. The voxel having the smallest distance (i.e. the voxel closest to the curve) has been frozen, and arrows point to its two non–frozen neighbours. C shows the state at the end of the iteration: The distance values at the two neighbours have been computed.

3 Computing Distances

Distances are computed by solving the Eikonal equation. In other words, we must find a distance value for the narrow band voxel so that the estimated length of the gradient $||\nabla T||$ is equal to $1/F$.

$$||\nabla T|| = 1/F \tag{2}$$

where we recall that T is the arrival time of the front and F is the speed. Sethian proposes the following formula (borrowed from the field of hyperbolic conservation laws) for the squared length of the gradient

$$||\nabla T||^2 = \left\{ \begin{array}{ll} \max(V_A - V_B, V_A - V_C, 0)^2 & + \\ \max(V_A - V_D, V_A - V_E, 0)^2 & + \\ \max(V_A - V_F, V_A - V_G, 0)^2 & \end{array} \right. \tag{3}$$

where V_A is the unknown distance value and $V_B, V_C, V_D, V_E, V_F, V_G$ are the distance values at the neighbouring voxels (in the six–connected neighbourhood). The stencil is illustrated in figure 5.

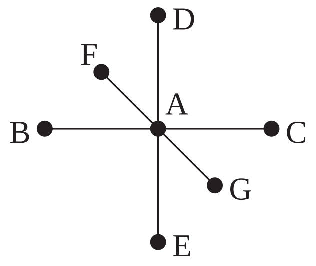

Figure 5: Stencil for the fast marching neighbourhood

To solve (2), we plug in (3) which leads to the following equation:

$$1/F^2 = \left\{ \begin{array}{ll} \max(V_A - V_B, V_A - V_C, 0)^2 & + \\ \max(V_A - V_D, V_A - V_E, 0)^2 & + \\ \max(V_A - V_F, V_A - V_G, 0)^2 & \end{array} \right. \tag{4}$$

To solve this equation, we look at each term of the form

$$\max(V_A - V_B, V_A - V_C, 0)^2$$

It is clear that we should choose to solve (4) using the smaller of the two values V_B and V_C for

$$V_B < V_C \implies V_A - V_B > V_A - V_C$$

In addition, we only use frozen values. If neither V_B nor V_C are frozen, this term drops out of the equation. It is possible to include non–frozen values in the computations, but tests indicate that it is detrimental to the quality of the solution[1].

With these things in mind, we form the quadratic equation. Assuming $V_B < V_C$, $V_E < V_D$, $V_F < V_G$, and that V_B, V_E, and V_F are frozen, the equation is

$$(V_A - V_B)^2 + (V_A - V_E)^2 + (V_A - V_F)^2 = F^{-2} \tag{5}$$

The largest solution (if there are two) to this equation is the one we want. This follows from the fact that V_A must be greater that the three known values (since they are frozen). If there are two solutions, it is easy to see that the smaller cannot fulfill this condition.

4 The High Accuracy Fast Marching Method

The precision of the FMM does leave something to be desired. A simple 2D example shows where the method goes wrong: In Figure 6 the front emanates from the frozen (black) vertex labeled 0, the distance has been computed at the two other frozen vertices, and the white vertex is being updated. From (4), we see that the value at the white vertex should be the larger solution to $(x-1)^2 + (x-1)^2 = 1vu$ which is $1 + \sqrt{1/2}vu$ Unfortunately, it is also clear that the correct distance is $\sqrt{2}vu$ which means that the value is wrong by almost 0.3 vu (vu means voxel unit, i.e. the distance between two adjacent voxels in the lattice). The error can be explained intuitively by observing that the FMM does not know the curvature of the front. If the front had been linear, the value would have been exact. This seems to indicate that the problem is worst when using the FMM to compute distance from high curvature boundary conditions.

[1]Using non–frozen values would also lead to situations where a voxel is used to update another voxel that has just been used to update itself. However, it appears that Sethian et al do use non–frozen values except in the higher accuracy version of the scheme [13] p. 96.

