Polygonization of connected areas in binary images

Nedim Šišić
Faculty of Mathematics, Natural Sciences and Information Technologies
University of Primorska
Koper, Slovenija
89202005@student.upr.si

***Abstract*—** In image processing, vectorization is the conversion of raster images to vector images. Polygonization, a type of vectorization, finds polygonal approximations of raster images. Most existing vectorization methods process entire images, thus performing unnecessary work if we are only interested in a single part of an image. In this paper, we propose an approach to polygonizing binary images, where only a single connected component of interest is polygonized. Our method reduces the time and space requirements in this problem setting compared to existing methods.

Keywords—*vectorization, polygonization, image manipulation, connected component*

1. Introduction

Computer images are most commonly represented as raster images or vector images. A raster image is rectangular grids of pixels, while a vector image is composed of geometric primitives such as points, lines, shapes, etc. Both formats have different advantages and disadvantages when it comes to storing and processing images. The raster format is more suitable for very complex images, such as photographs. On the other hand, unlike raster images, vector images can carry semantic information and allow for scaling without loss of resolution. In image processing, vectorization is the conversion of raster images to vector images. A special kind of vectorization is polygonization, which deals with finding polygonal approximations of a raster image. The resulting polygons can later be used in different image processing applications, such as image analysis [1], image matching [2][3][4], image and video retrieval [5], and image registration [6].

The approaches used behind existing vectorization methods depend on the type of raster images they intend to process. Some images, such as the ones that contain photographs, are noisy and contain imprecise region borders, so they are harder to vectorize. They are usually first segmented into different regions based on approximate color similarity, each region then being represented using uniform or transitional colors. Images such as pixel art and line drawings often have less noise and contain much more precise region borders. As such, they allow for the use of border or contour tracing. A review of vectorization techniques is given by Kopf and Lischinski [7] and Hoshyari et al. [8] for pixel art, and by Chen et al. [9] for line drawings. The methods that we will present this paper are mostly intended for images containing precise borders. An example of such an image, and one of its polygonal approximations, are given in Fig. 1.

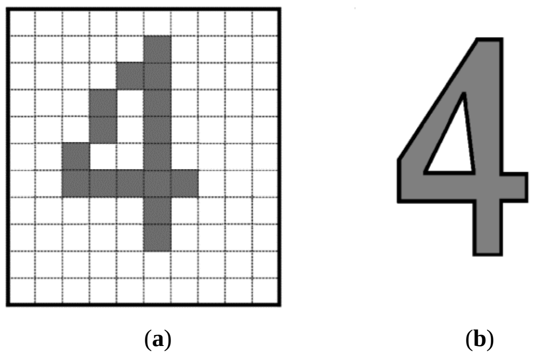

Fig. 1 (**a**) The binary image; (**b**) a polygonal approximation of (**a**).

To our knowledge, all existing vectorization methods vectorize entire images, which can be unnecessary if we are only interested in vectorizing a single part of an image. By restrictinging the scope of vectorization, we could decrease its time and memory requirements, especially if the area of interest is much smaller than the entire image. In this paper, we propose a novel approach to polygonizing connected areas in binary images. Binary images consist of pixels that can have one of exactly two colors, and as such are suitable for storing shapes with precise outlines. We present the *Polygonize* method, which finds a polygonal approximation of a single part of an image, without the need to process the rest of the image. We then present the *Reduce* method as a preprocessing step to *Polygonize*, which further reduces the space complexity of solving the problem. We analyze the time and space complexities of the methods. The polygonization methods we present can further be generalized to color images.

2. Preliminaries

In the paper, we sometimes reference a pixel P by its coordinates (x, y). We denote the two colors of the image by C_1 and C_2, and the color of a pixel P by $C(P)$. Color $\bar{C_i}$ is the converse color of C_i, meaning $\overline{C_1} = C_2$ and $\overline{C_2} = C_1$.

Definition 1 (Neighbors) Pixels P_1 and P_2 are:

a) *4-neighbors* if $(x_1, y_1) = (x_2 \pm 1, y_2)$ or $(x_1, y_1) = (x_2, y_2 \pm 1)$;

b) *8-neighbors* iff they are 4-neighbours or $(x_1, y_1) = (x_2 \pm 1, y_2 \pm 1)$.

Intuitively, pixels that are 4-neighbors are vertically or horizontally adjacent. Pixels that are 8-neighbors can also be diagonally adjacent.

