Voronoi toolpaths for PCB mechanical etch: Simple and intuitive algorithms with the 3D GPU

Marsette A. Vona*, Daniela Rus
Department of Electrical Engineering and Computer Science
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
*vona@mit.edu

***Abstract*— We describe *VIsolate* (*V*oronoi *I*solate), a system which performs geometric computations associated with toolpath planning for mechanical etch (also called isolation routing) of printed-circuit boards, including the computation of a novel Voronoi-based toolpath with some advantages over the current industry practice. We highlight how we use the 3D Graphics Processing Unit (GPU) to implement simple, intuitive algorithms in VIsolate, including polygon overlap detection, 2D offset, and constrained generalized Voronoi diagram computation, building on a method from [1]. Thus, this work also illustrates how we can employ the GPU as a rudimentary 'mind's eye' for the machine, allowing us to rapidly implement visually-intuitive geometric algorithms.**

I. Introduction

With the continued push for miniaturization the density of traces on PCBs is increasing, and the individual widths of the traces is decreasing. Industrial photochemical processes have been tuned to handle this increased density and decreased feature size. However, it can be challenging to produce such boards with the mechanical etch process[1], which is preferred in some rapid-prototyping and in-office fabrication environments. We must minimize the amount by which the cutter kerf encroaches into the copper intended to be part of the trace, which is challenging for extremely close-spaced and narrow traces.

The current industry practice is to cut a loop around the boundary of each trace. This will always produce *two* cutter passes between any two adjacent traces, even when the traces are so closely spaced that these passes may overlap. This results in an unnecessarily long toolpath, which translates to longer production time. Even when the passes do not quite overlap, we must still consider second-order effects which may degrade the geometry of such closely spaced cuts, e.g. play in the machine and flexing of the cutter.

By merging the two passes into one in such cases, we can get a single pass maximally distant from both traces. In many cases that would also effectively widen the traces, thus maximizing current carrying capacity, which is a concern in tiny circuits that must carry power. This motivation led us to the concept of *Voronoi toolpaths*: we compute a generalized Voronoi diagram using the PCB traces as Voronoi sites, and then we cut only along the boundaries between the resulting Voronoi regions.

We have implemented a prototype software system called *VIsolate* (*V*oronoi *I*solate) which computes both standard outline and our novel Voronoi toolpaths, and we have tested VIsolate by cutting several PCBs on a Sherline CNC mini-mill. Two of these PCBs are shown in Figure 1 (middle, right).

We also used VIsolate to experimentally measure the decrease in toolpath length and increase in trace width of the Voronoi toolpath method relative to the standard outline toolpath. Results for 7 boards are given in Section IV. The average toolpath length savings for these boards was about 37%, and the average approximate trace width increase was about 443%.

VIsolate accepts a board design in the standard Gerber (EIA RS-274X) format, and emits a cutter toolpath in the standard 'G-Code' (EIA RS-274D/ISO 6983) format. Most CNC milling machines can accept this output format directly, and we presume that it can also be easily adapted to run existing commercial mechanical etch machines.

A. Simple and Intuitive Algorithms with the 3D GPU

Algorithms exist to analytically compute generalized Voronoi diagrams, e.g. [2]. However, such algorithms can be tedious and tricky to implement. Instead, we adopt the intuitive and relatively simple GPU[2]-based method proposed by Hoff et al. in [1]. We also introduce two simple additions to this method which constrain the resulting Voronoi regions to maximal (Section III-D.1) and minimal (Section III-D.2) extents.

Continuing in this theme, we use simple and intuitive approximate GPU-based algorithms in VIsolate for

- finding overlaps among a set of planar polygons (Section III-A.2)
- computing the offset of planar polygonal geometry, including easy handling of offset self-intersections (Section III-C)

[1] In *mechanical etch* (also *isolation routing*), copper is removed mechanically by a high-speed rotary cutter, rather than photochemically. Specialized PCB mechanical etch systems are offered commercially by companies such as LPKF and T-Tech. However, it is also possible to use a general purpose CNC vertical milling machine with a high-speed spindle.

