Automatic 3D Mesh Generation from a Single Hand-Drawn Sketch

Nishant Shukla
University of California, Los Angeles
CS 268: Final Project
nxs@ucla.edu

Abstract

Reconstructing a watertight 3D model from a doodle is a severely ill-constrained problem. Since humans have little to no trouble understanding such sketches, this paper provides insights into human perception and geometric reasoning. Most notably, a novel approach to utilize a symmetry prior potentially reveals how humans interpret the most likely 3D representation of an image.

1. Introduction

With the prevalence of 3D printers, designing meshes has become an increasingly important skill. Unfortunately, learning how to use Computer Aided Design (CAD) software involves significant training, so we instead invent a way to automatically generate the most likely 3D mesh from a single hand-drawn sketch.

The process described in this paper is robust to a noisy depiction of an object, whether it be a hand-drawn sketch or the output of a Canny edge-detector. Most importantly, this pipeline is robust to self-occlusion of faces. For instance, the hidden faces of an octahedron can be correctly synthesized by utilizing a symmetry prior.

The contributions of this paper lie beyond 3D printing. With the proposed pipeline, a robot system can reconstruct watertight objects in a scene for collision avoidance and planning. Moreover, the generated mesh can be used for object detection, since it provides more cues than the original 2-dimensional image.

1.1. Related Work

Early works in line drawing interpretation [7] assume a graphical representation of the object is given. These techniques use relaxation to find satisfying interpretation of vertices to overcome the combinatorial search. Although highly accurate, the algorithms only make observations about visible vertices. Moreover, processing an arbitrary drawing into a usable graph-based wireframe is not considered.

Naya *et al.* [4] designed a technique to convert a hand-drawn sketch to 3D, but each vertex is required to be visible. This limitation forces one to explicitly specify all vertices and faces, crowding the drawing with too much detail. Since the location of hidden vertices must be known beforehand, this approach cannot be used on self-occluding images, such as outputs of edge-detectors.

Shesh and Chen [6] developed a system that can infer hidden vertices of simple 3D extrusions. However, inferring arbitrarily hidden vertices, such as the vertices of a pyramid, require consistent human interactions. Their system is a useful tool for interactively constructing a 3D mesh, but does not provide automatic generation other than of simple extrusions.

1.2. Contributions

The contributions of this paper include the following.

1. An efficient least-squares algorithm to fit a continuous piecewise linear function
2. A formulation for estimating the most likely topology of occluded faces using a symmetry prior

2. Method

Our process of converting a hand-drawn sketch to a 3D model is a multi-stage hierarchical pipeline. The first stage binarizes an input image to produce a set of relevant points to work with. Each following step of the pipeline interprets n-dimensional data into an $n+1$-dimensional structure, until finally a 3D mesh is generated. The following sections detailed each step from start to finish.

2.1. From Image to Points

Given an image of a hand-drawn sketch (Figure 1-A), we would like to identify and extract only the pixels associated with the sketch. Hand-written markings naturally have crisp distinctions between the ink and the background. We exploit this property by employing adaptive thresholding to binarize the image.

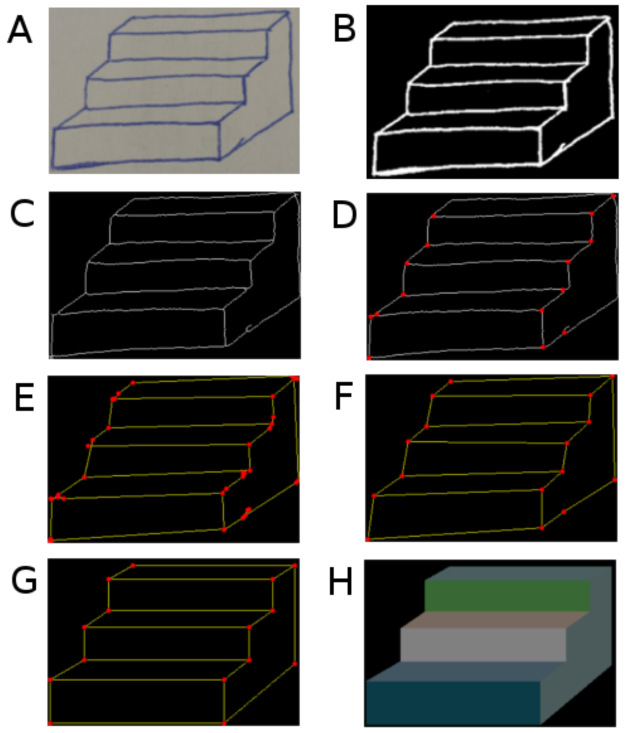

Figure 1. An overview of the image processing steps. (A) Raw input of a drawing on a whiteboard. (B) Binarized image from adaptive thresholding. (C) Line thinning result. (D) Detection of initial nodes. (E) Binary search and curvature detection to find more nodes. (F) Removal of redundant nodes. (G) Straighted edges. (H) Detection of 3D faces.

