ISSN 1392–124X (print), ISSN 2335–884X (online) INFORMATION TECHNOLOGY AND CONTROL, 2016, T. 45, Nr. 3

A Simple Centerline Extraction Approach for 2D Polygons

Aleksas Riškus, Armantas Ostreika, Antanas Lenkevičius, Vytautas Bukšnaitis

Department of Multimedia Engineering, Kaunas University of Technology
Studentų St. 50, LT−51368 Kaunas, Lithuania
e-mail: armantas.ostreika@ktu.lt

crossref http://dx.doi.org/10.5755/j01.itc.45.3.12964

Abstract. This paper describes a new two-task approach for extracting the centerline from simple 2D polygons. An algorithm of the first task, which generates a set of points for future centerline, is presented. The idea of the algorithm is to use the polygon hatching by parallel lines and creating paths from its middle points in different polygon rotations. The centerline is derived from the intersection points between these paths. The algorithm was developed for printed circuit board insulation process and its execution time for a few hundred polygons is less than one second. The algorithm is easy to understand and implement.

Keywords: Centerline; extraction; polygon; hatching; pcb; multimedia.

1. Introduction

The CAD software prepares the specific layout data for the printed circuit board (PCB) prototypes. For this, instead of chemical etching technology a milling technology is widely used in insulating process. Insulate milling is the process of removing not needed copper from coated base material by surface milling according to calculated insulation channels around interconnected tracks, pads and other areas. The mandatory primary tool is used for the primary insulation channel around all layout copper. Typically, a universal milling tool with a diameter of about 0.2 mm is used. To improve soldering properties, generate a minimum distance or remove other undesirable residual copper, other additional insulation tracks or free areas can be used [17, 19]. Such PCBs are widely used in the medical, multimedia, computer and other equipment.

Figure 1 shows a small PCB segment after insulation procedure. Pads and tracks are shown in grey color, blue paths are insulating channels.

Figure 2 shows the same PCB segment but insulating channels are shown in true width. It is obviously seen that in some places the copper is not removed (white areas between grey pads and tracks). The reason for this is that there is not enough space for the primary milling tool. To process these remaining areas either a smaller tool with rather low durability or laser beam is used. But in both cases a centerline for the smaller tool (or laser beam) motion needs to be generated in each free area.

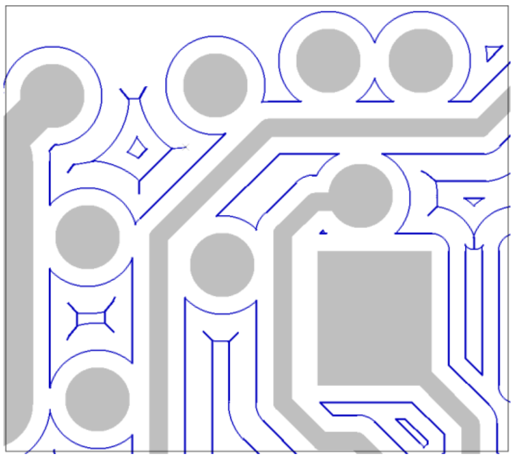
Figure 1. Illustration of the insulation procedure

The primary insulation channels can be generated by using the sequence of simple polygon operations "**union** + **oversize** by *toolWidth*/2 + **outline**". Invoking additional polygon operations **subtract** and **intersection** allows to prepare polygons for centerlines extraction (twelve black polygons in Figure 3).

The fast insulation procedure allows a user to work in "semi-interactive" mode – to analyze results with a different kit of milling tools and select a more suitable variant. Of course, the desired layout in parallel is checked by "*Design Rule Check*" module, but the final estimation can be done only after the production of assay. Usually, a few hundred centerline polygons can be generated in the small PCB. It means that centerline extraction time for one centerline polygon can be measured only in milliseconds.

The centerline is also the key in many practical applications, such as path planning in robotic navigation, virtual endoscopy and road detection.

The most common approaches for centerline extraction are based on topological thinning [5, 10], Euclidean distance transform (distance mapping) [2, 7, 11, 12], simulation of the wave propagation [3, 8] and Voronoi diagrams [4, 13].

The basic idea of thinning is iteratively to peel off the boundary voxels layer by layer (the procedure is very like of onion peeling). It takes very long time, because each time needs to ensure that its topological connection is not changed.