Definition 2 (Trail) A *trail* from pixel P_1 to pixel P_n is a sequence of pixels $P_1, P_2, \dots, P_n$ such that P_i and P_{i+1} are neighbours for all i in $\{1, \dots, n-1\}$, and $(P_i, P_{i+1}) = (P_j, P_{j+1}) \Rightarrow i = j$ (i.e. no pair of consecutive pixels appears twice in the sequence), for all i, j in $\{1, \dots, n-1\}$.

A trail $P_1, P_2, \dots, P_n$ is *closed* if $P_1 = P_n$. We call a pixel P a *border pixel*, if P is of color C_i and it has at least one neighbor of color $\bar{C_i}$. If at least one such neighbor is a 4-neighbor, we call it a *4-border pixel* (see Fig. 2). We call a closed trail of border pixels a *border*, and a closed trail of 4-border pixels a *4-border*.

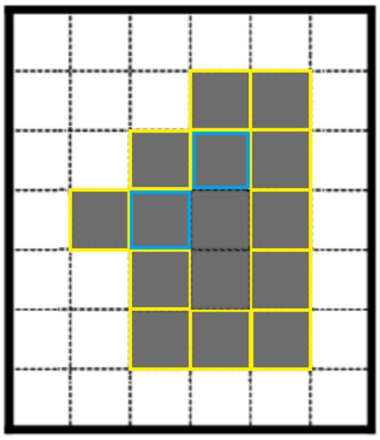

Fig 2. In the gray component, 4-border pixels are inside yellow frames, while border pixels that are not also 4-border pixels are inside blue frames.

Definition 3 (Connected pixels) Pixels P_1 and P_2, both of color C_i, are *connected* if there exists a trail from P_1 to P_2 containing only pixels of color C_i.

Definition 4 (Connected component) The set $S(P)$ of all pixels connected to P is called the *connected component* of P.

To our knowledge, there is no standardized approach to specifying a polygonal representation of a connected component. We decide on the following one. Abrash [10] specifies a polygon rasterization algorithm which, given a polygon as input, colors the pixels that belong to that polygon according to a specific criterion. An expanded version of this criterion that properly handles edge cases is described in [11]. The *polygon rasterization algorithm* (PRA) is the algorithm specified in [10] using criteria specified in [11].

Now, we define a polygonal representation as follows.

Definition 5 (Polygonal representation) A *polygonal representation* of a connected set S of pixels of color C_i is a set of simple planar polygons $T = \{T_0, T_1, T_2, \dots, T_n\}$, such that:

1) each of $T_1, T_2, \dots, T_n$ is inside T_0,
2) no two polygons from $T_1, T_2, \dots, T_n$ intersect, and
3) when T_0 is rasterized using *PRA* and color C_i, and each of $T_1, T_2, \dots, T_n$ is rasterized using *PRA* and color $\bar{C_i}$, a pixel P will have color C_i iff $P \in S$.

Intuitively, T_0 is the bounding polygon of S, the polygons $T_1, T_2, \dots, T_n$ are the polygons that correspond to the different "holes" in S. The order in which the holes are specified is not important. Note that $T = \{T_o\}$ if there are no holes in S. The advantage of this approach is the following. Assume we are given a connected set of pixels S, and we find its polygonal representation T. Now, by rasterizing T_0 using *PRA* and color C_i, and rasterizing each of $T_1, T_2, \dots, T_n$ using *PRA* and color $\bar{C_i}$, on a blank image, a pixel P will be colored if and only if it belongs to S. Every polygon T will correspond to some 4-border of S.

We now define our problem.

Polygonization of a Connected Component (*PCC*)

Input: A binary image I of width w_I and height h_I; coordinates (x_t, y_t) of a target pixel P_t.

Output: A polygonal representation T of the connected component $S(P_t)$.

3. Subtasks

The approach we present has a few subtasks which it needs to perform and that have already been extensively studied. First, the pixels encompassed by T must be exactly those in $S(P_t)$, so our methods need a way of identifying them. The number of pixels outside of $S(P_t)$ that the methods access should be small, to keep the time and space requirements low. Next, we have noted that every polygon T_i corresponds to some 4-border of S. Therefore, a simple way of constructing T_i is to follow the pixels along the corresponding 4-border and add a new vertex to T_i whenever appropriate. We follow the border pixels using one of the contour tracing algorithms discussed in Section 3.1; we discuss the construction of the polygons in Section 3.2 Finally, the methods need to determine if a border corresponds to the bounding polygon of S or to a hole inside S. We describe how this can be done in Section 3.3.

