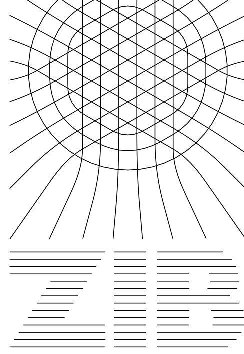


Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustr. 7, D-14195 Berlin-Dahlem

Henrik Battke   Detlev Stalling   Hans-Christian Hege

Fast Line Integral Convolution for Arbitrary Surfaces in 3D

Preprint SC-96-59 (Dec 1996)

Fast Line Integral Convolution for Arbitrary Surfaces in 3D

Henrik Battke Detlev Stalling Hans-Christian Hege

Abstract

We describe an extension of the line integral convolution method (LIC) for imaging of vector fields on arbitrary surfaces in 3D space. Previous approaches were limited to curvilinear surfaces, i.e. surfaces which can be parametrized globally using 2D-coordinates. By contrast our method also handles the case of general, possibly multiply connected surfaces. The method works by tesselating a given surface with triangles. For each triangle local euclidean coordinates are defined and a local LIC texture is computed. No scaling or distortion is involved when mapping the texture onto the surface. The characteristic length of the texture remains constant.

In order to exploit the texture hardware of modern graphics computers we have developed a tiling strategy for arranging a large number of triangular texture pieces within a single rectangular texture image. In this way texture memory is utilized optimally and even large textured surfaces can be explored interactively.

Contents

1 Introduction **1**

2 Background **2**

3 LIC on Surfaces **4**

3.1 Triangulation 4
3.2 Stream Line Integration 6
3.3 Local Texture Space 7
3.4 Triangle Tiling 9

4 Projecting 3D-Fields **11**

5 Applications and Results **12**

1 Introduction

Line integral convolution (LIC) is a simple but flexible method for generating textured images from vector data. After its invention by Cabral and Leedom [3] in 1993, the method found a wide range of applications. It is of particular interest in vector field visualization, where it has been used to compute comprehensive images from complex data sets.

Traditional approaches in vector field visualization comprise symbolic representations with arrows and probes [15], advecting particles, clouds [6] and stream surfaces [7]. Although important field characteristics can be extracted with all of these methods, in general they require a detailed knowledge about where to place symbols or seed points. Either these parameters have to be adjusted manually, or automatic methods for analysing and depicting vector field topology have to be applied [5]. However, no such problems arise if the placement of spatially extended objects could be avoided.

Texture based methods like line integral convolution offer exactly this opportunity. Textures have been used in visualization for the first time by van Wijk. In his spot noise algorithm [14] a random input texture is convolved with a 2D filter kernel. The shape of the filter kernel – or spot – is chosen to be a function of the field to be visualized. For example, an elliptic kernel might be aligned to the direction of a vector field. Line integral convolution is a further development of this method [3]. The convolution is performed using a 1D filter kernel oriented along the stream lines of a vector field. In contrast to standard spot noise the method resolves fine details of the field even in the vincinity of critical points, e.g. vortices or saddles. Texture based methods are most naturally applied to 2D fields. The resulting 2D textures encode information about the field at a length scale of a few pixels. Such highly condensed representations of vector fields cannot be obtained by traditional methods.

Visualization of vector fields defined on surfaces in 3D space remains an interesting and challenging problem. Examples comprise velocity fields on skin-friction walls in computational fluid dynamics, curvature fields on spline surfaces in CAD applications, or vector fields on manifolds occuring in differential geometry. While texture based visualization methods are hard to extend to real three-dimensional fields, they can be well applied to vector fields defined on surfaces.

Previously spot noise and line integral convolution have only been applied to curvilinear surfaces, i.e. surfaces which can be parametrized globally using 2D cartesian coordinates [4, 8]. In these methods the vector field is transformed into parameter space. Then a conventional 2D LIC image is generated. Finally the results are mapped back to the surface. However, this mapping in general is non-linear and therefore locally distorts the characteristic texture length. This may change severely the visual appearance of the mapped texture.

In this article we present an extension of the line integral convolution method for imaging vector fields on surfaces. Our method is not tainted with distortion problems since texture generation is performed in physical space rather than in parameter space. In addition the method is no longer restricted to curvilinear grids. Instead it works directly on a triangulation of an arbitrary surface.

For maximal performance we make use of a fast LIC algorithm which has been described in detail in [12]. Building on this paper, we will only summarize the fundamentals of the LIC method as well as its fast implementation in 2D. Then we will discuss how the method can be extended to arbitrary surfaces in 3D space. This will include remarks about required data-structures, stream line integration, definition of local texture coordinates, as well as an efficient strategy for triangle placing. Finally we will describe some applications and results.