In the distance transform approach, the minimum distance from inner voxels to surface voxels and end point of the navigation path are calculated. The point will be on the navigation path if its minimum distance to the surface is the largest one among the point set, in which the distances to the start point are equal.

In the wave propagation technique, a front propagates from one node with specified cost function until the other node is reached and then a backtracking procedure is used to compute the desired centerline.

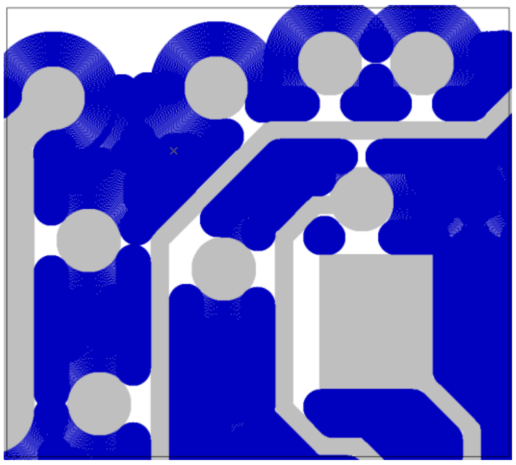

Figure 2. Illustration of the copper areas

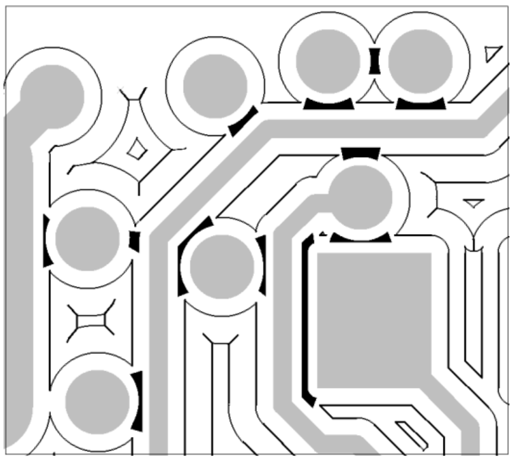

Figure 3. Illustration of the centerline polygons

The Voronoi diagram of a set of points is the set of edges that are equidistant from the two nearest points in the set. The edges of the Voronoi diagram of the vertices of a polygon that are interior to the polygon compose the Voronoi skeleton.

A more detailed analysis of existing publications on centerline extraction can be found in [1, 6].

The main problem of all mentioned approaches is speed - they are time-consuming. Typical running times of these methods range between tens of seconds and hours per dataset, which is too slow for interactive application with thousand objects. The speed problem can partially be solved by using the graphics processing unit (GPU) [9].

The centerline extracting method proposed in this paper is not related with any of the existing approaches and was developed for simple 2D polygons. The polygon consists of line and arc segments, has only one boundary and it doesn't cross over itself. To calculate a set of suitable points for future centerline takes only a few milliseconds for one polygon.

The paper is organized as follows. Section 2 explains the main tasks and steps of the centerline algorithm. Section 3 analyzes some implementation aspects of the proposed algorithm. Experimental experience and results are provided in section 4. Section 5 proposes a short discussion about extension of the proposed approach. Finally, some concluding remarks end the paper.

2. Centerline algorithm

The first task of the proposed centerline extraction approach is to calculate inside a polygon a set of points as more as possible compatible to the future centerline. The main algorithm steps of the first task are:

Step 1. **Hatching.** Create a set of parallel hatch lines with fixed distance *hatching_grid*. This process is illustrated in Figure 4.

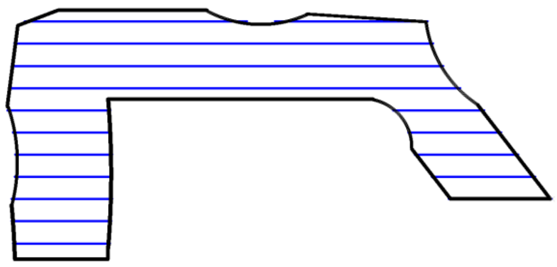

Figure 4. Polygon hatch lines

Step 2. **Paths composition.** Create a set of the simple paths from line segments only by using neighbors hatch lines middle points. If the distance between the neighbors middle points exceed the specified *dLimit* value, the composition of the path is stopped and the current hatch lines middle point becomes a start point of the new path. The *dLimit* parameter prevents from cutting of composed path with the polygon contour. This process is illustrated in Figure 5 (three paths are composed).

