Fast Polygonization of Implicit Surfaces

Triquet Frédéric, Meseure Philippe, Chaillou Christophe

Laboratoire d'Informatique Fondamentale de Lille
Bat. M3 UFR IEEA
Universite des Sciences de Lille
59655 Villeneuve d'Ascq (France)
triquet@lifl.fr

ABSTRACT

Our work is centered on the use of implicit surfaces in interactive applications (at least 10 frames per sec) running on high-end consumer architecture (modeling, simulation, deformable body animation, games). We focus on the Marching Cubes algorithm that we tried to implement in an optimized way. We restrict our work to blended iso-surfaces generated by skeletons, since this kind of implicit surfaces is the most handy to use for animations.
Our implementation optimizations deal with the following features: simplifying the field function, accelerating its evaluation for each point (voxel-based technique), generating automatically the triangles for any case of the Marching Cubes. Another point we have considered concerns tesselation ambiguities often resulting in holes appearing in the surface. We have coded a library which is very easy to use and can be downloaded freely.
All these optimizations allow us to sample implicit surfaces composed of 200 points in 45 ms on a 450 MHz Pentium II Xeon.

Keywords: implicit surfaces, marching cubes, real time, library

1 Introduction

Implicit Surfaces have proved to be one of the most powerful modelling tools to build complex surfaces. They have become very attractive and are widely used, especially for animation. However, they still suffer from the lack of a good algorithm which could draw them in realtime (say, at least 10 frames per second), with a high quality. Until now, few algorithms aim at displaying implicit surfaces. Either the surfaces are ray-traced or have to be tesselated before rasterisation. These methods are known to be time-consuming, so the implicit surfaces have been restricted to non real-time contexts (maybe except in some very specific forms like quadrics for instance). However, in a number of cases, implicit surfaces are necessary. They can convincingly represent fluids or highly deformable bodies. No other comparable modeling tool can handle these shapes as easily.

Since it is general-purposed, our algorithm is intended to be practical even on high-end consumer hardware: thus it might be useful for games for instance. Anyway it can be used in a totally different context: replace a raytracing rendering, render thousands of primitives, etc. We focus on the tesselation of these surfaces, for graphics cards can only display polygons.

Our work does not consist of a new method to tesselate implicit surfaces, but rather studies how to optimize the implementation of the well-known Marching Cubes algorithm. Even if our work shares the same topic and some basic ideas with [Sha00a, Sha00b], it proposes further optimization schemes and provides a faster method to display fluids.

In our context, we have only dealt with the implicit surfaces built with the blending of field functions. The skeleton is supposed to be based on points animated by physical laws [Gas93] but it could be composed of more complex primitives such as lines or planes with no real loss of speed. One of our constraints is that our surfaces may undergo important deformations: their shape and topology may change a lot between two consecutive rendering (= tesselating) passes.

In the next section, we draw up an inventory of the methods that allow the displaying of implicit surfaces, in section 3 are listed the problems encountered when using a classical implementation of the Marching Cubes algorithm. Section 4 details which improvements we applied, and the results are presented in the section 5.

2 State of the art

There are several methods to draw implicit surfaces, they seperate in three classes: ray-tracing, discrete rendering and projective rendering which requires to sample the surface.

2.1 Ray-tracing Implicit Surfaces

Ray-tracing produces high quality images. However, for each pixel to be rendered, we have to do lots of calculations to determine whether the ray intersects the surface or not. This method is rather slow and depends a lot on the resolution of the display unit. We'd rather benefit from the graphics hardware we can now find on any personnal computer.

2.2 Discrete rendering of Implicit Surfaces

Discrete rendering consists in dividing space into "small" cubes, determining whether they are inside or outside the surface and then render the cubes lying on the surface. This method does not provide a good rendering quality. That is why we have turned to a polygonization method that will provide a mesh of triangles and will take advantage of graphics hardware.

2.3 Sampling of Implicit Surfaces

2.3.1 Seed-based methods

Andrew P. Witkin and Paul S. Heckbert [WH94] proposed a particle-based approach to sample and control implicit surfaces: they apply constraints to particles. There are two constraints: one maintains the particles on the surface, and the other, which is a repulsion force, makes them move and recover uniformly the surface. The result of this algorithm is a cloud of points which cannot be displayed as is. A mesh has to be built, using a Delaunay triangulation which is a complex task and is consuming too much time for our purpose.

