A Primitive for Manual Hatching

GREG PHILBRICK and CRAIG S. KAPLAN, University of Waterloo, Canada

In art, hatching means drawing patterns of roughly parallel lines. Even with skill and time, an artist can find these patterns difficult to create and edit. Our new artistic primitive—the hatching shape—facilitates hatching for an artist drawing from imagination. A hatching shape comprises a mask and three fields: width, spacing, and direction. Streamline advection uses these fields to create hatching marks. A hatching shape also contains barrier curves: deliberate discontinuities useful for drawing complex forms. We explain several operations on hatching shapes, like the multi-dir operation, an easy way to depict 3D form using a hatching shape's direction field. We also explain the modifications to streamline advection necessary to produce hatching marks from a hatching shape.

CCS Concepts: • **Computing methodologies** → **Non-photorealistic rendering**; • **Human-centered computing** → *Gestural input*.

ACM Reference Format:
Greg Philbrick and Craig S. Kaplan. 2021. A Primitive for Manual Hatching. 1, 1 (December 2021), 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

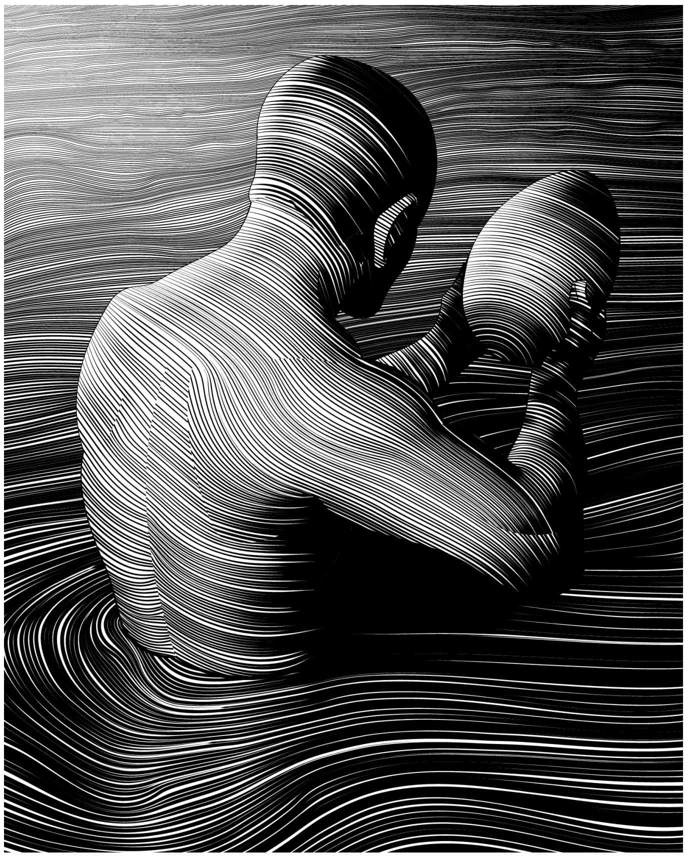


Fig. 1. The hatching marks come from hatching shapes: one shape for the human figure, one for the egg, and two for the water. Except where noted, art is by the first author.

1 INTRODUCTION

The term *hatching* refers to groups of roughly parallel curves in a drawing. This technique has a long history, particularly in print media like woodcut, and is still in use. It often serves as halftoning, with marks blending visually to suggest various tones. Hatching can also portray texture and the curvature of three-dimensional forms.

For an artist drawing marks one at a time, hatching is hard to create and even harder to edit. Altering a drawing means adding marks to an existing hatching pattern, which is prone to messiness; carefully erasing specific marks; or erasing the entire pattern and starting over. The difficulty is highest when hatching is dense, neat, and complex.

In a digital context, an artist might use a specialized primitive to direct a group of hatching marks collectively rather than adding or removing them individually. An experienced artist will want such a primitive to operate just above the level of individual marks, coordinating a group of marks without automating the entire drawing process.

The *hatching shape* (Figure 1) is our answer to this need. A hatching shape's purpose is to be more convenient to create and edit than a group of unorganized hatching marks. A hatching shape is a kind of *patch* [Philbrick and Kaplan 2019], which is just a group of marks that run together—the fundamental unit of hatching.

A hatching shape defines hatching within a bounding box. Three fields give hatching marks' widths, directions, and the spacings between them. The artist manages these fields and streamline advection [Jobard and Lefer 1997] uses them to produce hatching marks. A mask specifies where marks are visible within the bounding box.

A hatching shape also includes a collection of *barrier curves*, which induce discontinuities. Figure 2 shows two hatching shapes, one with and one without a barrier curve. Figure 1 shows the importance of barrier curves in an actual illustration. The torso has discontinuities in the picture plane which hatching marks should not cross, such as the edge of the right shoulder blade (Figure 3). Furthermore, a *single* hatching patch (produced by a single hatching shape) does the job of rendering the entire torso. Using a single patch to render a complex form like this is especially difficult. It is more common to break such a form into multiple patches, as in Figure 4. It is not necessarily *better* to use a single patch, but it is usually harder, a show of virtuosity. The show of virtuosity in Figure 1 would be impossible without the discontinuities created by barrier curves.

Authors' address: Greg Philbrick, gphilbri@uwaterloo.ca; Craig S. Kaplan, csk@uwaterloo.ca, University of Waterloo, Canada.

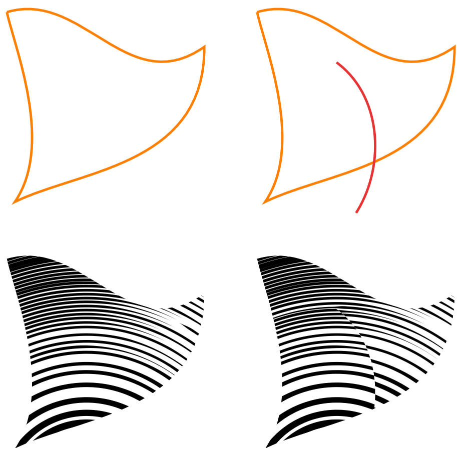


Fig. 2. At top left is a hatching shape mask. At top right is the mask plus a barrier curve. Below is the hatching produced (by our implementation) without the barrier curve (left) and with it (right).

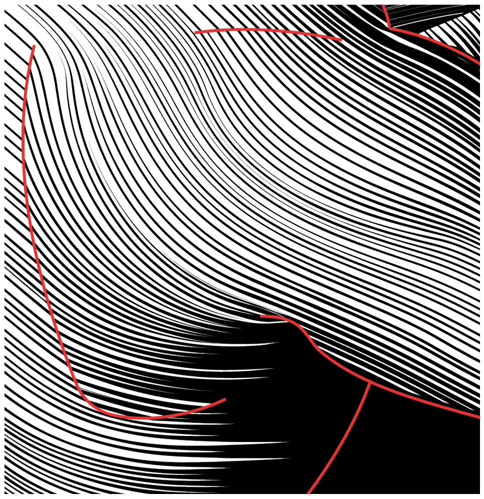


Fig. 3. This detail from Figure 1 shows how useful barrier curves are for portraying discontinuities in (imagined) three-dimensional surfaces.

Our primary contribution is the hatching shape primitive itself, which has the same type of research value as the diffusion curve primitive [Orzan et al. 2008] or the abstract computational-artistic technique of Modeling with Rendering Primitives [Schwarz et al. 2007]. Secondary contributions are (1) a suggested set of hatching shape operations with implementation details and (2) modifications to streamline advection for producing hatching marks from a hatching shape. These modifications primarily deal with discontinuities at barrier curves.

We have built a prototype interactive tool in C++ that takes input from a drawing tablet and uses OpenGL to render a virtual canvas. It has many of the features of an ordinary drawing tool along with the ability to create and edit hatching shapes.

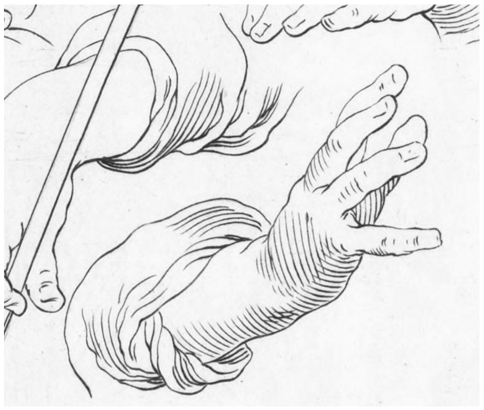


Fig. 4. The artist uses multiple hatching patches to render the single complex form of the upraised hand. Notice the seam between the patch on the back of the hand and patch running along the heel of the hand and the arm. Detail from *Ars Pictoria* [Browne 1669], copper plate.

The rest of this paper is organized as follows: Section 2 mentions prior literature. Section 3 explains the parts of a hatching shape. Sections 4 and 5 treat hatching shape operations: Section 4 covers basic operations and Section 5 explains more advanced ones that involve barrier curves. Section 6 gives our streamline advection details. Section 7 discusses the merits and limitations of hatching shapes. Section 8 considers future developments.

2 RELATED WORK

As a research problem, hatching usually falls in the purview of 3D rendering—the electronic kind. Lawonn et al. [2018] survey hatching applications in this context. Some 3D work outside of rendering, such as surface parametrization [Knöppel et al. 2015; Lichtenberg et al. 2018] essentially performs hatching without using the word. Our work has little to do with 3D research, though we should note that some 3D hatching renderers use streamline advection [Hertzmann and Zorin 2000; Lawonn et al. 2013; Singh and Schaefer 2010; Zander et al. 2004].

The immediate family of our work is hand-driven 2D hatching interfaces [Durand et al. 2001; Ostromoukhov 1999; Pnueli and Bruckstein 2005; Salisbury et al. 1996, 1994, 1997]. Our work is distinct within this family. One difference is superficial but worth noting for clarity: hatching shapes are meant mainly for drawing from imagination, not for creating hatching from a reference image. (We do describe one hatching shape operation that involves a reference image; see Section 4.4.)

Less superficially, hatching shapes give greater control over marks' widths, directions, and spacing than do previous alternatives. Control over the direction field is the most important, and therefore the best way to compare our contributions to others'.

One of the first hatching interfaces is DigiDürer [Pnueli and Bruckstein 2005], in which marks' directions come from a reference image. The same applies in Interactive Pen-and-Ink Illustration [Salisbury et al. 1994], though there the user additionally provides an offset angle. Orientable Textures [Salisbury et al. 1997] gives more direct control over marks' directions. As with our work, there

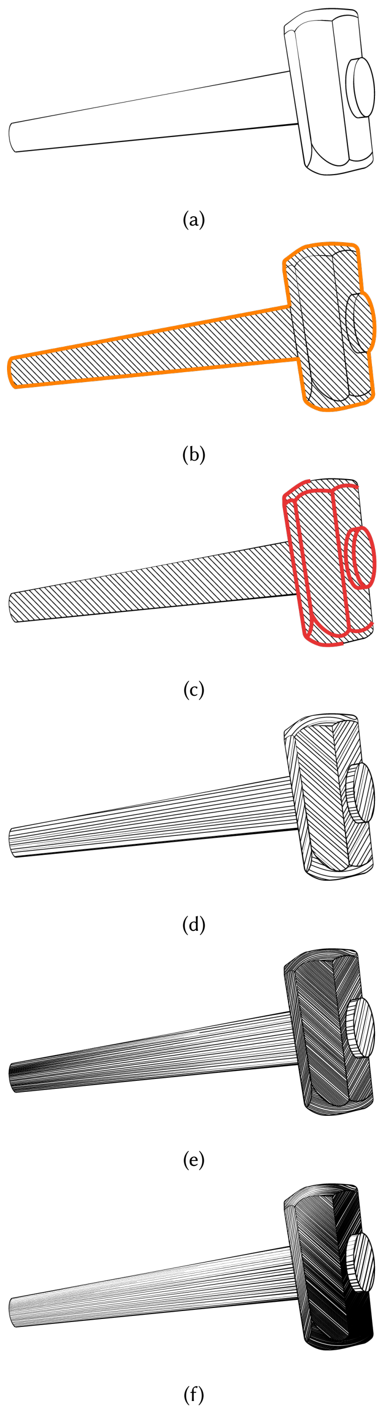


Fig. 5. Example process for an illustration that includes a hatching shape: Draw strokes in the canvas (a). Create a hatching shape (b), possibly using some of those strokes for its mask (Section 4.1). Specify barrier curves (c), again possibly snapping to preexisting strokes. Set the shape's direction field (d) using the multi-dir operation (Section 5). Tweak the spacing field (e) and the width field (f) using fill and blend-stroke operations (Sections 4.3 and 5).

is a rasterized direction field, editable via brush tools. The most interesting operation has the user draw two curves and fills the direction field between them by sampling from interpolating curves. A similar mechanism seems to feature in Strokes Maker [Apanovich 2021], a commercial application for creating hatching from a photograph. Adobe Illustrator, augmented with plugins like WidthScribe by Astute Graphics [2021], offers controls similar to Strokes Maker's.