Each pixel is compared to a constant offset of the mean of its 101×101 neighborhood. We make the assumption that the number of pixels associated with the background outnumbers the number of pixels associated with the sketch. The background pixels are set to black, and the foreground pixels are set to white, as seen in Figure 1-B.

The intended geometry of a drawing is invariant of line-thickness, so we incorporate the Hilditch thinning algorithm [3] to canonicalize the thickness of strokes (Figure 1-C). To account for noise caused by adaptive thresholding or line-thinning, only the pixels of the largest contour are considered relevant.

2.2. From Points to Lines

The 0-dimensional points accumulated in the previous step are assembled into 1-dimensional line segments. The objective of this stage in the pipeline is to construct a graphical representation of the hand-drawn sketch. A graph data structure ensures that line segments are not independently floating in space. Each node of the graph represents the vertices of line-segments, and each edge of this graph represents a best-fit line.

Since the strokes are each one pixel thick, the first few candidate vertices are points with three or more neighbors. With these starting nodes, we trace a hand-drawn stroke from one node to the next. The direction (in radians) being traversed is denoted by $\theta(t)$. If the path of this ray appears to suddenly curve, a new node is created at the moment of curvature. Curveness is defined as the second derivative of direction $d^2\theta/dt^2$. For instance, a straight line never changes direction $d\theta/dt = 0$. A sudden change in direction at time n is detected when

$$\sum_{t=0}^{n+1} \frac{d^2}{dt^2}\theta(t) \geq \frac{\pi}{4} \tag{1}$$

With these initial guesses of vertices, we use a least-squares approach to fit lines between these nodes. If the points connecting two nodes appear linear, an edge is established between the two nodes. Otherwise, new nodes will be discovered on the path to build a piecewise linear path. This process continues until a graph is generated that fully describes all the points.

One of the core contributions of the paper is an efficient algorithm to detect a change in linear model on a path. Tracing connected pixels from one node to another accumulates an ordered list of points p_i for $i = 1, ..., N$. We define q_i as the corresponding projection of p_i onto the line formed by p_0 and p_N. The cost of a list of points p_i for $i = a, ..., b$ is defined as follows.

$$Cost(p, a, b) = \frac{1}{b-a} \sum_{i=a}^{b} \frac{||q_i - p_i||^2}{||q_i - p_a||} \tag{2}$$

This formulation penalizes small line-segments. Given a list of points p_i for $i = 1, ..., N$, we use the cost function to determine whether adding another vertex on the path would improve fitting. These hidden vertices are found by binary search, as laid out in Algorithm 1. Faster than traditional approaches, the runtime is $O(k \log(N))$, where k is the number of linear models, which can be adjusted by a tuning parameter λ.

Figure 2 shows a qualitative comparison between using this proposed cost function and using a traditional least squares approach. In general, our algorithm produces results matching more closely to the ground truth.

2.3. From Lines to Faces

The next step in the pipeline is to interpret 1-dimensional line-segments as 2-dimensional shapes. The previous step provided a graphical data structure representing vertices and line-segments. However, the vertices are typically over-fitted, producing more nodes in the graph than necessary. A larger graph is more prone to error, so it is essential to simplify the graph as much as possible.

```
FindModelChange($p$):
Data: List of points $p_i$ for $i = 1, ..., N$
Result: Estimate best-fit piecewise linear function
$a = 0; b = N$
while $a \neq b$ do
    mid = $(a + b)/2$
    $a_{err} = Cost(p, a, mid)$
    $b_{err} = Cost(p, mid, b)$
    if $a_{err} > b_{err}$ then
        $b$ = mid;
    else
        $a$ = mid
    end
end
/* b is a newly found vertex        */
if $b_{err} > \lambda$ then
    FindModelChange($p_{mid:N}$);
end
```

Algorithm 1: Detect hidden vertices to form a best-fit piecewise linear function. $\lambda > 0$ is a tuning parameter.

First, nodes are deemed redundant and are removed if they lie collinear to their neighbors. A new edge is created between the two neighbors. This process repeats until no more modifications are mode to the graph.