From these tasks, we infer some lower bounds of the time and space complexities of our approach. The output of *PCC* is a set $T = \{T_0, T_1, T_2, \dots, T_n\}$. We denote the number of vertices (corner points) of T_i with h_i. Then, the size of the output is $\Theta(\sum_{i=0}^{n} h_i)$, so $\Omega(\sum_{i=0}^{n} h_i)$ is a lower bound on the space complexity. Because the vertices are added one by one, $\Omega(\sum_{i=0}^{n} h_i)$ is also a lower bound of the time complexity. Any attempt at solving *PCC* requires identifying the pixels belonging to $S(P_t)$. We cannot know if a pixel $P_j, j \neq t$ belongs to $S(P_t)$ before we know its color $C(P_j)$. It follows that our approach nees to read the color of every $P_j \in S(P_t) \setminus \{P_t\}$, so another lower bound of the time complexity and the number of pixel reads is $\Omega(S(P_t))$. We provide some upper bounds when discussing the complexity of our approach in Section 4.1.1.

3.1. Contour tracing

Seo et al. [12] provide an overview of contour tracing algorithms and classify them into three types: pixel following, vertex following, and run-data-based following; pixel following being the most common type. Pixel following algorithms visit the centers of pixels in a border, one by one and in sequence. They first visit a starting pixel P_s from a predefined direction. Then, for each border pixel they visit, they determine which neighbor border pixel to visit next. The algorithm stops when a specific stopping criterion is satisfied, ensuring that the algorithm has traversed all the pixels in the border. Vertex following algorithms work in the same manner, except that they visit the vertices of a pixel, i.e., they move along the edges of pixels. Both pixel following and vertex following algorithms are suitable when solving *PCC*. Run-data-based following algorithms do not follow along borders; rather, they trace the contours by processing the entire image. For that reason, they do not appear as an appropriate choice if the goal is solving *PCC* with a low time complexity.

3.2. Constructing a polygon from a contour

Multiple algoritms have been proposed for finding polygonal approximations of digital curves; an overivew is given by Pratihar and Bhowmick [13]. These can be used for constructing the polygons T_i in our problem. The algorithms are based on various techniques, such as area deviation, dominant point detection, curvature estimation, randomized techniques, etc. Polygonizations of digital curves are not unique, as different polygons can yield identical images when rasterized. Different algorithms are thus devised to achieve different approximation accuracies, and have different time and space complexities; in general, greater accuracies require more computational resources. For *PCC*, the choice of which algorithm to use depends on the desired properties of the resulting polygons. That may affect the time and space complexity of the corresponding algorithm.

3.3. Polygon classification

In the *Reduce* method we later describe, once a border is traced and its corresponding polygon is constructed, it is necessary to classify the polygon as the bounding polygon T_0 or as a hole polygon $T_i, i \neq 0$. The input pixel P_t is inside T_0, but not inside any $T_i, i \neq 0$. We can check if the point (x_t, y_t) is inside the constructed polygon using a point-in-polygon algorithm; then, that polygon is T_0 if the point is inside, otherwise it is the polygon of a hole. The point-in-polygon problem has been well reasearched and many algorithms and efficient implementations exist; an overview can be found in [14]. Some of the algorithms are specifically made for certain types of polygons. In our case, the polygons we find can be non-convex, but they are always simple (they do not intersect themselves). The appropriate algorithms for this case have a running time of $\Theta(h)$, where h is the number of vertices of the polygon, and are faster in practice than algorithms that have to work with non-simple polygons as well.

Instead of performing a point-in-polygon test after constructing a polygon, it is also possible to perform it during the process. For example, the crossings test [14, pp. 26-27] can be implemented as follows. We imagine a ray is shot from (x_t, y_t) along some axis. While tracing and polygonizing the the border, we check, whenever we add a new vertex to the resulting polygon, if the edge formed by the new vertex and the previous one intersects the ray. At the end of the process, the parity of the total number of crossings determines if (x_t, y_t) is inside the polygon or not.