[2] (3D) Graphics Processing Unit

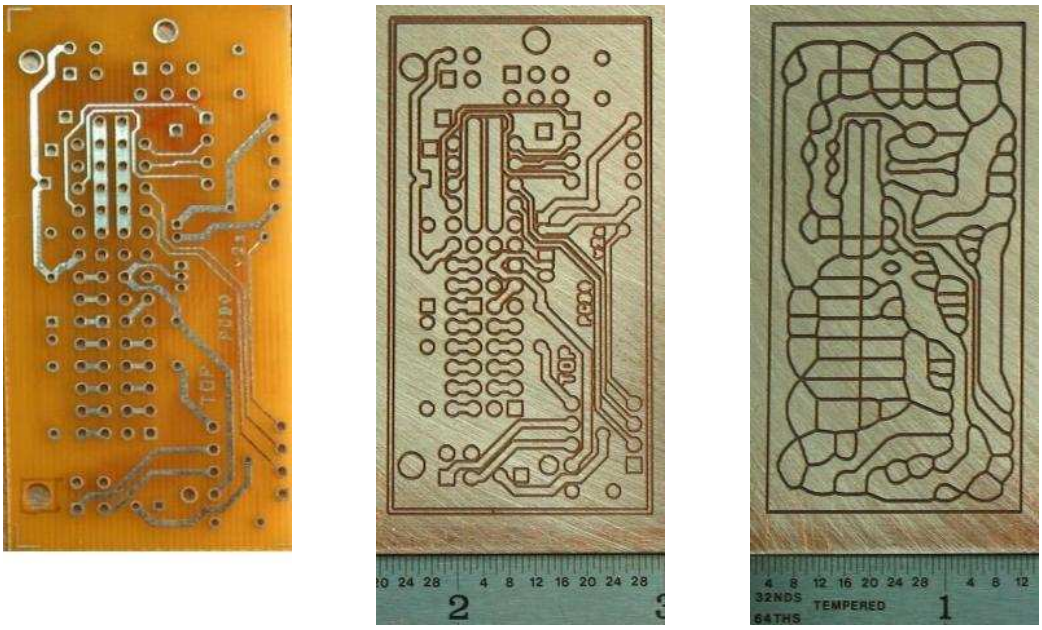

Fig. 1. The same printed circuit board (PCB) manufactured three different ways: by traditional photochemical process (left); by mechanical etch with standard outline toolpaths (middle); by mechanical etch with novel Voronoi toolpaths (right). The board on the left has also had additional processing: it has been drilled and tin-plated. The toolpaths for cutting the middle and right boards were computed with the system described in the text, and these boards were cut on a Sherline CNC mini-mill.

Taken together, this work gives several examples of how we can employ the GPU as a rudimentary 'mind's eye' for the machine, allowing us to rapidly implement visually-intuitive geometric algorithms.

The intended message of this paper is thus two-fold. First, we present Voronoi toolpaths as a novel method for PCB mechanical etch with some advantages over the current practice in this field. Second, we show practical examples of how to use the GPU to allow rapid implementation of simple and intuitive algorithms for some fairly generic geometric problems.

II. Related Work

While PCB mechanical etch systems are currently available commercially, there seems to be very little written about toolpath planning for mechanical etch in the research literature. The commercial systems utilize proprietary software, such as LPKF's 'CircuitCAM', to compute toolpaths. Discussion with users of this software [3], [4] suggest that the methods it employs are somewhat ad-hoc and can potentially be improved. For example, in many cases multiple very closely spaced toolpaths are produced where one would suffice, a problem we specifically address with our proposed Voronoi toolpaths.

Voronoi diagrams have long been used for motion planning. For example, Takahashi et al. proposed moving along Voronoi boundaries to avoid obstacles in [5]. To our knowledge, producing PCBs by separating traces along the boundaries of their associated Voronoi regions has not been previously reported.

Lengyel et al., also addressing the more general motion planning problem, propose an algorithm which employs the 3D GPU for path planning computations in [6]. This work, along with the work of Hoff et al., [1], is part of a growing body of research in the area of using the 3D GPU to perform computations other than direct visualization. While many of these works focus on simulated and idealized situations, we show that such methods can also be applied to a practical real-world motion planning problem.

III. Computations for Mechanical Etch

Two high-level operations must be performed to compute the etch paths:

1) we must recover the topology and geometry of each independent trace from the Gerber input
2) we must generate G-Code which creates sufficient cuts in the copper to electrically isolate each trace from all others

Due to the specifics of the Gerber format (see Section III-A.2 below), the first step is not trivial. It involves finding areas of overlap among a set of planar polygons. The second step is complicated by the necessity of offsetting the trace geometry by the radius of the cutter, which can lead to self-intersection of the offset outline (see Section III-C below). In general there is an infinite set of toolpaths which achieve the requisite electrical isolation, and we may want to choose a specific toolpath which has 'good' properties. In Section III-D we propose one such optimization, where the toolpath is computed to follow the boundaries of a Voronoi diagram induced by the traces.

A. Recovering Traces

The Gerber format was originally designed to control a machine called a photoplotter, which had an optical aperture of

selectable shape (most often a small circle, square, or rectangle with rounded corners is selected), a motion system to move the aperture relative to the PCB in the plane, and a switchable bulb for projecting light through the aperture. Controlled by a Gerber-format input, the photoplotter would 'draw' the traces which make up a PCB. As such, the format is essentially[3] a list P of primitives:

- *flashes*, which are produced when the aperture is set and the light is turned on and off without moving
- and *strokes*, which are produced when the aperture is set, the light is turned on, and the aperture is moved along a single line segment

If we further assert that all aperture shapes be specified as polygons (approximating where necessary), then each primitive is also a polygon. Flashes simply have the shape of the corresponding aperture polygon, and strokes have the shape of the corresponding aperture polygon swept along a line segment.

Maximal electrically connected regions, or *traces*, are composed of disjoint maximal connected subsets of the primitives in P, one subset per trace. Unfortunately there is no constraint which says these subsets must be individually contiguous in the input representation, or indeed identifiable in any other way than by finding all transitive closures of overlapping primitives.