The idea of interpolating curves crystallizes further in Digital Facial Engraving [Ostromoukhov 1999], where hatching marks' paths are isocurves in Coons patches. Digital Facial Engraving is the closest in the hand-driven hatching family to our work, since it focuses on hatching smooth, difficult 3D forms, though it does not give control over marks' directions beyond editing the boundaries of Coons patches.

An interesting approach not attempted in our work is to synthesize hatching from examples. This tack appears in the 2D domain [Barla et al. 2006; Jodoin et al. 2002; Xing et al. 2014] and in the 3D domain [Gerl and Isenberg 2013; Kalogerakis et al. 2012].

3 PARTS OF A HATCHING SHAPE

A hatching shape stores information for producing hatching within a bounding box: three fields, a mask, and barrier curves.

The three fields—width, spacing, and direction—are 2D arrays with a shared resolution. At each grid location, the hatching shape defines (1) a scalar *width*, the width of a hatching mark passing through that location; (2) a scalar *spacing*, the distance between the spines of two hatching marks; and (3) a *direction*, stored in our case as the three unique components of a structure tensor. We use structure tensors rather than 2D vectors or polar values since our directions need to be 180°-symmetric. Our tensors have the form provided by Knutsson [1989]: given a vector $\mathbf{v} = (x, y)$, the structure tensor is

$$\begin{bmatrix} x^2 & xy \\ xy & y^2 \end{bmatrix} \tag{1}$$

Structure tensors are a useful representation because the vectors $\mathbf{v}$ and $-\mathbf{v}$ come out identical. This matters not so much when storing a hatching shape in memory, but is crucial when editing its direction field. We do ultimately convert tensors back to 2D vectors when sampling the direction field during streamline advection.

To sample the width, spacing, or direction at a position inside the hatching shape's bounding box, we interpolate. When no barrier curves are nearby, this means standard bilinear interpolation; otherwise, it is more complicated—see Section 6.

The resolution of a just-created hatching shape's fields depends on the shape's relative size in the canvas. By default, a shape that completely fills a square canvas will have a 50×50 grid. The artist can change the resolution if desired. The torso hatching shape in Figure 1 has a 200×243 grid.

In Sections 4 and 5, we discuss operations that modify a hatching shape's three fields. Often, the goal of these operations is to interpolate a smooth field from a sparse set of constraints. In such cases we solve grid-Laplacians [Chaudhuri 2017]. A grid-Laplacian is a linear system that interpolates a scalar field f on a grid. The field has known values—Dirichlet constraints—at some grid locations

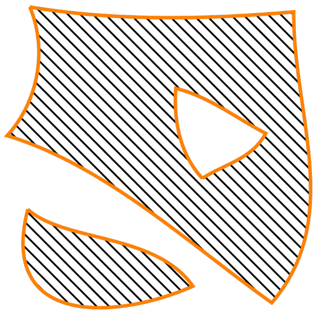


Fig. 6. A hatching shape's mask is a collection of closed curves—three in this case.

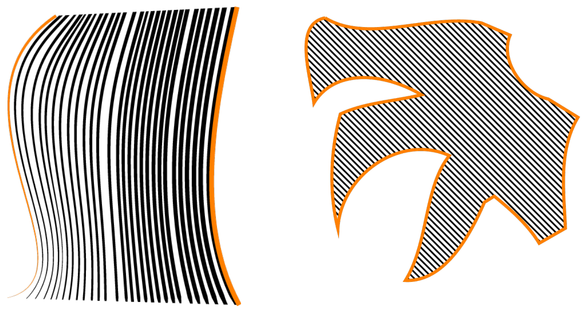


Fig. 7. One way to create a hatching shape (left) is to draw two strokes defining opposite sides of a Coons patch. Another way (right) is to draw a closed curve, which becomes a new hatching shape's mask.

and Δf should be zero (f should be smooth) elsewhere. For each unknown $f_{i,j}$, the system contains the equation

$$-4f_{i,j} + f_{i-1,j} + f_{i+1,j} + f_{i,j-1} + f_{i,j+1} = 0 \tag{2}$$

When interpolating a smooth direction field, we solve separate grid-Laplacians for the three unique components of a structure tensor. Our grid-Laplacians all take the four-neighbor form shown in Equation 2, though an eight-neighbor form exists. The importance of this restriction becomes clear in Section 5, when barrier curves get involved. We solve grid-Laplacians with the sparse simplicial LDLT solver from the Eigen library [Guennebaud et al. 2010].

Since an artist might want to confine hatching marks to a shape other than a rectangle, a hatching shape also has a mask, which is a group of closed curves that do not intersect but may contain each other (Figure 6).

We perform the masking itself in two ways. In interactive OpenGL rendering, we fill a stencil buffer by rendering each polyline-approximated mask curve using a triangle fan, incrementing the stencil value at drawn locations [Woo et al. 1997]. This produces stencil values that are odd (with the least significant bit set to one) inside the mask, where hatching marks should be visible. Later, when we export the artist's work in PostScript form, we use the mask curves as clipping paths.

The last part of a hatching shape is zero or more barrier curves (Figure 2). These must extend inside the bounding box to have effect.

4 CREATING AND EDITING HATCHING SHAPES

With a hatching shape's parts defined, we now present operations on this primitive. In this section, we cover simpler operations that do not involve barrier curves (but do not rule out their presence). These operations are hatching shape creation (Section 4.1), mask editing (Section 4.2), field fill operations (Section 4.3), sourcing a hatching shape's width field from a reference image (Section 4.4), field multiplication (Section 4.5), and crosshatching (Section 4.6).

Section 5 discusses more complex operations that respect barrier curves.

4.1 Creating a Hatching Shape

One way to create a hatching shape is to draw two source strokes, as in the left half of Figure 7. The strokes' combined bounding box becomes the shape's bounding box. The spacing field starts out with a user-chosen uniform value; the irregular spacings in the figure are an artifact of our streamline placement algorithm (Section 8.2). The width and direction fields come from solving grid-Laplacians, where the Dirichlet constraints are widths and directions sampled from the strokes.

Another way to create a hatching shape is to draw a closed curve, possibly snapping to strokes in the canvas as in Figure 5. The bounding box of the curve becomes the bounding box of the hatching shape. The three fields start out with uniform values: a diagonal direction for the direction field and user-chosen values for the width and spacing fields. The initiating curve becomes the shape's single mask curve (Figure 7, right).

4.2 Mask Editing

The artist can draw a closed curve and remove its enclosed area from a hatching shape's visible area (Figure 8b). This affects only the hatching shape's mask. We use planar arrangements (provided by CGAL [The CGAL Project 2020]) to derive a new collection of mask curves. *Adding* area to a hatching shape (Figure 8c) is more involved, since it might require expanding the hatching shape's bounding box. In that event, we recompute all three fields, setting them to interpolated original values in the space covered by the original box and to the solution of a grid-Laplacian (with Dirichlet constraints along the border of the old box) elsewhere.

4.3 Field Fill Operations

We offer several means of editing a hatching shape's fields, including three kinds of fill operations. A simple fill takes a fill value and an alpha in $[0, 1]$. A linear gradient or radial gradient fill operation takes start and end positions, two fill values, and two alphas. All fill operations reduce to updating each of a field's values to an alpha-weighted mixture of its old value and a fill value. No grid-Laplacian is involved. Figure 9 shows linear and radial gradient fill operations applied to the width and spacing fields of a hatching shape. Figure 10

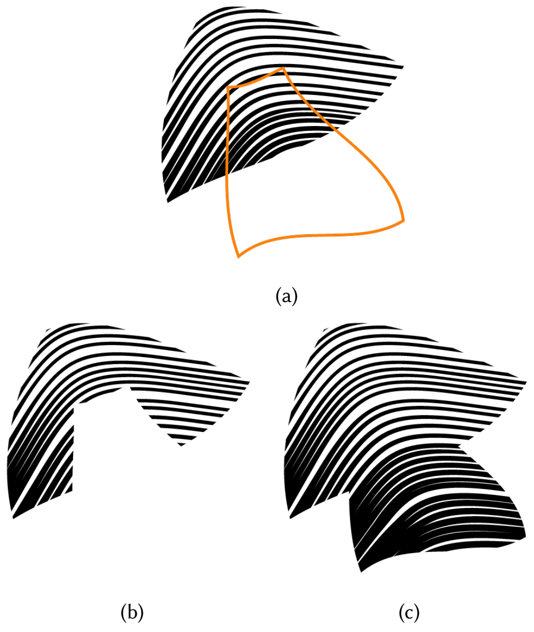


Fig. 8. Removing the orange shape from the hatching shape's mask requires only coming up with a new mask curve using planar arrangements (b). Adding the orange shape to the mask is more complicated, since it expands the hatching shape's bounding box, requiring new versions of the width, spacing, and direction fields.

shows a linear gradient fill on a hatching shape's spacing field in the context of an illustration.

The user explicitly chooses alpha values and fill values, except for direction-field fill values, which are inferred from the input stroke. The simple fill uses the direction from one end of the stroke to the other. The linear gradient fill interpolates between the stroke's endpoint tangents. In the radial gradient fill, fill values point either toward the stroke's start position (source/sink mode) or perpendicular to that direction (vortex mode). Figures 11 and 12 show fill operations on the direction field.

4.4 Setting the Width Field From an Image

The artist can set a hatching shape's width field using a maximum width value and a greyscale reference image. This option is a fallback for modifying the width field manually is too tedious. Figure 13 presents such a case. In a widths-from-image operation, the hatching shape's resolution changes: its cell width becomes twice the size of a pixel from the reference image, assuming the reference image is mapped to the bounding box of the canvas.

