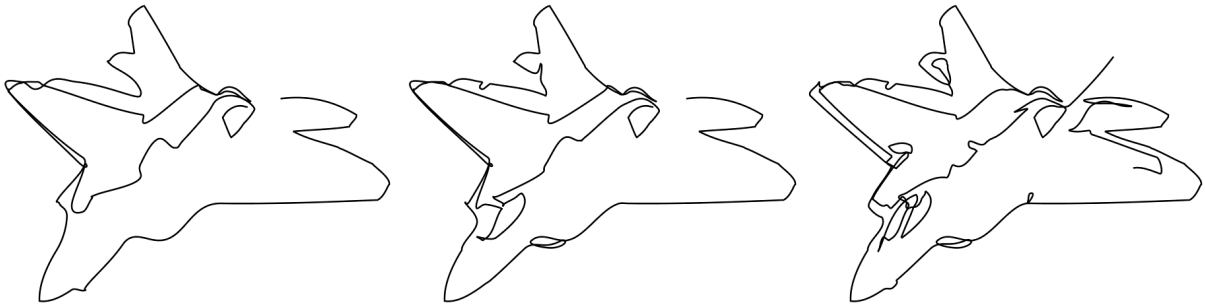
Pacific Graphics 2011
Jan Kautz, Tong-Yee Lee, and Ming C. Lin
(Guest Editors)

Volume 30 (2011), Number 7

A Graph-based Approach to Continuous Line Illustrations with Variable Levels of Detail

Fernando J. Wong and Shigeo Takahashi

The University of Tokyo, Japan



(a) Detail level 1 (b) Detail level 2 (c) Detail level 3

Figure 1: Continuous line illustration of a plane image generated with three levels of detail.

Abstract

This paper introduces a method for automatically generating continuous line illustrations, drawings consisting of a single line, from a given input image. Our approach begins by inferring a graph from a set of edges extracted from the image in question and obtaining a path that traverses through all edges of the said graph. The resulting path is then subjected to a series of post-processing operations to transform it into a continuous line drawing. Moreover, our approach allows us to manipulate the amount of detail portrayed in our line illustrations, which is particularly useful for simplifying the overall illustration while still retaining its most significant features. We also present several experimental results to demonstrate that our approach can automatically synthesize continuous line illustrations comparable to those of some contemporary artists.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation J.5 [Computer Applications]: Arts and Humanities—Fine Arts

1. Introduction

A *continuous line illustration* (CLI) is a type of drawing consisting of a single line. This technique is commonly taught at the early stages of many art courses in order to help students to loosen up their artistic senses [Nic90]. In many cases, these simple drawing exercises evolve into beautiful works of art. Apart from artistic purposes, CLIs have other applications such as in quilting designs [Fri01], steel wire sculptures [Loh09], and connect-the-dots puzzles.

Although relatively simple, these illustrations possess a unique charm that captures the heart of whoever sees them. This in turn has given birth to different artistic styles. For example, the "labyrinthine projection" style of Morales [Mor05] consists in tracing a single non-intersecting line and creating shading effects by controlling the density of lines in certain areas. In the case of Slater's drawings [Sla01], a non-intersecting line fills the entire drawing space while varying in thickness and color as necessary, resulting in beautiful and colorful works of art.

Figure 2: A continuous line illustration by professional illustrator Rachel Ann Lindsay [Lin10].

On the other hand, Sable's works [Sab09] are dominated by continuous lines with several self-intersections and slight variations in thickness, thus providing more insight on the shape of depicted objects as well as shading effects.

Our CLI results are closer in appearance to the works of professional illustrator Rachel Ann Lindsay [Lin10], on which a self-intersecting continuous line with uniform thickness is skillfully utilized in order to portray an object or scene, as shown on Figure 2. It is astonishing to see how much variety and beauty can be born from something as simple as tracing a single line.

Motivated by the works of the above mentioned artists, as well as by the technical difficulties involved in creating such drawings, we have devised a method for creating self-intersecting CLIs. Our approach essentially consists in inferring a graph based on a set of extracted image contours and finding a path that traverses through all edges of the graph. A series of simplification steps are employed in order to reduce visual cluttering and also a method for controlling to some extent the level of detail in the final illustration is proposed as well. A result of this approach is shown in Figure 1.

This paper is organized as follows: Related work relevant to our research is presented in Section 2. An overview of our CLI algorithm is given in Section 3, and further details are provided in Sections 4, 5 and 6. Results of our approach are briefly discussed in Section 7, followed by conclusions and pointers to future work in Section 8.

2. Related Work

Although the creation of continuous line illustrations has been popular among artists, very few works on this kind of drawings can be found in the field of computer graphics.

In [BH04] was proposed the creation of continuous line drawings by first obtaining a set of points whose distribution along the drawing space was based on the intensity of an input image. An instance of the traveling salesman problem (TSP) was then solved over the resulting set of points. This work was later extended in [KB05] through the use of modern image stippling techniques, in order to obtain a point distribution that better resembled the original image.

CLIs are also related to labyrinths and mazes. The organic labyrinthine structures proposed in [PS06] are basically CLIs. Their method is based on the evolution of a set of input curves into a labyrinth-like pattern, through the application of several forces. CLIs can also be used as solution paths for picture mazes, as seen in [WT09]. In this case, a hybrid maze is created by placing maze walls according to a primary image, and approximating the shape and shading of a secondary one with the solution of the maze.

