6.838: Shape Analysis, 2017
Instructor: Justin Solomon

Direction Fields on Triangle Meshes for Non-Photorealistic Rendering

Parker Greene[1]

[1]MIT Department of Electrical Engineering and Computer Science

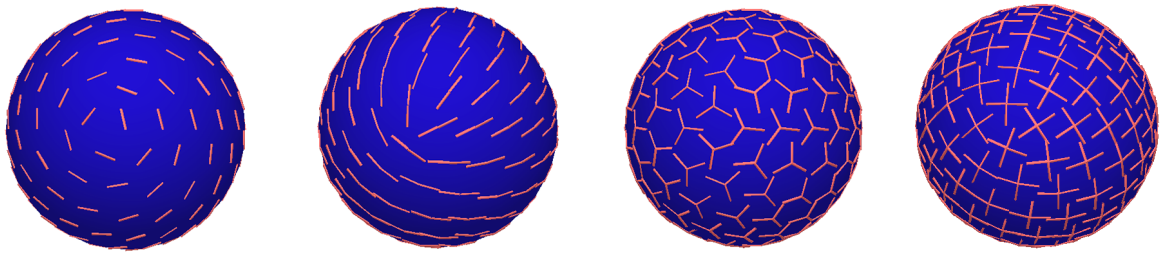

Figure 1: *Globally smoothest direction fields with degrees 1, 2, 3, 4 (right to left) on a sphere*

Abstract

We present an implementation of the state-of-the-art direction field from [KCPS13] and further extend it for non-photorealistic rendering. By utilizing this approach, we can render triangle meshes in real-time with smooth lines or hatching textures. These textures can vary depending on the degree of the direction field and also whether the field is curvature aligned or not.

1. Introduction

A direction field refers to a collection of vectors tangent to a surface, and an n degree direction field refers to such a field, but at each point, p, there are n such vectors with origin p and directions evenly spaced. That is, a degree 1 direction field will have a single vector at each point, a degree 2 vector field will have 2 vectors π radians apart, a degree 3 field will have 3 vectors rotated $\frac{2\pi}{3}$ radians apart, and so on. See Figure 1 for examples of direction fields of varying degrees.

Direction fields with degree ≥ 2 have nice visual properties, and because of that, they work well for rendering non-photorealistic (NPR) textures on triangle meshes. In particular, direction fields with degree 2 resemble line drawings with smooth, non-connecting lines, and direction fields with degree 4 resemble cross hatches. We can also get direction fields that align with principal curvature directions of the mesh, which, in the case of degree 4 fields, helps to convey shape over the smooth fields.

The algorithm for computing the globally optimal direction field on the surface involves essentially four steps: **(1)** choose an arbitrary edge (i, j_0) adjacent to each vertex v_i and compute the *scaled* angle between (i, j_0) and all the other edges (i, j) adjacent to v_i; **(2)** compute the parallel transport coefficients r_{ij} between each pair of vertices (i, j) that share an edge and use this to compute the holonomy Ω per-face; **(3)** Use the holonomy and parallel transport coefficients to assemble a mass matrix M and an energy matrix A; **(4)** Use M and A to solve either a minimum eigenvalue problem (smoothest field) or a linear system (curvature aligned field).

2. Related Work

Previous work in computing direction fields have mainly been in applications involving non-photorealistic rendering [HZ00], remeshing [KNP07], parameterization of meshes [CSM03], and texture synthesis [LH00]. Previous approaches to approximate optimal direction fields have typically formulated energies in such a way that their optimization problems are non-convex and therefore NP-hard to minimize.

In the area of NPR, Hertzmann and Zorin [HZ00] were concerned with achieving a degree 4 direction field for illustrating

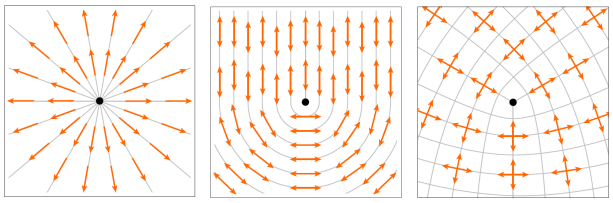

Figure 2: *Examples of direction fields of degree 1 (left), 2 (middle), and 3 (right) near singularities. These types of singularities can also be seen in the spheres from Figure 1. Figure from [KCPS13].*

smooth surfaces while mimicking line-art drawing with cross-hatching. Their energy formulation was nonconvex, however, which meant the output was not necessarily optimal and was largely dependent on initialization. [PHWF01] introduces an NPR technique capable of rendering hatching textures on triangle meshes in real-time using tonal art maps. This approach renders NPR styles that are locally coherent, but don't necessarily have global coherence or continuity, and it also requires the user to specify tonal art maps to be used in the rendering process.

The current state-of-the-art technique for computing direction fields comes from Knöppel et. al. [KCPS13] in which they formulate an energy matrix whose smallest eigenvalue corresponds to the eigenvector u that formulates the globally smoothest direction field. Each element u_i of the minimum eigenvector u is a complex number whose angle determines the rotation of the corresponding vertex's basis vector. This result is an improvement upon a previous technique from [RVLB08], in which they formulate the smoothness energy using the angles between edges, but still are left with a mixed integer optimization problem, which is NP-hard to minimize. [BZK09] improves upon Ray et. al. slightly and tends to produce lower energies, but is still not globally optimal. [KCPS13] compares their "globally optimal" direction field to the "mixed integer" direction field from Bommes et al., determining that their approach yields much better results.