1) Reducing the Number of Overlaps: We observe experimentally that PCB design systems do often produce Gerber output which *mostly* has the form shown in Figure 2, which is amenable to rapid extraction of the traces by a simple graph-based algorithm that does not require overlap detection. We give this as Algorithm 1 and run it as a pre-processing step in order to significantly reduce the number of overlaps that must be detected.

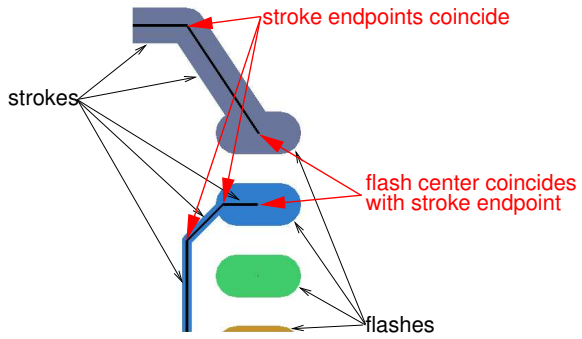

Fig. 2. We can consider two strokes which share a common vertex to be parts of the same trace. Similarly, we can consider a flash sharing its center point with another flash and/or with a stroke vertex to be parts of the same trace.

The first loop in SIMPLE-TOPOLOGY-RECOVER constructs an adjacency-list representation of the full embedded planar graph of all the primitives, with stroke endpoints[4] and flash center points as vertices. The second loop repeatedly extracts connected components from this graph by DFS. Each such connected component is considered to be a trace.

[3]For now we will ignore features such as the ability to draw arcs, arbitrary polygons, complex apertures, and to draw in 'inverse' mode.

SIMPLE-TOPOLOGY-RECOVER(P)
▷ P is the list of primitives from the Gerber
for p **in** P
  for v **in** VERTICES-OF(p)
    $adj\text{-}prims[v] \leftarrow adj\text{-}prims[v] \cup \{p\}$
while $P \neq \emptyset$
  $T \leftarrow T \cup \{$EXTRACT-TRACE$(\text{remove-first}(P), \emptyset)\}$
▷ T is the set of traces
return T

EXTRACT-TRACE(p, t)
▷ p is a primitive
▷ t is a trace (a set of primitives)
$t \leftarrow t \cup \{p\}$
$adj \leftarrow \emptyset$
for v **in** VERTICES-OF(p)
  $adj \leftarrow adj \cup (adj\text{-}prims[v] \cap P)$
$P \leftarrow P \ / \ adj$
for p **in** adj
  $t \leftarrow$ EXTRACT-TRACE(p, t)
return t

VERTICES-OF(p)
if p is a flash
  then return center-of(p)
  else return endpoints-of(p) ▷ p is a stroke

Algorithm 1: A simple algorithm to recover most of the topology of the traces from the Gerber in the common case.

Theorem 1: SIMPLE-TOPOLOGY-RECOVER returns a correct set of traces T (*only*) where the Gerber input has the form shown in Figure 2.

Proof Sketch: SIMPLE-TOPOLOGY-RECOVER's criteria for identifying overlapping primitives is conservative but not complete: if the primitives have coincident vertices then they *must* overlap; however, two primitives *may* overlap even if they share no vertices. Thus SIMPLE-TOPOLOGY-RECOVER correctly assigns primitives to traces except in cases where overlap is not accompanied by coincident vertices (as in Figure 3 (left)). ■

Theorem 2: The total time complexity of SIMPLE-TOPOLOGY-RECOVER is $O(|P|)$, i.e. linear in the complexity of the Gerber input.

Proof Sketch: For real-world PCB designs, the out-degree of the graph defined by the $adj\text{-}prims$ adjacency lists

[4]When we speak of an 'endpoint' of a stroke, we mean the center point of the aperture used to create the stroke when positioned at one end or the other of the stroke.

in SIMPLE-TOPOLOGY-RECOVER is bounded by a constant because traces almost never have many branches starting from the same point. Thus the adjacency lists can all be constructed in total $O(|P|)$ time. EXTRACT-TRACE is a DFS in this graph, and since each call to it extracts a disjoint subgraph (i.e. trace), a total of $O(|P|)$ time is spent there as well. ■

2) Detecting Overlaps: As it relies on the above-stated assumption about the Gerber input, Algorithm 1 does not work in all cases. Some parts of some traces may not have segment endpoints and flash center points which precisely match-up, even when most parts do. An example is shown in Figure 3 (left), in which Algorithm 1 has already run and assigned primitives to traces as best as it could. Such cases can only be detected by searching for geometric overlaps.

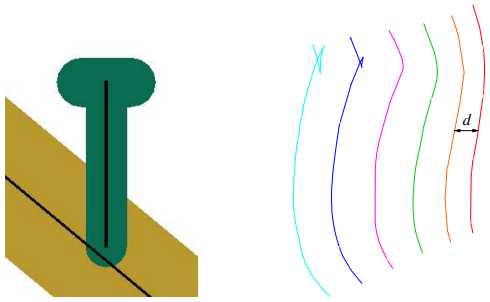

Fig. 3. Left: An example where a segment endpoint does not coincide with either endpoint of an adjacent segment, even though the geometries of the segments overlap (and therefore the segments are electrically connected and are parts of the same trace). Right: A sequence of offset curves. The base curve at right is offset at successive distances d. The two leftmost offset curves show self-intersections (adapted from [7]).