4.5 Field Multiplication

The artist can multiply either the width or the spacing field by some factor e. Or they can perform a density-change operation by specifying a density factor e such that both the width and spacing fields get multiplied by $1/e$. Though trivial, these operations make clear the value of procedural hatching. Consider how long it would

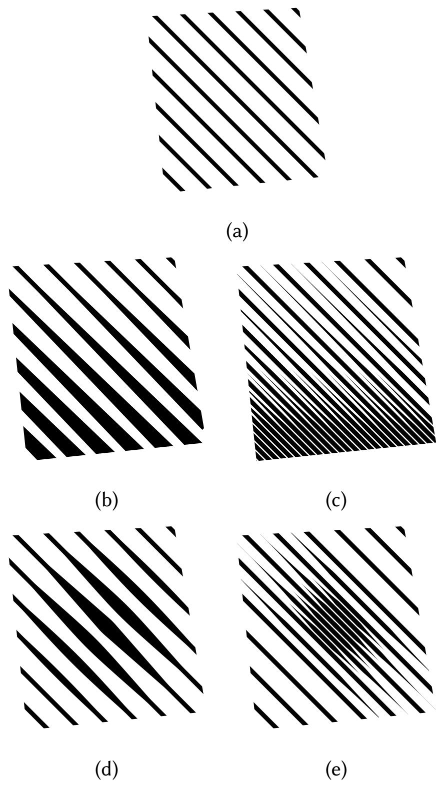


Fig. 9. An initial hatching shape (a) modified with linear (b,c) and radial (d,e) gradient fill operations on the width and spacing fields.

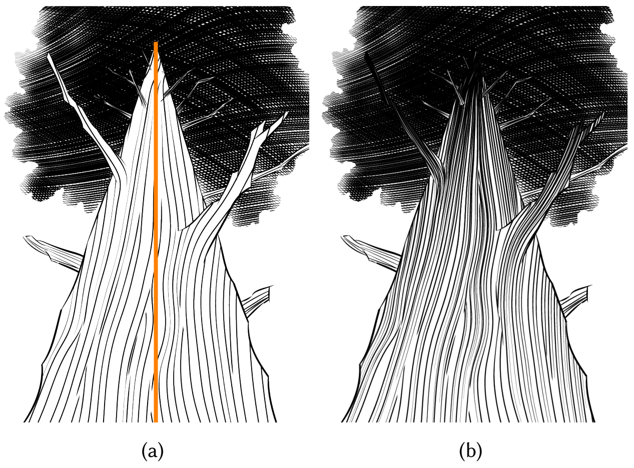


Fig. 10. Initially, the hatching shape representing the tree has a uniform spacing field (a). A linear gradient fill operation guided by the orange stroke applies a smaller spacing value with an alpha that decreases from top to bottom. This produces foreshortening (b).

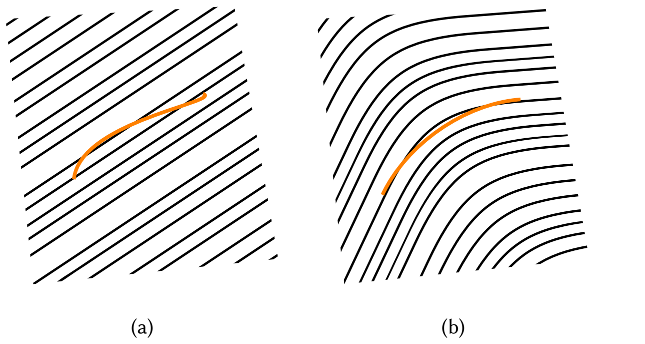


Fig. 11. Simple fill and linear gradient fill operations on the direction field. The simple fill operation uses a stroke's endpoint-to-endpoint direction (a). The linear gradient fill operation uses a stroke's endpoint tangents (b).

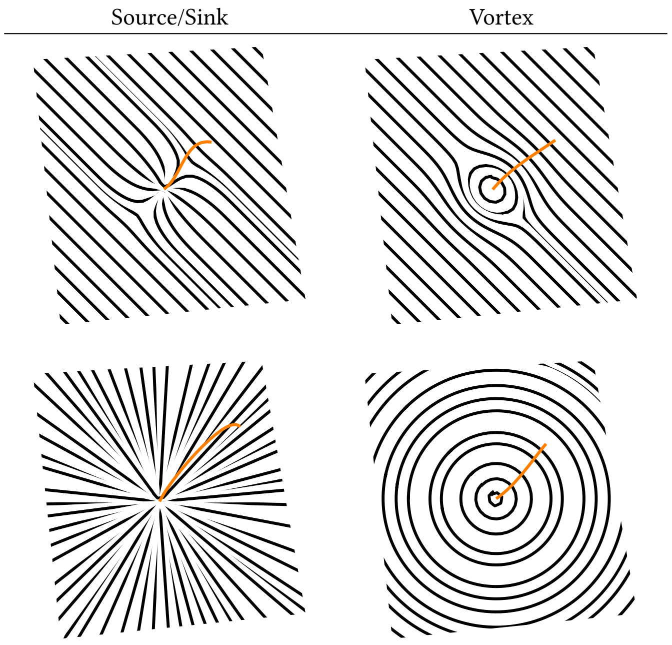


Fig. 12. Radial gradient fills on the direction field, shown with the precipitating strokes. The first column shows source/sink mode; the second shows vortex mode. In all four cases, the first alpha value, associated with the beginning of the stroke (the center of the field), is 1. The second alpha value is 0 in the first row and 1 in the second row.

take to effect the edit shown in Figure 14 drawing each mark by hand.

4.6 Crosshatching

Our crosshatching operation, demonstrated in Figure 15 and also discernible in the foliage of Figure 10, takes an angular range α between 0 and 90°, plus a number n of crosshatching layers to create. We make that many copies of the selected hatching shape. Each

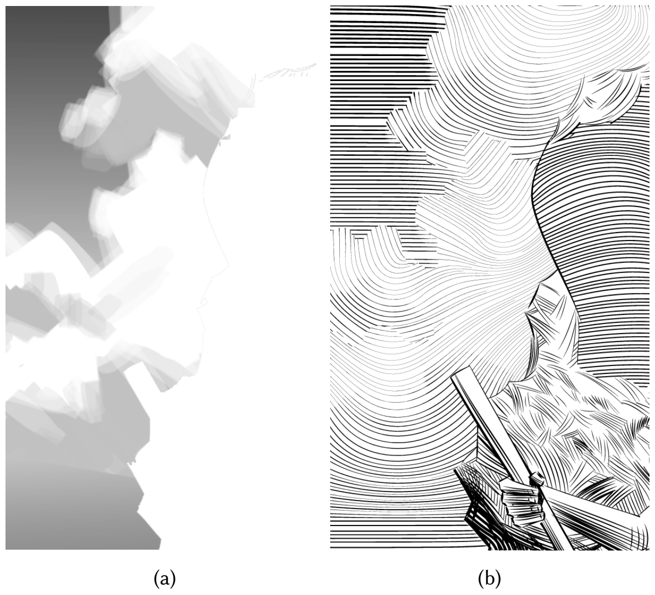


Fig. 13. A quick Photoshop painting (a) provides the width field for the hatching shape depicting clouds and sky in a detail from a larger composition (b).

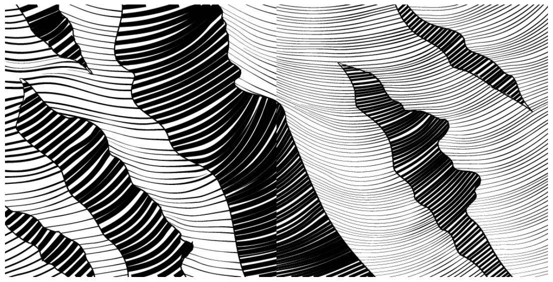


Fig. 14. The artist can modify the density of a hatching shape, doubling it here.

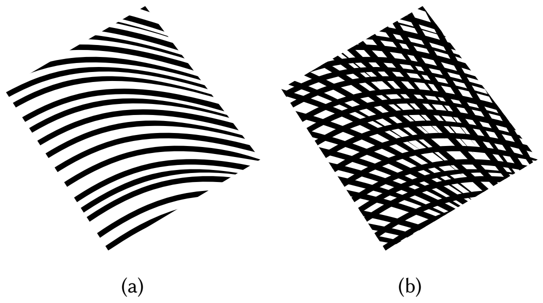


Fig. 15. The crosshatching operation copies a hatching shape and rotates the direction field in the copies. Here, the crosshatching angle α is 45° and the number of copies n is 1.

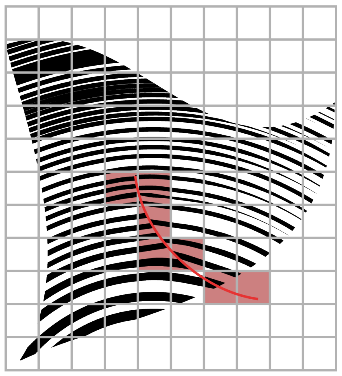


Fig. 16. Barrier curves create “dead cells” (red) in a hatching shape’s grid. Widths, spacings, and directions from dead cells are unreliable for use in streamline advection.

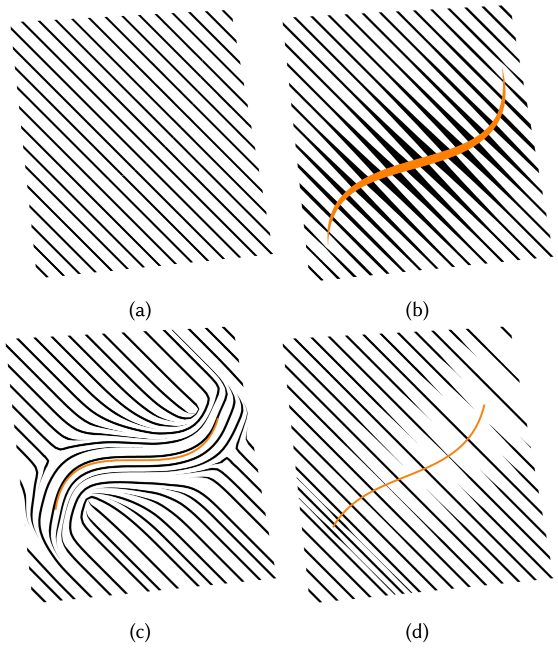


Fig. 17. Blend strokes applied to the width (b), direction (c), and spacing (d) fields of an initial hatching shape (a).

copy has its direction field (not its marks) rotated clockwise by $i\alpha/n$, where $i = 1 \ldots n$.

5 BARRIER CURVES AND OPERATIONS INVOLVING THEM

Having explained the simpler hatching shape operations, we now discuss two more complex ones: *blend-stroke* and *multi-dir*. These operations respect a hatching shape’s barrier curves, if any. Remember that barrier curves are for creating discontinuities.

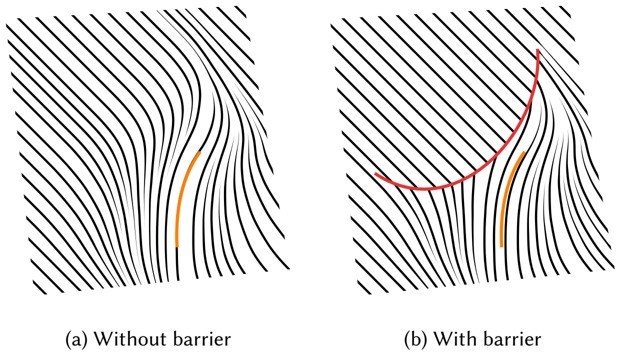


Fig. 18. Blend strokes’ effects stop at barriers.

Before describing these two operations, we must introduce “dead cells” (Figure 16). We find these by rasterizing barrier curves to a hatching shape’s grid. Dead cells let barrier curves have influence in this section’s two operations. They also influence streamline advection (Section 6).

5.1 Blend Stroke

A blend-stroke operation lets the user locally and smoothly modify one of a hatching shape’s fields with a stroke. Effects propagate from the stroke up to a user-specified radius (Figure 17). Achieving this means updating each of the field’s values using a fill value and an alpha value, like the fill operations described earlier. Here, however, fill values and alpha values are more complex to calculate.