2 Background

Consider a vector field $\mathbf{f}$ defined on an arbitrary two–dimensional differentiable manifold M, for example a plane, a sphere, a torus, or some more complex surface. We assume all vectors to be tangential to the surface, i.e. $\mathbf{f}$ is mapping from M to the tangent bundle TM of the manifold. By definition around each point of M there is a non–empty region which can be parametrized locally using 2D euclidean coordinates. However, in general no global parametrization does exist. We assume M to be embedded in 3D–space, so that the euclidean metric of $\mathbf{R}^3$ can be applied to M. Excluding the case of time–dependent fields, stream lines or integral curves σ of a vector field are defined by an autonomous ordinary differential equation

$$\frac{d\sigma(t)}{dt} = \mathbf{f}(\sigma(t)), \quad \sigma(t) \in M.$$

In the following we assume σ to be parametrized by arc–length s instead of the arbitrary parameter t. The directional structure of a vector field (up to its orientation) is fully characterized by its integral curves σ. Therefore, depicting stream lines is an important technique in scientific visualization.

Instead of drawing individual curves, the idea in line integral convolution is to blur a noise function T along stream line direction. In this way a filtered image of intensity

$$I(\mathbf{x}) = \int_{s_0-L}^{s_0+L} k(s-s_0)\; T(\sigma(s))\; ds, \quad \sigma(s_0) = \mathbf{x}$$

is obtained. Here k denotes a one–dimensional filter kernel of length $2L$. The convolution results in highly correlated intensity values in stream line direction, whereas perpendicular to this direction there is almost no correlation. Thereby the

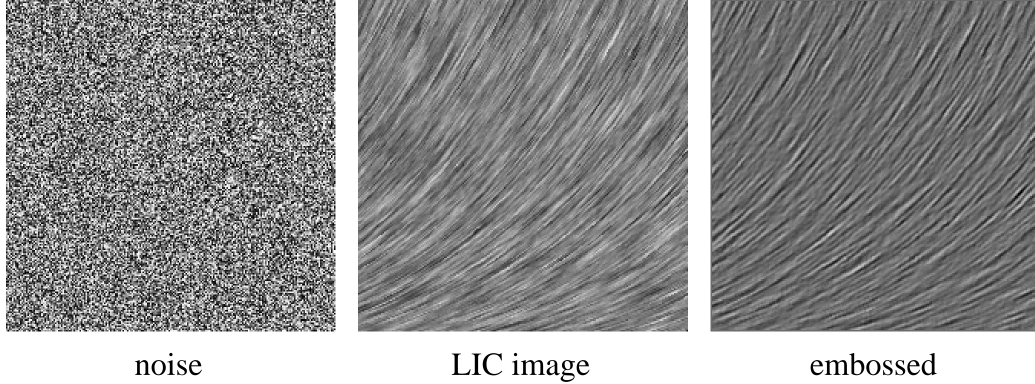


Figure 1: In line integral convolution a noise function is blurred in direction of the vector field's integral curves.

directional structure of the whole field emerges visually. Fig. 1 shows an example of a LIC image together with an unfiltered noise function. The results from raw LIC may be further improved by applying an emboss filter.

Solving the convolution integral for each pixel from scratch is prohibitively slow. However, in case of a constant filter kernel k a fast evaluation strategy can be established by exploiting the coherence between the intensities of neighbouring points on a stream line. Choosing a constant filter kernel does not disturb visual results and therefore is no limitation. The convolution integral is approximated by $2N+1$ equi-distant discrete samples,

$$I(\mathbf{x}_0) = \frac{1}{2N+1} \sum_{i=-N}^{N} T(\mathbf{x}_i), \quad \mathbf{x}_i = \sigma(s_0 + i\frac{L}{N}),$$

with N being a free parameter of the algorithm. Using a sample distance of half a pixel length, typical values of N are 20–80. After having computed $I(\mathbf{x}_0)$ it is easy to update intensity values along the stream line via

$$I(\mathbf{x}_{i\pm1}) = I(\mathbf{x}_i) - \frac{T(\mathbf{x}_{i\mp N})}{2N+1} + \frac{T(\mathbf{x}_{i\pm(N+1)})}{2N+1}.$$

Usually a single pixel will be covered by multiple stream lines. Therefore all samples falling into that pixel have to be averaged. This makes it necessary to intermediately store accumulated intensities as well as the number of hits per pixel. For performance reasons it is favourable to cover all pixels of an image using a a minimal number of stream lines. Heuristics to achieve such an optimization and other details of the so-called fast-LIC algorithm are described in [12].

3 LIC on Surfaces

Although previous algorithms for computing LIC images were restricted to two-dimensional cartesian grids, the technique can be extended conceptually to arbitrary two-dimensional manifolds.

If a global parametrization is known for a particular surface, it would be possible to compute LIC images in cartesian parameter space and map the results to the surface. This approach has been investigated in [4]. However, many surfaces like arbitrary stream surfaces or surfaces obtained from subdivision schemes cannot be parametrized globally. In addition, an arbitrary mapping from parameter space to physical space in general is not linear. Thus the appearance of the mapped LIC image varies significantly across the surface, impairing the visual impression. Of course one could try to compensate the distortions implied by the mapping while generating LIC images in parameter space. However, this turns out to be quite difficult because it requires not only to locally adjust filter length, but also to locally modify the characteristics of the input noise function.