Another technique to place seeds is proposed by Mathieu Desbrun et al. [DTG95]: their method takes benefits from temporal coherence for accelerating the sampling of a surface that progressively moves and deforms. Their algorithm could surely work in realtime but no efficient method has been proposed to build polygons.

2.3.2 Marching Cubes method

This is a well-known method to tesselate implicit surfaces. It was proposed by William E. Lorensen and Harvey E. Cline [LC87]. It consists in dividing the space into cubes, evaluating the field function at all the vertices of the obtained grid, and then in building the polygons inside each cube according to the relative position of the vertices regarding the surface.

We chose to base our real-time algorithm on this method since it provides directly the polygons we want to display.

3 Time cost of a classical Marching Cubes algorithm

Our purpose is to test how fast the Marching Cubes algorithm can be. The algorithm Jules Bloomenthal proposed [Blo94] is already optimized in one important way: his program deals only with the ”useful” cubes (those that intersect the surface). However, this program was aiming to show how the Marching Cubes algorithm was working and it could not really be used on its own in a real time context (see section 5).

When we analyse what is costing calculation time in the basic Marching Cubes algorithm, we find that there are two classes of improvements we can carry out. On one hand, some calculations are redundant and should be avoided, on the other hand some other steps (which are required) are not optimal and should be accelerated.

Some calculations are done repeatedly:

- most of the points on the marching cubes grid are shared by eight cubes, when we treat cubes sharing a same vertex we evaluate the field function at this vertex for each cube. This is not true if we have the whole grid filled with the values of the field function, anyway this requires the algorithm to fill *the whole grid* which can be useless for a lot of nodes (see 4.4),
- a similar problem occurs when the position of the intersection points between the surface and the edges of our cubes is determined (since these edges are shared by four cubes, we do the same computations four times).

Some great improvements can be applied to the computation of the field function:

- there are different expressions of field functions, some of them are more or less complex to evaluate,
- for the field functions, we chose finite potentials which allow us to precisely model our surfaces: in this model the potential of a source decreases and drops to zero when the distance is larger than the radius of influence of the source. Thus, the potential of a given point is only determined by the surrounding skeleton points and the movement of a primitive only affects the shape of the surface locally.

4 Our optimizations

In this section, some improvements of the Marching Cubes method are presented in detail.

4.1 Timestamping results

At each step we avoid doing the same computations several times. Any computed value is tagged with a counter which is incremented each time we call the whole algorithm. This timestamp tells whether a value (field function evaluation, intersection points, etc.) has already been computed or not for the current tesselation: if one certain value has already been calculated (in the current polygonisation), then it has been stocked and timestamped. We just have to check whether it is in our table. In this case we just pick it up and avoid calculating it twice. Any value that could be computed twice benefits from this method.

4.2 The field function

We use a field function with limited area of influence [WMW86, NHK+85]. Our field function is based on an expression proposed by Blanc and Schlick [BS95] but has been slightly modified:

$$\mathbf{d_i}^2 = (x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2$$

$$F_i(p) = \begin{cases} A_i * \left[1 - \left(\frac{\mathbf{d_i}}{R_i}\right)^2\right]^2 & if\ \|\mathbf{d_i}\| < R_i \\ 0 & otherwise \end{cases} \tag{1}$$

where $\mathbf{d_i}$ is the vector between the i^{th} primitive of coordinates (x_i, y_i, z_i) and the point p of coordinates (x, y, z), R_i the ray of influence of the primitive and A_i a scale factor.

It is a piecewise polynomial function which is identically null beyond a certain ray of influence R_i. A vector normal to the isosurface is given by:

$$\mathbf{grad}\ F_i(p) = \begin{cases} \frac{4 * A_i}{R_i} * \mathbf{d_i} * \left[1 - \left(\frac{\mathbf{d_i}}{R_i}\right)^2\right] & if\ \|\mathbf{d_i}\| < R_i \\ 0 & otherwise \end{cases} \tag{2}$$

Our field function has a very low calculation cost: we only use simple operations (multiplications and additions). Indeed any exponential function or square root evaluation should be avoided in real-time applications. The shapes obtained in the resulting blended surfaces are satisfying.