There are three steps to a blend-stroke operation. First, we compute a distance transform of the stroke at the same resolution as the hatching shape. When there are no barrier curves, we use a simple two-pass distance transform [Toivanen 1996]. Otherwise we use Dijkstra’s algorithm modified to ignore dead cells. This means that for every cell position, we find the length of the shortest path to the stroke, where paths are a mixture of horizontal and vertical steps that do not visit dead cells.

Second, we compute an alternate version of the field using a grid-Laplacian with Dirichlet constraints along the stroke. When editing the direction field or width field, we densely sample the stroke’s tangents or widths for our Dirichlet values. In a spacing field edit, we interpolate between two user-specified spacing values along the stroke. Third, we blend this alternate version of the field with the current field. Each cell’s alpha value comes from the distance map, the effect radius, and an overall alpha value chosen by the user.

Because our Dijkstra distance transform uses four-neighbor paths only and considers dead cells non-traversable, effects do not propagate across barriers (Figure 18). Dead cells are unchanged by blend-stroke operations, which means their values are liable to stop agreeing with their neighbors. To be safe, then, we do not use dead cells’ values when interpolating data from the three fields (Section 6.3).

5.2 Multi-Dir

The multi-dir operation is an expansion of the blend-stroke operation, meant just for the direction field. Bending hatching marks to show the curvature of an imagined 3D surface is a crucial part of

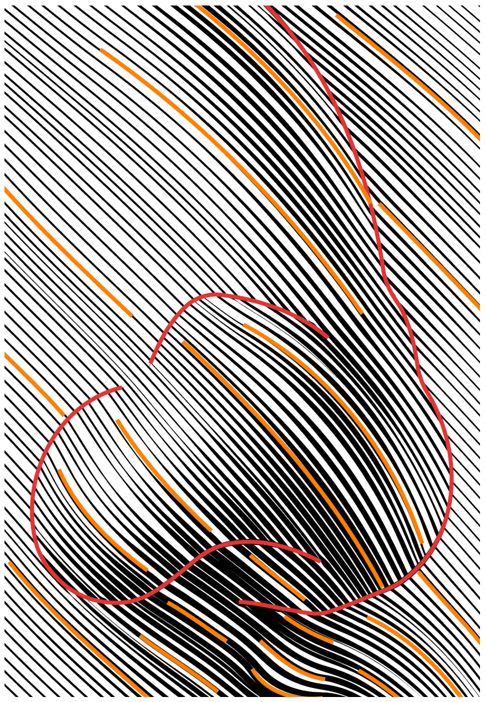


Fig. 19. The multi-dir operation fills the direction field using the barrier curves (red) and multiple guiding strokes (orange).

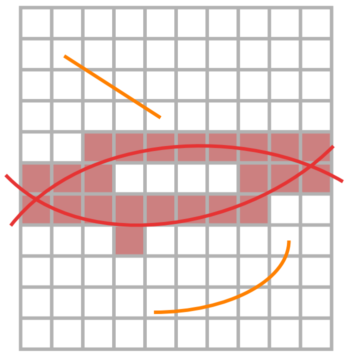


Fig. 20. In the multi-dir operation, the grid-Laplacian leaves out dead cells. There might be pockets of live cells that are unreachable from Dirichlet constraints (orange). The system leaves these out as well to prevent the solver from failing.

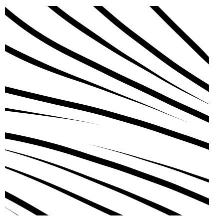


Fig. 21. Streamlines taper as they approach each other.

hatching, and we find it too difficult using individual blend strokes. In multi-dir, we do away with the effect radius and do not alpha blend with the old direction field. Instead we produce a completely new direction field that interpolates the densely sampled directions of one or more guiding strokes (Figure 19). The input in multi-dir evokes the "directional guides" of FLOWPAK [Saputra et al. 2017], the "curvature lines" of BendFields [Iarussi et al. 2015], the "cross sections" of CrossShade [Shao et al. 2012], and the "hatching strokes" of Bui et al. [2015].

Multi-dir requires another grid-Laplacian, with Dirichlet values set to guiding strokes' tangents. There is a critical difference between this grid-Laplacian and those discussed earlier. We exclude dead cells from this linear system, which lets barrier curves exert an effect: cells on opposite sides of a barrier are no longer as closely connected, if at all. A similar technique creates discontinuities in the context of diffusion curves [Bezerra et al. 2010; Finch et al. 2011].

To clarify, we do *not* exclude dead cells from the grid-Laplacian used for a blend stroke. Blend strokes do respect barrier curves, but by way of the distance map used for blending. With a blend stroke, the grid-Laplacian only needs to produce a smooth field that interpolates sample values from an input stroke. We could use the more advanced, barrier-respecting grid-Laplacian for this interpolation, but we have not yet seen the need.

In the dead cell-excluding grid-Laplacian, the four-neighbor versus eight-neighbor distinction is just as critical as it is in the Dijkstra distance transform used in the blend-stroke operation. If we used eight-neighbor paths in our distance transforms or an eight-neighbor grid-Laplacian for multi-dir, the rasterized barriers would not be effective unless we used a "thick" rasterization algorithm that marked *every* cell touched by a barrier curve as dead. We use "thin" rasterization, keeping as many cells as possible live and thereby throwing away no more data than necessary. Section 6.4 explains our rasterization further.

Removing dead cells from the grid-Laplacian sometimes means that a small pocket of cells is unreachable from any of the Dirichlet constraints (Figure 20). These pockets can make the linear solver fail. Thus, the system includes only live cells reachable from Dirichlet constraints via four-neighbor paths that do not include dead cells.

6 STREAMLINE ADVECTION

We have explained a hatching shape's parts and showed ways to manipulate this primitive. Now we explain how to produce hatching marks from a hatching shape using our version of Jobard and Lefer's streamline placement algorithm [1997].

The original placement algorithm fills a rectangle with streamlines one at a time, popping seed locations from a queue. If a candidate location is too close to already placed streamlines, discard it; otherwise, integrate through a vector field (a hatching shape's direction field, in our case) and terminate the streamline upon going out of bounds or coming within a stopping distance of an older streamline.

There are many low-level implementation choices, of course. Here are the the simpler details of our approach:

- Streamline density is variable [Chen et al. 2007; Li et al. 2008; Mebarki 2016; Schlemmer et al. 2007; Singh and Schaefer

2010]. The separation distance between streamlines, called d_{sep} by Jobard and Lefer, comes from a hatching shape's spacing field.
- Like Jobard and Lefer, we use midpoint integration.
- For a given hatching shape, the integration step size is a constant: the smallest d_{sep} found in the spacing field.
- Streamlines taper when they come close together, again as per Jobard and Lefer. The local value d_{test} is the minimum spacing allowed between two streamlines: the "stopping distance" mentioned above. A streamline in progress will taper to nothing and terminate if its distance to a previous streamline falls to this value. We set d_{test} to half of the local spacing value ($d_{\text{sep}}/2$). See Figure 21.
- A streamline seed must be at least d_{sep} (not d_{test}) away from previous streamlines in order not to be discarded.
- Before placing any hatching marks, we fill our seed queue, borrowing the approach of Suggestive Hatching [Singh and Schaefer 2010]. We take seeds from the vertices of a coarse-to-fine series of grids, the finest grid having a cell size equal to the step size. As Suggestive Hatching explains, this is a cheap way to get the benefits of farthest-point seeding [Mebarki et al. 2005]. A second seed queue handles gaps caused by barriers (Section 6.5).
- We use a minimum length test: if, after advection in both directions, a streamline contains fewer than 5 points, we discard it.

There are several advanced streamline placement features that our implementation lacks, such as the adaptive step size and loop prevention of ADVESS [Liu et al. 2006] and the critical-point handling of Verma et al. [2000].

With these lighter points covered, we move to the more involved parts of our streamline placement algorithm. Most of the complexity comes from the need to respect barrier curves.

6.1 The Dead Grid and the Collider Grid

We use auxiliary 2D arrays for streamline placement, such as the *dead grid*, a boolean image to which we rasterize barrier curves. We have already alluded to checking the dead-or-alive status of cells; that is the dead grid's role. The *collider grid* cheaply determines whether a given line segment intersects any barrier curve. Each cell in the collider grid stores line segments to use in intersection tests. Both the dead grid and the collider grid have the same resolution as a hatching shape's three fields.

To fill these two grids, we approximate each barrier curve with a series of line segments. For each line segment $\overline{AB}$, we find the integral coordinates it rasterizes to (Section 6.4) and mark these in the dead grid. We also add $\overline{AB}$ to the bin at each of the same positions in the collider grid. Finally, we dilate the information in the collider grid with a 3×3 kernel (Figure 22). Dilating the collider grid is necessary because of the thin rasterization we use. Without it, we would miss diagonal intersections.

We can find the first intersection of any directed line segment $\overline{CD}$ with any barrier curve by finding the rasterized coordinates of $\overline{CD}$ and, for each, checking the intersection between $\overline{CD}$ and the line

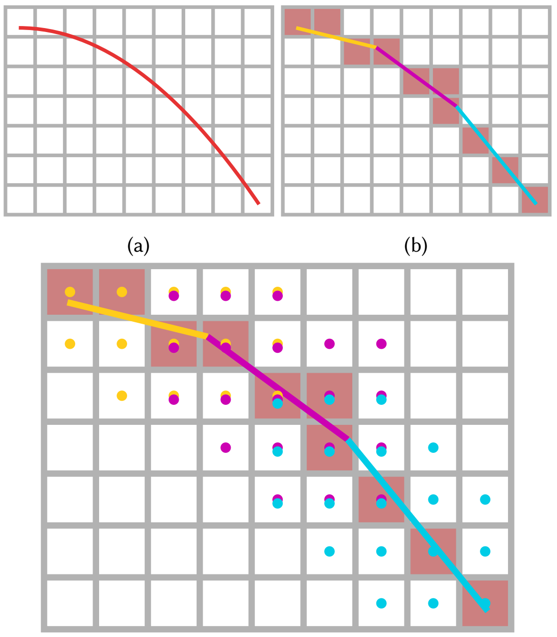


Fig. 22. Given a barrier curve (a), we split it into line segments. The boolean *dead grid* marks cells visited by any barrier line segment (b). The *collider grid* (c) stores copies of barrier line segments. For example, a cell with a yellow and a purple dot contains a copy of the yellow line segment and a copy of the purple line segment.

segments stored in the collider grid there, returning the intersection closest to C, if any.

6.2 Occupancy Grid

The *occupancy grid* is a third auxiliary array, meant for finding the distance to the nearest streamline at a query position. Ours resembles the occupancy grid of ADVESS: each cell stores points visited by completed streamlines (not the points from the streamline in progress). The choice of cell size for this grid can be made without regard to the cell sizes of other grids, permitting a trade-off between more memory usage and more distance calculations. We choose as our occupancy grid cell size the minimum value found in a hatching shape's spacing field, clamped so that the occupancy grid's minimum dimension is no smaller than 15 and its maximum dimension is no larger than 500.

There are two use cases for the occupancy grid: checking a streamline in progress and checking a streamline seed.

6.2.1 Streamline in Progress. Every time a streamline takes a step, we need to check how close it is to previous streamlines. If the nearest streamline is d_{test} away—recall that d_{test} is half of d_{sep}—then the current streamline tapers to nothing and takes no more steps in the current direction.

The occupancy grid does not need to return the true nearest distance to an already placed streamline at the current location; it just needs to return the true distance clamped to the range $[d_{\text{test}}, d_{\text{sep}}]$.