We propose Algorithm 2 as a simple and intuitive GPU-based algorithm to be run on the output of Algorithm 1 to detect overlaps in cases like that shown in Figure 3 (left).

TOPOLOGY-FROM-TRANSLUCENCY renders all traces in different *translucent* colors, so that areas of overlap will have a color that is a mix of the underlying trace colors. It then scans through the resulting image looking for pixels with mix colors. When it finds such a pixel, it performs a 3D pick operation using the GPU to determine the set of traces which overlap at that point.

Since it operates by scanning pixels, TOPOLOGY-FROM-TRANSLUCENCY returns only a discrete approximation: overlap areas with minimum dimension less than about $\epsilon = 1/(\text{rendered dots per inch})$ will not be detected. In practice, we have found that with real-world PCB designs, setting ϵ at about 0.002 (500 DPI) is sufficient to catch all overlaps.

The set *seen* is used to greatly accelerate TOPOLOGY-FROM-TRANSLUCENCY by keeping track of colors which have already been 'seen' and which can thus be ignored. *seen* is seeded with the base colors of the traces, and every time a new mix color is found it is added to *seen*. Such a mechanism is critical to the performance of the algorithm. However, the use of *seen* makes TOPOLOGY-FROM-TRANSLUCENCY a Monte Carlo randomized algorithm. It is possible that a mix color will happen to be identical to one of the base colors (call

TOPOLOGY-FROM-TRANSLUCENCY(T, ϵ, w, h)

▷ T is the set of traces as computed by SIMPLE-TOPOLOGY-RECOVER
▷ ϵ is the rendering resolution in inches per dot
▷ w and h are the board dimensions in inches
for t **in** T
  assign t a unique random color other than black
simultaneously render all $t \in T$ 50% translucent parallel to the xy-plane at staggered depths in z against a black background
with an orthogonal projection and a viewpoint looking straight down the z-axis, acquire an $\lceil h/\epsilon \rceil \times \lceil w/\epsilon \rceil$ bitmap B of the rendering
$seen \leftarrow \emptyset$
for t **in** T
  $seen \leftarrow seen \cup \lceil \text{color-of}(t)/2 \rceil$
for i **in** $[0\,..\,(\lceil h/\epsilon \rceil - 1)]$
  for j **in** $[0\,..\,(\lceil w/\epsilon \rceil - 1)]$
    while $B[i,j] \notin seen$
      use GPU to do a 3D pick along a z-line at pixel (i,j); let U be the set of picked traces
      $m \leftarrow \emptyset$
      for t **in** U
        $T \leftarrow T/\{t\}$
        $m \leftarrow m \cup t$
      $T \leftarrow T \cup \{m\}$
      $seen \leftarrow seen \cup B[i,j]$
      re-render and re-acquire B
return T

Algorithm 2: A GPU-based algorithm to detect geometrically overlapping traces which were not detected by Algorithm 1 (as shown, for example, in Figure 3 (left)) and merge them.

the probability of such a collision with a base color P_{bc}), and it is also possible that more than one combination of base colors will form the same mix color (P_{mc}). However, the probability of either of these occurring is low.

Assuming 24-bit RGB rendering, the value that the GPU hardware computes for one of the three color components of a pixel p in bitmap B is given by

$$c(p) = \sum_{i=0}^{O(p)} \lceil 2^{-i} c_i \rceil$$

where $O(p)$ is the number of overlapping traces at p, $c_0 = 0$, and $c_{i>0}$ is the value of the corresponding color component of the ith overlapping trace at p[5].

Thus, components of the base colors ($O(p)$=1) will actually have only 7 significant bits. However, mix colors (1<$O(p)$<8)

[5]Note that for $i \geq 8$ all bits will be lost. Thus our algorithm may fail in cases where more than 7 traces overlap at the same pixel; however we observe that in practice the number of overlapping traces at a pixel is almost never more than two.

have 8 significant bits, because they are always made by a sum of positive integers where one of the addends is a 7-bit number.

Lemma 3: The probability of a mix color component having a particular value is at most $1/128$, independent of $O(p)$. Thus the probability that a mix color has a particular value is at most $(1/128)^3 = 2^{-21}$.

Proof Sketch: One way to prove this is by brute-force exploration of the entire space of mix color components. ■

Theorem 4: TOPOLOGY-FROM-TRANSLUCENCY will never report an overlap where none exists, and the probability that TOPOLOGY-FROM-TRANSLUCENCY will fail to identify an overlap is low.

Proof Sketch: TOPOLOGY-FROM-TRANSLUCENCY only reports an overlap where a 3D pick along a z-line indicates that two or more trace geometries are stacked on top of each other; thus TOPOLOGY-FROM-TRANSLUCENCY can only falsely report an overlap if the 3D pick falsely reports an intersection with the z-line.