4. Method

In Section 4.1, we describe the *Polygonize* method, which is the method we propose for solving *PCC*; we analyze its complexity in Section 4.1.1. In Section 4.1.2, we propose a way of reducing the space complexity of the *Polygonize* method by describing a preprocessing method *Reduce*.

4.1. Polygonize

The *Polygonize* method uses depth-first search (*DFS*) and a Boolean matrix we denote by *M*. DFS views the image I as a graph whose vertices represent pixels, two vertices sharing an edge if and only if the pixels they represent are neighbors in I. *M* is a Boolean matrix of size $w_I \times h_I$, where the element $M(x,y)$ is $true$ if pixel (x,y) was already visited by the search algorithm, and $false$ otherwise. All pixels are labeled as not visited in the beginning.

We assume P_t is of color C_i. *Polygonize* performs the following. The DFS algorithm is run starting with pixel P_t. Whenever the algorithm visits an unvisited pixel P, it labels P as visited in *M*. For a pixel P of color C_i visited by DFS, DFS also visits all of the yet unvisited neighbors of P. When such a neighbor P' of color $\bar{C_i}$ is visited, the *PolygonizeHandleBorder* procedure, described below, is performed to process a border whose neighbor is P'. *Polygonize* terminates when the DFS has finished visiting pixels and the last *PolygonizeHandleBorder* procedure terminates.

PolygonizeHandleBorder performs multiple tasks. It traces along a border using a contour tracing algorithm, and constructs a polygonal approximation of the border. During the tracing, whenever an unvisited pixel of color $\bar{C_i}$ is visited, it gets labeled as visited in *M*, so each border is processed only once. The construction of the polygonal approximation of the border is performed after tracing. The border pixels need to be stored in a buffer array during tracing, and the approximation will then be performed on the array. The resulting polygon is added to the output set T.

We give the pseudocode of *Polygonize* below. Depth-first search is implented using a stack.

```
Algorithm Polygonize
Input: image $I$ of size $w_I \times h_I$; pixel $P_t$
Output: a polygonal representation $T$ of $S(P_t)$

    $T = \{\}$
    define a matrix $M$ of size $w_I \times h_I$
    set $M(x,y) = 0$ for all pixels $(x,y)$
    set $S$ to an empty stack
    $S.push(P_t)$
    while $S$ is not empty:
        $P = S.pop()$
        if $P$ is of color $C(P_t)$ and $M(x,y) = false$:
            $(x,y) = true$
            push all neighbors of $P$ onto $S$
        else if $P$ is of color $\overline{C(P_t)}$ and $M(x,y) = false$:
            $T.add(PolygonizeHandleBorder(p))$
    return $T$
```

4.1.1. Complexity analysis

The running time of *Polygonize* depends on the time needed for allocating and initializing the matrix M, time taken by the search, and time taken by running *PolygonizeHandleBorder* for each of the borders of $S(P_t)$. Time needed for allocating and initializing the matrix M may depend on different factors, such as which computing platform is used, or the size the matrix. The time complexity of depth-first search is linear in the number of different pixels visited, because every pixel can be visited from at most 8 of its neighbors. Because of the choice of pixels that the search algorithm visits, the visited pixels will be exactly the ones in $S(P_t)$ and the pixels of color $\overline{C_\iota}$ that are neighbors of border pixels in $S(P_t)$. Because every border pixel has at most eight 8-neighbors, the time complexity of the search is $O(|S(P_t)|)$. The time needed for each instance of the *PolygonizeHandleBorder* procedure depends on which contour tracing and polygonization algorithms are used.

Polygonize uses $O(w_I \times h_I)$ space for the matrix *M*. Depth-first search can implemented using a stack of size linear in the number of different pixels visited, i.e., with space complexity of $O(|S(P_t)|)$, where of course $|S(P_t)| = O(w_I \times h_I)$. The space complexity of the *PolygonizeHandleBorder* procedure depends on the choice of tracing and polygonization algorithms.

4.1.2. Reducing the size of matrix M

In comparison to polygonizing the entire image, *Polygonize* succeeds in polygonizing only the borders of the connected component $S(P_t)$. However, the space needed for the matrix *M* still depends on the size of the entire image. Here, we describe a preprocessing method, *Reduce*, that can reduce this space requirement.