Step 3. Rotate the polygon by specified *rotation_step* angle and repeat **Step 1–Step 2** until the last rotation angle is still less the 180 degree. Figure 6 shows hatch lines and corresponding paths when the polygon was rotated by 45^0.

Step 4. **Intersection points calculation.** Calculate intersection points between all paths composed in **Step 2**. This process is illustrated in Figure 7.

The second task of the proposed centerline extraction approach is to draw a centerline based on the results of the first task. The input data are: polygon start point, polygon end point and the set of intermediate intersection points from the first task. But this task is out of the scope of this paper.

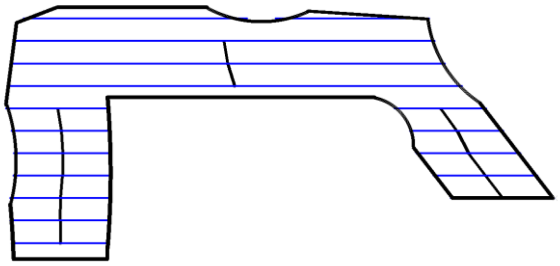

Figure 5. Illustration of the paths composition (composed 3 paths)

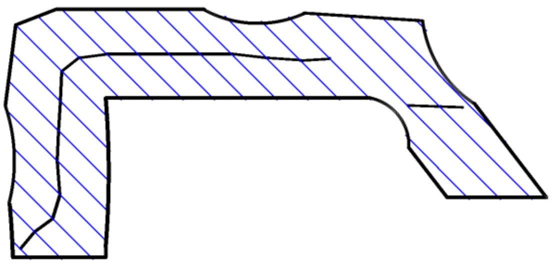

Figure 6. Hatch lines and composed paths with 45^0

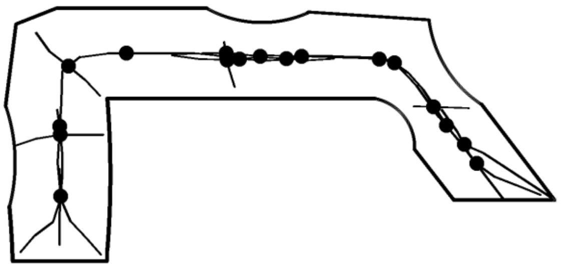

Figure 7. Results after 0, 45, 90 and 135 degree rotations have been performed (*rotation_step* = 45)

3. Some implementation notes

1. Hatching. At least two hatching strategies can be used.

1) *Hatching of rotated polygon*:

– rotate the polygon by angle

$\boldsymbol{\theta}$ += rotation_step;

– generate lines parallel to **x**-axis (or **y**-axis) by using the *hatching_grid*;

– use transformation and store resulting hatch lines in real coordinates according to the following formula:

$$xt = cos(\theta) * (xo - xr) - sin(\theta) * (yo - yr) + xr,$$

$$yt = sin(\theta) * (xo - xr) + cos(\theta) * (yo - yr) + yr, \quad (1)$$

where $(\boldsymbol{xo, yo})$ is the original point, $\boldsymbol{\theta}$ is the rotation angle, $(\boldsymbol{xr, yr})$ is polygon rotation point (e.g. rotation around polygon center point or around 0/0 point) and $(\boldsymbol{xt, yt})$ is already transformed $(\boldsymbol{xo, yo})$ point, respectively. The formula (1) implements counter-clockwise rotation (CCW).

2) *Hatching of initial polygon*. The polygon is not rotated at all but hatch lines are generated with current rotation angle. The distance between hatch lines is specified by the same *hatching_grid* value.

The complexity of the hatching step depends on how the hatch line middle point is calculated. A standard case is *Line and Polygon intersection* - calculate hatch line intersection points with the polygon contour and take a center of each resulting line segment between neighbor's intersection points. Indeed, these are simple *line-line* and *line-arc* intersection operations. The complexity of hatching step in the worst case is *O(NM)*. Here *N* is the number of the hatch lines and *M* is the total number of line and circular arc segments on the polygon contour, respectively. Additional sorting of polygon contour segments can decrease this complexity.

Note. If the length of hatch line is defined by the bounding-box of the polygon, then a few resulting hatch line segments can be composed (e.g. two segments for polygon of shape "**V**").

Another possibility is to use graphical operations of programming languages. For example, Java has class **Area** with intersection and other operations.