3. Method Overview

Our approach generates a CLI from an image through the use of image processing techniques and the application of concepts in graph theory. We extract the edges of the image and create a graph based on them. At this point, the graph undergoes a series of modifications and then a path that traverses all of its edges is found. This path is then transformed into a CLI after several post-processing operations.

Depending on the application, users might find a lack or excess of details in the resulting CLI. A few examples are: controlling the amount of dots in connect-the-dots puzzles and changing the amount of detail in the CLI according to the distance from the viewer. For this reason, we have also incorporated a method for allowing users to specify levels of detail in the final illustration, while maintaining the coherence of the CLI between detail levels at the same time.

In short, our approach takes an image I and a desired number of detail levels n as input, and does the following:

- Extract the contours of the input image.
- Classify the contours into n detail levels.
- Infer a graph from the extracted contours.
- Find a path that traverses through all graph edges.
- Trace a continuous line based on the obtained path and the specified detail level.

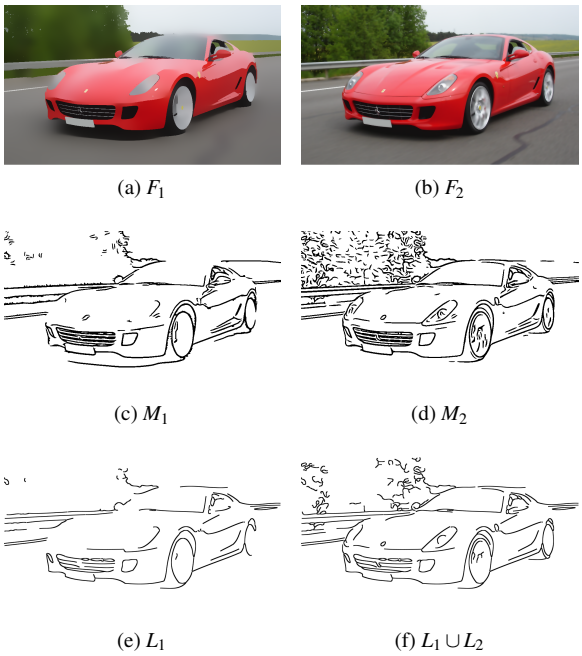
A more detailed explanation of these steps is given in the following sections.

4. Image Processing and Graph Construction

4.1. Initial Filtering and Edge Detection

Our method starts by obtaining a set of images derived from the original, which gradually vary in the amount of features they portray. Edge detection is then applied to each of them.

A set of images $\mathbf{F} = \{F_1 \dots F_n\}$ (Figures 3a and 3b) is created from the original image I, each of them obtained through bilateral filtering [TM98] with decreasing kernel



(a) F_1 (b) F_2

(c) M_1 (d) M_2

(e) L_1 (f) $L_1 \cup L_2$

Figure 3: Edge Detection and Classification: Two bilateral filtered images of a car, created with different kernel sizes are shown in (a) and (b), while their edge detection results are shown in (c) and (d). Their classified edges at two different detail levels are shown in (e) and (f).

sizes. The kernel sizes for domain and range filtering at detail level i are given by $\sigma_f(i) = r_o + (n-i)r_{dec}$ and $\sigma_g(i) = s_o + (n-i)s_{dec}$, respectively, where r_o and s_o are the initial kernel sizes while r_{dec} and s_{dec} are the amounts by which the kernel sizes are decreased. In this way, most significant features are retained in F_1, while most minor details are preserved in F_n. We apply 5 iterations of the filter per detail level. Since filtering becomes prohibitely expensive as the kernel grows in size, we employ a separable kernel implementation [PvV05].

Edges are then detected for each F_i by making use of a flow-based difference of Gaussians (FDoG) filter [KLC09]. This produces a set of binary maps $\mathbf{M} = \{M_1 \dots M_n\}$, where detected edges are marked in black (Figures 3c and 3d). Three iterations of the FDoG filter are performed per detail level with the default parameters specified in [KLC09].

The edges detected in map M_n are then processed through a line thinning algorithm [HWL03] in order to obtain a collection of one-pixel-wide segments S.

4.2. Edge Classification into Detail Levels

The next step in our algorithm is to classify each of the extracted edges into one of n levels of detail. The idea is that edges belonging up to a certain level are kept in the final CLI, while others are suppressed or simplified.

Segments in S are classified into a set of detail levels $\mathbf{L} = \{L_1 \dots L_n\}$ (Figures 3e and 3f) in two steps. The first step consists in comparing each segment against each binary map, from the coarsest to the finest scale (from M_1 to M_n). In other words, we compare each segment $s \in S$ against M_i, and assign it to L_i if most parts of the segment belong to black areas in M_i (at least 80% of the pixels in the edge in our implementation). The process is then repeated for all unclassified edges against $M_i + 1$ and so on.

During the second step, we reclassify the edges according to length, since longer edges have a tendency to correspond to significant features in general. Starting from L_n to L_1, all edges in L_i with lengths exceeding $\eta_i = \mu_i + \sigma_i$ are reassigned to L_{i-1}, where μ_i and σ_i are the average and standard deviation of edge lengths in L_i, respectively. The process is then repeated for L_{i-1} and so on. After this step, L_1 should have a majority of the long and significant edges, while most minor edges are left at higher levels.

We proceed to infer a graph from the edges in S after all of them have been classified into detail levels.