The number of base colors is the number of traces, $|T|$. For the boards we are working with, $|T|\sim 100$. Let N_o be the total number of overlapping regions on the board ($N_o \geq$ the number of mix colors). We observe $N_o \sim 10$ for the boards we have tried.

Since the probability that a mix color has a particular value is at most 2^{-21}, if we consider the sum of the collision probabilities across all distinct mix color/base color pairs we get

$$P_{bc} \leq \frac{N_o|T|}{2} 2^{-21} = N_o|T|2^{-22} \sim (100)(10)2^{-22} \sim \frac{1}{4000}$$

For P_{mc}, we have essentially the birthday problem[6]. Since the mix colors come from a gamut of 2^{24} possible colors (8 significant bits per color component), using a standard approximation from the birthday problem [8], we get

$$\begin{aligned} P_{mc} \leq 1 - \frac{2^{24}!}{2^{24^{N_o}}\,[2^{24} - N_o]!} &\approx 1 - \left[1 - \frac{N_o}{2 \cdot 2^{24}}\right]^{N_o - 1} \\ &\sim 1 - \left[1 - \frac{10}{2^{25}}\right]^{10-1} \\ &\sim 1/372000 \end{aligned}$$

■

The time complexity of TOPOLOGY-FROM-TRANSLUCENCY is not generally optimal. However it is an extremely intuitive algorithm, it works in practice, and, importantly, it leverages the software framework used by the rest of the application to such a great extent that it took only a few hours to implement.

Theorem 5: The time complexity of TOPOLOGY-FROM-TRANSLUCENCY is

$$O(|T| + \lceil h/\epsilon \rceil \lceil w/\epsilon \rceil + N_o \cdot (t_{pick} + t_{frame}))$$

where N_o is the number of overlapping regions in the traces T returned by SIMPLE-TOPOLOGY-RECOVER, t_{pick} is the time required to do a 3D pick in the scene, and t_{frame} is the time to render the scene and acquire the $\lceil h/\epsilon \rceil \times \lceil w/\epsilon \rceil$ bitmap B[7]. t_{pick} and t_{frame} depend on hardware and software implementation as well as board complexity, and are typically on the order of fractions of a second on modern hardware for the low to medium complexity boards we have tested so far.

Proof Sketch: For real-world PCB designs, the number of overlapping traces at any one point is bounded by a constant because traces do not usually have redundant overlapping primitives. Thus, the number of picks and renders performed in TOPOLOGY-FROM-TRANSLUCENCY at each overlap point is constant, so the total time spent at each of the N_o overlaps is $O(t_{pick} + t_{frame})$. ■

B. Toolpath Generation

Algorithms 1 and 2 constitute the first of the two phases of processing we described above. We now have the complete topology and geometry for each trace represented independently, so we can turn to the second phase of processing in VIsolate: generating a toolpath which produces cuts in the copper that electrically isolate each trace.

We first give a base implementation which cuts a loop around the boundary of each trace; below in Section III-D we extend this to generate our Voronoi-based toolpaths.

Since we have the collections of polygonal primitives which make up each trace, we could analytically determine the geometry of simple polygons that tightly bound each trace. However, we propose a simple, intuitive, GPU-based algorithm to solve this problem, given as Algorithm 3.

VECTORIZE-BOUNDARIES renders every trace in a different color and then extracts the boundaries between differently colored pixels (the background color is also unique). We use an adaptation of Selinger's 'Potrace' algorithm ([9]) to convert these (chains of) pixel boundaries into a set of connected line segments forming an embedded planar graph O, and then we break O into 2-connected subgraphs S.

S is easily turned into an overall toolpath for etching a PCB with traces T: simply cut along each path in S in some order $s_0, s_1, .., s_{|S|-1}$. We greedily pick s_{i+1} as the uncut path whose start is closest to the end of s_i.

VECTORIZE-BOUNDARIES returns a discrete approximation of the true boundary which is correct only up to the rendering resolution ϵ; we find in practice that a resolution of 0.002 [in] recovers boundaries with sufficient fidelity.

VECTORIZE-BOUNDARIES is correct as long as the raster-to-vector algorithm it employs is correct. Refer to [9] for analysis.

Although the time complexity of VECTORIZE-BOUNDARIES is not optimal, we find that this algorithm works in practice with reasonable efficiency, and we were able to code it correctly with only a moderate effort.

Theorem 6: The total running time of VECTORIZE-BOUNDARIES, including our adaptation of the Potrace algo-

[6] In a room with n people, what is the probability that at least two people have the same birthday?

[7] The rendering can be performed in tiles if the resolution exceeds hardware limits, with a corresponding increase in run-time.