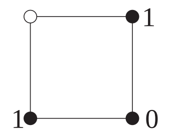

Figure 6: 2D illustration of the problem with the FMM

To explore the problem further, a simple experiment was conducted. The distance from a point to all voxels within a radius of 20 vu was computed. This yields a max error of 1.48 vu and a mean error of 0.89 vu. If the exact distances at all voxels in the 26–neighbourhood[2] of the centre voxel are precomputed, the results are only slightly better: The max error drops to 1.24 vu and the mean error drops to 0.73 vu.

This has motivated the implementation of the higher accuracy version of the scheme [13]. To explain this scheme, I have to change the notation a little bit: The normal FMM is based on the use of one sided derivatives computed using forward and backward differences:

$$\begin{aligned} D^{-x}G &= G[x,y,z] - G[x-1,y,z] = V_A - V_B \\ D^{+x}G &= G[x+1,y,z] - G[x,y,z] = V_C - V_A \end{aligned}$$

where D^{-x} and D^{+x} are the standard notation for backward and forward differences and G is the voxel lattice (note that we implicitly assume that the voxel distance is unit). The squared partial derivative of T with respect to x is approximated

$$T_x^2 \approx \max(D^{-x}G, -D^{+x}G, 0)^2 = max(V_A - V_B, V_A - V_C, 0)^2$$

The partial derivatives with respect to y and z are computed similarly, and the approximated squared gradient length becomes

$$||\nabla T||^2 = T_x^2 + T_y^2 + T_z^2 \tag{6}$$

which amounts to (3). Thus, we arrive at the ordinary FMM. The main difference between FMM and the higher accuracy version (FMMHA) is that the first order approximations (D^{-x} and D^{+x}) to the partial derivatives

[2]The 6–neighbourhood contains all voxels at a distance of $\leq 1vu$. Similarly, the 26–neighbourhood contains all voxels at a distance of $\leq \sqrt{3}vu$.

are replaced by second order approximations:

$$\begin{aligned} D_2^{-x}G &= \frac{3G[x,y,z]-4G[x-1,y,z]+G[x-2,y,z]}{2} \\ D_2^{+x}G &= -\frac{3G[x,y,z]-4G[x+1,y,z]+G[x+2,y,z]}{2} \end{aligned}$$

When these second order approximations are used, the scheme still works in exactly the same way – except that we get different polynomial coefficients. To use the scheme, the voxels at 2 *vu* distance must be frozen and have smaller distance values than those at 1 *vu* distance, e.g. $G[x-1] \geq G[x-2]$. If these two conditions are not met, the first order approximations to the derivative can be used instead.

It turns out that with few alterations, it is possible to change the function that recomputes distance values to use the high accuracy scheme. We are now trying to solve

$$1/F^2 = \begin{cases} \max(D_2^{-x}G, -D_2^{+x}G, 0)^2 + \\ \max(D_2^{-y}G, -D_2^{+y}G, 0)^2 + \\ \max(D_2^{-z}G, -D_2^{+z}G, 0)^2 \end{cases} \tag{7}$$

and the method is almost the same as for the normal FMM. Each of the three max terms in (8) are analyzed in a way that is analogous to the ordinary FMM. We shall look at the first term. For instance, if $G[x-1,y,z]$ is frozen and $G[x-1,y,z] < G[x+1,y,z]$ we pick the left argument to the first max term. If $G[x-2,y,z]$ is frozen and $G[x-2,y,z] \leq G[x-1,y,z]$ we form the equation

$$(D_2^{-x}G)^2 + \ldots = 1/F^2 \tag{8}$$

where the dots indicate that we add similar terms from the other max terms of (4). In case $G[x-2,y,z]$ is not frozen or not smaller than $G[x-1,y,z]$ we simply add $(D^{-x}G)^2$ instead of $(D_2^{-x}G)^2$.

Following these steps, a second order polynomial is formed, and this polynomial may be solved analytically. If there are two solutions, the larger is picked.

To simplify the notation, it is possible to write the coefficients in a compact way [6]:

$$(D_2^{-x}G)^2 = \frac{9}{4}G[x,y,z]^2 - 2\frac{9}{4}KG[x,y,z] + aK^2 \tag{9}$$

	FMM	FMMHA
Average error	0.00467565 *vu*	0.000496425 *vu*
Maximum error	0.120639 *vu*	0.0270829 *vu*

Table 1: Comparison of the Fast Marching Method and the higher accuracy Fast Marching Method.

where

$$K = \frac{1}{3}(4G[x-1,y,z] - G[x-2,y,z]) \tag{10}$$

Pseudo–code for the high accuracy recompute function is provided in Appendix A. A simple compile time switch can be used to disable FMMHA and revert to FMM for testing.