3. Technical Approach

Note that we will leave out some details that are described in [KCPS13], and instead mainly focus on details that were omitted there but necessary to the implementation.

The approach relies on representing the direction fields as n evenly numbers per-point on the complex plane, where n is the degree of the direction field. After computing the n-degree direction field, we get as output a vector u of complex numbers and one complex number $u_i = \phi$ per vertex. We can then recover n per-point vectors by taking the n^{th} root of ϕ. That is, we compute

$$|\phi|^{1/n} e^{\mathrm{i}(\arg(\phi)/n + 2\pi k/n)} \text{ for } k = 0, 1, \ldots n-1$$

Where $\mathrm{i} = \sqrt{-1}$, $|\phi|$ is the magnitude of the complex number ϕ, and $\arg(\phi)$ is the angle that ϕ makes on the complex plane. This notation will be used throughout the rest of this paper.

Each per-point complex number ϕ encodes a direction, but each direction is with respect to its basis vector. Since we chose each basis vector arbitrarily, we need to *parallel transport* vectors between tangent spaces in order to compare them.

In practice, when we discretize the surface into the sets of vertices, edges, and faces (V, E, T, resp.) our tangent spaces are defined per-vertex. We define the tangent space in the discrete setting for a vertex i as i and all its neighboring vertices. Because we define tangent spaces this way, we must take into account the fact that each vertex's tangent plane doesn't actually exist in a plane at all. That is, the sum of angles around a vertex i in general do not sum to 2π. By "scaling" all the angles around a vertex so that they do sum to 2π, we essentially flatten the tangent space for a vertex. The scaling factor s_i for vertex i is defined as

$$s_i := \frac{2\pi}{\sum_{t_{ijk} \ni i} \alpha_i^{jk}}$$

Where $t_{ijk} \ni i$ refers to the set of all triangles adjacent to i, and α_i^{jk} is the angle opposite edge (j,k) in t_{ijk}.

We then can compute parallel transport from vertices i and j. Let

$$\rho_{ij} = n(s_j \theta_{ji} - s_i \theta_{ij} + \pi)$$

where θ_{ij} is the total angle (resp. ccw) measured between the reference vector X_i for vertex i and the edge e_{ij}, and as before, s_i is the scaling factor for i that flattens the tangent space. Therefore, $s_i\theta_{ij}$ refers to the rescaled angle between X_i and e_{ij} (and similarly for $s_j\theta ji$). We then need to add π to the difference, since θ_{ij} is in reference to e_{ij}, but θ_{ji} is in reference to e_{ji}, since we do not take into account edge orientation when computing each θ_{ij}. Since e_{ij} is exactly π radians from e_{ji} after rescaling, adding π essentially "flips" the edge in the calculation.

The parallel transport factors, r_{ij} are defined per-edge as

$$r_{ij} := e^{\mathrm{i}\rho_{ij}}$$

In other words r_{ij} is the complex number with angle in the complex plane equal to ρ_{ij}. We can then compute holonomy per-face. Holonomy for triangle t_{ijk} is defined as

$$\Omega_{ijk} = \arg(r_{ij} r_{jk} r_{ki})$$

Holonomy per-face is related to the gaussian curvature of the face in that $\Omega_{ijk} = nK_{ijk}|t_i jk|$, where K_{ijk} is the Gaussian curvature and $|t_{ijk}|$ is the area of triangle t_{ijk}. An example of this relationship on a triangle mesh is shown in Figure 3

Using these values, then, we can assemble the mass matrix M and energy matrix A as they are defined in [KCPS13].

The computation for calculating the smoothest direction field over the triangle mesh boils down to solving