VECTORIZE-BOUNDARIES(T, ϵ, w, h)

▷ T is the set of traces as computed by TOPOLOGY-FROM-TRANSLUCENCY
▷ ϵ is the rendering resolution in inches per dot
▷ w and h are the board dimensions in inches

1 **for** t **in** T
2 assign t a unique random color other than black
3 simultaneously render all $t \in T$ parallel to the xy-plane at staggered depths in z against a black background
4 with an orthogonal projection and a viewpoint looking straight down the z-axis, acquire an $\lceil h/\epsilon \rceil \times \lceil w/\epsilon \rceil$ bitmap B of the rendering
5 apply a raster-to-vector algorithm (e.g. [9]) to extract the boundaries between differently colored regions in B as an embedded planar graph O
6 separate O into a set of disjoint open 2-connected paths S
7 **return** S

Algorithm 3: A (partly) GPU-based algorithm to extract the planar boundaries separating the 2D projections of differently colored 3D geometries.

rithm, is approximately

$$O(|T| + t_{frame} + \lceil h/\epsilon \rceil \lceil w/\epsilon \rceil + (L_{boundary}/\epsilon)^2)$$

where t_{frame} is the time to render the bitmap B, as above, and $L_{boundary}$ is the total length of all inter-color boundaries in inches.

Proof Sketch: The first term in this expression is due to the **for** loop at the beginning of VECTORIZE-BOUNDARIES; the third term is the time required to scan B for all boundaries between pixels of different colors; and the fourth term is (approximately) the time required for Potrace to simplify that network of pixel boundaries into a clean vector representation. In practice, the running time seems to be dominated by this last term. ■

C. Adding a Cutter-Radius Offset

We can add an important feature to the toolpath extraction algorithm with a surprisingly small added effort, and without compromising intuitiveness: a geometric offset which compensates for the cutter radius.

Even the small diameter cutters used for PCB mechanical etch have some *kerf*, i.e., they leave a groove in the copper of some measurable width (typically 0.004 to 0.008 inches). If we were to simply cut around the exact boundary of each trace, we will cut into the area of the trace by half the kerf width, r, i.e. the cutter radius. For boards with small feature sizes, as are now common, this can significantly reduce the width of a trace, even to the point where its electrical integrity is compromised.

To avoid this, we need to 'fatten' the geometry of all traces by r before computing toolpaths. This is a typical 2D offset operation, and it comes with the usual problem of 2D offset: self-intersections of the offset boundary. See Figure 3 (right) for an example. This is again a mostly solved geometric problem, but standard analytic algorithms (see e.g. [10]) can be tedious to implement correctly.

We can implement cutter-radius offset, including correct handling of self-intersections of the offset boundary, with a simple tweak to Algorithm 3. Recall that traces are composed of flash and stroke primitives, so that the geometry of a trace is the union of the geometry of the constituent primitives. Before constructing the geometry for each primitive, we simply fatten the associated aperture by r[8,9]. Self-intersections result in overlapping geometry for primitives in the trace, which when rendered will not be visible because the involved primitives are the same color.

While we can safely ignore *self*-intersections of the offset geometries, we still must contend with intersections between adjacent fattened traces. Such cases are uncommon except on extremely dense boards, and it is not entirely clear what the correct behavior should be when the distance between two traces is less than the cutter diameter. If we adopt the notion that the best we can do is to split the distance between the traces, then the extension we describe next, Voronoi toolpaths, can inherently find such splits, in addition to its other properties.

D. Voronoi Toolpaths

Instead of cutting a loop around the boundary of each trace, we can compute a generalized Voronoi diagram with the traces as sites[10], and then cut along the boundaries separating all the Voronoi regions. Figure 4 shows a PCB layout and the associated generalized Voronoi diagram. While the resulting board will look different than the original design, it will still have the same electrical connectivity, and it will still admit all the components to be soldered in the same places as in the original design.

To compute the generalized Voronoi diagram, we adopt the intuitive and relatively simple GPU-based method proposed by Hoff et al. in [1]. Basically, we construct a planar generalized Voronoi diagram by building a 3D geometry g_s with a unique random color for each Voronoi site s and concurrently view all g_s with an orthogonal projection. We construct the g_s in such a way that the image we see in this view *is* (a discrete approximation to) the generalized Voronoi diagram.

Construction of the g_s according to [1] is easy. For a point site s_p, g_{s_p} is a right circular cone with apex at s_p and extending back along the z-axis (i.e. away from the camera). For a simple polygon site s_o, g_{s_o} consists of two pieces: the

[8]We actually compute a polygonal approximation to the fattened aperture.

[9]The technique can be generalized to arbitrary polygonal geometry by adding an additional 'band' of width r around each polygon, explicitly ignoring self-intersections in the band and between the band and the polygon.

[10]Actually we use the approximate core medial axes of the traces as sites, as described below.

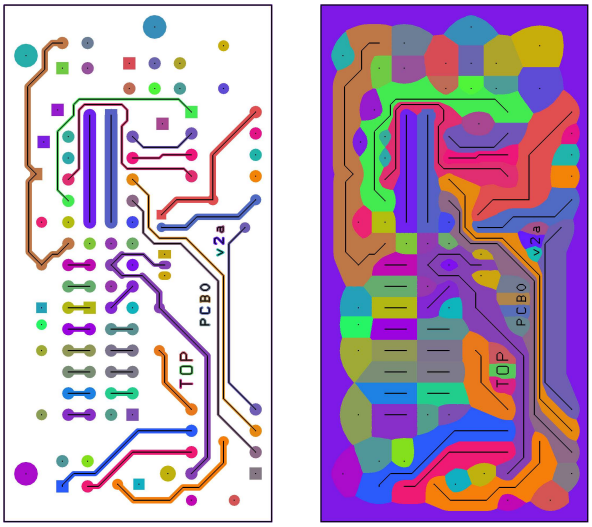

Fig. 4. A PCB layout (left) and (a discrete approximation to) the corresponding generalized Voronoi diagram (right), computed using the GPU with an algorithm based on [1]. Each Voronoi region corresponds to one of the traces in the layout: the Voronoi sites are the approximate core medial axes of the traces (shown in black).

planar geometry of s_o, and a 'curtain' attached around the perimeter of s_o and extending back along the z-axis at a $45°$ angle.