4.3. Inferring a Graph From a Set of Contours

Our approach infers a graph $G = (E,V)$ from a set of line segments S in a straightforward manner. Assuming no line $s \in S$ intersects with any other line $t \in S$, we regard the endpoints of each segment as graph vertices. If two or more endpoints are within a certain distance from each other, we merge them as a single graph vertex. Additional nodes are created at points where the line curvature presents sudden changes. An example of such a graph is shown in Figure 4a.

In case more than one connected component exists in the graph, we create additional *auxiliary edges* between pairs of vertices so that it becomes a fully connected graph. However, there are cases in which we would prefer to connect two graph components at points along their edges rather than between vertices. Our model attempts to simulate this behavior during the creation of auxiliary edges.

Auxiliary edges are created by first obtaining a set of segments S' from subdividing each segment $s \in S$ into smaller segments of length h. Each of these small segments belongs to the same detail level of its original. A graph $G' = (E',V')$ is then inferred from S' (Figure 4b). Please note that $V \subseteq V'$ in this graph. Two Delaunay triangulations $T(V)$ and $T(V')$ (Figures 4c and 4d) are computed over the vertex sets V and V', respectively. The weight of each triangulation edge $(u,v) \in T(V)$ is computed as

$$w(u,v) = d(u,v) \tag{1}$$

where $d(u,v)$ denotes the length of (u,v). In contrast, the weight of each edge $(u',v') \in T(V')$ is computed as

$$w(u',v') = \alpha \times d(u',v') \tag{2}$$

where $\alpha \geq 1$ is a user-specified penalty parameter that provides a trade-off between the usually longer edges connecting original vertices and the usually shorter ones connecting vertices created after line subdivision. At each iteration, the edge $(u,v) \in [T(V) \cup T(V')]$ with the lowest weight among both triangulations is processed. If vertices $u \in V'$ and $v \in V'$ are in different components, an auxiliary edge (u,v) is inserted in E'. Edges are processed in this way until the graph becomes a single component or no more triangulation edges remain unprocessed.

After this procedure, graph G' becomes fully connected, nevertheless, it usually presents a high amount of degree-one vertices. These are particularly problematic, since this means their incident edge will have to be forcefully retraced in the final CLI, thus increasing the amount of line density. In order to reduce the amount of these vertices, we process once more all edges $(u,v) \in [T(V) \cup T(V')]$. If either u or v has degree one, $d(u,v) < \beta$ and (u,v) has a difference in orientation of at most 30 degrees to the edge incident to the degree-one vertex, we insert auxiliary edge (u,v) in E'. Parameter β controls the maximum length allowed for these edges, while the orientation constraint is imposed in order to avoid sudden changes in line curvature on the CLI.

At this stage, we reduce graph G' by merging all edges sharing a common vertex of degree two, as long as the vertex in question is not one of the original vertices and both edges belong to the same level of detail. Figure 4e shows an example of a graph created through this approach. For ease of explanation, we will refer to this reduced graph as $G = (E,V)$ throughout the rest of this paper.

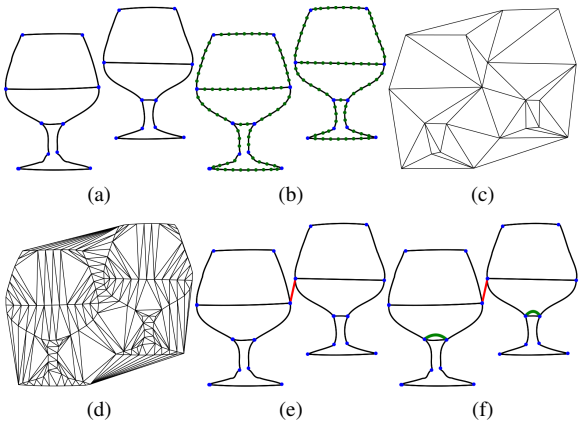


Figure 4: Constructing the Graph: (a) An initial graph is inferred according to the edges of the image. (b) Additional vertices are created at fixed intervals within each line segment. (c) A Delaunay triangulation $T(V)$ of the original vertices is calculated. (d) A second triangulation $T(V')$ is calculated from the new set of vertices. (e) Auxiliary edges (in red) are inserted in order to merge the disconnected graph components, based on the edges of both triangulations. (f) Duplicate edges (in green) are created in order to Semi-Eulerize the graph.

5. A Path that Traverses All Edges in a Graph

An Eulerian path in a graph $G = (V,E)$ is a sequence of vertices $v \in V$ that traverses through all edges $e \in E$ exactly once. G is called Semi-Eulerian if only two of its vertices are of odd degree, which implies that an Eulerian path can always be found in it. If all vertices are of even degree, the graph is then Eulerian, and an Eulerian cycle can be found. An Eulerian path or cycle can be found in G by making use of Fleury's algorithm [Fle83].

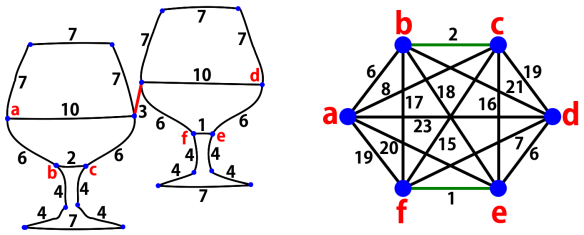
Thus, if G is a graph inferred from the features of an input image, then a CLI could be considered as an Eulerian path in G. Usually, our graphs will contain more than two odd-degree vertices, as seen in Figure 5a. In such cases, odd vertices are removed through a Semi-Eulerization process.