A common way to deal with general surfaces is to use triangular approximations. Modern 3D graphics computers are designed to render very fast large numbers of triangles. Therefore triangular approximations are the ideal representation of surfaces for visualization and other graphical purposes. A planar triangle can be parametrized without any distortion using the identity mapping. This leads to the idea of computing a separate piece of LIC texture for each triangle. The local texture space has to be cartesian in order exploit current rendering soft- and hardware. Stream lines have to be followed across neighbouring triangles to give the impression of a smooth surface, but within a triangle convolution can be performed as in the planar case. The idea is illustrated in Fig. 2. On the first sight it might appear that such an approach would suffer from the non-matching texture grids. However, if the algorithm is implemented carefully, accurate results can be obtained, as demonstrated in Color Plate 1. In the following we will describe some of the details of this method.

3.1 Triangulation

Our starting point is a triangular approximation of an arbitrary surface. For stream line integration we will need to access quickly neighbouring triangles. Usually the connectivity information required for this is available when the triangulation is generated. However, many popular geometry file formats simply include a vertex list and a triple of indices per triangle, but no explicit connectivity information. To be able to process such kind of data as well our algorithm determines the adjacency relations.

This is done by inserting the edges of all triangles into an AVL search tree

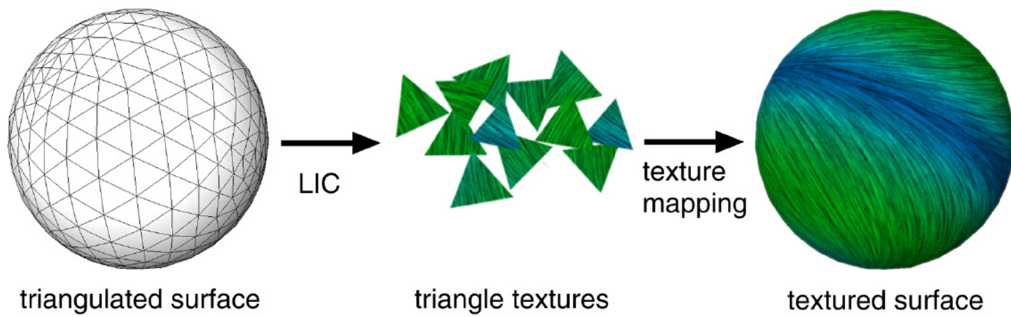
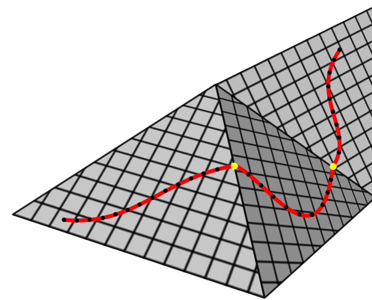


Figure 2: LIC–textures are computed for each element of a triangulated surface (top). For each triangle local euclidean texture coordinates are defined (right). Following stream lines across neighbouring triangles ensures a smooth impression of the resulting textured surface.



[1]. The data structure allows us to quickly check if a single edge is used by two neighbouring triangles. In principle more than two triangles may share a common edge. However, in this case we can't determine the topology of the underlying 2D-manifold without further information. In the current implementation we simply assume that an edge either belongs to the boundary of the manifold or that it connects two triangles. For performance reasons the inter-triangle connectivity is stored directly without referencing any edges.

We have also implemented an interface of our method to the Open Inventor graphics toolkit. This class library provides not only means for defining triangular surfaces but also for quadmeshes and spline surfaces. It is possible to generate triangulations of such surfaces at different levels of detail. In practice for each triangle a user-defined callback method is invoked. However, the callback provides only vertex coordinates but no vertex indices. This slightly complicates identification of shared edges, since a single vertex may be represented by different floating point coordinates due to round-off errors. We tackle this problem by taking into account a small tolerance when comparing coordinates.

The time consumption for computing the connectivity information is usually quite low. In fact it was well below 5% of the overall computation time for all our examples.

3.2 Stream Line Integration

Since we can't rely on a global parametrization we assume the vector field $\mathbf{f}$ to be defined in 3D coordinates. For a field $\mathbf{f} : M \to TM$ defined on a 2D manifold the 3D field vectors are tangential to M. To obtain the field in local coordinates these vectors are projected onto the individual elements of the triangulation.

Because this projection has to be performed anyway, also vector fields with non-vanishing normal component can be handled without further expense. This allows us to visualize the projection of 3D vector fields on arbitrary surfaces without having to compute an intermediate representation of the projected field.

For ease of simplicity we assume $\mathbf{f}$ to be defined at the vertices $\mathbf{p}_i$ of the triangulation. Then the question is how to interpolate the field within the triangles. One approach would be to interpolate linearly in spherical coordinates. Less expensive is a linear interpolation in cartesian coordinates, using barycentric coordinates $\lambda_0, \lambda_1, \lambda_2$ whith $\lambda_0 + \lambda_1 + \lambda_2 = 1$. Then for a point $\mathbf{p}$ represented as

$$\mathbf{p} = \sum_i \lambda_i \, \mathbf{p}_i \qquad \text{we have} \qquad \mathbf{f}(\mathbf{p}) = \sum_i \lambda_i \, \mathbf{f}(\mathbf{p}_i).$$

However, interpolating each component of the vector field independently results in vectors that are shortened in an unnatural way if the directions of the vectors $\mathbf{f}(\mathbf{p}_i)$ are rather different. A simple remedy is to first interpolate the vectors and then scale them to linearly interpolated magnitude. Notice that such a scaling does not change the shape of a stream line, but merely its native parametrization. In line integral convolution the vector field is normalized anyway so that stream lines are parametrized by arc-length. However, if vector magnitude is to be encoded by color or by LIC texture animation as described in [12], interpolated lengths should be used rather than lengths of interpolated vectors.