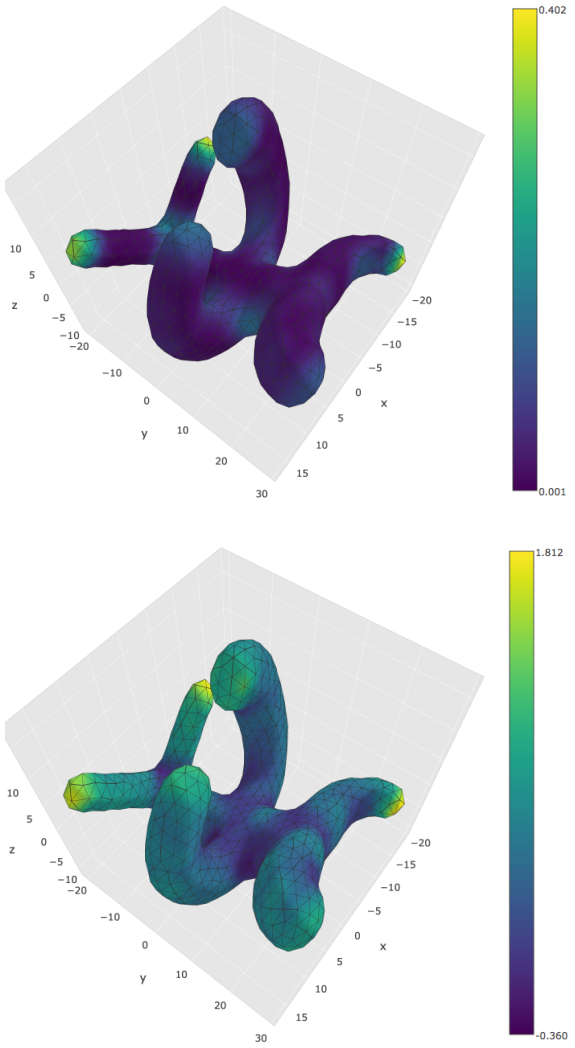

Figure 3: *Direct holonomy computation per-face (top) and the corresponding Gaussian curvature calculated from this holonomy (bottom)*

$$Au = \lambda Mu$$

for the length $|V|$ vector u of complex numbers.

The computation for calculating a curvature aligned direction field is a bit more involved, as we have to first compute q according to [KCPS13]. The linear system to solve is

$$(A - \lambda_t M)\tilde{u} = Mq$$

Where u is $\tilde{u}$ normalized and λ_t is iteratively found, usually starting with $\lambda_t = 0$.

4. Implementation

This section details the implementation of the algorithm (section §4.1) as outlined above, then proceeds to detail how to visualize the field on a triangle mesh in practice (section §4.2), and then provides an extension of the algorithm for NPR (section §4.3).

4.1. Direction Field Computation

The implementation for computing either the smoothest field or the curvature-aligned field follows the approach from section §3, for the most part. An outline of the process is shown in Figure 4.

To begin, we compute the sparse adjacency matrix G for the triangle mesh, which we can then use to choose an arbitrary edge X_i for every vertex i, which will be our reference vector for parallel transport. We then use X_i to compute θ_{ij} for all j where an edge exists between i and j (note that this simply comes from the non-zero elements in row i of G). Recall that θ_{ij} is the euclidean angle between X_i and e_{ij} *before* scaling. Because the tangent plane around i is not flat, to compute each θ_{ij}, we must sum the tip angles of individual triangles around vertex i. This allows us to keep a running total as we iterate between adjacent triangles counterclockwise around i, where each subtotal at edge e_{ij} becomes θ_{ij}, and the full total $\theta_{i,\text{total}}$ is related to s_i in that $s_i = \frac{2\pi}{\theta_{i,\text{total}}}$.

We can then compute r_{ij} straightforwardly, as described in section §3 for each i, j pair (where $e_{ij} \in E$) using s_i, s_j, θ_{ij}, and θ_{ji}. Once we have all r_{ij}, we compute the holonomy per-face. Detailed pseudocode on computing s, r, θ, and Ω is listed in algorithm 1.

In practice, to compute the mass matrix M and the energy matrix A, we can omit some details of the problem a bit. Specifically, we will only consider the case where $s = 0$, and therefore, from [KCPS13], we have that $A = \Delta$ (Note that s is just a parameter that allows some control over the number of singularities in the field–setting $s = 0$ still gives us a globally optimal direction field).

While not explicitly stated in [KCPS13], we can take the mass matrix to be $\frac{1}{2}$ the standard diagonal mass matrix from FEM. Again, here we do a bit of simplification, but it still produces the correct results. We then iterate over faces of the triangle mesh to assemble A using r and Ω.

4.2. Visualizing the Direction Field

Though there are a few ways to visualize the direction field, the simplest way is to treat the output vector u as rotations of the basis vectors: for a given vertex i, u_i determines the rotation of basis vector X_i. That is, after we recover the k^{th} root of the complex number u_i (call this number $u_i^{(k)}$), we can rotate X_i by $arg(u_i^{(k)})$ radians.

This would work for the continuous case, where the tangent plane is locally flat around a point; however, in the discretized case, the tangent plane for a vertex is not flat in general. In treating the rotations in this way, we're ignoring the local geometry around vertex i, and instead treating the tangent plane of vertex i as the plane containing X_i. Because we're essentially ignoring the local curvature of vertices, problems can arise in areas of the mesh where we have vertices with adjacent triangles with large variations in their normals, as seen in Figure 5.

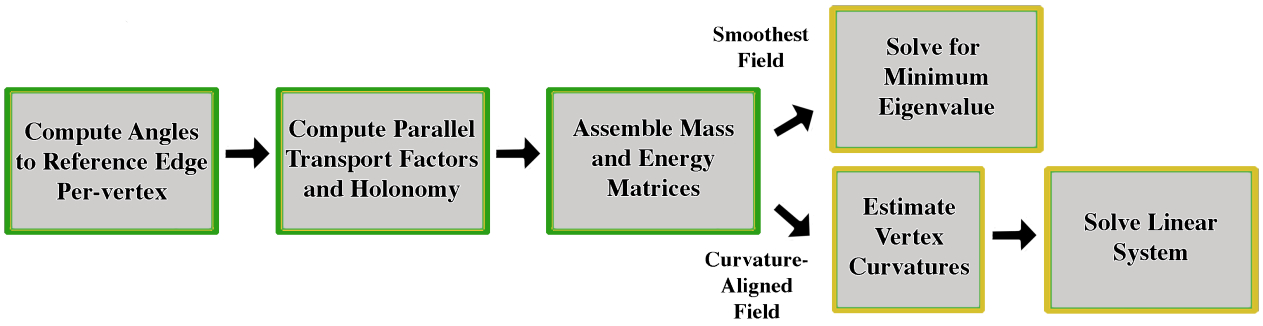

Figure 4: *An outline of the algorithm for computing direction fields. After computing the mass and energy matrices, we can either solve a minimum eigenvalue problem for smoothest field or solve a linear system involving the principal curvature estimates per-vertex.*