5.1. Semi-Eulerization

Semi-Eulerizing a graph consists in creating *duplicate edges* in order to remove all but two odd-degree vertices. Ideally speaking, we would like to find an optimal Semi-Eulerization that minimizes the sum of the length of all duplicate edges. Although efficient algorithms for finding optimal graph Eulerizations have been proposed [EJ73], obtaining an optimal Semi-Eulerization is still computationally expensive. Also, these methods tend to rely solely on edge duplication, while the nature of CLIs grants us more flexibility, allowing direct connections between vertices and even the removal of some edges. For these reasons, we apply instead a greedy approach to Semi-Eulerization.

A complete graph $G_o = (E_o, V_o)$, whose set of vertices $V_o \subseteq V$ contains only the vertices of odd degree in G, is created (Figure 5b). The length $l(u,v)$ of each edge $(u,v) \in E_o$ is calculated as the total length of the shortest path $P(u,v)$ in the original graph G. Each edge $(u,v) \in E_o$ is processed in ascending weight order, until only a single edge remains. If neither $u \in V_o$ nor $v \in V_o$ have been processed already, the shortest path $P(u,v)$ is found in G and the distance $d(u,v)$ is calculated. If $l(u,v) > \kappa \times d(u,v)$ and $d(u,v) < \varepsilon$, then a new auxiliary edge (u,v) is inserted in G, otherwise all edges traversed by $P(u,v)$ are then either duplicated (if no duplicate of the edge exists) or removed (if a duplicate exists) in G. This means each edge in the final graph is duplicated at most once. Parameter κ provides some balance between merely retracing edges and making direct connections between vertices, while ε keeps direct connections within a certain length.

In addition, the length of a duplicate edge is set to the negative value of its original, so that $P(u,v)$ has a tendency to retrace over previously duplicated edges, thus removing them



(a) Weighted contour graph (b) Odd-vertex graph

Figure 5: Semi-Eulerization steps: (a) A graph with several odd-degree vertices (labeled in red). (b) A complete graph G_o is built upon the odd-degree vertices of the input graph. Vertex matchings (in green) are chosen in a greedy manner in our approach.

from the graph. We are aware that current shortest path algorithms are unable to guarantee the returned path is indeed the shortest one in undirected graphs with negative edges, but they still provide a good approximation measure for such paths. In an attempt to reduce line cluttering, all auxiliary edges and short edges duplicated during this process are removed along with their duplicates, as long as the graph remains connected. Figure 4f shows a Semi-Eulerized version of the graph in Figure 4e.

5.2. Modified Fleury's Algorithm

Once a Semi-Eulerian graph is obtained, Fleury's algorithm [Fle83] can be used in order to find an Eulerian path in G. The logic behind Fleury's algorithm for Semi-Eulerian graphs is to start at any of the two odd-degree vertices, then traverse through an edge $e \in E$ whose removal would not disconnect G, remove e, and repeat the procedure for the following vertex.

We propose a modification to the algorithm in order to avoid sudden changes in line direction, while also minimizing the amount of cluttering at graph vertices. Before removing an edge, the next edge to be traversed is selected by performing a minimum weighted matching of all edges still incident to the current vertex. A complete graph $G_v = (E_v, V_v)$ is built, where each vertex $v_i \in V_v$ corresponds to an edge $e_i \in E$ incident to the vertex $v \in V$. The weight of an edge $(v_i, v_j) \in E_v$ is defined as

$$w_e(v_i, v_j) = \overrightarrow{o_i} \cdot \overrightarrow{o_j} \tag{3}$$

where $\overrightarrow{o_i}$ denotes the normalized orientation vector of edge e_i. A matching of minimum weight is then found in G_v. Since the amount of edges incident to a vertex in any of our inferred graphs is very low (at most 8 edges in all the cases we have seen), we have opted for examining all possible matchings in G_v and selecting the one with lowest weight. The next edge to be traversed is the one matched to the last traversed edge. If the next edge disconnects the graph upon removal, then the matching is discarded and the next one with lowest weight is selected.

5.3. Custom Endpoints

Specifying custom endpoints for an Eulerian path is relatively simple. Let p and q denote the desired endpoints of the path. Then, prior to Semi-Eulerizing the graph, we insert a fictitious edge (p,q) into G. The Semi-Eulerization process is then performed as explained before, only that the goal now is to completely remove all odd vertices from G. If we remove (p,q) from the graph, we end up once more with a Semi-Eulerian graph and then we can apply our modified Fleury's algorithm to obtain an Eulerian path from p to q.

6. Drawing a Continuous Line

6.1. Path Modification According to Detail Level

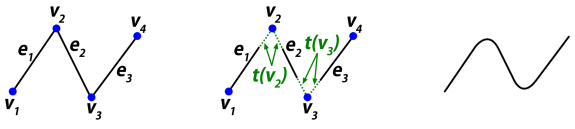
Before tracing the CLI, users specify a detail level d in the range $[1,n]$ and the path is modified according to it. For each edge sequence in the Eulerian path containing only edges in $L_{d+1} \cup \ldots \cup L_n$ and auxiliary edges, we do the following:

1. Let $P_e(i,j) = \{e_i \ldots e_{j-1}\}$ denote such a sequence of edges and $P_v(i,j) = \{v_i \ldots v_j\}$ denote the corresponding sequence of vertices spanned by $P_e(i,j)$.
2. Create a subgraph G^* from G, composed only from vertices in $P_v(i,j)$.
3. Obtain shortest path $P(v_i, v_j)$ in G^*.
4. Replace $P_v(i,j)$ for $P(v_i, v_j)$ in the Eulerian path.
5. Modify $P_e(i,j)$ according to $P(v_i, v_j)$, and set all of its edges as auxiliary edges.