2. Paths composition. This step is related to the time consuming 4-th step of the proposed algorithm (intersection points calculation). To reduce the complexity of Step 3, it is better to have more paths represented by two-point line instead of the paths with many intermediate points. This can be achieved by adding some additional conditions. The goal is to interrupt the creation of current path and start a new path. There are two possible solutions.

1) *Path with fixed length.* When a number of points in the path reaches the specified limit, then the creation of the path is stopped. The path stays with all intermediate points, but shorter paths are better than

less but longer paths. The complexity of the paths composition is *O(NK)*, where *N* is the number of the hatch lines and *K* is the number of composed paths.

2) *Check perpendicular distance on the fly:*

- each time after adding a new point in the current path calculate perpendicular distance of each already added point to line (*Start*-point, *Last-added*-point);
- if any distance exceeds the given error tolerance, remove the last added point and stop the path creation;
- remove all intermediate points from *Start*-point until *End*-point (convert the path into two-point line).

The complexity of the paths composition in this case is *O(NKL)*, where *N* is the number of the hatch lines, *K* is the number of composed paths and *L* is the number of points in the longest path, respectively.

3. Calculation of paths intersection points.

The constraint "path with two points only" in Step 2 allows the slow function "two paths intersection points" to be transformed into fast function "two lines intersection point".

By using the smaller *rotation_step* (e.g. 10 degree) there can be composed some almost parallel and intersected paths in the same polygon region. This fact will give a few intersection points with similar coordinates. These redundant intersection points can be eliminated by the introduction of additional parameter *minAngle*: if the angle between two paths at their intersection point is less than the *minAngle*, then the intersection point is skipped. Experiments showed that depending on the complexity of polygon shape *minAngle* value can vary from 5 to 15 degrees.

The complexity of the step depends on the number of paths and the number of segments in each path and in the worst case is *O(PK)*. Here *P* is the number of the composed paths and *K* is the number of line segments in the longest path, respectively.

Using *mas points* instead of *middle points*

The discussed path composition was based on the usage of hatch line's middle point. When channel (polygon) contour is roof, then another path composition strategy can be used:

- select an area, restricted by two neighbors hatch lines and two corresponding polygon contour segments between the hatch lines, respectively;
- calculate a mass point of the selected area.

The mass point (Cx, Cy), also known as the "centroid", can be calculated as follows [18]:

$$Cx = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

$$Cy = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i) \quad (2)$$

$$A = \tfrac{1}{2}\textstyle\sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)\,.$$

In the formula (2), the vertices are numbered in the order of their occurrence along the polygon's perimeter. The polygon is closed (the first point is repeated at the end). But the formula (2) is valid only for polygon from line segments. The insulate channel mainly consists of circular arc segments. Two solutions are possible:

1. approximate each arc segment by a set of line segments;
2. modify the formula (2) to fit for both line and arc segments. Figure 11 in **Appendix** shows the main piece of C++ code from mass point (Cx, Cy) calculation function, successfully used in [17].

Some data formats, used in computer-aided design, besides the circular arcs support Bezier curves as well. Under the necessity the Bezier curve easily can be approximated by circular arcs or vice versa [14].

4. Experimental experience

There are two important parameters in the proposed centerline algorithm – *hatching_grid* and *rotation_step*. The *hatching_grid* directly depends on the width of polygon the centerline to be generated. It should be selected with the ability that a few hatch lines will be done in the narrowest polygon place. On the other hand, too small *hatching_grid* impacts on hatching time. The optimal grid is

hatching_grid = minWidth / 3

where the *minWidth* is the narrowest place of the centerline polygon. Centerline points in Figure 7 were generated with *hatching_grid* 0.2 mm and *minWidth* 0.58 mm.

The *rotation_step* impacts on the number of composed paths (Step 2 of the algorithm). A smaller *rotation_step* creates more paths which in turn increases the number of their intersection points (Step 3 of the algorithm). Optimal *rotation_step* is 45 and 30 degrees. Figure 8 shows resulting intersection points, generated with *hatching_grid* 0.3 mm and *rotation_step* 30 degree.