input : triangle mesh K and corresponding adjacency matrix G
output: r, Ω, θ
for $v_i \in V$ **do**
  // choose basis vector X_i and compute scaling factors s_i for each vertex
  choose arbitrary edge $X_i \in G[i]$;
  get all neighbors of v_i from $G[i]$;
  sort neighbors of v_i by ccw angle from X_i;
  $s_i \leftarrow 0$;
  for $v_j \in$ *sorted neighbors of* v_i **do**
    $\theta_{ij} \leftarrow s_i$;
    $s_i \leftarrow s_i$ + angle between X_i and e_{ij};
  end
  $s_i \leftarrow \frac{2\pi}{s_i}$;
end
// parallel transport factors per edge
for $e_{ij} \in E$ **do**
  $\rho_{ij} \leftarrow n(s_j\theta_{ji} - s_i\theta_{ij} + \pi)$;
  $r_{ij} \leftarrow e^{i\rho_{ij}}$;
end
// holonomy calculation
for $t_{ijk} \in T$ **do**
  $\Omega_{ijk} \leftarrow \arg(r_{ij}r_{jk}r_{ki})$
end

Algorithm 1: Computing holonomy Ω per-face, and r_{ij} (parallel transport factors) and θ_{ij} for all edges $e_{ij} \in E$

The alternative, then, is to first flatten the triangles around vertex i using the scaling factor s_i from section §3 then rotate the basis vector X_i in this flattened space. Once we have the rotated basis vector, which represents our direction field at vertex i, we can map it to the plane of the triangle when we "unflatten" the triangles around i.

However, in practice, it is fine to use the former method mentioned above and just simply rotate X_i naïvely in its plane. As shown in Figure 5, globally, the direction field is still largely visible, except for some areas of positive or negative Gaussian curvature. To render a degree n direction field, we simply map cylindrical primitives to each vertex v_i and align them with the n vectors corresponding to the n directions of the field at v_i.

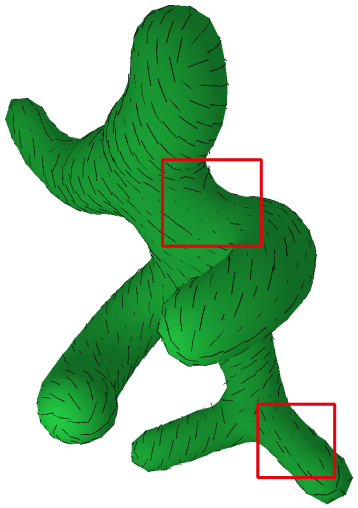

Figure 5: *Areas with large curvature tend to occlude the direction field vectors when visualizing with naive rotations, making them difficult to see (a few of those areas shown in red boxes). Globally, however, the direction field is very easy to see.*

4.3. Non-Photorealistic Rendering

Direction fields lend themselves quite nicely to non-photorealistic rendering (NPR). This area of computer rendering is concerned with displaying scenes that mimic hand-drawn artistic styles. In particular, direction fields closely mimic line drawings (or similarly, contour-hatching) in the case where $n = 1$ and $n = 2$ and cross-hatching in the case where $n = 4$. Hand drawn examples of these styles can be seen in Figure 4.3.

An alternative approach to rendering hatching textures would be to use a technique similar to Praun et. al. [PHWF01] in which they convey a value gradient on the surface by varying the density of their drawn lines, which works quite well for conveying light and

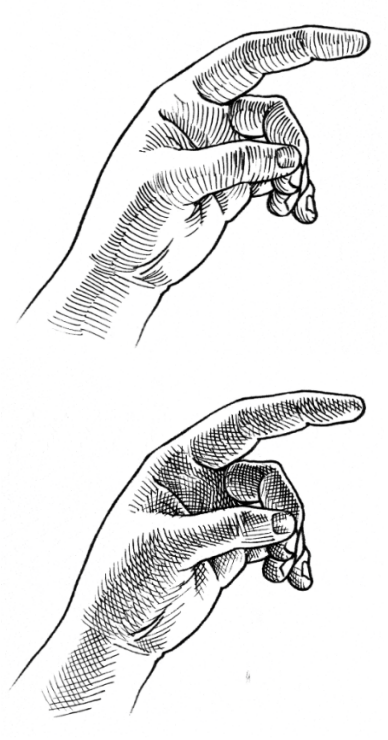

Figure 6: *examples of hand-drawn contour-hatching (top) and cross-hatching (bottom). Link to source.*

dark areas of the shape, qualitatively. In the context of direction fields, this would be equivalent to varying the field degree, n to get light and dark values. Figure 8 shows the gradient of the field as we vary n as output by my implementation. The problem with using fields of varying degree on the same shape, however, is that a degree $n = 2$ field is not necessarily aligned with an $n = 4$ field, so if we use the $n = 4$ field for darker areas and the $n = 2$ field for lighter areas, they won't align in general. Also, this would mean we have to compute multiple sets of matrices M and A to calculate each direction field. Recall from §3 that the computation of M and A is dependent on the value for n. Therefore, in practice, we use attenuation technique as mentioned above on a single direction field of some degree n.