In this way, all edges at detail levels higher than d are either removed or replaced by auxiliary edges, which are rendered differently as will be explained in Section 6.3, while suppressing any particular details in the CLI that are attributed to such levels.

6.2. Line Trimming

In order to draw the CLI, each edge is trimmed by a certain distance from its endpoints, and then a new connecting



(a) (b) (c)

Figure 6: Trimming and reconnecting: (a) Original segments in the Eulerian path. (b) Line segments are trimmed by a distance $t(v_i)$. (c) Connecting curves are traced between the endpoints of subsequent line segments in the path.

curve is traced between the current edge and the next in the Eulerian path (Figure 6).

The maximum trimming distance from an endpoint of edge $e \in E$ is calculated as

$$d_{trim}(e) = min(\lambda, \gamma l_e) \quad (4)$$

where λ is the maximum allowed trimming distance, γ is a control parameter in the range [0,0.5] and l_e denotes the length of edge e. If γ is close to 0, almost no trimming occurs. On the other hand, if we select a value close to 0.5 the line might be trimmed in its entirety.

Edges are then trimmed in the following manner:

1. Let the Eulerian path be defined by a vertex sequence $P_v = \{v_1 \dots v_m\}$ and a sequence of edges $P_e = \{e_1 \dots e_{m-1}\}$ (Figure 6a).
2. Define the trimming distance at each vertex $v_i \in P_v$ as
$$t(v_i) = min(d_{trim}(e_{i-1}), d_{trim}(e_i)) \quad (5)$$
3. For each vertex $v_i \in P_v$, trim edges $e_{i-1} \in P_e$ and $e_i \in P_e$ by a distance $t(v_i)$ from vertex v_i (Figure 6b).

We set $d_{trim}(e_0)$ and $d_{trim}(e_m)$ to 10 in our implementation. Also, if e_i is an auxiliary edge, we set $d_{trim}(e_i) = \lambda$, but do not trim this edge as it does not yet exist.

6.3. Connecting Curves and Additional Edges

Once edge trimming has been completed, a connecting curve is created between each pair of consecutive edges in P_e (Figure 6c). Catmull-Rom splines [CR74] are employed for this purpose, since they are easy to compute and pass through all their control points.

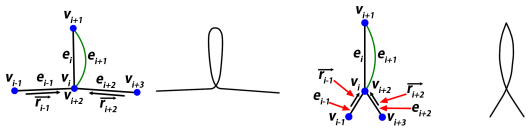
Auxiliary edges and *duplicate edges* in our CLIs are traced as well with Catmull-Rom splines. In the case of *auxiliary edges*, a new curve is created from the last point of the previous edge to the first point of the next one. For *duplicate edges* a different approach is taken. Since these edges are copies of other ones, we would like them to have similar curvature to that of their originals, but at the same time we would like them to be different enough so as to avoid excessive overlapping of curves. For these reasons, we estimate the points at which the edge presents changes in curvature, and use these points as control points of a new curve. Before tracing the curve, each control point, except for the first and last, is displaced by a random distance in a direction perpendicular to the tangent of the curve at the control point in question. In this way, duplicate curves that completely overlap their original counterparts are avoided to some extent.

6.4. Curls and Cusps

We have noticed that most continuous line drawings, such as the ones from Sable [Sab09] and Lindsay [Lin10], usually contain several curl and cusp features at different parts of the line. Our approach attempts to reproduce such features by taking advantage of *duplicate edges* in the Eulerian path. Before actually tracing the CLI, all pairs of subsequent edges (e_i, e_{i+1}) in P_e are checked. If e_i and e_{i+1} correspond to the same edge, that is, one is a duplicate edge that immediately traces over its original, then this pair of edges is turned into either a curl or cusp feature. After studying line drawings of several artists, we chose the following decision function for determining whether to draw a curl or a cusp:

$$D = \begin{cases} curl & \text{if } arccos(|\overrightarrow{o_{i-1}} \cdot \overrightarrow{o_{i+2}}|) > \frac{\pi}{4} \\ cusp & \text{otherwise} \end{cases} \quad (6)$$

where $\overrightarrow{o_{i-1}}$ and $\overrightarrow{o_{i+2}}$ are the normalized orientation vectors of edges e_{i-1} and e_{i+2}, respectively, at the point they meet with either e_i or e_{i+1}. Figure 7 provides more detail in the selection of these features.



(a) Curl (b) Cusp

Figure 7: Curl or cusp?: (a) A sequence of line segments is converted into a curl feature. (b) A sequence of line segments is converted into a cusp feature.

We represent both curls and cusps as two different Catmull-Rom splines. As can be seen in Figure 7, the difference between both type of features lies at vertex v_{i+1}, in other words, at the tip of the curl or cusp. To trace a curl, the tangent vectors of both curves at v_{i+1} are estimated as vectors perpendicular to the orientation of e_i at this vertex. Cusps are done similarly, with the difference that, after estimating the tangent vectors, they are rotated so as to give the impression of a sharp cusp being present in the curve.