Although stream lines can be found analytically for a linear vector field, we use numerical integration instead. This is much simpler and more stable than evaluation of the analytical expressions. In conventional 2D line integral convolution we are using a Runge-Kutta integrator with adaptive step-size control and error monitoring to quickly pass through homogeneous regions of the field. Since we have to take many samples along a stream line, an additional interpolation polynomial has to be evaluated. Adaptive integrators are somewhat less efficient if the solution is only requested for a relatively small domain such as a single patch of a triangulated surface. Nevertheless, the adaptive method is still superior to an one-sample wide, fixed-sized midpoint rule, e.g. second-order Runge-Kutta. Simple Euler integration may be faster, but is too inaccurate.

To detect whether a stream line has left the current triangle we perform a point-in-triangle test for every sample. Internally the adaptive integrator may well evaluate the field outside the triangle. Then linear interpolation in fact becomes

linear extrapolation. The point-in-triangle test is performed by checking if one of the three barycentric coordinates λ_i is smaller than zero. If a point lies outside we have to find the location where the streamline crosses the border and in which neighbouring triangle the point is located. This means computing the intersection of a stream line with one or two possible edges. Let $\mathbf{p}$ be the last sample inside the triangle and $\mathbf{q}$ the first one outside. We look for a number s such that $\mathbf{p}+s(\mathbf{q}-\mathbf{p})$ lies on a triangle edge. In order to find s quickly we store triangle edges in implicit form $ax+by+c=0$, which leads to $s=(ap_x+bp_y+c)/(a(p_x-q_x)+b(p_y-q_y))$. If two barycentric coordinates are smaller than zero this equation has to be evaluated for both edges. The smallest s corresponds to the edge sought and determines the actual exit point. After this point has been computed, stream line integration is continued in the neighbouring triangle. In general the projected vector field is discontinuous at the border of two adjacent, non-coplanar triangles. However, for sufficiently fine triangulations the discontinuities will not be visible.

If a stream line leaves the domain of M we continue the curve artifically as a straight line, such that filter boxes of sufficient length can be used.

3.3 Local Texture Space

Modern rendering software and graphics hardware requires textures to be defined on a uniform cartesian grid. Therefore we have to fit every triangle into a local coordinate system of this type, although cartesian coordinates are not the most natural ones for triangles. For convenience we choose the cartesian grid to be aligned to the longest edge of a triangle. Like in the fast-LIC algorithm for the planar case we maintain for every pixel of the texture grid an accumulation variable and a hit counter. All pixels with centers inside the triangle are traversed using a scan-conversion algorithm. If a pixel has not already been hit before, a new stream line is started and the convolution integral is computed for that pixel as well as for other samples lying on the same curve.

Some care has to be taken when pixels outside the current triangle are processed. A pixel may cover the triangle partially, even if its center lies outside. Such a pixel may be covered by some stream lines, but it is not guaranteed to receive any hits. Depending on the type of texture interpolation, such pixels as well as pixels which are completely outside may influence the final color inside the triangle. This is illustrated in Fig. 3 for the case of constant and bilinear interpolation. Intensity of these outer pixels is set to the value which has been computed for the same location in a neighbouring triangle. It may happen that the center of such a border pixel does not fall into any of the three neighbouring triangles. Instead of looking for second-nearest neighbour triangles in this case we simply take the value of the nearest texture pixel for which a value has been computed. If no such pixel exists we assign a constant gray value. However, usually this

doesn’t happen unless the triangles are very small or degenerate.

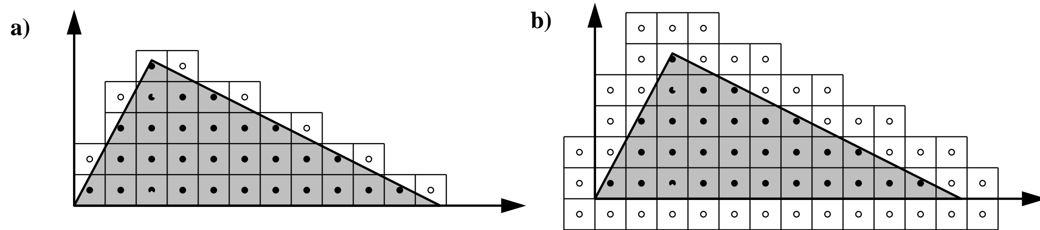


Figure 3: Pixels outside the triangle influence final color depending on the type of texture interpolation: a) constant interpolation b) bilinear interpolation.