In order to render surfaces with NPR textures, we use a white background and triangle mesh, and render the direction field on the surface as n primitives at each vertex $v_i \in V$ aligned with the n degree direction field at v_i, similar to what we did to visualize the direction field in the previous section. We then attenuate the of the primitives' values at vertex v_i based on the location of the light source using standard diffuse shading. One property of rendering the textures in this way is that, as opposed to mapping the texture directly to each triangle, such as in [PHWF01], the primitives aren't bound to the plane of the triangles–that is, the texture in some sense can leave the surface, giving a much rougher looking texture, similar qualitatively to the NPR textures generated in [MKG*97]. In

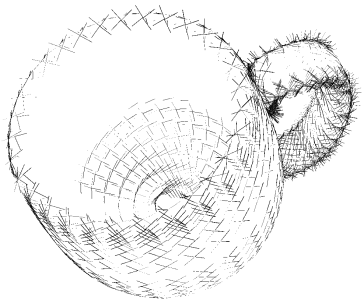

Figure 7: *example of the hatching texture exhibiting bad behavior on areas of large Gaussian curvature, particularly on the handle and rim of the cup. This results in a "spiky" texture in those areas.*

areas of large positive Gaussian curvature; however, the textures can extrude too far from the surface as in Figure 4.3.

Using direction fields to guide hatching textures gives some guaranteed global coherence among all hatches on the surface, which isn't necessarily given from hatching methods such as [PHWF01]. The downside to rendering the entire direction field for every vertex is that the texture resolution is entirely dependent on the resolution of the input triangle mesh. See section §5.2 for examples of renderings of different objects with hatching textures.

5. Results

This section details the output of my implementation for computing direction fields (section §5.1) and qualitative results in using the computed direction fields for NPR applications (section §5.2).

All of the code for computing the direction fields was written in Python. The visualization tools were written in C++ using OpenGL. The eigenvalue problem and linear system were solved using Python's Scipy sparse matrix methods. All times were recorded on a 2.6 GHz Intel Core i5 Macbook. The methods for building M and A are not optimized, and therefore we reference computation times for only the eigenvalue and linear system solving (for smoothest and curvature-aligned fields, respectively), assuming M and A are pre-computed.

5.1. Fields

Figure 9 shows curvature-aligned direction field results for a mesh with varying resolutions. Fields on meshes of $|T| = 1$k, $|T| = 2$k, and $|T| = 4$k, were computed in 5ms, 10ms, and 25ms on average, respectively. Figure 10 shows varying fields of varying degrees on a mesh with $|T| = 4$k, which were all computed in 25ms, on average. Figure 11 shows a genus 1 surface with $|T| = 10k$ that was computed in 159ms on average. Figure 12 compares a smoothest field with a curvature-aligned field on a mesh with $|T| = 2k$. The curvature-aligned field computation took about 10ms, whereas the smoothest field computation took about 170ms. In general, the smoothest field eigenvalue problem takes a bit longer to compute versus the curvature-aligned linear system.

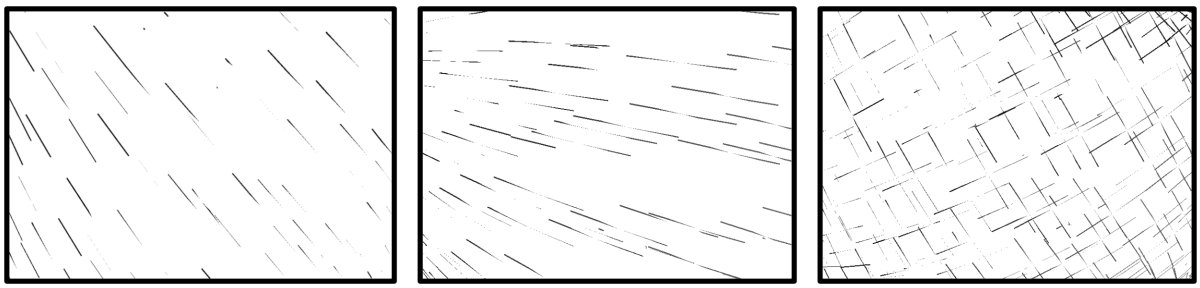

Figure 8: *Examples of NPR textures generated on a triangle mesh. The direction fields have degrees $n = 1$, 2, and 4 (right to left, respectively). As n increases, the texture overall gets a bit darker.*