After converting parts of P_e into either curl or cusp features, the CLI is drawn by tracing the edges in order as well as the connecting curves between them.

7. Results and Discussion

Our prototype system was implemented in C++ on an Intel Core 2 Duo E6550 2.33 Ghz CPU with 2 GB of RAM. The CGAL library was used for efficient computation of Delaunay triangulations [cga].

Several results are shown on Figures 1 and 10. These examples show the capability of our approach in suppressing or providing details as the level is changed. All of these examples were generated with values $n = 3$, $r_o = 1$, $r_{dec} = 3$, $s_o = 10$, $h = 25$, $\alpha = 4$, $\beta = 20$, $\gamma = 0.12$, $\varepsilon = 100$, $\kappa = 3$ and $\lambda = 25$. The "Arc" results were generated with $s_{dec} = 5$, while the "Motorcycle" results were obtained with $s_{dec} = 15$

Table 1: Step-by-step timing results of CLI generation.

	Arc	Motorcycle	Ukiyoe
Size	**885 × 960**	**1024 × 681**	**1024 × 693**
Bilateral Filter	52.726s	43.283s	44.006s
Flow-based DoG	14.394s	12.082s	12.303s
Edge Classification	0.031s	0.032s	0.035s
Graph Construction	0.148s	0.173s	0.212s
Delaunay Triangulation	0.678s	0.593s	0.865s
Graph Merging	0.082s	0.119s	0.199s
Semi-Eulerization	1.191s	1.072s	2.041s
Fleury's Algorithm	0.024s	0.029s	0.058s
Line Drawing	0.147s	0.190s	0.215s
Total	**69.421s**	**57.573s**	**59.934s**
Detected Edges	817	829	1078
Odd Vertices	374	346	458

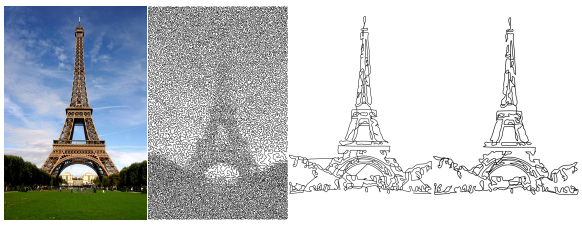
and the "Ukiyoe" ones with $s_{dec} = 10$. A summary of the running times for generating these results is given on Table 1.

Although our approach is indeed able to generate a CLI from an image, usually there are several unnatural lines at some parts, such as in the front wheel and the main body of the bike in Figure 10e. This is partly due to our approach being based on edge detection techniques, which are not accurate enough for our purposes, and to edge classification not taking into account the shape of the object in question.

7.1. Comparison to Previous Approaches

The proposed method is based on the contours of the input image, resulting in smooth continuous lines that better represent object shapes. This is a major difference from [BH04] and [KB05], which attempt to approximate the overall shading of the image with a single line. As explained earlier, a set of points is distributed according to the intensity of the image in these approaches, then the CLI is created by computing a TSP tour over the set of points. In contrast, our approach uses edge detection to obtain the contours of the image and infers a graph based on these contours. The resulting CLI is then created from an Eulerian path found in a Semi-Eulerized version of the graph. Although object contours are somewhat noticeable in the results of [BH04], this comes at the expense of an increased number of cities for the TSP, slowing the running time considerably. On the other hand, our approach creates CLIs that portray these contours without much computational overhead, but is unable to approximate the shading of the image.

Figure 8 shows a comparison between results from [BH04] and our proposed approach. The result in Figure 8b took about 15 minutes to compute while the result in Figure 8d only took 51.94 seconds. This, however, is quite an unfair comparison given that both approaches intend to reproduce two different CLI styles with different applications. While [BH04] shows results similar to the drawing style of Morales [Mor05], which is useful as a half-toning technique, our approach produces results more akin to the works of Lindsay [Lin10], which are more suitable for producing connect-the-dots puzzles and wire sculptures. We also attempted to compute a TSP tour (Figure 8c) over the same set of edges used in Figure 8d. The TSP tour took 36.5 seconds to compute, as opposed to our Semi-Eulerization approach, which generated the CLI in 1.27 seconds from the same set of edges. Also, some unnatural edges are present in Figure 8c, particularly at the left and the bottom of the tower.



(a) (b) (c) (d)

Figure 8: Comparison between CLI approaches: (a) Original input image. (b) Result from [BH04]. (c) A CLI obtained by solving the TSP over a set of detected edges from the image. (d) Our approach.

7.2. Application to Connect-the-Dots Puzzles

We applied our CLI algorithm for the generation of connect-the-dots puzzles such as the one shown in Figure 9, which was generated from the CLI in Figure 1c. Furthermore, we conducted a user study in which 10 participants were handed 5 puzzles generated through this approach and asked to identify the object behind each puzzle before and after solving it. The results of this study are summarized in Table 2.

As you can see, the experiment demonstrates that our puzzles, and thus, the CLIs created with our method, are able to portray the objects in a visually plausible manner. Work on the generation of these puzzles is still at an early stage, as we cannot completely hide the shape of some objects with the current method, as was the case for Puzzle 3.

Figure 9: A connect-the-dots puzzle derived from a CLI.