Since most traces are relatively narrow and most pads are relatively small, we find that in practice we can make the additional simplification of considering the Voronoi sites to be not the actual polygon shapes of the traces, but rather the core medial axes of these polygons. This is convenient in practice because we can extract approximate core medial axes directly from the Gerber as the centers of the flashes plus the segments connecting the endpoints of the strokes. Figure 5 shows an example of such a geometry.

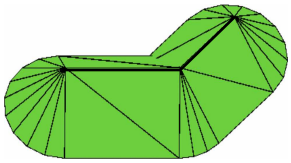

Fig. 5. Oblique view of 3D geometry used to help compute the Voronoi region associated with the core medial axis (shown with a thick line) of a trace. For the actual Voronoi computation, this geometry would be viewed with an orthogonal projection directly from above, and would not be rendered with shading or with the black triangle edges.

Hoff et al.'s Voronoi diagram algorithm computes a discrete approximation which is only correct up to the rendering resolution ϵ, and its time bounds are likely less than optimal (refer to [1] for a more complete analysis of time complexity and correctness). However, we find that the algorithm works in practice, and, importantly, it was not difficult to implement correctly within the framework of our application. Note, for example, that Algorithm 3 can be re-used to extract the cutter toolpath from the discrete approximation of the generalized Voronoi diagram (which is just a raster with each Voronoi region shown in a different color).

As far as we know, this Voronoi toolpath method for PCB mechanical etch is novel. It offers several advantages over the current industry practice of cutting the outline of each trace:

- The Voronoi method results in a single cut between two adjacent traces, whereas the outline method produces two cuts in such cases[11], even when those cuts may overlap. Since the involved copper features can be tiny, reducing the number of overlapping cuts can improve feature quality when real-world issues such as flexibility in the cutter and play in the machine are considered.
- The Voronoi method often produces a toolpath whose overall length is shorter than the corresponding outline toolpath. Relative to the standard outline toolpath method, the Voronoi method resulted in an average 37% decrease in total toolpath length across 7 test boards. This result is discussed further below in Section IV.
- The Voronoi method results in traces which are, in some sense, the widest possible, thus maximizing current carrying capacity. As discussed below in Section IV, the Voronoi method resulted in an average 443% increase in approximate trace width across 7 test boards.

1) Limiting the Voronoi Diagram: One downside to the Voronoi method is that it may increase the parasitic capacitance and inductance between traces because it often results in a longer section of close proximity along the boundaries of adjacent traces. This may be alleviated somewhat by limiting the extents of the Voronoi regions.

We can exploit the structure of the GPU-based Voronoi algorithm to provide a simple limit to the extent of the Voronoi regions. To limit the extent to δ we simply clip the g_s geometries so that they extend only δ back along the z-axis. Figure 6 shows an example where such clipping has been applied.

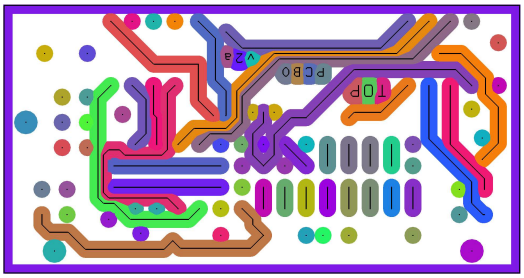

Fig. 6. The generalized Voronoi diagram corresponding to the PCB layout in Figure 4 (left), with the extent of the Voronoi regions limited to 0.04 inches from the core medial axis of the corresponding site. The core medial axes are shown in black.

This is yet another example of how we can use a simple and intuitive method to achieve what might otherwise be a tedious geometric computation.

[11] And discussion with a user of the proprietary commercial software 'CircuitCAM' indicate that even more than two overlapping cuts are often generated [4].

2) Ensuring Minimum Trace Widths: Since we compute the Voronoi diagram of the core medial axes of the traces, it may occur that the Voronoi region for a trace does not fully include all of the original geometry of the trace. Consider a wide trace closely adjacent and parallel to a narrow one—after computing the Voronoi diagram of their core medial axes, the boundary between the resulting Voronoi regions will be equidistant from the core medial axes, and thus the trace which was originally wide might be made more narrow.

We can avoid this with a simple modification to the GPU-based Voronoi algorithm. Simply render the original flat geometry of the traces above (i.e. nearer to the viewer) the Voronoi geometries g_s. If we use the same color for both the flat and the Voronoi geometry for each trace, then the image the viewer sees will appear to be the Voronoi diagram with each Voronoi region constrained to include at least the full geometry of the corresponding trace.