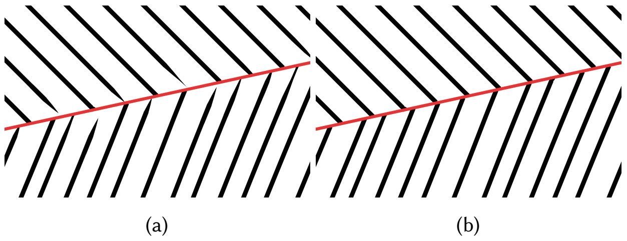


Fig. 23. Failing to account for barriers in occupancy-grid tests causes unnecessary tapering (a). When checking the distance to previously placed hatching marks from a query position, we ignore marks on the other side of barriers (b).

This observation yields the general routine DistToNearestStreamline, which takes a query position p along with lower and upper bounds on the distance to return: $d_{\min}$ and $d_{\max}$. First, we look at the occupancy grid cell containing p and check if it has any stored points, then the 3×3 ring of cells around that cell, then the 5×5 ring of cells around that cell. We proceed until we complete the $k \times k$ ring, where $k = \min(2l_{\max} + 1, 2\lceil d_{\max}/w_{\text{occ}} \rceil + 1)$, $l_{\max}$ is the maximum dimension of the occupancy grid, and w_{occ} is the width of one of the occupancy grid's cells. Any points stored outside of the $k \times k$ ring will be at least $d_{\max}$ away from p, so there is no need to check any more cells. If we find a stored point less than or equal to $d_{\min}$ away from p, we immediately return $d_{\min}$. Otherwise we return the minimum distance found, or $d_{\max}$ if we encounter no stored positions.

Barrier curves add a wrinkle. The occupancy information from one side of a barrier should not affect the other side, or streamlines will taper unnecessarily as they approach a barrier and detect previously placed streamlines on the other side (Figure 23). We use the collider grid to test whether the line segment from the query position to each stored position intersects any barrier curve, and ignore stored positions blocked by barriers.

If the next integration step for a streamline has it run into a barrier, as indicated by the collider grid, then the streamline terminates 5% short of reaching the intersection. The padding ensures that the last stored position for the streamline is on the correct side of the barrier. Without it, subsequent streamlines on the other side might spuriously detect the just completed streamline's influence in the occupancy grid.

6.2.2 Streamline Seed. The second use case for the occupancy grid is when creating a new streamline, given a seed position. If the seed is less than d_{sep} from an existing streamline, then we discard it. Here we use our occupancy grid routine atLeastThisFarAway, which takes a query position p and a minimum distance $d_{\min}$, returning a boolean. This routine is simply the boolean expression DistToNearestStreamline$(p, (1-\epsilon)d_{\min}, d_{\min}) \geq d_{\min}$. The constant $\epsilon = 0.001$ is necessary to keep the expression from always evaluating to true. This constant actually features twice: when we call AtLeastThisFarAway, we set $d_{\min}$ to $(1-\epsilon)d_{\text{sep}}$ instead of just d_{sep}, as floating point error can otherwise lead perfectly placed seeds to be discarded for being detected as just barely too close to existing streamlines.

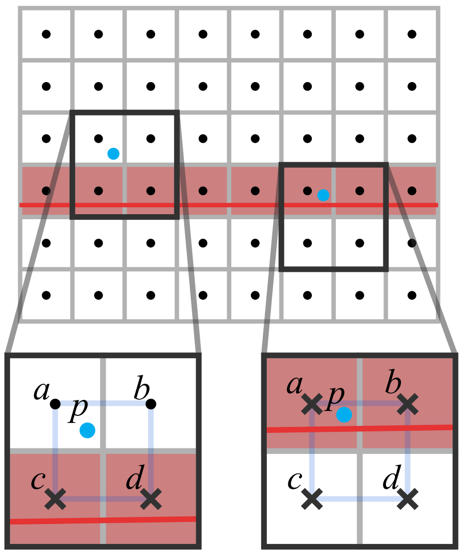


Fig. 24. In the left case, we exclude cells c and d from interpolation because they are dead. In the right case, we exclude a and b for the same reason. There we also exclude c and d since their centers are separated from p by a segment of a barrier curve.

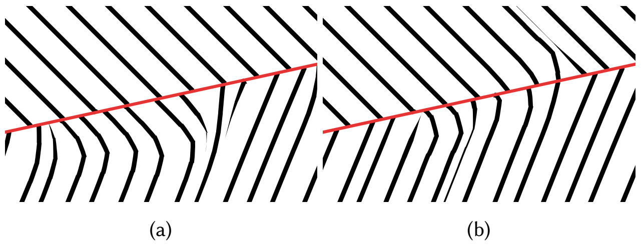


Fig. 25. In the left case, we allow dead cells to participate in interpolation. Some of the streamlines at bottom veer undesirably as they enter the dead cells' zone of influence and start incorporating out-of-date direction values. In the right case, we allow cells on the wrong side of barrier curves to participate. Some of the streamlines below the barrier receive influence from direction values above the barrier, and one streamline from above the barrier receives influence from below it.

6.3 Interpolating Field Data

The width, direction, and spacing values for a query position inside a hatching shape come from interpolating the values surrounding that position. Barrier curves pose a couple of problems: we must avoid sampling dead cells' values or sampling values from the wrong side of a barrier.

By default, we use bilinear interpolation between four cells' values. To prevent interpolation problems caused by barriers, we may disqualify some of these four cells. Two tests can disqualify a cell. First, we ignore dead cells, since their values are unreliable, as explained in Section 5. Second, we ignore cells whose centers are blocked from the query point by barriers. Figure 24 shows both of these tests. When using the collider grid to answer the necessary intersection queries, we err on the false-positive side. Remember that the collider grid returns the first collision between a directed line segment $\overline{CD}$ and barriers. In this case we start with C equal to the query point and D equal to the center of the cell of interest. To avoid missing any intersections, we extend the length of $\overline{CD}$ by a factor of 1.001 in both directions before checking it against the collider grid.

Figure 25a shows the consequence of leaving out the first test and letting dead cells participate in the interpolation of streamline directions: old data pollute the pattern. Figure 25b shows the result of leaving out the second test: "leaks" result when streamlines approach a barrier and start sampling data from the other side of it.

If three of the four cells are considered valid, we use distance-weighted averaging to interpolate their values. In two-cell cases where the two passing cells have the same X or Y coordinate, we linearly interpolate using the query position's X or Y coordinate. In diagonal two-cell cases, we use distance-weighted averaging.

6.4 Rasterization Issues

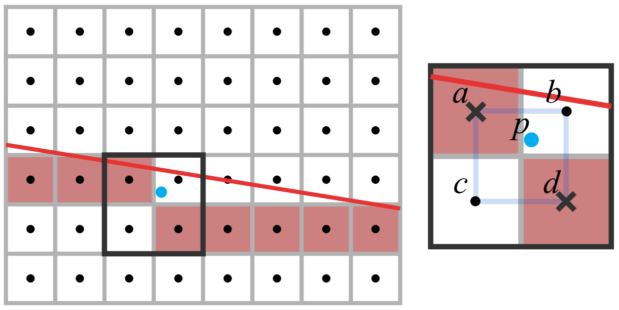
We must take care when choosing how to rasterize barriers to the dead grid. A subtle problem arises if we use integer-based algorithms. Situations arise like that shown in Figure 26. Here, a barrier segment is so far from its rasterized cells that there are areas of live space below the true barrier segment yet above the rasterized version. In this situation, the collider test for excluding interpolation values is unreliable, resulting in leaks. We use a floating-point DDA algorithm [Watt 2000] that steps a distance of one cell width until capping off with a step less than or equal to one cell width. This algorithm keeps barrier segments closer to their rasterized versions.

6.5 When Interpolation Fails

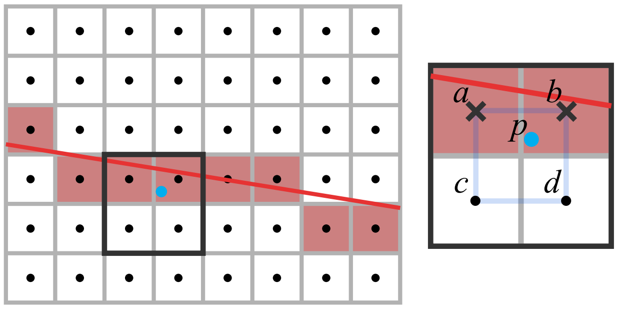
Sometimes a query point is surrounded by four disqualified cells, meaning that width, spacing, and direction are indeterminate there. If a streamline enters indeterminate space, it keeps advecting using its last known data until it terminates or reenters determinate space.

When we pop a seed from the primary queue and find that it falls in indeterminate space, we discard it. This can cause gaps in hatching. To fill these, we introduce a secondary seed queue, which we do not use until the primary queue is empty. While advecting a streamline, at each step we look d_{sep} to the left and right, creating two candidate positions p_l and p_r. If a position falls within a dead cell and there is no barrier curve between p and the streamline's current position, we tentatively create a secondary seed at p. (We do not enqueue the secondary seeds produced by a streamline until that streamline finishes and passes our minimum length test. Otherwise an infinite loop can result from zero-length streamlines reseeding each other.) Secondary seeds, unlike primary seeds, store initial spacing, width, and direction data, sourced from parent streamlines. When we pop a seed from the secondary queue, the resulting streamline starts out using these stored data. Figure 27 shows how secondary seeds fill some gaps, but not areas of indeterminate space completely walled off by barriers.

Hatching inevitably degrades near barrier curves. This degradation owes not just to indeterminate space, but more generally to the loss of interpolation data as query points approach barriers. Reducing artifacts caused by barrier curves is one of the main reasons to increase the resolution of a hatching shape (Figure 28).

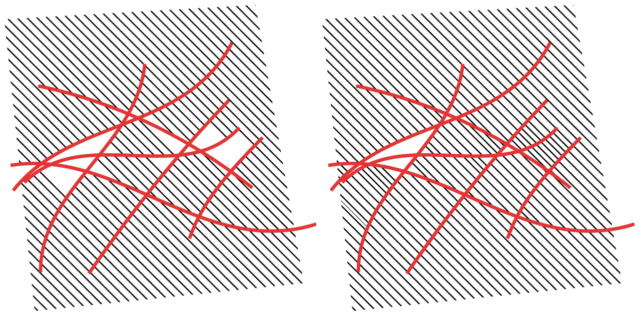


(a) Integer-based midpoint [Pradhan 2020]



(b) Floating point DDA [Watt 2000]

Fig. 26. If we mark dead cells using integer-based rasterization, situations arise where we cannot rely on our tests to prevent the wrong cells from participating in interpolation. The first scenario is an example (a). Cells b and c qualify to participate. Neither is dead and neither is separated from the query point by the barrier. But these cells are on opposite sides of the *rasterized* barrier, so using them both at once means leaking data across. Our floating-point rasterization algorithm (b) avoids situations like this by keeping the rasterized barrier curve closer to the actual barrier curve.



(a) (b)

Fig. 27. Normal streamlines cannot sprout in indeterminate space, which causes gaps where multiple barrier curves come close together (a). Secondary seeds, which can sprout in indeterminate space, fill some of these gaps (b).

	Breeze	Twee	Houppelande
Hatching Shapes	8	4	2
Hatching Marks	3425	3669	1294
Hand-Drawn Marks	5462	3450	3883
Total Draw Time (s)	3.85893	1.77037	0.35684
Hatching-Only Draw Time (s)	3.8353	1.76357	0.328343

Table 1. Data for "Breeze" and "Twee" (Figure 29), and for "Houppelande" (Figure 32). Each time figure is an average of ten runs.