Table 2: Results of the study with connect-the-dots puzzles.

Puzzle	BS	AS
Puzzle 1 (Figure 9)	2/10	9/10
Puzzle 2	0/10	10/10
Puzzle 3	9/10	10/10
Puzzle 4	5/10	10/10
Puzzle 5	5/10	10/10

BS: Subjects able to recognize the object before solving.
AS: Subjects able to recognize the object after solving.

8. Conclusions

We have introduced a novel approach to create continuous line illustrations from input images, based on edge detection and graph Semi-Eulerization. Moreover, our method allows users to specify the amount of detail shown on our line illustrations. The results portray rather smooth continuous lines that contain important features such as curls, cusps and self-intersections. Also, we have demonstrated that continuous line illustrations created through our approach resemble the objects depicted in the original images in a visually plausible manner, and shown a possible application of these illustrations to the generation of connect-the-dots puzzles.

Our approach is limited by the current state of edge detection algorithms, which are prone to subdetection and overdetection of lines. Even with our level-of-detail approach, this incurs in line cluttering at some areas of the illustration, while lacking enough detail in some others.

We plan to further extend this work by introducing visual attention elements [IKN98] to decide which segments should be considered as part of the continuous line illustration, as well as determining automatically which areas should be free of line cluttering. We are also working on improving the visualization of these drawings by introducing additional effects such as the shading of certain areas by controlling the density of line segments, varying the line width at certain parts, and line haloing [EBRI09].

Acknowledgements

We are grateful to all the participants that took part on our user study, to anonymous reviewers for their helpful comments and to Rachel Ann Lindsay for allowing us to show her work in this paper. This work has been partially supported by Japan Society for the Promotion of Science under Grants-in-Aid for Scientific Research (B) No. 20300033 and No. 21300033.

References

[BH04] BOSCH R., HERMAN A.: Continuous line drawings via the traveling salesman problem. *Operations Research Letters 32*, 4 (2004), 302–303. 2, 7

[cga] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org. 6

[CR74] CATMULL E., ROM R.: A class of local interpolating splines. In *Computer Aided Geometric Design* (1974), pp. 317–326. 6

[EBRI09] EVERTS M. H., BEKKER H., ROERDINK J. B. T. M., ISENBERG T.: Depth-dependent halos: Illustrative rendering of dense line data. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1299–1306. 8

[EJ73] EDMONDS J., JOHNSON E. L.: Matching, euler tours and the chinese postman. *Mathematical Programming 5* (1973), 88–124. 4

[Fle83] FLEURY: Deux problemes de geometrie de situation. *Journal de Mathematiques Elementaires* (1883), 257–261. 4, 5

[Fri01] FRITZ L. L.: *250 Continuous-Line Quilting Designs for Hand, Machine & Long-Arm Quilters*. C&T Publishing, Inc., 2001. 1

[HWL03] HUANG L., WAN G., LIU C.: An improved parallel thinning algorithm. In *ICDAR '03: Proceedings of the Seventh International Conference on Document Analysis and Recognition* (Washington, DC, USA, 2003), IEEE Computer Society, p. 780. 3

[IKN98] ITTI L., KOCH C., NIEBUR E.: A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence 20*, 11 (1998), 1254–1259. 8

[KB05] KAPLAN C. S., BOSCH R.: TSP Art. In *Proceedings of Bridges 2005, Mathematical Connections in Art, Music and Science* (2005), pp. 301–308. 2, 7

[KLC09] KANG H., LEE S., CHUI C. K.: Flow-based image abstraction. *IEEE Transactions on Visualization and Computer Graphics 15*, 1 (2009), 62–76. 3

[Lin10] LINDSAY R. A.: Rachel Ann Lindsay - illustrator, 2010. http://www.rachelannlindsay.com/. 2, 6, 7

[Loh09] LOHMAN S.: LineArtGallery.com, 2009. http://www.lineartgallery.com/. 1

[Mor05] MORALES J. E.: Virtual Mo, 2005. http://www.virtualmo.com/. 1, 7

[Nic90] NICOLAÏDES K.: *The Natural Way to Draw*. Houghton Mifflin Company, Boston, Massachusetts, USA, 1990. 1

[PS06] PEDERSEN H., SINGH K.: Organic labyrinths and mazes. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering* (New York, NY, USA, 2006), ACM, pp. 79–86. 2