4.3 Accelerating the computation of the blending function

A classical way to compute the potential of a given point in space is to sum the F_i values associated with **all** the primitives: $F(p) = \sum_{i=1}^{N} F_i(p)$.

The influence of a primitive on a point requires the computing of the distance between this point and the skeleton. However, beyond a certain distance (the ray of influence) a primitive does not affect the potential of the point any longer. We have found a way to take this property into account for the evaluation of the potential of a point.

In our method, the space is divided into voxels containing a list of all the primitives which influence this area. Before sampling the iso-surface we run through the list of primitives (composing the skeleton of the surface we have to sample). Each primitive is added to the lists of the voxels which are covered by the sphere defined by the ray of influence of its field function. It is a well-known technique for accelerating the ray-tracing on implicit surfaces.

To evaluate the potential of a point we just have to peek at the list of primitives stored in the corresponding voxel: we thus consider only the local primitives.

The size of the voxels has to be chosen judiciously since the ratio between the size of the voxels and the size of the Marching Cubes highly influences the performance of the improvement. Anyway it also depends a lot on the ray of influence of the primitives we tesselate [1].

[1] we still do not know, for a given surface and a given grid, what voxel size will be the best and we have to test it

4.4 Tracking of the surface

Our implementation only deals with the ”useful” cubes (those that intersect the surface). We start our work in one of those cubes and go on only to the ”useful” neighbours. Our program pushes and pops cubes in a stack, it ends when the stack gets empty. Since a cube may be pushed several times (average of 4 times), the timestamp described in section 4.1 is used to know whether it has already been considered or not. Thus, we only need to compare two integers to verify that a cube has already been treated and we do not evaluate the field function.

Since our surfaces may be composed of several disconnected pieces, we have to find a starting point on each of them. Our method is not really neat but always works and does not require too much time: we start our search from each primitive lying inside the surface (i.e. on the skeleton) and search in an arbitrary direction for a cube cut by the surface (we get outside of the surface), we then push these cubes for further use. The figure 1 shows an example of a multi-piece surface.

Methods tracking the surface have already been proposed [Blo88], however they are rarely tolerant to non-connective surfaces. Our solution to this problem is trivial yet efficient and can manipulate identicaly any kind of surface.

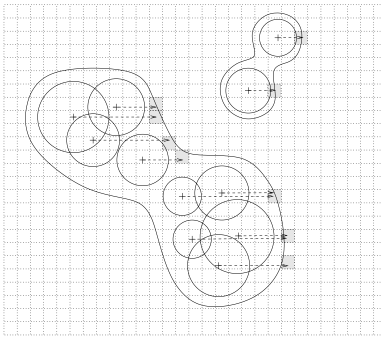


Figure 1: Finding starting points on the surface consists, for each primitive, in searching in a given direction for a cube cut by the surface.

4.5 Incorrect tesselation

There is an important problem which is inherent in the Marching Cubes algorithm: the position of the vertices relative to the surface is not always sufficient to determine the local topology of the surface inside a cube and thus to generate triangles fitting to the local topology of the real surface. Figure 2 shows a typical case of a such ambiguity.

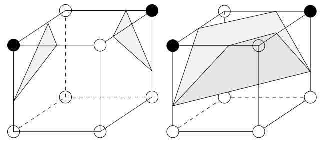
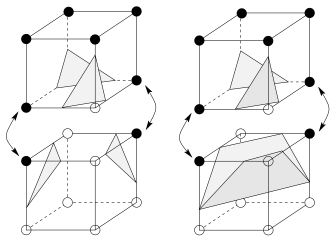


Figure 2: Ambiguous cube with two different possible tesselations and thus two different topologies.

