A general and efficient method for finding cycles in 3D curve networks

Yixin Zhuang[1,2] Ming Zou[2] Nathan Carr[3] Tao Ju[2]

[1]National University of Defense Technology, China
[2]Washington University in St. Louis, USA
[3]Adobe, USA

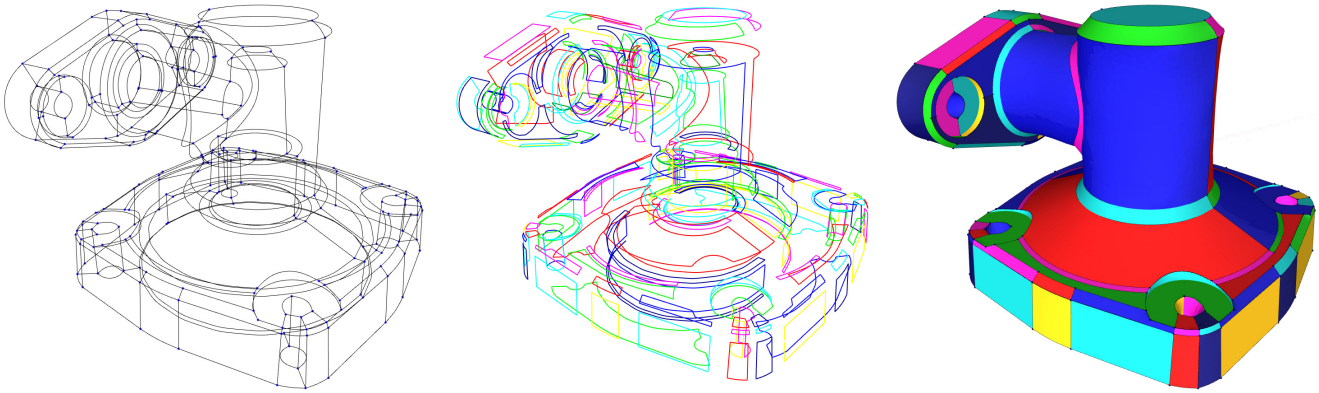

Figure 1: *A curve network (*misc2*) representing a genus-7 mechanical part (left), cycles found by our algorithm (middle), and surface patches generated from the cycles (right). The curve network contains 410 curves. Our algorithm completed in half a second.*

Abstract

Generating surfaces from 3D curve networks has been a longstanding problem in computer graphics. Recent attention to this area has resurfaced as a result of new sketch based modeling systems. In this work we present a new algorithm for finding cycles that bound surface patches. Unlike prior art in this area, the output of our technique is unrestricted, generating both manifold and non-manifold geometry with arbitrary genus. The novel insight behind our method is to formulate our problem as finding local mappings at the vertices and curves of our network, where each mapping describes how incident curves are grouped into cycles. This approach lends us the efficiency necessary to present our system in an interactive design modeler, whereby the user can adjust patch constraints and change the manifold properties of curves while the system automatically re-optimizes the solution.

CR Categories: I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Display Algorithms

Keywords: curve networks, wireframe, sketch-based modeling, lofting, dynamic programming

Links: DL PDF

1 Introduction

Creating surfaces from a network of 3D curves, often known as *lofting* or *skinning*, is a classical and fundamental problem in computer-aided design (CAD). In a traditional CAD pipeline, engineers start by defining a wireframe of the desired model, which then needs to be turned into a surface representation for analysis, simulation, and manufacturing. More recently, sketching tools [Wesche and Seidel 2001; Bae et al. 2008; Schmidt et al. 2009; Grimm and Joshi 2012] offer intuitive means for drawing 3D curves using 2D input devices, allowing artists to quickly come up with concept designs. Even though these sketches may not be used for defining the final product, surface visualization could help users better appreciate their design [Abbasinejad et al. 2011; Bessmeltsev et al. 2012].

An important first step in lofting is identifying a cycle of curves in the network that bounds an individual surface patch. Although humans can quickly "see" such cycles in a typical curve network created by a designer, the actual process of selecting curves and grouping them into cycles can be rather tedious. The process may be significantly shortened if the computer can suggest most, if not all, of the cycles, requiring only minimal input from the user in the form of quick inspection and small adjustments.