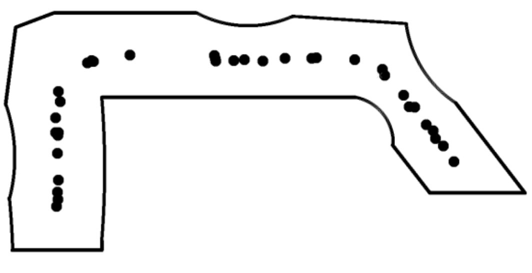

Figure 8. Paths intersection points: *rotation_step* = 30, *hatching_grid* = 0.3 mm

Figure 9 shows the results for the layout segment with 12 polygons from Figure 3.

As it was mentioned before, the algorithm must provide information for centerline in short time. For example, the small PCB, whose segment was used for illustrations in Figure 1 - Figure 3, has 1144 pads, 1767 tracks and 508 polygons were prepared for centerline extraction. Table 1 shows how total execution time depends on the number of polygons, hatching grid and rotation step. Some properties (total number of hatch lines, paths and intersection points) are provided in the table as well. The proposed algorithm was implemented in C++ and was run on a Pentium IV PC, 3.47GHz CPU and 8GB RAM.

Additionaly, was tested a colon, used as the benchmark in [2, 15]. The centerline algorithm in [2] took 5 minutes, the centerline algorithm in [15] took 16 seconds, respectively. The 2D shape of the colon and our result is shown in Figure 9.

Centerline for road map segment with algorithm in [16] was generated in 468 seconds. The result of the proposed algorithm is shown in Figure 10.

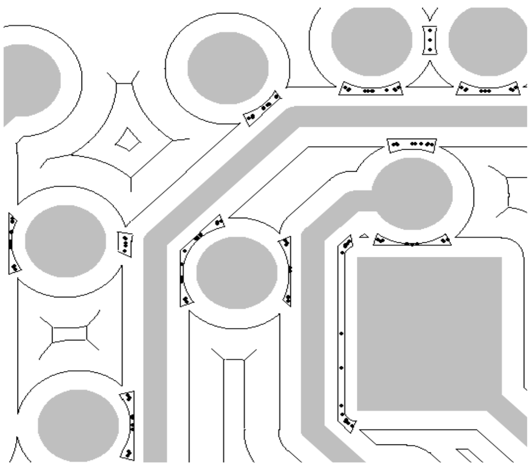
Figure 9. Paths intersection points: *rotation_step* = 45, *hatching_grid* = 0.005 mm, *minWidth = 0,009*

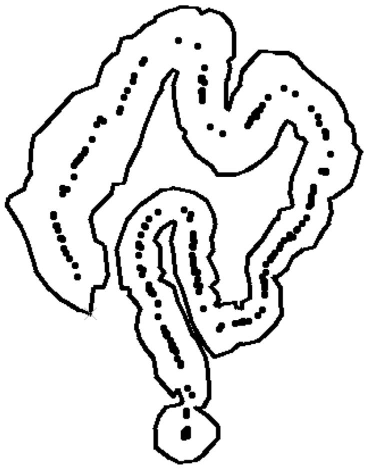
Figure 9. Paths intersection points: *rotation_step* = 25, *hatching_grid* = 0.1 mm. Calculation time = 0.047 sec

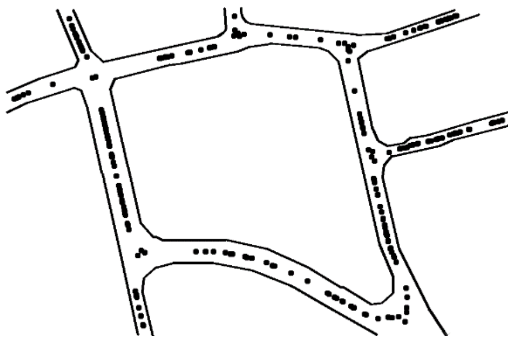
Figure 10. Paths intersection points: *rotation_step* = 30, *hatching_grid* = 0.02 mm. Calculation time = 0.23 sec