We again assume P_t is of color C_i. *Reduce* begins at pixel P_t, and starts moving along pixels in an arbitrary direction, until it reaches the bounding border of $S(P_t)$. Then, *Reduce* defines a Boolean matrix *M'* of size $(w_{S(P_t)} + 2) \times (h_{S(P_t)} + 2)$, where $w_{S(P_t)}$ and $h_{S(P_t)}$ are the height and width (in pixels) of $S(P_t)$. Constructed this way, *M'* is the smallest rectangular matrix that can encompass the component $S(P_t)$ and its neighboring pixels.

Without loss of generalization, we assume the direction in which pixels are visited is "upwards", i.e., the next pixel to be visited after pixel (x,y) is pixel $(x,y+1)$. Starting from P_t, *Reduce* moves upwards along pixels until it visits a pixel of color $\overline{C_\iota}$, which means that it has encountered a border. *ReduceHandleBorder*, a modified version of the *PolygonizeHandleBorder* procedure, is then ran. *ReduceHandleBorder* traces along the encountered border, and finds its polygonal approximation T_i. In addition, it also keeps track of the minimum and maximum x and y coordinates of the border pixels it visits, as well as the coordinates of the pixel with the maximum y coordinate. However, unlike *PolygonizeHandleBorder*, *ReduceHandleBorder* cannot yet label the border pixels of color $\bar{C}_i$ as visited. Rather, it tests if the processed border is the bounding border of $S(P_t)$, or the border of a hole. This is done using a point-in-polygon test on polygon T_i and point (x_t, y_t), as described in Section 3.3. After the test, the instance of *ReduceHandleBorder* terminates, and *Reduce* continues in a manner which depends on the result of the test.

If the test determined the processed border is that of a hole, *Reduce* continues visiting pixels upwards, starting from the pixel with the largest y coordinate it visited thus far, and performs *ReduceHandleBorder* on the next border it encounteres. Starting from the pixel with the largest y coordinate insures that no border will be processed twice, and that *Reduce* will eventually reach the bounding border of $S(P_t)$. On the other hand, if the test determines the border processed by *ReduceHandleBorder* is the bounding border of $S(P_t)$, the method can define the matrix *M'* of size $\left(w_{S(P_t)} + 2\right) \times \left(h_{S(P_t)} + 2\right)$, so that the matrix encompasses the component $S(P_t)$ and its neighboring pixels. Therefore, *M'* now has sufficient size to enable *Polygonize* to polygonize $S(P_t)$.

Before running *Polygonize*, however, *Reduce* can perform one additional step. For each border that was processed, *Reduce* can trace along the edges of that border's polygonal approximation, and label the pixels of color $\bar{C}_i$ along the border as visited in *M'*. In this way, borders processed by *Reduce* will not be processed again in *Polygonize*. As *Reduce* has already computed the polygonal approximations of these borders, it simply adds them to the final output. Now, *Polygonize* can be run, using the matrix *M'* and starting from pixel P_t, to visit all pixels in $S(P_t)$ and process borders which have not processed by *Reduce*, if any such borders exist.

Reduce reduces the space needed for the matrix *M'* from $\mathcal{O}(w_I \times h_I)$ to $\mathcal{O}(w_{S(P_t)} \times h_{S(P_t)})$. Assuming the direction of visiting pixels is *upward*, the number of pixels visited in *Reduce* is bounded by $h_{S(P_t)}$. Point-in-polygon tests ran in time proportional to the number of vertices of the tested polygon. Thus, the cumulative time taken by all point-in-polygon tests performed in *ReduceHandleBorder* is $\Theta(\sum_{i=0}^{n} h_i)$ (where h_i is the number of vertices of polygon T_i). Because $\sum_{i=0}^{n} h_i = \mathcal{O}(|S(P_t)|)$, the tests do not increase the overall time complexity of solving the problem; the same analysis is true for labeling pixels as visited in *M'* just before running *Polygonize*. Therefore, *Reduce* reduces the space complexity of solving *PCC* without increasing the time complexity.

5. Conclusion

In this paper, we have presented a novel approach of polygonizing connected areas in binary images. As the literature discussing the specific problem is sparse, we have first formally defined the problem using established concepts in image processing and computer graphics, to avoid ambiguity and provide foundation for future work.