In conventional 2D line integral convolution the initial noise function to be convolved is discretized usually on a 2D grid of the same resolution as the final image. For surface LIC we use a procedurally defined 3D noise function instead. This has a number of advantages. First, a procedural texture saves a lot of memory. Second, we exclude any artifacts which might occur because of partially visible texture cells at the triangle borders. Third, a 3D texture can also be evaluated outside the surface domain. This is important because stream lines may have to be continued artifically beyond the surface boundary to provide a filter box of sufficient length.

A procedural noise function proposed by Perlin [10] has become quite popular in computer graphics. An efficient implementation of this function is described in [13]. The method works by mapping floating point coordinates to an integer grid. From the integer location a pseudo-random value is generated using arithmetic and binary operations. To get a continuous noise function random values for neighbouring grid points have to be interpolated. In our case a discontinous noise function is sufficient. Therefore we can abstain from the interpolation step. This makes the implementation very simple:

```
int noise(float x, float y, float z) {
  int random;
  x=(int)x;
  y=(int)y;
  z=(int)z;
  float r1=x*-67.0+y*59.0+z*71.0;
  float r2=x*73.0+y*79.0-z*83.0;
  float r3=x*89.0-y*97.0+z*101.0;
  random=*((int*)(&r1))<<12^*((int*)(&r1));
  random^=*((int*)(&r2))<<12^*((int*)(&r2));
  random^=*((int*)(&r3))<<12^*((int*)(&r3));
  return (random&0x7fffffff)%GREY_LEVELS;
}
```

A proper scaling by a factor r, i.e. by calling `noise(r*x,r*y,r*z)`, allows us to adjust the granularity of the resulting noise to the resolution of the triangle textures. On default the unit length of the noise is chosen to be equal to the the length of a texture pixel. Somewhat smoother results are obtained using a coarser noise.

3.4 Triangle Tiling

Modern graphics computers provide hardware support for texture mapping. This feature makes it possible to display textured objects at interactive frame rates. However, there are some restrictions with hardware texture mapping. First, all texture images have to fit into a special memory area of limited size[1]. Today this size is typically between 4 and 16 MB. Moreover, width and height of individual textures are restricted to powers of 2, i.e. textures have to be $2^i \times 2^j$ pixels large (sometimes i and j even must be equal).

Obviously under these constraints it is not a good idea to reserve a separate block of texture memory for every element of a surface triangulation. A large amount of expensive texture memory would be wasted. In addition the use of many small textures instead of a few big ones may affect rendering speed. Therefore we would like to place many small triangular patches into a single block of texture memory. The packing should be as dense as possible. If we require the longest edge of a triangle to be aligned horizontally then for each triangle only four orientations are possible, including left-handed arrangements. Although this greatly reduces the number of possible packings, we cannot expect to find an optimal solution. In fact, it can be shown that the problem is NP-complete [2].

To find a sub-optimal tiling of reasonable quality several heuristics can be pursued. The approach we have chosen relies on arranging similar triangles within rows. Two versions of this strategy have been implemented. The first one works on a discrete level by considering the exact shape of a triangle's texture region shown in Fig. 3. The second one only takes into account the analytical shape of a triangle. For triangles containing a large number of texture cells this is a sufficiently good approximation.

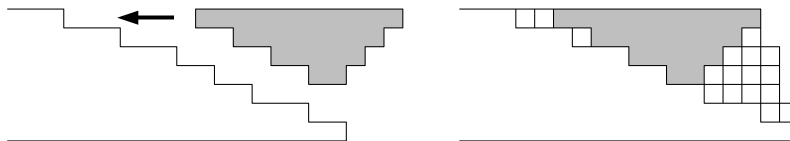
We start by placing the triangle of maximal height into the lower left corner of a quadratic texture region. The aim is to build a row of similar triangles. We consider only a limited set of orientations. The longest side of a triangle is either aligned to the bottom or to the top of a row. Our algorithm proceeds in a greedy way, i.e. it tries to find a triangle which optimally fits to the end of a row. The

[1]We expect that in future graphics architectures it will be possible to use main memory as texture memory. The memory saving techniques described in this section however will still be of practical interest.

local strategy is to minimize the wasted area between two subsequent elements. If a quadratic texture is completely filled we take a new one and proceed until no triangle is left. The height of the last filled texture might be reduced to some smaller power of 2. Our algorithm can be summarized by the following pseudo code:

(1) let T be a set of triangles
(2) **while** $T \neq \emptyset$
(3) take a new quadratic texture of size $s \times s$
(4) $height := 0$
(5) **while true**
(6) choose $t \in T$ with maximal height
(7) $height := height + height(t)$
(8) **if** $height > s$ **goto** 3
(9) $T := T \setminus \{t\}$
(10) insert t in a new row
(11) $length := length(t)$
(12) **while true**
(13) choose $t \in T$ yielding minimal waste
(14) $length := length + length(t)$
(15) **if** $length > s$ **goto** 6
(16) $T := T \setminus \{t\}$
(17) append t to end of row
(18) **end** (while)
(19) **end** (while)
(20) **end** (while)
(21) reduce size of last filled texture if possible

In the discrete version of the algorithm we run through all remaining triangles, simply counting the number of possibly wasted pixels. We do not only consider pixels to the left of a triangle, but also pixels filling a gap on the right hand side, as illustrated here:



Our strategy ensures that the right hand side of a row approximately takes the form of a straight line segment. This makes it easier to continue the row with minimal loss. Notice, that the number of possibly wasted pixels is just a criterium for finding the next triangle. The actual number of wasted pixels might be somewhat smaller because new triangles may occupy the empty area on the right.