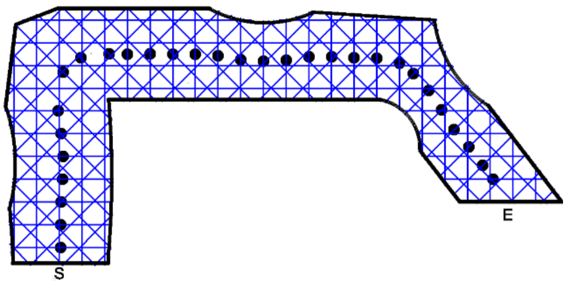

Figure 10. Hatch lines and centerline points

5. Discussion

The algorithm calculates a set of points in a polygon and these points late will be used to construct a centerline between specified start and end points, respectively. The main data source is polygon hatch lines, created in a few different polygon rotations. Perhaps there might be other solutions how to apply these hatch lines. By visual inspection of the hatch lines it is clearly seen, that some middle points of the hatch lines already correspond with future centerline points – one only needs to take suitable hatch lines from every polygon rotation and skip not needed hatch lines (e.g., too long and too short). Figure 10 shows hatch lines, created with 45 degree rotation step. The middle points of the most suitable hatch lines are marked with dots. 'S' and 'E' mean centerline start and end points, respectively.

Before doing the hatching, the polygon can be undersized ("shrank") by some value. Of course, this value must be less than 2* *minWidth*, otherwise the polygon will be split in a few separate sub-polygons.

6. Concluding remarks

In this paper, the new centerline extraction approach for simple 2D polygons (with non-intersecting lines but with arc segments) is discussed. The approach is not related with any of the existing approaches. It was developed for one of the PCB design tasks - insulate milling, but can be successfully used for other tasks on 2D shapes as well. The centerline points calculation algorithm is fast and simple in both understanding and implementation sense.

Experiments with real PCB of multimedia devices proved the sufficient approach speed and quality. Calculation of the set of points for centerline takes up to one second for 500 polygons. Empirical results are presented in Table 1. Possible implementation variants and their complexity were analyzed for every step of the algorithm.

Finally, the discussion about the extension of the proposed approach was started.

Table 1. Execution speed

Number of polygons	*hatching_grid* (mm)	*rotation_step* (degree)	Number of hatch lines	Number of paths	Number of intersection points	Calculation time (sec)
12	0.005	45	764	132	173	0.024
12	0.007	45	545	116	164	0.019
12	0.005	30	815	195	358	0.036
100	0.005	45	5908	1018	1367	0.088
100	0.007	45	4222	925	1277	0.070
100	0.005	30	6301	1548	2901	0.156
508	0.005	45	34229	5492	7500	0.565
508	0.007	45	24451	4857	6787	0.374
508	0.005	30	36703	8132	20906	1.042

References

[1] **O.K.-C. Au, C.-L. Tai, H.-K. Chu, D. Cohen–Or, T.-Y. Lee.** Skeleton Extraction by Mesh Contraction. *ACM Transactions on Graphics*, 2008, Vol. 27, No. 3, -10.

[2] **I. Bitter, M. Sato, M. Bender, K. McDonnel, A. Kaufman, M. Wan.** CEASAR: A Smooth, Accurate and Robust Centerline Extraction Algorithm. In: *Proc. IEEE Visualization*, 2000, pp. 45-52.

[3] **H. Blum**. A Transformation for Extracting NewDescriptors of Shape. *Models for the perception of speech and visual form, MIT Press*, 1967, 362–380.

[4] **T. K. Dey and W. Zhao**. Approximate medial axis as a Voronoi subcomplex. *Computer-Aided Design*, 2004, Vol. 36, No. 2, 195–202.

[5] **M. Ding, R. Tong, S. Liao, J. Dong**. An extension to 3D topological thinning method based on LUT for colon centerline extraction. *Computer Methods and Programs in Biomedicine*, 2009, Vol. 94, No. 1, 39–47.

[6] **D. Jiménez, D. Labate, I.A. Kakadiaris, M. Papadakis**. Improved Automatic Centerline Tracing for Dendritic and Axonal Structures. *Neuroinformatics*, 2015, Vol. 13, No. 2, 227-244.

[7] **S. Ferchichi, S. Wang, S. Grira**. New algorithm to extract centerline of 2D objects based on clustering. In*: The International Conference on Image Analysis and Recognition*, 2007, Vol. 4633, pp. 364–374.

[8] **M.S. Hassouna, A.A. Farag**. Robust centerline extraction framework using level sets. In: *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2005, pp. 458–465.

[9] **B. Liu, A.C. Telea, J. B. T. M. Roerdink, G. J. Clapworthy, D. Williams, P. Yang, F. Dong, V. Codreanu, A. Chiarini**. Parallel Centerline Extraction on the GPU. *Computers & Graphics*, 2014, Vol. 41, 72–83.