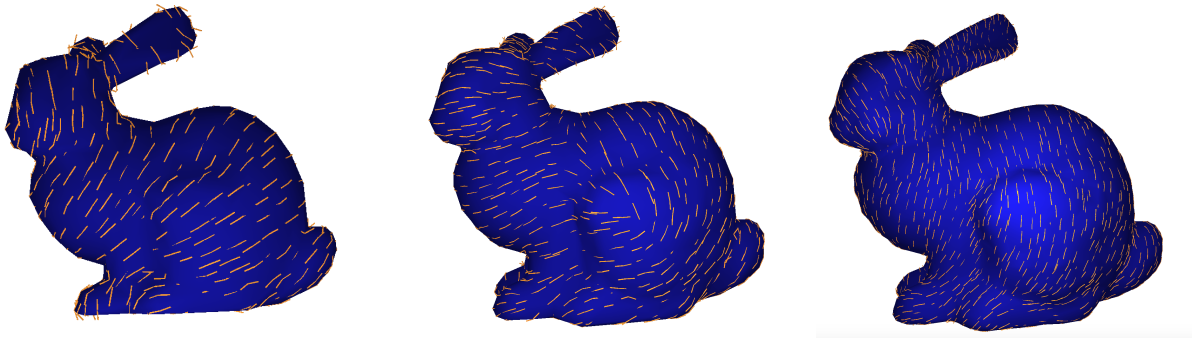

Figure 9: *Curvature-aligned direction field of degree $n = 2$ for a mesh with $|T| = 1k$, 2k, and 4k (left to right, respectively). The field's direction changes slightly for different resolutions of the mesh, but the overall quality doesn't change much when increasing the resolution.*

5.2. NPR

In general, I found, qualitatively, that smoothest direction fields tend to produce better NPR texture results when $n = 2$ (mimicking contour hatching), and that curvature-aligned direction fields are better when $n = 4$ (mimicking cross hatching). Figures 13 and 14 show these results. The fields are both fairly smooth and have nice global coherence.

As mentioned in section §4.3, since we are displaying the direction field at every vertex, as we increase the number of vertices of the mesh, the texture will become quite small. This is good if we want a very fine texture, but unfortunately its bad if we don't. Figure 15 displays the texture meshes of different resolutions.

6. Conclusion and Future Work

Though the field (pun intended) of non-photorealistic rendering has been around for decades, methods that relied on direction fields to generate hatching textures could not create any global coherence, since earlier techniques to generate direction fields were not guaranteed any global optimality. Using a Python implementation of [KCPS13], we generated globally optimal direction fields that we then, in turn, used these to generate hatching textures that have global coherence. Depending on the computation of the direction field, these hatching textures can be either curvature-aligned or smoothest on the surface, and we can also adjust the degree n of the direction field to change the hatching style.

In the future, I would like to change actually map the NPR texture to the triangle mesh, rather than display it as primitives. This would be similar to what [PHWF01] does in their method. Another possible extension for NPR with direction fields would be to render painterly textures, as the direction fields nicely capture the curvature of the surface, which is essential to a lot of painting styles, such as impressionism and Fauvism.

All source code is available at https://github.com/pwgreene/ DirectionFields

References

[BZK09] BOMMES D., ZIMMER H., KOBBELT L.: Mixed-integer quadrangulation. *ACM Trans. Graph. 28*, 3 (2009). 2

[CSM03] COHEN-STEINER D., MORVAN J.: Restricted delaunay triangulations and normal cycle. *Proc. Symp. Comp. Geom.* (2003), 312–321. 1

[HZ00] HERTZMANN A., ZORIN D.: Illustrating smooth surfaces. *Proc. ACM/SIGGRAPH Conf.* (2000), 517–526. 1

[KCPS13] KNÖPPEL F., CRANE K., PINKALL U., SCHRÖDER P.: Globally optimal direction fields. *ACM Trans. Graph. 32*, 4 (2013). 1, 2, 3

[KNP07] KÄLBERER F., NIESER M., POLTHIER K.: Quadcover - surface parameterization using branched coverings. *Comp. Graph. Forum 25*, 3 (2007), 375–384. 1

[LH00] LEFEBVRE S., HOPPE H.: Appearance-space texture synthesis. *ACM Trans. Graph. 25*, 3 (2000), 541–548. 1

[MKG*97] MARKOSIAN L., KOWALSKI M. A., GOLDSTEIN D.,

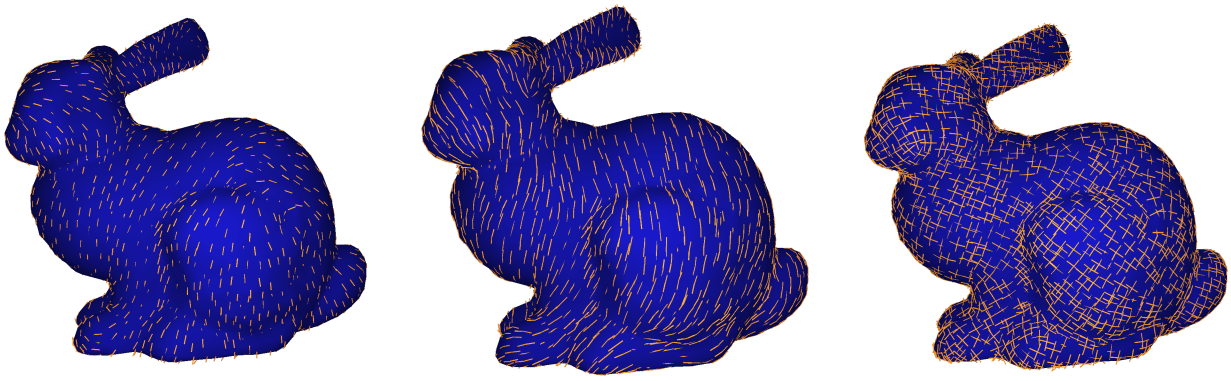

Figure 10: *Curvature-aligned field of degrees $n = 1$, 2, and 4 on a mesh with $|T| = 4k$*