With the greedy algorithm very efficient packings can be obtained. Typically 85-95% of the pixels in a quadratic texture region are covered with triangles. Examples of resulting arrangements are shown in Color Plate 3. The efficiency could

be further improved by filling the empty regions at the front and back of a row. This would require to allow $90°$ rotations instead of aligning all longest edges horizontally.

Counting the number of wasted pixels on a discrete level yields good results, but becomes quite slow if the number of triangles is large or if the triangles contain many texture cells. An alternative method would be to select the next triangle from a simple angle criterium. Assuming that the right hand side of the row looks like a straight line, we would like to find a matching triangle edge with similar slope. However, in addition to the angle α also the height h of the triangle is important. Ideally the height should be equal to the height of the row. In a simple ad-hoc method we can try to find a compromise between both criteria by minimizing the weighted sum

$$d^2 = c_\alpha(\alpha_{\text{row}} - \alpha)^2 + c_h(\frac{h_{\text{row}} - h}{h_{\text{row}}})^2.$$

The coefficients c_α and c_{row} are chosen such that the mean contribution of both terms in d^2 is similar. We evaluate this distance measure for both base angles of a triangle. The minimum value of all remaining triangles determines the next element to be added to the row together with its orientation. While this method produces somewhat sparser arrangements (80-90% coverage), in general it is faster than the discrete approach. Implementing the method by a linear search results in a time complexity of $O(n^2)$, where n is the total number of triangles.

A pair of values (α, h) can be interpreted as a point in the plane. Appending a triangle requires to solve a *proximity* or *closest–point problem* in (α, h)-space. Problems of this kind can be solved fast and efficiently using Voronoi-diagrams [11]. Using such data-structures it becomes possible to reduce the time complexity of the algorithm to $O(n \log n)$, at least for a fixed value of h_{row}.

4 Projecting 3D-Fields

As already been pointed out the projection of a truely 3D vector field onto an arbitray surface can be visualized in a straightforward way using our algorithm. The reason is that the projections of the 3-component vectors onto the triangles have to be computed anyway. However, it is a well-known fact that the structure of a 3D vector field can hardly be imagined from its projection onto a surface [9]. Using line integral convolution somewhat better results can be obtained by varying the length of the convolution filter according to the magnitude of the vector component tangential to the surface. In areas where the vector field is oriented almost perpendicular to the surface only very little blurring occurs, i.e. the input noise is visible instead of a convolved texture. An example of this technique is shown

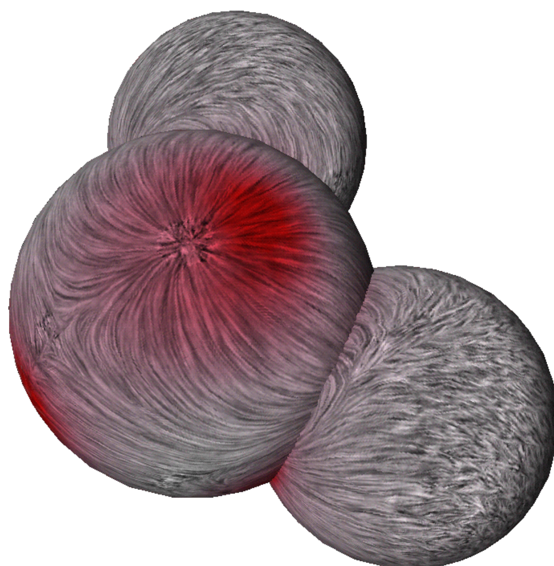


Figure 4: Electrostatic field of a water molecule projected onto a model of the molecular surface. Filter length in line integral convolution varies according to the magnitude of the tangential field component.

in Fig. 4. The image depicts the electrostatic field of a water molecule. Vanishing filter lengths indicate locations where the field is oriented perpendicular to the molecule's surface. Note, that it is still necessary to perform the convolution along projected stream lines. Convolving along the original 3D curves results in a poor pixel correlation, unless the field is almost tangential to the surface.

5 Applications and Results

We have applied our method to a number of data sets from various scientific areas. Color Plate 2 shows an example from differential geometry. The textures are of good quality. No artifacts due to the non-aligned texture grids occur between neighbouring triangles. Such artifacts can only be observed at high magnifications. They are even less apparent if bilinear texture interpolation is used instead of constant interpolation, as illustrated in Fig. 5.