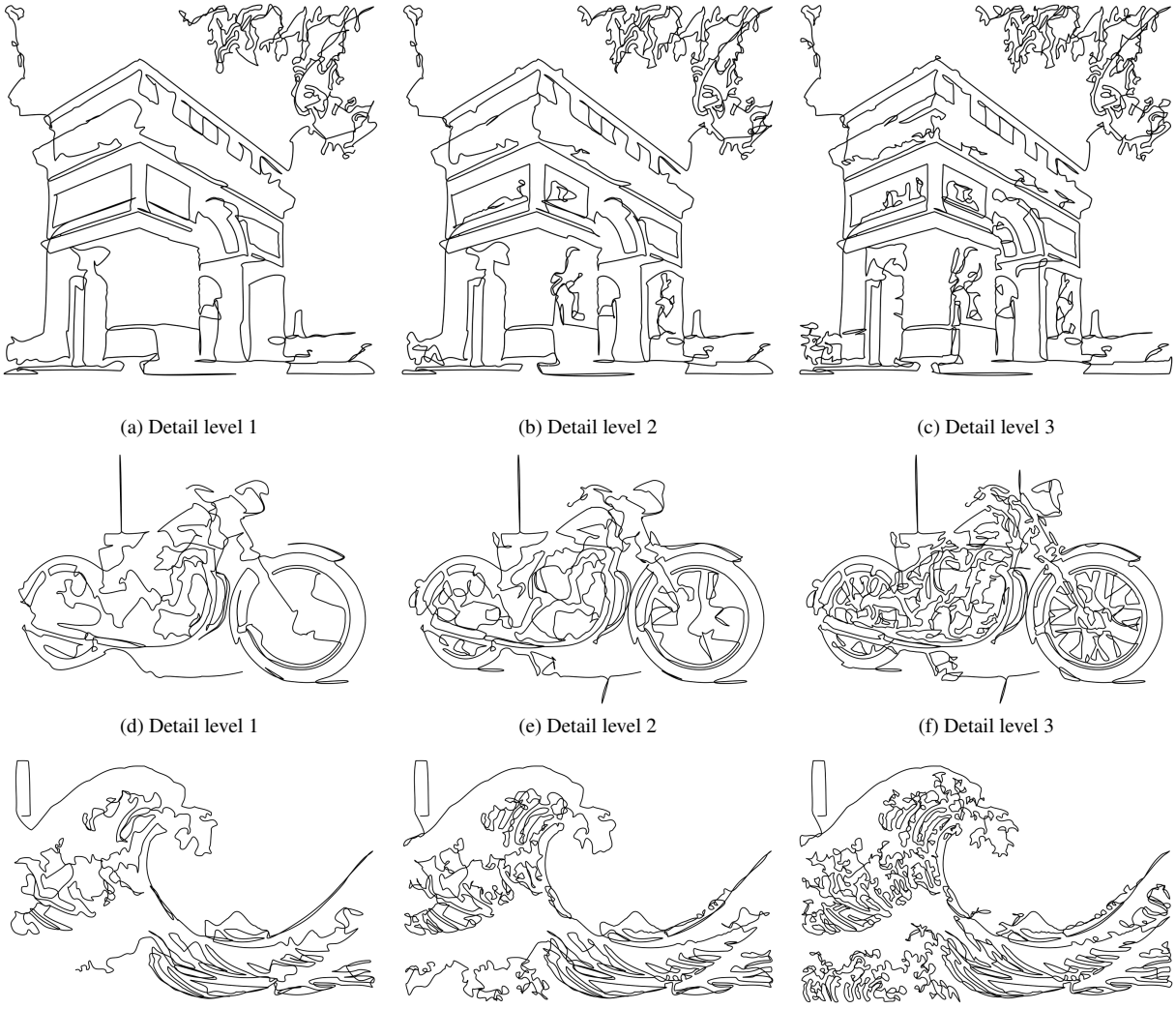
[PvV05] PHAM T. Q., VAN VLIET L. J.: Separable bilateral filtering for fast video preprocessing. In *Proc. IEEE Conf. Multimedia and Expo (ICME)* (2005), pp. 1–4. 3

[Sab09] SABLE P.: Single line artwork by Pamela Sable, 2009. http://www.pameline.com/. 2, 6

[Sla01] SLATER G.: Geoff Slater: Contemporary artist, 2001. http://www.geoffslater.com/. 1

[TM98] TOMASI C., MANDUCHI R.: Bilateral filtering for gray and color images. In *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision* (Washington, DC, USA, 1998), IEEE Computer Society, pp. 839–846. 2

[WT09] WONG F. J., TAKAHASHI S.: Flow-based automatic generation of hybrid picture mazes. *Computer Graphics Forum 28*, 7 (2009), 1975–1984. 2

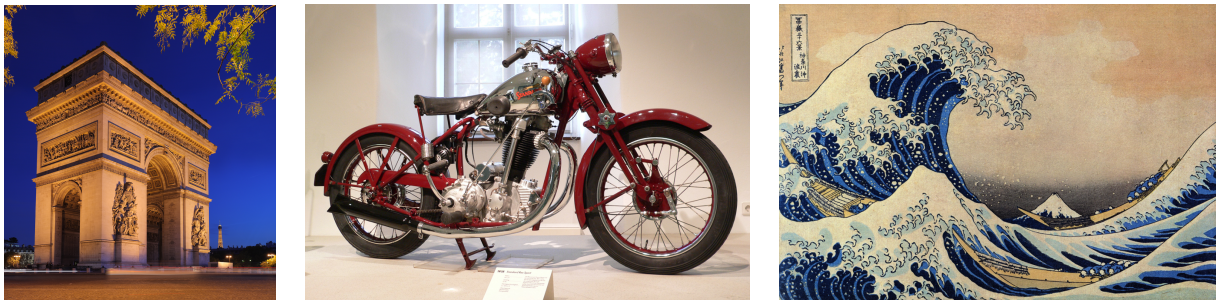


(a) Detail level 1 (b) Detail level 2 (c) Detail level 3

(d) Detail level 1 (e) Detail level 2 (f) Detail level 3

(g) Detail level 1 (h) Detail level 2 (i) Detail level 3

Figure 10: Some of our continuous line illustration results with varying detail levels.



(a) Arc de Triomphe (b) Motorcycle (c) Japanese Ukiyoe Painting

Figure 11: Original images used for generating our results.