We have proposed a method, *Polygonize*, which, given an image and a pixel of interest, polygonizes only the borders of the connected component the pixel belongs to. This is unlike existing polygonization techniques (and vectorization techniques in general), which polygonize entire images, thus performing unnecessary work. Existing techniques also necessitate extracting the polygons of interest from the final output. Our method removes this need, and, by limiting the scope of polygonization, reduces the time and space requirements of the process.

The time complexity of the *Polygonize* method depends only on the size of the connected component of the pixel of interest. However, the space complexity of the method depends on the entire image, due to the image-size Boolean matrix the method uses. For that reason, we have also proposed a preprocessing method, *Reduce*, which finds the minimum size of the matrix such that *Polygonize* still performs properly. The *Reduce* method does not increase the time complexity of the whole process. Thus, when *Reduce* is used previously to *Polygonize*, both the time complexity and the space complexity of the approach depend only on the connected component of interest. The *Reduce* method could potentially be used for reducing the space complexity in applications outside of vectorization, e.g., in applications where traversing a connected component is required.

Future work on this problem includes generalizing the proposed polygonization approaches to arbitrary types of vectorization, and to color images. As polygonization is one of the simplest forms of vectorization, and as binary images are simpler than color images, vectorizing connected components in color images could introduce new challenges, but it also may provide space for further optimization. Lastly, applying parallel techniques using graphic processors on this class of problems asserts itself as a natural option. That being said, we cannot rule out the possibility of some parts of the problem being inherently sequential.

References

[1] Ye, Su, R. Pontius and Rahul Rakshit. "A review of accuracy assessment for object-based image analysis: From per-pixel to per-polygon approaches." Isprs Journal of Photogrammetry and Remote Sensing 141 (2018): 137-147.

[2] Avrahami, Yair, Y. Raizman and Y. Doytsher. "A Polygonal Approach for Automation in Extraction of Serial Modular Roofs." Photogrammetric Engineering and Remote Sensing 74 (2008): 1365-1378.

[3] Soysal, Ömer M., B. Gunturk and K. Matthews. "Image Retrieval using Canonical Cyclic String Representation of Polygons." 2006 International Conference on Image Processing (2006): 1493-1496.

[4] Ruiz-Lendínez, J. J., M. Ureña-Cámara and F. J. Ariza-López. "A Polygon and Point-Based Approach to Matching Geospatial Features." ISPRS Int. J. Geo Inf. 6 (2017): 399.

[5] Laban, Noureldin, M. El-Saban, A. Nasr and H. Onsi. “System refinement for content based satellite image retrieval.” The Egyptian Journal of Remote Sensing and Space Science 15 (2012): 91-97.

[6] Wang, Ke, T. Shi, G. Liao and Qi Xia. “Image registration using a point-line duality based line matching method.” J. Vis. Commun. Image Represent. 24 (2013): 615-626.

[7] Kopf, J. and Dani Lischinski. “Depixelizing pixel art.” ACM SIGGRAPH 2011 papers (2011)

[8] Hoshyari, Shayan, E. Dominici, A. Sheffer, N. Carr, Zhaowen Wang, Duygu Ceylan and I-Chao Shen. “Perception-driven semi-structured boundary vectorization.” ACM Transactions on Graphics (TOG) 37 (2018): 1 - 14.

[9] Chen, Jiazhou, Qi Lei, Yongwei Miao and Qunsheng Peng. “Vectorization of line drawing image based on junction analysis.” Science China Information Sciences 58 (2014): 1-14.

[10] Abrash, M.. “Michael Abrash's Graphics Programming Black Book, Special Edition.” (1997): Ch. 40, 536-539.

[11] McCool, M., C. Wales and K. Moule. “Incremental and hierarchical Hilbert order edge equation polygon rasterizatione.” HWWS '01 (2001): Ch. 2.1, Fig. 2

[12] Seo, Jonghoon, Seungho Chae, Jinwook Shim, Dong-Chul Kim, Cheolho Cheong and T. Han. “Fast Contour-Tracing Algorithm Based on a Pixel-Following Method for Image Sensors.” Sensors (Basel, Switzerland) 16 (2016): n. pag.

[13] Pratihar, Sanjoy and Partha Bhowmick. “Fast and Direct Polygonization for Gray-Scale Images Using Digital Straightness and Exponential Averaging.” Int. J. Image Graph. 16 (2016): 1650007:1-1650007:36.

[14] Haines. "Point in Polygon Strategies," Graphics Gems IV, ed. Paul Heckbert, Academic Press (1994): 24-28