Like in the planar case an additional scalar quantity can be visualized using color coded textures. Three times more memory is needed compared to greyscale textures. Usually we want to use hardware texture mapping to manipulate the textured geometry interactively. If 4MB texture memory is available, five color coded texture images of size 512^2 can be allocated. Most of our examples con-

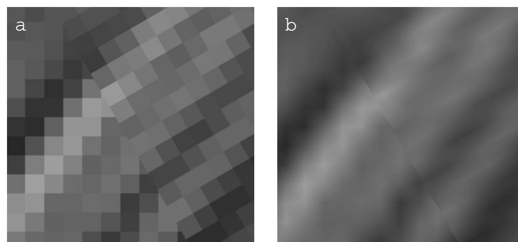


Figure 5: Close-up of textured triangles in case of (a) constant texture interpolation and (b) bilinear texture interpolation. Artefacts due to different texture space orientations are only visible at large magnifications.

tain about 1 million texture cells. Therefore using color coding a single surface will just fit into texture memory. If a higher resolution is requested or additional textured geometry is to be displayed more texture memory is needed.

In the 2D case image quality can be improved significantly by applying an emboss filter, cf. Fig. 1. This is no longer possible for surfaces. An emboss or gradient filter simulates the reflection of light shining diagonally onto the plane. In contrast, textured surfaces are rendered within a 3D scene which has its own lighting conditions. Depending on the orientation of the surface parts of it appear brighter than others. The lighting cannot be simulated on texture level because it is view dependent. However, a very similar effect can be obtained using bump mapping. There the surface normals are tilted according to the gradient of a 2D scalar texture, simulating the effect of a rough surface. Unfortunately, bump mapping isn't supported by the OpenGL graphics interface. However, many software renderers do support bump mapping. In Color Plate 4 an example of a bump-mapped LIC texture is depicted. The image contains a stream surface computed for the electro-static field of a benzene molecule. This is also an example of a surface which has no natural parametrization. The surface has been computed using an algorithm similar to the one described in [7].

The computation times for the examples shown did not exceed one minute on a SGI Indigo^2 250 MHz workstation. As already mentioned, about 1 million texture cells have typically been processed. The surfaces contained 1600-4000 triangles. Our code has not been fully optimized. We believe that it can be further accelerated in future.

Acknowledgements: Thanks to Malte Zöckler and Roland Wunderling for useful discussions, to Uwe Schwarz for generating the Klein bottle, and to Thomas Steinke for providing molecular electrostatic field configurations.

References

[1] G. ADELSON-VELSKII AND Y. LANDIS, *An algorithm for the organization of information*, Soviet Math. **3** (1962), 1259–1263.

[2] H. BATTKE, *Entwicklung texturbasierter Methoden zur Vektorfeldvisualisierung (in German)*, Master's thesis, Konrad-Zuse-Zentrum Berlin und Humboldt-Universität zu Berlin, Fachbereich Informatik, 1996.

[3] B. CABRAL AND L. C. LEEDOM, *Imaging vector fields using line integral convolution*, Computer Graphics (SIGGRAPH '93 Proc.), **27** (1993), 263–272.

[4] L. K. FORSSELL, *Visualizing flow over curvilinear grid surfaces using line integral convolution*, Visualization '94, 1994, 240–247.

[5] A. GLOBUS, C. LEVIT, AND T. LASINSKI, *A tool for visualizing the topology of three-dimensional vector fields*, Visualization '91, 1991, 33–40.

[6] A. J. S. HIN AND F. H. POST, *Visualization of turbulent flow with particles*, Visualization '93, 1993, 46–52.

[7] J. P. M. HULTQUIST, *Constructing stream surfaces in steady 3d vector fields*, Visualization '91, 1992, 171–177.

[8] W. C. DE LEEUW AND J. J. VAN WIJK, *Enhanced spot noise for vector field visualization*, Visualization '95, 1995, 233–239.

[9] N. MAX, R. CRAWFIS, AND C. GRANT, *Visualizing 3d velocity fields near contour surfaces*, Visualization '94, 1994, 248–255.

[10] K. PERLIN, *An image synthesizer*, Computer Graphics **19**:3 (1985), 287–296.

[11] F. P. PREPARATA AND M. I. SHAMOS, *Computational geometry, an introduction*, Springer Verlag, New York, 1985.

[12] D. STALLING AND H.-C. HEGE, *Fast and resolution independent line integral convolution*, SIGGRAPH 95 Conference Proceedings, 1995, 249–256.

[13] G. WARD, *A recursive implementation of the perlin noise function*, Graphics Gems 2 (J. ARVO, ed.), Academic Press Inc., 1991, 396–401.

[14] J. J. VAN WIJK, *Spot noise*, Computer Graphics **25**:4 (1991), 309–318.

[15] J. J. VAN WIJK AND W. C. DE LEEUW, *A probe for local flow field visualization*, Visualization '93, 1993, 39–45.