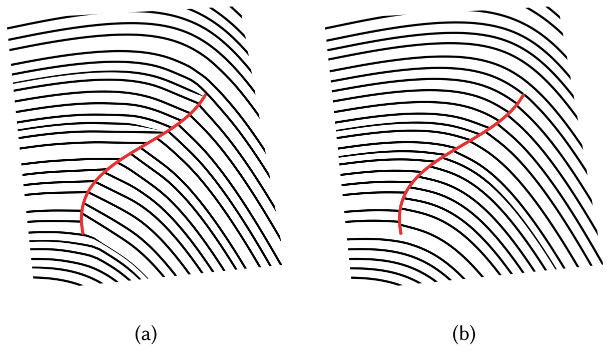


Fig. 28. The fields of the first hatching shape (a) are 8×9. There are artifacts caused by streamlines losing interpolation data as they approach the barrier curve. Increasing the resolution to 20×21 masks these artifacts.

7 RESULTS

We have explored hatching shapes' utility in full illustrations like "Breeze" and "Twee" (Figure 29). Not all marks in these pictures are hatching, and not all hatching comes from hatching shapes (Figure 30).

In our exploration we have found a couple of criteria for deciding if a hatching task would benefit from hatching shapes. First, the best subjects for hatching shapes are forms that read visually as smooth surfaces, like the cliffs in "Breeze" and the concrete walls in "Twee." Good subjects may have many discontinuities, as long as there are smooth areas visible between. Figure 31 shows an extreme example: the jagged interior of the egg.

Second, individual hatching marks should have low pictorial weight. In "Houppelande" (Figure 32), hatching shapes only feature in the dark drapery and the egg; other hatching is completely manual. The woman's face is not ideal for a hatching shape, despite being a smooth form, because individual marks have too much pictorial weight here: marks' widths and spacings are large relative to the size of crucial facial features, which makes it possible for the placement of individual marks to ruin the rendering. A hatching shape *could* produce the hatching here, but because individual marks are so visually powerful, this would require a lot of careful editing, enough to justify hatching manually instead. The same goes for the hatching on the turban (which needs to jibe with the hatching on the face), plus the hatching on the arms and hands.

Hatching produced by hatching shapes has a distinct style. It is often so regular that it draws the eye more than hatching normally does. This can be pleasing or painful to the eye, and may harm a composition. In "Breeze," the marks on the cliffs are primarily from hatching shapes, but there are additional hand-drawn marks to break up the uniformity and add naturalism. Future work with hatching shapes could impart such naturalism automatically by mimicking traditional hatching, whether at the level of individual marks [AlMeraj et al. 2009] or at the level of a group of hatching marks [Gerl and Isenberg 2013; Jodoin et al. 2002], i.e., at the level of a hatching shape.

Table 1 gives render times for "Breeze," "Twee," and "Houppelande." It lists the time required for our interface to completely redraw the canvas, plus the time required just to redraw all hatching shapes. Drawing times for hatching shapes include the streamline advection process.

7.1 Artist Interviews

We conducted semi-structured interviews with three professional artists to help assess hatching shapes' value. In each interview, we explained what hatching shapes are, showed some operations, and looked at example illustrations. We solicited reactions and asked about prior experience with hatching. Responses to hatching shapes were encouraging.

Artist 1 has over forty years of hatching-oriented professional experience. His preferred medium is scratchboard. He could be considered a master of hatching, experienced enough not to benefit much from hatching shapes, though he did muse that his lines might be more "consistent" if he had them at his disposal. When we showed him our illustrations, most of his criticisms were around draftsmanship and unrelated to our research. His reaction to the hatching shape primitive itself was encouraging: "You are on your way to creating extraordinary effects with your hatching."

Artist 2 has about 13 years of professional experience, primarily digital. His work has a clean, op art style. He often creates hatching with Adobe Illustrator "blends"—interpolations between two curves, like the Coons patch-based hatching of Digital Facial Engraving [Ostromoukhov 1999]. The affordance these blends give him is limited. He said that if he could use hatching shapes, he would feel less averse to doing intricate hatching, especially with organic subject matter.

Artist 3 had a similar reaction. He also has 13 years of professional art experience, also largely digital, though little if any involving hatching. He admires hatching, but feels intimidated by its difficulty, saying hatching can seem like a "magic trick" when done by experts. Early in the interview, as we showed operations on hatching shapes, he called what he saw "phenomenal," and later said he would do much more hatching if hatching shapes were on the table. However, he disliked the untapered ends of hatching marks that terminate

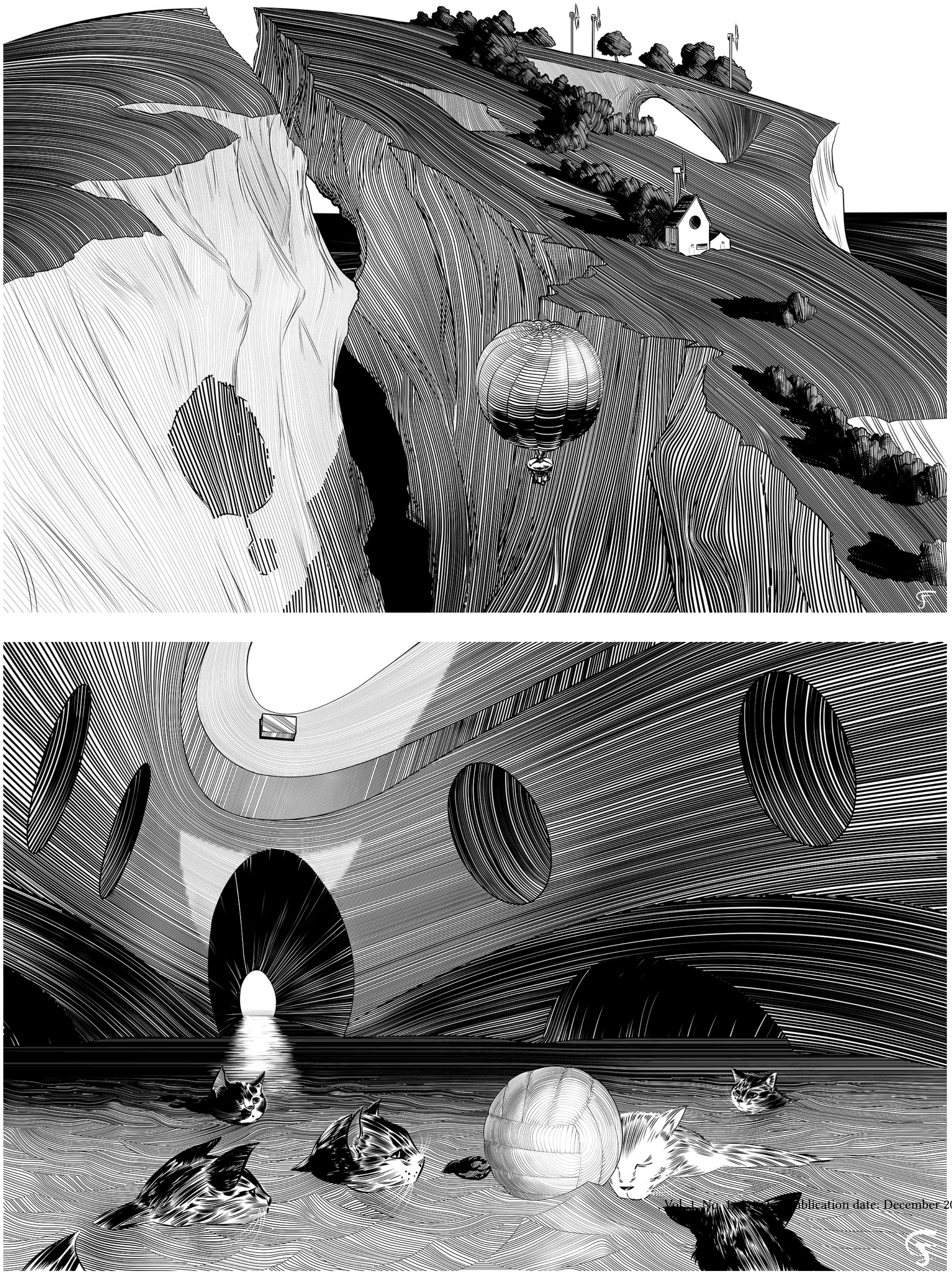


Fig. 29. “Breeze” (top) and “Twee” rely heavily on hatching shapes.

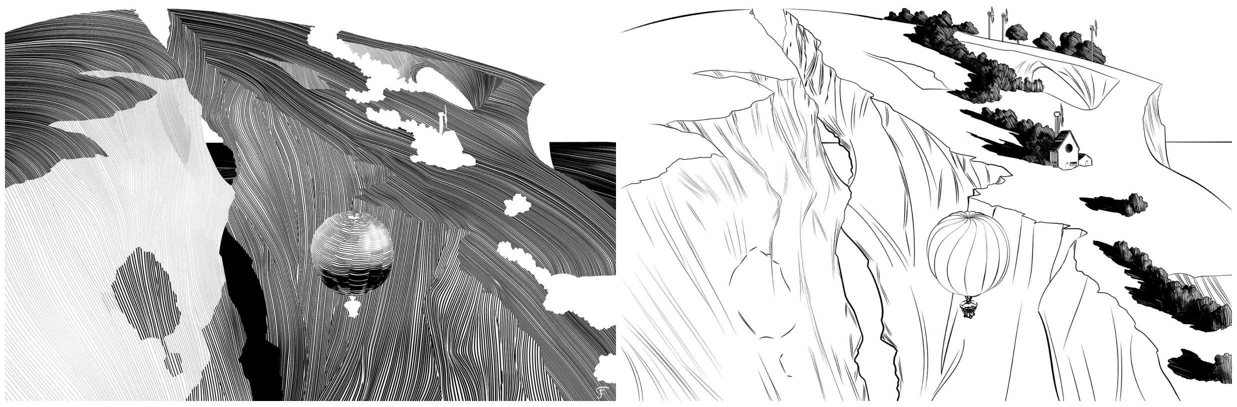


Fig. 30. “Breeze” contains procedurally generated marks from hatching shapes (left) and individually hand-drawn marks (right).

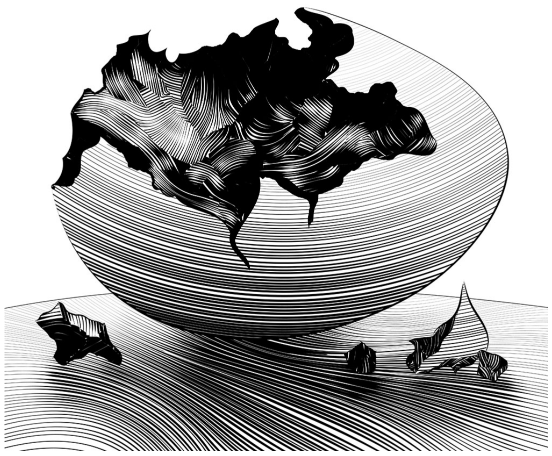


Fig. 31. Hatching shapes are ideal for this form because it is easy to visually separate into smooth surfaces, plus individual hatching marks have little pictorial importance. One hatching shape covers the outside of the egg and another handles the exposed interior.