The main problem consists in choosing the correct set of triangles within the cube to avoid topology problems (see figure 3). Guaranteeing

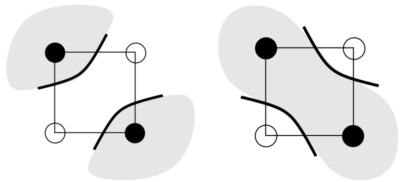


(a) (b)

Figure 3: Figure a shows two adjacent cubes where the respective tesselations are not coherent and exhibits discontinuities. Figure b shows a better choice providing C^0 continuity.

the C^0 continuity of the surface between two neighbouring cubes relies on the fact that the triangles share common edges and not only vertices. Whatever the configuration cube, a given kind of cube face configuration will always generate the same edges. If two cubes are adjacent, by construction, the associated triangles will share edges on the common face. Thus it is not possible to generate an edge belonging to only one triangle (like in figure 3a).

The problem consisting in choosing "a good" tesselation arrives when the considered cube presents at least a face where two diagonaly opposed vertices are inside the surface and the two others are outside (see figure 4). The choosen tes-



(a) Separation of the surfaces is privileged. (b) Connection of the surfaces is privileged.

Figure 4: The black vertices are inside the surface and the white ones are outside.

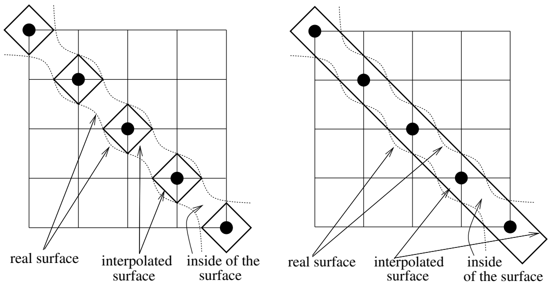
selation will either privilege the separation (figure 4a) or the fusion (figure 4b) of the surfaces. Thus, the only constraint is to be coherent and to make always the same kind of choice (which was not done in the figure 3a). We choose to let the user decide whether separation or fusion is to be prefered.

This choice is directly coded in the table (see section 4.6). This solution does not require any additional computations such as in [PPP88, HR95] for example. Should our global approach be too restrictive, such methods could be used to improve local approximation of the surface.

Our way to build the triangles ensures C^0 continuity. However it does not ensure to obtain a correct approximation of the surface (in terms of topology). This is a well-known problem of the Marching Cubes algorithm and can be reduced by increasing the grid resolution (= decreasing the cubes size). The figure 5 shows a typical case where the topologies of the approximated and the real surfaces can be very different: in figure 5(a) the obtained surfaces are disconnected although the real surface is composed of only one part.

4.6 Descriptive table

In the light of all these improvements, when we treat a cube, we have to do the following processes:



(a) The approximated and real surfaces do not match

(b) The approximated and real surfaces match

Figure 5: Differences between the topologies of the approximated and real surfaces (see also figure 4).

- determine the coordinates of the intersection points between the surface and the grid,
- generate the triangles inside a cube,
- follow the treatment for the ”useful” neighbours only.

For that purpose, we use a table which contains all the informations we may need, this table is composed of 256 entries and addressed by a byte[2] built like this: if a vertex is outside of the surface (its potential is lower than 0), the associated bit is cleared, in the other case the corresponding bit is set. For each entry of our table we get:

- the number of triangles to be generated,
- the description of each of these triangles,
- the number of the ”useful” neighbours of this cube,
- the list of these cubes.

This table is built using the 20 base cube configurations we obtained by adding 6 new cubes to the 14 proposed by Lorensen and Cline [LC87] (see figure 6).

[2] 8 bits corresponding to the 8 vertices of the cube

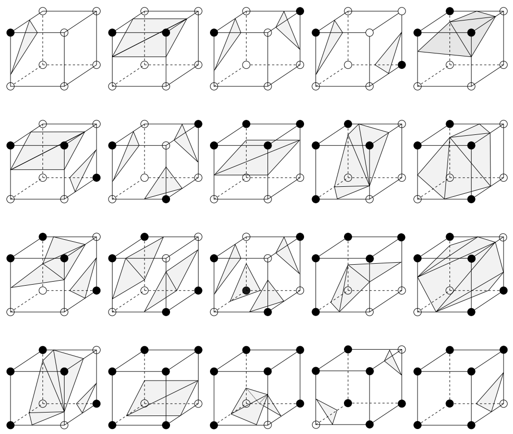


Figure 6: Our table consists of 20 cubes

4.7 Memory considerations

Our main goal is the real-time display of moving implicit surfaces. We first did not take into account memory problems and we allocated the memory for each cube of the whole marching cubes grid. However, the amount of necessary memory can quickly get prohibitive[3]. Since this grid is the biggest structure in the program and its size depends on the rendering quality the user wants, we were highly limited.