Appendix: Color Plates

Plate 1 *No distortions occur if a separate LIC texture is computed for each triangle. The texture charateristics remain constant throughout the whole surface.*

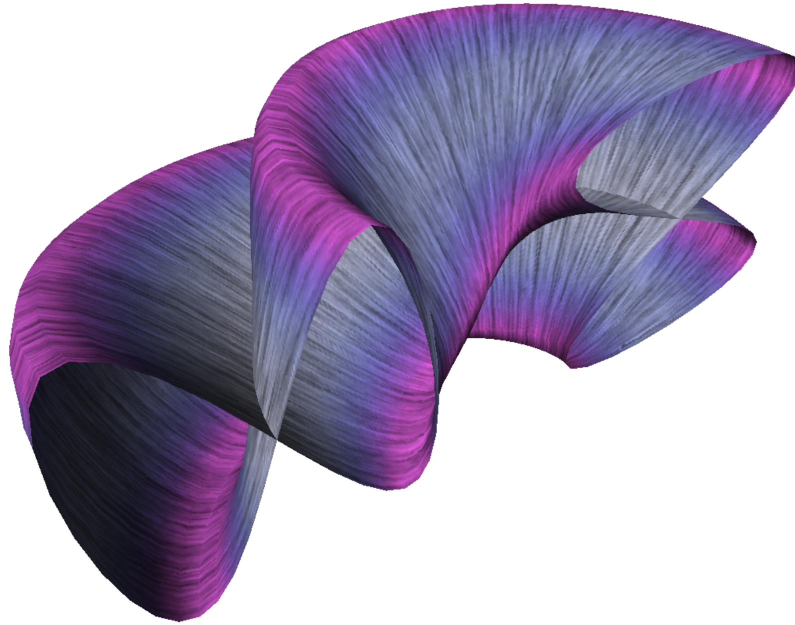


Plate 2 *Constant directional field in parameter space on a surface with the topology of a Klein bottle.*

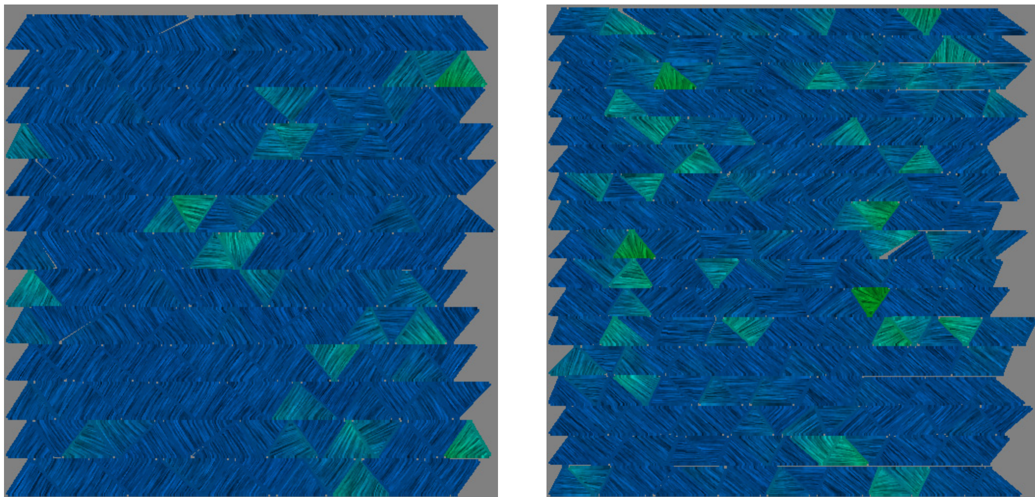


Plate 3 *To make optimal use of texture memory similar triangle patches are arranged in rows. The packing algorithm proceeds in a greedy way: rows are filled sequentially by taking the optimal triangle according to some local cost function.*

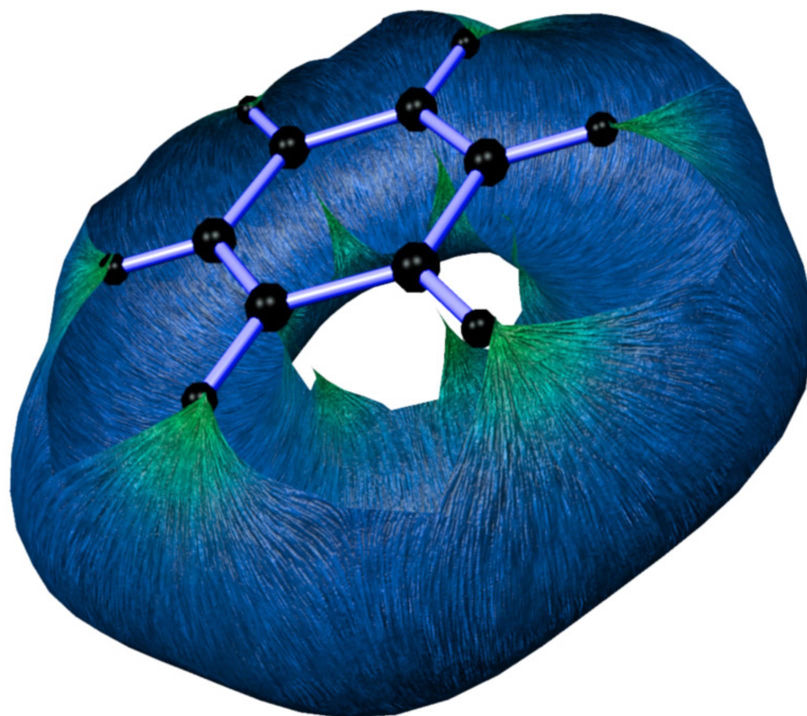


Plate 4 *Electrostatic field of a benzene molecule — displayed using a bump-mapped LIC texture. The surface has been computed by connecting multiple field lines.*