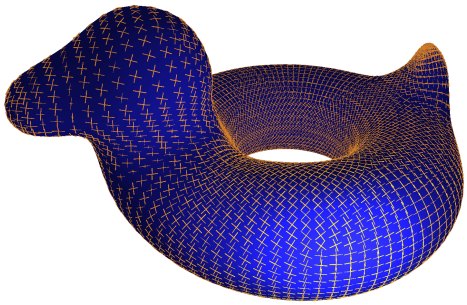

Figure 11: *A degree $n = 4$ curvature-aligned direction field on a mesh with genus 1 and $|T| = 10k$. The degree 4 field does pretty well at aligning with the principal curvature directions*

.

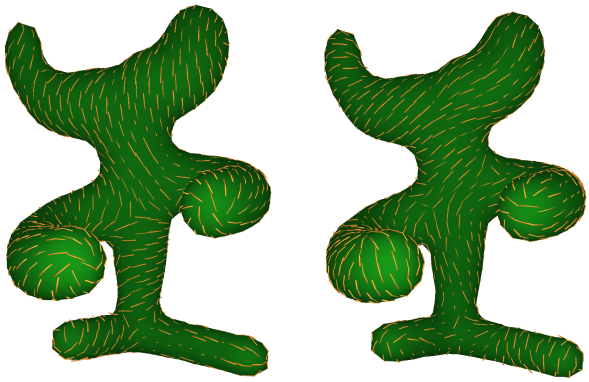

Figure 12: *Smoothest direction field (left) compared against a curvature-aligned field (right). Both are degree $n = 2$. The smoothest field tends to avoid singularities except in areas of large Gaussian curvature magnitude, whereas the curvature-aligned field is more concerned with following the principal curvature directions.*

TRYCHIN S. J., HUGHES J. F., BOURDEV L. D.: Real-time nonphotorealistic rendering. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 415–420. URL: http://dx.doi.org/10.1145/258734.258894. 5

[PHWF01] PRAUN E., HOPPE H., WEBB M., FINKELSTEIN A.: Real-time hatching. *ACM SIGGRAPH* (2001). 2, 4, 5, 6

[RVLB08] RAY N., VALLET B., LI W. C., B. L.: N-symmetry direction field design. *ACM Trans. Graph. 27*, 2 (2008), 10:1–10:13. 2

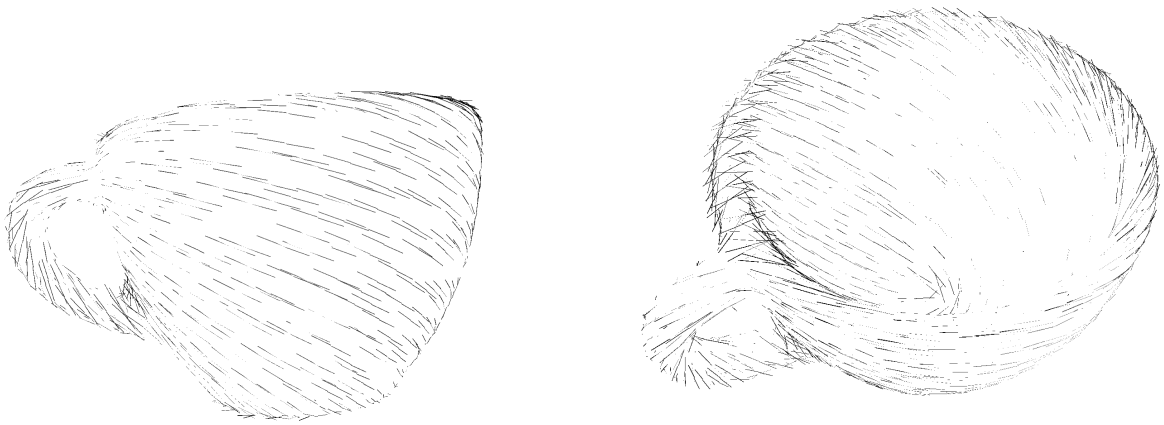

Figure 13: *NPR texture on a triangle mesh viewed at two different angles. A globally smooth field of degree* 2 *resembles contour hatching.*