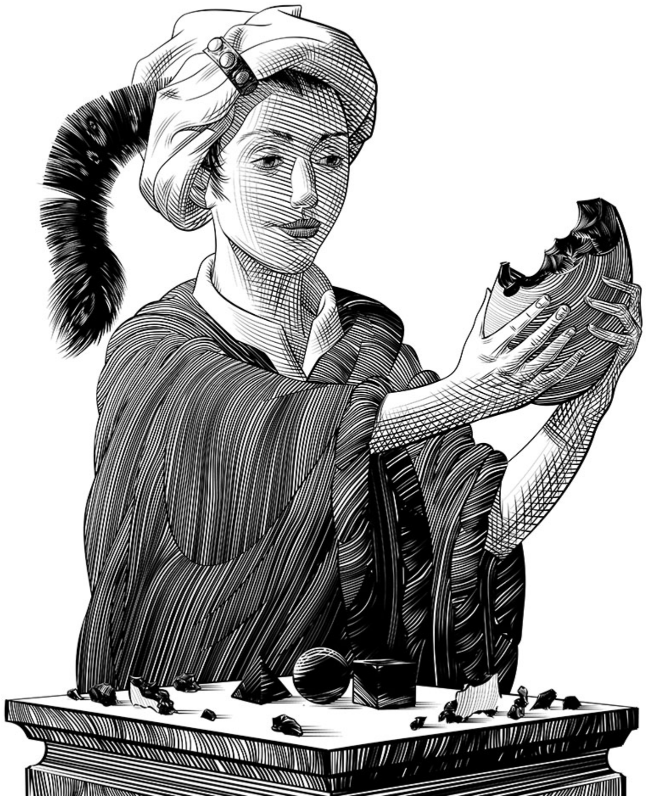


Fig. 32. “Houppelande” contains two hatching shapes, which produce the hatching of the dark fabric and of the egg’s outer surface. Everything else was more practical to accomplish with individual hand-drawn strokes.

against barrier curves (this is discernible in Figures 1 and 2; normally we mask this effect by placing a hand-drawn mark on top of a barrier curve).

These interviews, plus the pictures we have made, suggest that the hatching shape and its operations have value. Our finished pictures are almost able to communicate this on their own, but remember that much of the hatching shape’s value is the ease of repeated editing. That, if nothing else, is the hatching shape’s reason for being.

8 LIMITATIONS AND FUTURE WORK

Hatching shapes could grow to cover a broader range of hatching tasks. Some enhancements might be simple. For example, currently a barrier curve applies to all three hatching fields at once, but there are situations where it would be better for a barrier to apply only to one or two of the fields, such as in the dunes of Figure 14. Here, streamlines stop at the barriers representing shadows, which makes the forms harder to read. The hatching would be more legible if those barriers applied only to the width field, so that streamlines did not terminate at them but abruptly changed their widths instead.

Other improvements could be much more involved, entailing the addition of completely new data fields, a more sophisticated mask definition, or new mechanisms for seeding and advecting streamlines. Below we consider a few broad areas for future work.

8.1 Tonal Controls

The previous 2D hatching solutions that inspire our work tend to make tone an explicit part of their interfaces and algorithms. In practice this means that hatching patterns must approximate the tones of reference images. With hatching shapes, however, the artist operates below the level of tone, instead directly accessing hatching marks' widths and spacing, which are the two components of tone. We choose this approach partly because we are not specifically concerned with matching tones from reference images, but more importantly because we want to keep control over hatching as low-level as we can. However, it would be useful to wrap hatching shapes with higher-level tone-based controls. These controls ought to be tunable by the artist, since it is possible to achieve a desired tonal pattern by adjusting just a hatching shape's width field, just its spacing field, or both at once.

8.2 Spacing Artifacts

Hatching shapes sometimes produce spacing artifacts, as shown earlier in the left half of Figure 7. Figure 33 shows two more cases (first two rows). These artifacts stem from our streamline seeding strategy. As a reminder, we enqueue seed positions on a coarse-to-fine series of grids. This process is almost completely oblivious to the ideal spacing between marks.

Consider the first case in Figure 33, where the direction and spacing fields are constant. Streamlines initially grow from far-apart seed positions taken from a very coarse grid, then from a finer but still coarse grid, and so on. At some point, two streamlines are placed such that the vertical distance between them is not an integer multiple of d_{sep}, which leads to the artifacts. (Sometimes a constant-spacing, constant-direction hatching shape does not manifest this problem—see the right half of Figure 7.) In the second case, the spacing field varies linearly from top to bottom but the resulting pattern features banding in inter-mark spacings. The bands correspond to the different grids used for enqueuing seed positions.

As a stopgap, we have implemented an optional, line-based seeding mode where we fill the primary seed queue by starting at the center of the shape and placing seeds while walking perpendicular to the direction field, using the spacing field for each step distance. Just in case this approach leaves gaps, we append all the seeds produced by the default method. Line-based seeding works well if the direction field is constant and the spacing field only varies perpendicular to the direction field. However, it is too crude to handle more complex hatching shapes in a visually uniform way. The bottom-right square of Figure 33 shows a failure case. Seeds from the line-based method produce the hatching in the bottom left of the square, while seeds from the default method fill in the top right. An unwanted false contour separates the two regions.

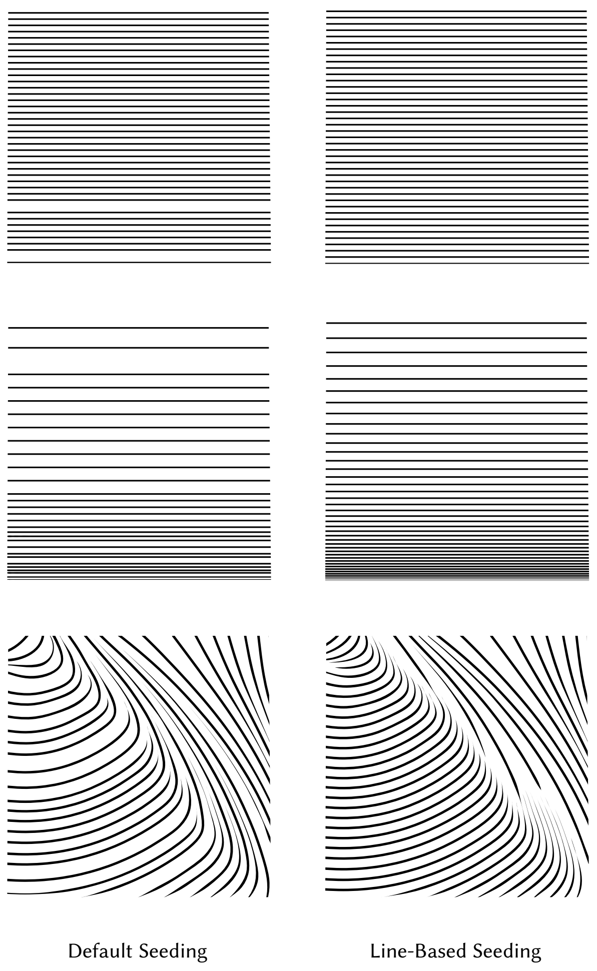
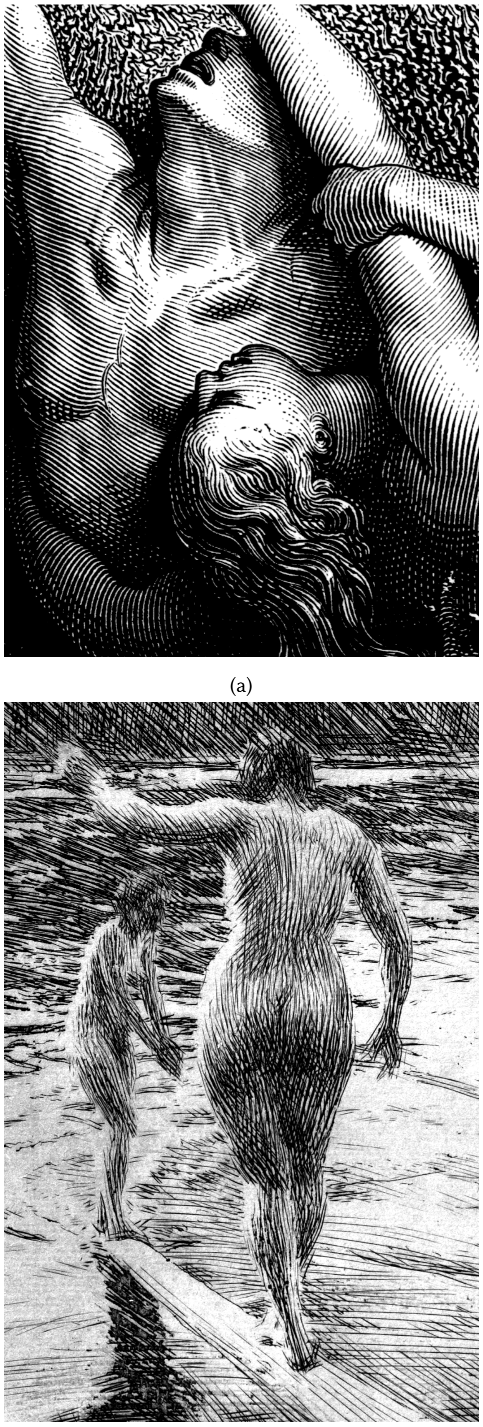


Fig. 33. The left column shows the default seeding method while the right shows the alternate, line-based seeding method. The line-based method fixes spacing artifacts for simpler hatching shapes (first two rows), but is too simplistic for more complex ones (bottom row), where it fails to produce a uniform look.

Moving beyond stopgaps, there may be multiple valid ways to resolve spacing artifacts. We might adopt a more intelligent seeding strategy that accounts for the structure of both the spacing and the direction fields. Or we might fix spacing irregularities in a post-processing step.

8.3 Messy Hatching

Hatching shapes favor "clean" hatching (Figure 34a), where marks are as long as possible and a single mark has clear left and right neighbors. In "messy" hatching (Figure 34b), marks are more chaotically placed and choppy. Subjects like fur or dense vegetation are not separable into smooth surfaces divided by barrier curves. For now, hatching shapes are insufficient for tackling such hatching. Enabling that would require augmentations, like a choppiness field.



(a)

(b)

Fig. 34. “Clean” and “messy” hatching, demonstrated respectively by details from *The Deluge*, Gustave Doré (a), circa 1866, engraving, and from *Balans*, Anders Zorn, 1919, etching (b).

It would also help to give hatching shapes soft rather than hard masks.

ACKNOWLEDGMENTS

We thank Linda Carson for her cross-disciplinary input on this work. We also acknowledge support from the Natural Sciences and Engineering Research Council of Canada (NSERC) Grant No. 2020-04024.

REFERENCES

Zainab AlMeraj, Brian Wyvill, Tobias Isenberg, Amy A. Gooch, and Richard Guy. 2009. Automatically Mimicking Unique Hand-Drawn Pencil Lines. *Comput. Graph.* 33, 4 (Aug. 2009), 496–508. https://doi.org/10.1016/j.cag.2009.04.004

Dmitry Apanovich. 2021. Strokes Maker. http://www.strokesmaker.com.

Pascal Barla, Simon Breslav, Joëlle Thollot, François Sillion, and Lee Markosian. 2006. Stroke Pattern Analysis and Synthesis. *Computer Graphics Forum* 25, 3 (2006), 663–671. https://doi.org/10.1111/j.1467-8659.2006.00986.x

Hedlena Bezerra, Elmar Eisemann, Doug DeCarlo, and Joëlle Thollot. 2010. Diffusion Constraints for Vector Graphics. In *Proc. Non-Photorealistic Animation and Rendering* (Annecy, France) *(NPAR)*. Association for Computing Machinery, New York, NY, USA, 35–42. https://doi.org/10.1145/1809939.1809944

Alexander Browne. 1669. *Ars Pictoria.* Redmayne, Tooker, Battersby.