A good improvement avoids storing the whole grid in memory since all cubes are not useful at the same time. Instead, we propose to take benefit of this property by allocating an entire grid of more simple cube structures. These cubes contain an index to the corresponding complex cube structures which are allocated only if necessary (this index is time-stamped as described in 4.1). This improvement requires about a half of the memory needed by the previous implementation.

An other way to allocate only the necessary cubes structures consists in using a hash-table instead of our look-up table. This method requires even less memory than ours but is slightly slower since accessing a cube requires to scan a chained-list. Thus, though it is compatible with our algorithm we choose not to implement it yet.

5 Results

We modified a little bit the implementation Jules Bloomenthal proposed [Blo94] to be able to

[3] about 64MB for a $100 * 100 * 100$ grid

compare it to our work: we changed the field function and some tesselation parameters in order to obtain the same tesselation from the two programs. We get more or less the same number of triangles (for 1000 triangles generated the two tesselations vary of one or two triangles). The graph of the figure 7 compares the tesselation times of two implementations of the Marching Cubes method. It shows Bloomenthal's implementation (without any of our improvements) and the first implementation we coded (without using the voxels' optimization). We can see that our code is already more than 6 times faster than Bloomenthal's. Figure 8 compares Bloomenthal's implementation benefiting from our acceleration of the blending function computation (see 4.3) to our two implementations (the *"Classical" one*[4] and *"Our"* implementation[5]). The "classical implementation" follows the surface but does not benefit from voxel-based evaluation of the blending function and the problem of redundant computations is not dealt with (no time-stamping).

This figure shows that our voxel-based improvement divides the tesselation time by 5 or 6.

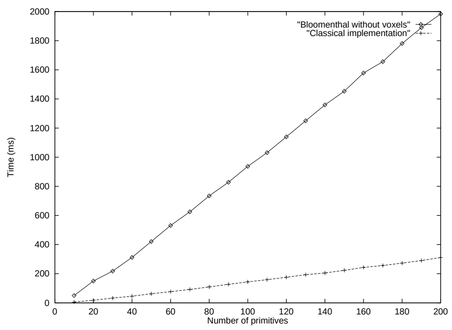


Figure 7: Tesselation times

These results have been measured on a 450 MHz Pentium II Xeon for a $20 * 20 * 20$ Marching Cubes grid. We can see that the tesselation time is linear with respect to the number of primitives. We can expect to animate one or more surfaces where the overall number of primitives is less than 200. Naturally, the number of primitives can be much higher for more powerful processors.

[4] with some of our improvements

[5] with all the improvements we implemented

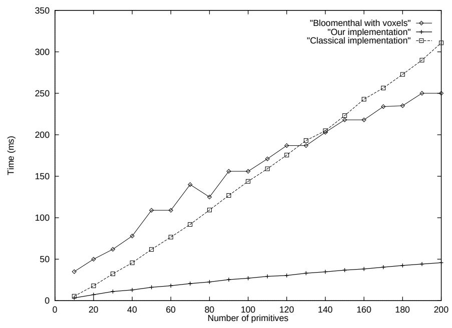


Figure 8: Tesselation times

We have managed to display a surface of 150 primitives at a 15 Hz framerate on a Titan II graphics card (but this framerate highly depends on the graphics hardware). However these results show that the display of complex implicit surfaces on a personnal computer is possible.

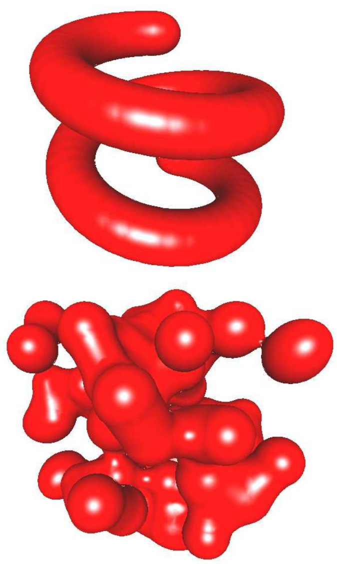


Figure 9: Screenshots: 80 primitives sampled whithin a $80 * 80 * 80$ Marching Cubes grid

We are using this library to display organic bodies and fluids in surgical simulators. This method enables us to display blood flow realistically.