[10] **M. C. Ma, M. Sonka**. A fully parallel 3d thinning algorithm and its applications. *Computer Vision and Image Understanding*, 1996, Vol. 64, No. 3, 420–433.

[11] **D.S. Paik, C.F. Beaulieu, R.B. Jeffery, G.D. Rubin, S. Napel**. Automated Flight Path Planning for Virtual Endoscopy. *Medical Physics*, 1998, Vol.25, No. 5, 629-637.

[12] **Y. Peng, J. Shi**. A new fast algorithm for extracting center path. In: *Proceedings of SPIE-The International Society for Optical Engineering*, 2003, Vol. 5286, Issue 2, pp. 735-740.

[13] **O. Sharma, F. Anton, D. Mioc**. Level Sets and Voronoi based Feature Extraction from any Imagery. In: *GEOProcessing 2012, The Fourth International Conference on Advanced Geographic Information Systems, Applications, and Services*, 2012, pp. 89-97.

[14] **A. Riškus, G. Liutkus.** An Improved Algorithm for the Approximation of a Cubic Bezier Curve and its Application for Approximating Quadratic Bezier Curve. *Information Technology and Control*, 2013, Vol. 42, No. 4, 303-308.

[15] **M. Wan, F. Dachille, A. Kaufman**. Distance-field based skeletons for virtual navigation. In: *Proc. IEEE Visualization*, 2001, pp. 239-245.

[16] **P. Yuqing, C. Wenchao, S. Yehua.** The shortest hypotenuse-based centerline generation algorithm. *IJCSNS International Journal of Computer Science and Network Security*, 2009, Vol. 9, No. 8, 155–159.

[17] CircuitCAM. Available *from WWW*: *< http://www.circuitcam.com/ >*

[18] **P. Bourke**. Calculating the area and centroid of a polygon. Available *from WWW*: *http://paulbourke.net/geometry/polygonmesh*>

[19] Milling and Drilling of Printed Circuit Boards. Available from WWW: http://www.lpkf.com

Received August 2015.

Appendix

```
VERTEX *pv1 = pvStart;
double A = 0.0;
do {
    double r2, angle;
    VERTEX *pv2 = pv1->pNext;
    double x1 = pv1->x;
    double y1 = pv1->y;    // (x1, y1) – start point of the line/arc
    double x2 = pv2->x;
    double y2 = pv2->y;    // (x2, y2) – end point of the line/arc
    if (pv1->pArc) {  // Arc segment
        double cpx = pv1->pArc->x;
        double cpy = pv1->pArc->y;  // (cpx, cpy) – center point of the arc
        double t1 = atan2(y1 - cpy, x1 - cpx) ;
        double t2 = atan2(y2 - cpy, x2 - cpx) ;
        if (pv1->fCCW) {
            while (t2 < t1) { t2 += 2*PI ; }
        } else {
            while (t1 < t2) { t1 += 2*PI ; }
        }
        angle = t2 - t1 ;
        r2 = (cpx - x1)*(cpx - x1) + (cpy - y1)*(cpy - y1);
        *Cx += (cpx*r2*angle - cpx*((y2 + cpy)*(x2 - cpx) - (y1 + cpy)*(x1 - cpx)) +
              (y2 - y1)*((y2 - cpy)*(2*y2 + y1) + (y1 - cpy)*(y2 + 2*y1))/3.0)/2.0 ;
        *Cy += (cpy*r2*angle + cpy*((y2 - cpy)*(x2 - cpx) - (y1 - cpy)*(x1 - cpx)) +
              cpx*(y2 - y1)*(y2 + y1) - 2*(x2 - x1)*((x2 - cpx)*(x2 - cpx) +
              (x2 - cpx)*(x1 - cpx) + (x1 - cpx)*(x1 - cpx))/3.0)/2.0 ;
        A += r2*angle + cpx*(y2 - y1) - cpy*(x2 - x1);
    } else {    // Line segment
        *Cx += (x1 - x2)*(x1*(2*y1 + y2) + x2*(y1 +2*y2))/6.0 ;
        *Cy += (y2 - y1)*(x1*(2*y1 + y2) + x2*(y1 +2*y2))/6.0 ;
        A += x1 * y2 - y1 * x2;
    }
    pv1 = pv2;
} while (pv1 != pvStart); // Start vertex is duplicated at the end
A /= 2.0;
*Cx /= A;
*Cy /= A;
```

Figure 11. A main fragment from the mass point calculation function