Minh Tuan Bui, Junho Kim, and Yunjin Lee. 2015. 3D-Look Shading from Contours and Hatching Strokes. *Comput. Graph.* 51, C (Oct. 2015), 167–176. https://doi.org/10.1016/j.cag.2015.05.026

Siddhartha Chaudhuri. 2017. Laplacian Mesh Processing. https://www.cse.iitb.ac.in/~cs749/spr2017/lecs/18_laplace.pdf.

Yuan Chen, Jonathan Cohen, and Julian Krolik. 2007. Similarity-guided streamline placement with error evaluation. *Trans. Visualization and Computer Graphics* 13, 6 (2007), 1448–1455. https://doi.org/10.1109/TVCG.2007.70595

Frédo Durand, Victor Ostromoukhov, Mathieu Miller, Francois Duranleau, and Julie Dorsey. 2001. Decoupling strokes and high-level attributes for interactive traditional drawing. In *Rendering Techniques 2001*. Springer, 71–82. https://doi.org/10.1007/978-3-7091-6242-2_7

Mark Finch, John Snyder, and Hugues Hoppe. 2011. Freeform Vector Graphics with Controlled Thin-Plate Splines. *ACM Trans. Graph.* 30, 6 (Dec. 2011), 1–10. https://doi.org/10.1145/2070781.2024200

Moritz Gerl and Tobias Isenberg. 2013. Interactive Example-Based Hatching. *Computers & Graphics* 37, 1 (2013), 65–80. https://doi.org/10.1016/j.cag.2012.11.003

Astute Graphics. 2021. WidthScribe. https://astutegraphics.com/plugins/widthscribe.

Gaël Guennebaud, Jacob Benoît, et al. 2010. Eigen. http://eigen.tuxfamily.org.

Aaron Hertzmann and Denis Zorin. 2000. Illustrating Smooth Surfaces. In *Proc. Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 517–526. https://doi.org/10.1145/344779.345074

Emmanuel Iarussi, David Bommes, and Adrien Bousseau. 2015. BendFields: Regularized Curvature Fields from Rough Concept Sketches. *ACM Trans. Graph.* 34, 3, Article 24 (May 2015), 16 pages. https://doi.org/10.1145/2710026

Tobias Isenberg. 2016. Interactive NPAR: What Type of Tools Should We Create?. In *Proc. Computational Aesthetics, Sketch-Based Interfaces, Modeling, Non-Photorealistic Animation, and Rendering* (Lisbon, Portugal) *(Expressive)*. Eurographics Association, Goslar, DEU, 89–96. https://doi.org/10.2312/exp.20161067

Bruno Jobard and Wilfrid Lefer. 1997. Creating Evenly-Spaced Streamlines of Arbitrary Density. In *Visualization in Scientific Computing*, Wilfrid Lefer and Michel Grave (Eds.). Springer Vienna, Vienna, 43–55. https://doi.org/10.1007/978-3-7091-6876-9_5

Pierre-Marc Jodoin, Emric Epstein, Martin Granger-Piché, and Victor Ostromoukhov. 2002. Hatching by Example: A Statistical Approach. In *Proc. Non-photorealistic Animation and Rendering* (Annecy, France) *(NPAR)*. ACM, New York, NY, USA, 29–36. https://doi.org/10.1145/508530.508536

Evangelos Kalogerakis, Derek Nowrouzezahrai, Simon Breslav, and Aaron Hertzmann. 2012. Learning Hatching for Pen-and-Ink Illustration of Surfaces. *ACM Trans. Graph.* 31, 1, Article 1 (Feb. 2012), 17 pages. https://doi.org/10.1145/2077341.2077342

Felix Knöppel, Keenan Crane, Ulrich Pinkall, and Peter Schröder. 2015. Stripe Patterns on Surfaces. *ACM Trans. Graph.* 34, 4, Article 39 (July 2015), 11 pages. https://doi.org/10.1145/2767000

Hans Knutsson. 1989. Representing Local Structure Using Tensors. *Proc. Scandinavian Conference on Image Analysis* s. 244-251 (1989). https://doi.org/10.1.1.140.7699

K. Lawonn, T. Moench, and B. Preim. 2013. Streamlines for Illustrative Real-Time Rendering. *Comput. Graph. Forum* 32, 3pt3 (2013), 321–330. https://doi.org/10.1111/cgf.12119

Kai Lawonn, Ivan Viola, Bernhard Preim, and Tobias Isenberg. 2018. A Survey of Surface-Based Illustrative Rendering for Visualization. *Computer Graphics Forum* 37, 6 (2018), 205–234. https://doi.org/10.1111/cgf.13322

Changjian Li, Hao Pan, Yang Liu, Xin Tong, Alla Sheffer, and Wenping Wang. 2017. BendSketch: Modeling Freeform Surfaces through 2D Sketching. *ACM Trans. Graph.* 36, 4, Article 125 (July 2017), 14 pages. https://doi.org/10.1145/3072959.3073632

Liya Li, Hsien-Hsi Hsieh, and Han-Wei Shen. 2008. Illustrative Streamline Placement and Visualization. *IEEE Pacific Visualisation Symposium*, 79 – 86. https://doi.org/10.1109/PACIFICVIS.2008.4475462

Nils Lichtenberg, Noeska Smit, Christian Hansen, and Kai Lawonn. 2018. Real-time field aligned stripe patterns. *Computers & Graphics* 74 (05 2018). https://doi.org/10.1016/j.cag.2018.04.008

Zhanping Liu, Robert Moorhead, and Joe Groner. 2006. An Advanced Evenly-Spaced Streamline Placement Algorithm. *IEEE Trans. Visualization and Computer Graphics* 12 (10 2006), 965–72. https://doi.org/10.1109/TVCG.2006.116

Abdelkrim Mebarki. 2016. Adaptive Distance Grid Based Algorithm for Farthest Point Seeding Streamline Placement. *Open Computer Science* 1, open-issue (2016). https://doi.org/10.1515/comp-2016-0007

Abdelkrim Mebarki, Pierre Alliez, and Olivier Devillers. 2005. Farthest Point Seeding for Efficient Placement of Streamlines. In *VIS 2005*. IEEE Computer Society, 479–486. https://doi.org/10.1109/VISUAL.2005.1532832

Alexandrina Orzan, Adrien Bousseau, Holger Winnemöller, Pascal Barla, Joëlle Thollot, and David Salesin. 2008. Diffusion Curves: A Vector Representation for Smooth-Shaded Images. *ACM Trans. Graph.* 27, 3 (Aug. 2008), 1–8. https://doi.org/10.1145/1360612.1360691

Victor Ostromoukhov. 1999. Digital Facial Engraving. In *Proc. Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 417–424. https://doi.org/10.1145/311535.311604

G. Philbrick and C. S. Kaplan. 2019. Defining Hatching in Art. In *Proc. Computational Aesthetics, Sketch-Based Interfaces, Modeling, Non-Photorealistic Animation, and Rendering* (Genoa, Italy) *(Expressive)*. Eurographics Association, Goslar, DEU, 111–121. https://doi.org/10.2312/exp.20191082

Yachin Pnueli and Alfred Marcel Bruckstein. 2005. DigiDürer — a digital engraving system. *The Visual Computer* 10 (2005), 277–292. https://doi.org/10.1007/BF01901584

Shivam Pradhan. 2020. Mid-Point Line Generation Algorithm. https://www.geeksforgeeks.org/mid-point-line-generation-algorithm/?ref=rp.

Mike Salisbury, Corin Anderson, Dani Lischinski, and David H. Salesin. 1996. Scale-Dependent Reproduction of Pen-and-Ink Illustrations. In *Proc. Computer Graphics and Interactive Techniques (SIGGRAPH)*. Association for Computing Machinery, New York, NY, USA, 461–468. https://doi.org/10.1145/237170.237286

Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. 1994. Interactive Pen-and-ink Illustration. In *Proc. Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM, New York, NY, USA, 101–108. https://doi.org/10.1145/192161.192185

Michael P. Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin. 1997. Orientable Textures for Image-based Pen-and-ink Illustration. In *Proc. Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 401–406. https://doi.org/10.1145/258734.258890

Reza Adhitya Saputra, Craig S. Kaplan, Paul Asente, and Radomír Měch. 2017. FLOWPAK: Flow-Based Ornamental Element Packing. In *Proceedings of the 43rd Graphics Interface Conference* (Edmonton, Alberta, Canada) *(GI '17)*. Canadian Human-Computer Communications Society, Waterloo, CAN, 8–15. https://doi.org/10.20380/GI2017.02

Michael Schlemmer, Ingrid Hotz, Bernd Hamann, Florian Morr, and Hans Hagen. 2007. Priority Streamlines: A Context-Based Visualization of Flow Fields. In *Proc. Visualization* (Norrköping, Sweden) *(EUROVIS)*. Eurographics Association, Goslar, DEU, 227–234. https://doi.org/10.2312/VisSym%2FEuroVis07%2F227-234

Martin Schwarz, Tobias Isenberg, Katherine Mason, and Sheelagh Carpendale. 2007. Modeling with Rendering Primitives: An Interactive Non-Photorealistic Canvas. In *Proc. Non-Photorealistic Animation and Rendering* (San Diego, California) *(NPAR '07)*. Association for Computing Machinery, New York, NY, USA, 15–22. https://doi.org/10.1145/1274871.1274874

Cloud Shao, Adrien Bousseau, Alla Sheffer, and Karan Singh. 2012. CrossShade: Shading Concept Sketches Using Cross-section Curves. *ACM Trans. Graph.* 31, 4, Article 45 (July 2012), 11 pages. https://doi.org/10.1145/2185520.2185541

Mayank Singh and Scott Schaefer. 2010. Suggestive Hatching. In *Proc. Computational Aesthetics in Graphics, Visualization and Imaging* (London, United Kingdom) *(Computational Aesthetics)*. Eurographics Association, Goslar, DEU, 25–32. https://doi.org/10.2312/COMPAESTH/COMPAESTH10/025-032

The CGAL Project. 2020. *CGAL User and Reference Manual* (5.1 ed.). CGAL Editorial Board. https://doc.cgal.org/5.1/Manual/packages.html.

Pekka J. Toivanen. 1996. New geodosic distance transforms for gray-scale images. *Pattern Recognition Letters* 17, 5 (1996), 437 – 450. https://doi.org/10.1016/0167-8655(96)00010-4

Vivek Verma, David Kao, and Alex Pang. 2000. A Flow-Guided Streamline Seeding Strategy. In *Proceedings of the Conference on Visualization '00* (Salt Lake City, Utah, USA) *(VIS)*. IEEE Computer Society Press, Washington, DC, USA, 163–170. https://doi.org/10.1109/VISUAL.2000.885690

A.H. Watt. 2000. *3D Computer Graphics*. Addison-Wesley. https://books.google.ca/books?id=tQJEAQAAIAAJ

M. Woo, J. Neider, T. Davis, and OpenGL Architecture Review Board. 1997. *The Official Guide to Learning OpenGL* (1.1 ed.). Addison Wesley.

Jun Xing, Hsiang-Ting Chen, and Li-Yi Wei. 2014. Autocomplete Painting Repetitions. *ACM Trans. Graph.* 33, 6, Article 172 (Nov. 2014), 11 pages. https://doi.org/10.1145/2661229.2661247

Johannes Zander, Tobias Isenberg, Stefan Schlechtweg, and Thomas Strothotte. 2004. High Quality Hatching. *Comput. Graph. Forum* 23 (09 2004), 421–430. https://doi.org/10.1111/j.1467-8659.2004.00773.x