Besides, our work is presently experimented at the LERI in Reims (France) for the rendering of about 7000 primitives. The tesselation requires about 1 second on a general-purpose PC which is sufficient for previewing before raytracing. This shows that our library has a very general purpose and can be used in very different contexts, for interactive but non real-time applications.

The whole code has been implemented as a library which can be downloaded at the following URL: `www.lifl.fr/~triquet/implicit`.

6 Further Work

We are working just now on some of the following topics.

Another improvement we will soon take an interest in is avoiding unwanted blending, our voxel structure will give us important informations on the local topology of the surface.

Our whole program has been coded in C language, some parts of the code could attractively be written in assembly language.

At least we intend to apply texture mapping to our surfaces, which is a delicate problem since they may move, deform, and even change of topology at any time. For now we just display our surfaces with an environment mapping to apply a specular effect.

One improvement which is often taken an interest in consists in taking an advantage of the previous tesselation to build the next one. This improvement is based on temporal coherency of the surfaces. We did not take this into account at all since this did not improve enough our results: our surfaces may move and deform a lot between two frames. Moreover, even if we manage to find parts of the surface that did not move too much, we still have to perform all the computations we presently do.

7 Acknowledgements

We wish to thank Kadi Bouatouch at the IRISA in Rennes (France) for his precious comments about this paper and Stéphanie Prévost and Éric Bittar from the LERI in Reims (France) for their help and encouragements.

REFERENCES

[Blo88] Jules Bloomenthal. Polygonisation of implicit surfaces. *Computer Aided Geometric Design*, 5:341–355, 1988.

[Blo94] Jules Bloomenthal. An implicit surface polygonizer. *Graphics Gems*, IV, 1994.

[BS95] Carole Blanc and Christophe Schlick. Extended field functions for soft objects. In *Implicit Surfaces'95*, pages 21–32, Grenoble, France, April 1995. Proceedings of the first international workshop on Implicit Surfaces.

[DTG95] Mathieu Desbrun, Nicolas Tsingos, and Marie-Paule Gascuel. Adaptive sampling of implicit surfaces for interactive modelling and animation. In *Implicit Surfaces'95*, pages 171–186, Grenoble, France, April 1995. Proceedings of the first international workshop on Implicit Surfaces.

[Gas93] Marie-Paule Gascuel. An implicit formulation for precise contact modelling between flexible solids. *Computer Graphics*, 27:313–320, 1993.

[HR95] Steve Hill and Jonathan C. Roberts. Surface models and the resolution of n-dimensional cell ambiguity. *Graphics Gems*, V, 1995.

[LC87] William Lorensen and Harvey Cline. Marching cubes: a high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987. Proceedings of SIGGRAPH'87 (Anaheim, California, July 1987).

[NHK+85] Hitoshi Nishimura, Makoto Hirai, Toshiyuki Kawai, Toru Kawata, Isao Shirakawa, and Koichi Omura. Object modeling by distribution function and a method of image generation. *The Trans. of the Inst. of Elec. and Comm. Eng. of Japan*, J68-D(4):718–725, 1985. In Japanese (transl. into English by T. Fujiwara).

[PPP88] Alexander Pasko, V. Pilyugin, and V. Pokrovskij. Geometric modeling in the analysis of trivariate functions. *Computers and Graphics*, 12(3/4):457–465, 1988.

[Sha00a] Brian Sharp. Go with the flow. *Game Developper*, pages 26–35, July 2000.

[Sha00b] Brian Sharp. Moving fluid:*the conclusion of a two-part series about implicit surfaces*. *Game Developper*, pages 40–51, August 2000.

[WH94] Andrew Witkin and Paul Heckbert. Using particles to sample and control implicit surfaces. *Computer Graphics*, pages 269–278, July 1994. Proceedings of SIGGRAPH'94.

[WMW86] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, pages 227–234, August 1986.