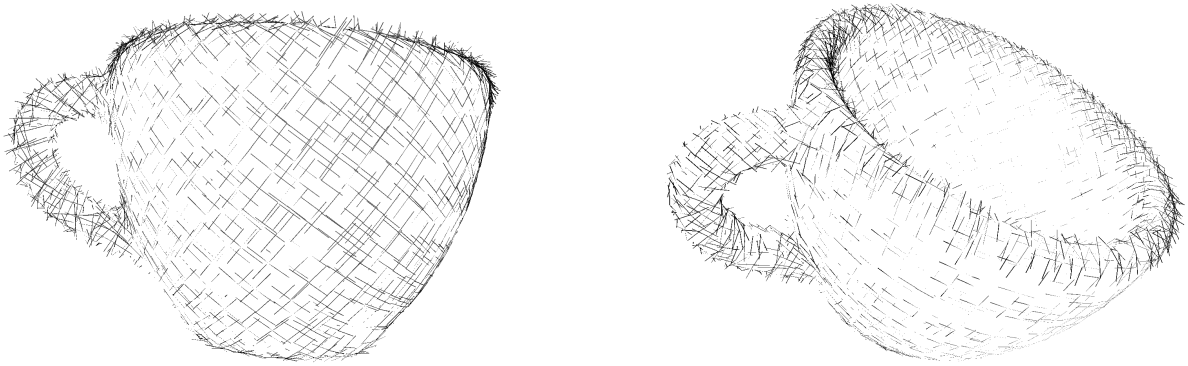

Figure 14: *NPR texture on a triangle mesh viewed at two different angles. A curvature-aligned field of degree* 4 *resembles cross hatching. A video of this result can be found at https://youtu.be/3P3TZ3UBPak*

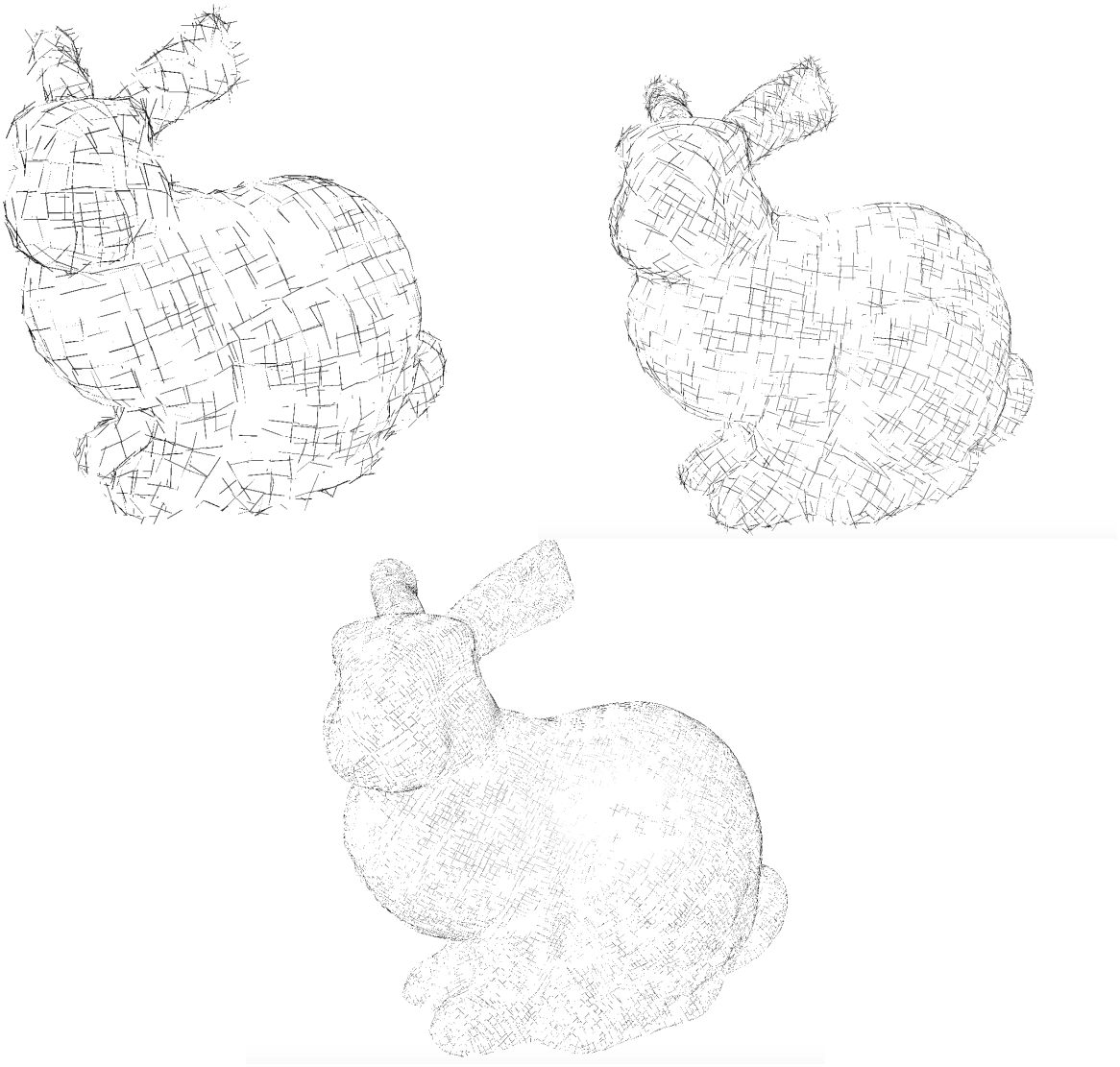

Figure 15: *Cross hatching texture on meshes of $|T| = 2k$, $|T| = 4k$, and $|T| = 28k$. As we increase the number of triangles, the number of vertices also increases and so we get a much more fine-grained texture*