When the experiment above is repeated using FMMHA we get far better results. Of course, it does not make sense to use the high accuracy scheme starting from a single voxel, because in that case it must resort to the first order approximations to the derivative for the first few steps where voxels at 2 *vu* distance are not available. Consequently, when the FMMHA scheme is tested, the exact distances are computed at the centre voxel and in its 26–neighbourhood. For this experiment we obtain a max error of 0.27 *vu* and a mean error 0.07 *vu*. Notice that the mean error is an order of magnitude better than using plain FMM.

A practical volume graphics experiment was also conducted. An ellipsoid with principal axes of length 20 *vu*, 80 *vu*, 120 *vu* was voxelized. The voxels adjacent to the surface (meaning that the voxel has a 6–neighbour on the other side of the surface) of the ellipsoid had their distance values computed numerically using Hartmann's foot point algorithm [5]. The remaining voxels to within a distance of 2.5 *vu* were computed using the FMM or the high accuracy FMM. The results are summarized in Table 1. The average error is the average difference between the distance as computed by the foot point algorithm and the distance stored in the voxel (i.e. computed using FMM). The maximum error is the greatest of these differences. It is noticeable that the average error has dropped by an order of magnitude and that the maximum error by about half an order of magnitude.

Visually, both ellipsoids are indistinguishable from the same ellipsoid voxelized using only the foot point algorithm. However, in some cases there can be a visual difference between the result of the two Fast Marching Methods. This is illustrated in Figure 7 which shows two spheres that have been created by running the FMM (or FMMHA) starting from a single voxel. The sphere created using the high accuracy method is clearly more round although not perfect. This example was timed to get an idea about the

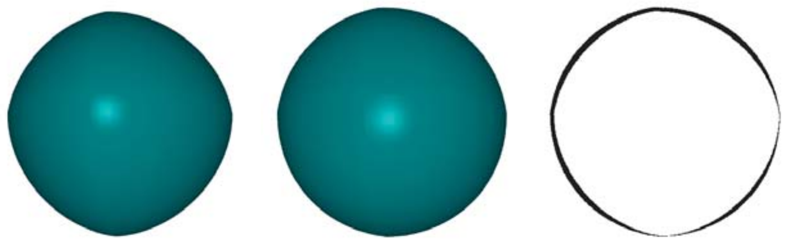

Figure 7: Comparison of two spheres voxelized using (left) FMM and (centre) the high accuracy variant. A difference image is shown on the right.

difference in performance. The actual "marching" took 1.4 seconds using FMM and 1.62 seconds using FMMHA (measured using the `clock` system call) on the Linux platform. Thus, the performance difference between the two methods is not great.

5 Discussion

This note has discussed Sethian's Fast Marching Method and the high accuracy variant. The main aim has been to provide a clear explanation of how to implement the method.

It seems that the FMMHA is a very good method for computing distance fields but it has yet to be compared against vector distance transforms, e.g. [8]. Of course, the FMM and FMMHA can also be used for other things than distance transforms and on non–Cartesian lattices. None of these issues have been discussed, and the reader is referred to [13] for a survey of applications.

A Pseudo–code

Pseudo–code for recomputing the distance value at a given voxel is presented below. The code assumes the existence of a number of functions whose prototypes are given. The C++ pseudo–code is almost identical to the code I actually use, but some simplifications have been made in the interest of clarity.

In the code below, a number of definitions are assumed. First of all,

```
STATE state(const Vec3i& p);
```

is a function which returns the state of a voxel given a 3D integer vector. STATE may be either *far*, *narrow band*, or *frozen* depending on whether the voxel is un-visited, in the narrow band or frozen, respectively. Another function

```
double value(const Vec3i& p);
```

returns the distance value of the voxel. If the state is *far*, this value is, of course, undefined. Finally, we assume a function

```
int solve_quadric(double coeff[3], double sol[2]);
```

which computes the solution to second order polynomials. The coefficients are given as an array of doubles where the array index corresponds to the degree of the coefficient.