IV. Experimental Results

We ran VIsolate on 7 PCB designs, with the results shown in Table V. Dataset 0, a board for a robot in our lab, is shown in Figures 1,4, and 6. Dataset 1 is the back side of that board, and the rest of the datasets are example boards from the internet.

The average toolpath length savings for these boards was about 37%, and the average approximate trace width[12] increase was about 443%. For Dataset 0 in particular, the Voronoi toolpath (44.9 in) took about 60 minutes to cut on our Sherline CNC mini-mill, whereas the outline toolpath (63.6 in) took about 87 minutes to cut.

V. Summary and Future Work

We have presented VIsolate, a new system for computing toolpaths for PCB isolation routing, including a novel Voronoi-based toolpath algorithm with some advantages over the current industry practice. Along the way we illustrated several intuitive, simple, and re-usable algorithms—polygon overlap detection, 2D offset, constrained generalized Voronoi diagram—which all leverage the 3D GPU for straightforward implementation.

In the future, we may explore the possibility of selectively using Voronoi regions for only a subset of traces.

Also, we would like to better understand how to predict whether the Voronoi toolpath will be shorter than the outline toolpath and by how much.

Currently our algorithms work only for basic PCB flash and stroke geometries. In the future we may extend them to also handle PCB features such as ground planes and general polygonal traces. So far we have only tested VIsolate on fairly small boards ($\sim$ 100 traces, $\sim 10\text{in}^2$), in the future we may test it on larger designs and address any scaling issues.

VIsolate is implemented in Java™ and a demo applet is available at

`http://www.mit.edu/~vona/Visolate-info.html`

Acknowledgments

We are grateful to Joe Edelman and Keith Kotay for constructive discussions about this project. Keith also supplied the board design shown in the figures.

References

[1] K. E. Hoff, III, J. Keyser, M. Lin, D. Manocha, and T. Culver, "Fast computation of generalized Voronoi diagrams using graphics hardware," in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co., 1999, pp. 277–286.

[2] C. M. Gold, P. R. Remmele, and T. Roos, "Voronoi diagrams of line segments made easy," in *Proceedings of CCCG'95*, 1995, pp. 223–228.

[3] Personal communication with Jeff Hoff, an undergraduate student and LPKF CircuitCAM user at the MIT Computer Science and Artificial Intelligence Laboratory.

[4] Personal communication with Mike Garcia-Webb, a graduate student and LPKF CircuitCAM user at the MIT Bioinstrumentation Laboratory.

[5] O. Takahashi and R. J. Schilling, "Motion planning in a plane using generalized Voronoi diagrams," *IEEE Transactions on Robotics and Automation*, vol. 5, no. 2, pp. 143–150, April 1989.

[6] J. Lengyel, M. Reichert, B. R. Donald, and D. P. Greenberg, "Real-time robot motion planning using rasterizing computer graphics hardware," in *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*. ACM Press, 1990, pp. 327–335.

[7] Http://www.lems.brown.edu/vision/Presentations/Wolter/figs3.html.

[8] M. Sayrafiezadeh, "The birthday problem revisited," *Mathematics Magazine*, no. 67, pp. 220–223, 1994.

[9] P. Selinger, "Potrace: a polygon-based tracing algorithm," September 2003, unpublished, available at http://potrace.sourceforge.net/potrace.pdf.

[10] N. M. Patrikalakis and T. Maekawa, *Shape Interrogation for Computer Aided Design and Manufacturing*. Springer-Verlag New York, Inc., 2002.

TABLE V

Results for 7 PCB designs. Each dataset corresponds to a different Gerber input. **Overlap (s)** is the runtime for Algorithm 2 on a commodity workstation; **Toolpaths (s)** is the runtime for Algorithm 3; **Outline (in)** and **Voronoi (in)** are the total lengths for outline and Voronoi toolpaths, respectively; the **width** columns give approximate trace widths as described in the text.

Dataset	Traces	Area (in^2)	Overlap (s)	Toolpaths (s)	Outline (in)	Voronoi (in)	Length Savings	Normal Width (in)	Voronoi Width (in)	Width Increase
0	87	3.0	3.9	11.0	63.6	44.9	42%	0.04	0.14	278%
1	86	12.1	8.7	30.6	112.8	85.4	32%	0.07	0.25	271%
2	83	5.0	14.8	14.5	51.1	48.3	6%	0.03	0.30	806%
3	83	3.0	4.4	10.3	60.9	42.0	45%	0.03	0.11	238%
4	38	4.1	7.7	11.7	57.1	37.7	51%	0.03	0.31	1088%
5	21	0.6	2.1	2.8	15.0	10.4	44%	0.04	0.09	156%
6	13	0.6	1.9	2.7	12.4	8.9	38%	0.04	0.14	267%
avg	59	4.1	6.2	11.9	53.3	39.7	37%	0.04	0.19	443%

[12]Trace widths were approximated (only) for traces containing at least one stroke by dividing the approximate area of the trace by the sum of the lengths of the strokes forming the trace.