Next, nearby nodes are clustered together to relieve redundancy. K-means clustering is out of the question since the number of clusters may range widely and mistakes in guessing the wrong number of clusters is costly. Instead, we scan each vertex to record all neighboring nodes located within a threshold distance ϵ. For efficiency, a k-d tree is preprocessed to examine only nearby nodes. If multiple nodes exist within ϵ distance, the nodes are merged into a newly created node at the centroid.

Now that the number of vertices is settled, we shift our attention to straightening the edges. If the slope of an edge is nearly horizontal or vertical, then one of the vertices is drifted to snap the edge straight. This process repeats until no more edges require straightening.

If a cycle in a graph contains no other cycles, it is a minimal cycle. All minimal cycles of our graph correspond to visible surfaces of the underlying mesh. We use Paton's algorithm [5] to detect cycles, and record all cycles that do not contain another cycle. This process completely identifies all visible 2-dimensional faces.

2.4. From Faces to Surfaces

The 2-dimensional faces detected in the previous step will infer the most likely 3-dimensional mesh. We exploit topological symmetry to synthesize faces of the mesh that are not visible due to self-occlusion. The technique in this section describes another main contribution of the paper.

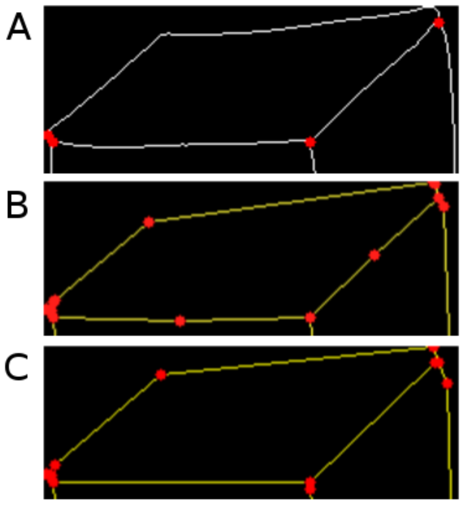

Figure 2. A comparison of finding hidden vertices. (A) A drawing with initial vertices detected. (B) The graph produced by using traditional least squares approach to find hidden vertices. (C) The graph produced by instead using our cost function to find hidden vertices.

The graph we have generated so far demonstrates how the vertices are related to each other by connected line-segments. We now construct the dual of this vertex-line graph, where the nodes are minimal cycles and edges connect neighboring cycles. This face-neighbor graph encapsulates the surface topology of the underlying 3D mesh as seen in Figure 3-B.

We call this initial dual graph G the *generator*. The nodes of degree 1 are called *open* nodes, while all other nodes are called *closed*. A face-neighbor graph F is considered *stable* if there are no open nodes, as seen in Figure 3-D.

Let's say a planar graph F is *valid* if the union of all subgraphs of F which are isomorphic to G are equivalent to F. In other words, let $\hat{F}$ be a set of all subgraphs isomorphic to the generator G. If the union of each $S \in \hat{F}$ is isomorphic to the original planar graph, then it is valid.

$$\hat{F} = \{S \mid S \subseteq F, S \cong G\} \tag{3}$$

$$\bigcup_i \hat{F}_i \cong F \Rightarrow \text{Valid} \tag{4}$$

The topology filling algorithm starts from graph G, which is innately valid but not stable, and applies constructive graph operations until it reaches a configuration that is both stable and valid. The goal is to achieve the smallest possible stable and valid graph. As per extensions of Steinitz's theorem, such a graph defines a valid polyhedron of spherical topology [2].

The approach we employ performs graph operations at each step that persist graph validity using VFLib [1], an ef-

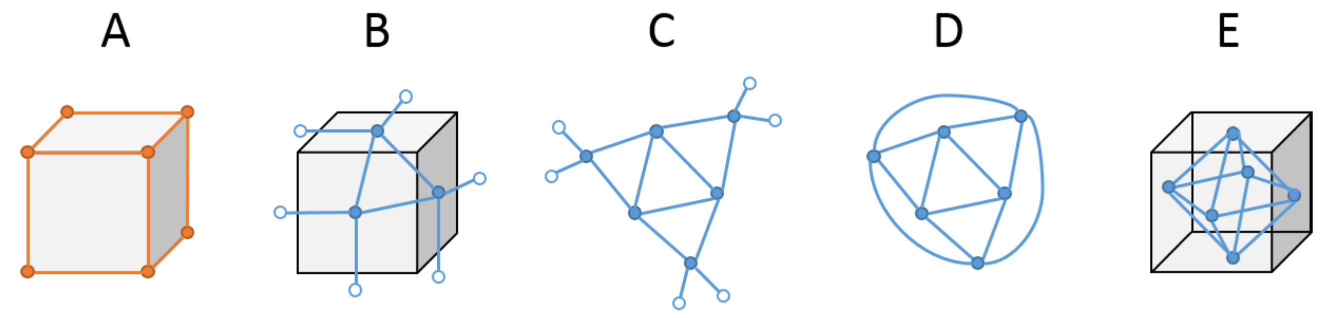

Figure 3. A step-by-step diagram showing how to synthesize the full topology of a 3D mesh. (A) A primary vertex-line graph. (B) Its dual face-neighbor graph. (C) A valid but unstable graph during the topological filling algorithm. (D) The final topology discovered. (E) Corresponding faces from synthesized topology.