The neighbours of a voxel are those reachable by taking one step in any of the six directions parallel to the primary axes. These six different steps are stored in an array of 3D integer vectors

```
const Vec3i N6i[6] =
{
  Vec3i(-1,0,0),
  Vec3i( 1,0,0),
  Vec3i( 0,-1,0),
  Vec3i( 0, 1,0),
  Vec3i( 0,0,-1),
  Vec3i( 0,0,1)
};
```

Finally, the pseudo code for recomputing the distance value at a voxel follows below. Note that the function is intended as an implementation of the FMMHA. However, by removing (e.g. using conditional compilation) a part of the code, we can fall back to the ordinary FMM.

```
bool recompute(const Vec3i& pi, double& max_sol)
{
  double coeff[3] = {-1,0,0};

  for(int j=0;j<3;j++)
    {
      double val1 = DBL_MAX;
      double val2 = DBL_MAX;

      for(int i=0; i<2;i++)
        {
          Vec3i pni = pi+N6i[2*j+i];
          if(state(pni) == FROZEN)
            {
              double _val1 = value(pni);
              if(_val1<val1)
                {
                  val1 = _val1;
                  Vec3i pni2 = pi+2*N6i[2*j+i];
                  double _val2 = value(pni2);
                  if(state(pni2) == FROZEN && _val2 <= _val1)
                    val2 = _val2;
                  else
                    val2 = DBL_MAX;
                }
            }
        }
#if HIGH_ACCURACY
      if(val2 != DBL_MAX)
        {
          // Note that the values of the coefficients are expressed in a
          // compact fashion described in a short note by James Rickett [6]
          // and Sergey Fomel.
          double tp = (1.0/3) * (4*val1 - val2);
          double a = 9.0/4;
```

```
          coeff[2] += a;
          coeff[1] -= 2 * a * tp;
          coeff[0] += a * sqr(tp);
        }
      else
#endif
        if(val1 != DBL_MAX)
          {
            coeff[2] += 1;
            coeff[1] -= 2 * val1;
            coeff[0] += sqr(val1);
          }
    }

  max_sol = DBL_MAX;
  double sol[2] = {DBL_MAX,DBL_MAX};
  if(int no_solutions = solve_quadric(coeff, sol))
    {
      if(no_solutions==2)
        max_sol = max(sol[1],sol[0]);
      else
        max_sol = sol[0];

      return true;
    }

  return false;
}
```

References

[1] G. Borgefors. Distance transformations in digital images. *Computer Vision, Graphics, and Image Processing*, 34(3):344–371, 1986.

[2] David E. Breen, Sean Mauch, and Ross T. Whitaker. 3D scan conversion of csg models into distance volumes. In Stephen Spencer, editor, *Proceedings of IEEE Symposium on Volume Visualization*, October 1998.

[3] D.E. Breen and R.T. Whitaker. A level-set approach for the metamorphosis of solid models. *Visualization and Computer Graphics, IEEE Transactions on=20*, 7(2):173–192, 2001.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[5] Erich Hartmann. On the curvature of curves and surfaces defined by normalforms. *Computer Aided Geometric Design*, 16(5):355–376, 1999.

[6] James Rickett and Sergey Fomel. Short note: A second–order fast marching eikonal solver. Technical report, Stanford Exploration Project, 2000.

[7] R. Satherley and M. Jones. Extending hypertextures to non-geometrically definable volume data. *International Workshop on Volume Graphics. Preprint*, pages 77–88 vol.1, 1999.

[8] R. Satherley and M. W. Jones. Vector-city vector distance transform. (found on authors' homepage) Submitted to Computer Vision and Image Understanding, 2001.

[9] Robert Sedgewick. *Algorithms*. Addison–Wesley, 2 edition, 1988.

[10] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences of the USA - Paper Edition*, 93(4):1591–1595, 1996.

[11] J.A. Sethian and A. Mihai Popovici. 3-d traveltime computation using the fast marching method. *Geophysics*, 64(2):516–23, 1999.

[12] James A. Sethian. Fast marching methods. *SIAM Review*, 41(2):199–235, 1999.

[13] James A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, second edition, 1999.