ficient subgraph matching library. The algorithm finishes at the moment a stable graph is discovered. The dual of the resulting face-neighbor graph specifies the topology of the original vertex-line graph, as seen in Figure 3-E.

If the face-neighbor graph is not connected, we examine each connected component independently, as demonstrated in Figure 4. This allows us to resolve the topology of substructures in parallel. To reassemble the mesh, relaxation may be used to achieve a configuration that minimizes the geometric variance between components of identical generators. For instance, the position of the legs of the table must be at the four corners of the surface in order for the legs to have the least variance in geometry.

2.5. From Surfaces to 3D Object

The final step is to generate a 3D mesh. Our approach lends itself to seek a boundary representation of the solid as opposed to Constructive Solid Geometry (CSG).

First, the x, y, and z axes are identified. For simplicity, we assume the x-axis lies horizontally and the y-axis lies vertically. The direction of the z-axis is assumed to be the mode of the directions of all diagonal lines. The mode is calculated by classifying lines into eight equal bins.

Knowing the coordinate axes enables assigning every vertex a z-value. We list each vertex and its x, y, and z values using the Wavefront OBJ format.

```
v 0 0 0
v 0 10 0
...
```

The face topology lets us enumerate all faces, which we specify in clockwise order to adhere to the Wavefront OBJ format.

```
f 1 2 3 4
f 1 4 5 6
...
```

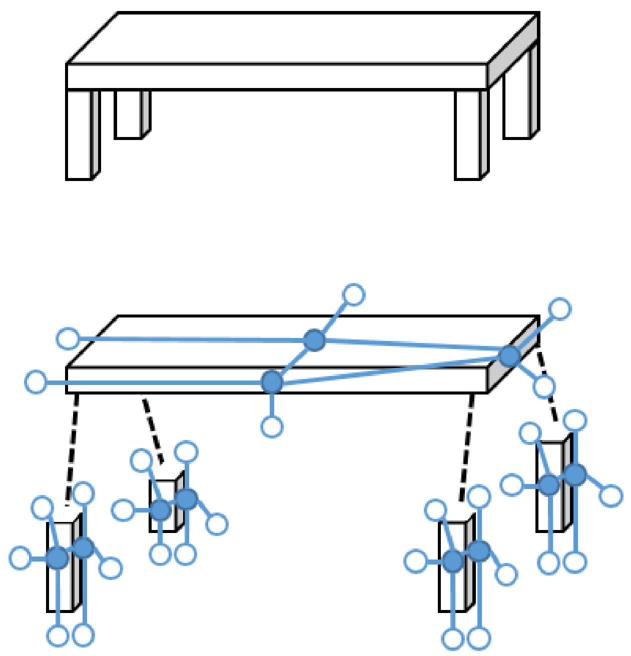

Figure 4. Automatic disassembly and reconstruction of a table.

Finally, the produced file completely specifies a 3D surface, and can be opened in any CAD software. Due to the topological constraints, the mesh is also watertight.

References

[1] L. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10):1367–1372, 2004.

[2] D. Eppstein and E. Mumford. Steinitz theorems for orthogonal polyhedra. In *Proceedings of the Twenty-sixth Annual*

Symposium on Computational Geometry, SoCG '10, pages 429–438, New York, NY, USA, 2010. ACM.

[3] L. Lam, S.-W. Lee, and C. Y. Suen. Thinning methodologies-a comprehensive survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(9):869–885, Sept. 1992.

[4] F. Naya, J. A. Jorge, J. Conesa, M. Contero, and J. M. Gomis. Direct modeling: From sketches to 3d models. In *Proc. of the 1st Ibero-American Symposium in Computer Graphics*, pages 109–117, 2002.

[5] K. Paton. An algorithm for finding a fundamental set of cycles of a graph. *Commun. ACM*, 12(9):514–518, Sept. 1969.

[6] A. Shesh and B. Chen. Smartpaper: An interactive and user friendly sketching system, 2004.

[7] D. Waltz. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, page pages. McGraw-Hill, 1975.