Finding cycles automatically is a challenging task. First of all, there is usually a huge number of cycles to choose from in a curve network [Biondi et al. 1970]. Second, while the complete perceptual model of how humans identify the "good" cycles is not yet well-understood, it is evident that we evaluate the goodness of one cycle not in isolation, but in the context of other cycles. Consider the example in Figure 1 (left): among the many circular rings in the network, only a few of them describe faces of the model while others are cross-sections or ends of cylindrical shafts. Last but not least, the specific application context often imposes additional constraints on the cycles. For example, wireframe designs of mechanical models usually define closed manifolds (often times with high genus, like the one in Figure 1), meaning that each input curve needs to be

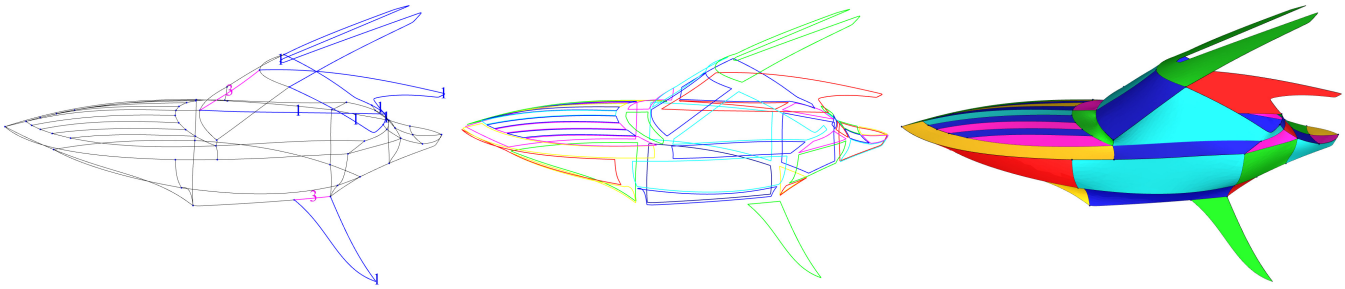

Figure 2: *A sketched design containing non-manifold curves with prescribed degrees (left), cycles found by our algorithm (middle), and surface patches generated from the cycles (right). The curve network contains 102 curves. Our algorithm completed in 0.1 second.*

used exactly twice in the resulting cycles. These factors make cycle discovery a highly complex search problem.

Many algorithms for cycle discovery have been proposed in the past few decades (see a more detailed discussion in Section 2). To tame the search complexity, they rely on stringent assumptions on either the geometry or the topology of their output. Some methods only look for planar cycles, while others are usually limited to cycles that define either a manifold surface with genus 0 or an open, non-manifold surface that does not bound any solid space. It remains difficult to model surfaces with both interesting geometry (e.g., with highly curved faces) and complex topology (e.g., a manifold with high genus) in an efficient manner.

Contribution We present a new algorithm for finding cycles that bound potential surfaces in a curve network. Following previous works on this problem, we take as input a "clean" network, in the sense that a curve does not self-intersect and no two curves intersect unless at their joining vertices.

Unlike existing methods, the output of our algorithm is not limited to any particular class of geometry or topology. The cycles can be highly non-planar, and the surface defined by these cycles can be either manifold or non-manifold and have an arbitrary genus. In addition, the user can exactly control the non-manifold topology of the output by prescribing the degree of each non-manifold curve in the input network (see Figure 2). The algorithm is highly efficient and capable of achieving interactive speeds even on large networks (like the one in Figure 1).

The key idea behind our method, which lends us efficiency without sacrificing generality, is exploring a different search space. While previous methods search in the space of cycles, our algorithm looks for mappings locally at a vertex or a curve that describes how the incident curves are grouped into cycles. A collection of such mappings over all vertices and curves, called a *routing system*, uniquely determines a set of cycles, and vice versa. The advantage of considering the routing system is that the number of mappings at each vertex or curve is far fewer than the number of cycles in a network.

We formulate the problem of finding an optimal set of cycles as finding a routing system that minimizes the sum of costs associated with the mappings. Our cost functions capture both the quality of individual cycles and the compatibility between adjacent cycles, even though we never explicitly find a cycle. The optimal routing system can be found using a simple dynamic programming search. While the full search has exponential cost, we observe that, for a typical curve network, the optimal solution can usually be found very efficiently by considering only the few best mappings at each vertex and using a banded search.

We tested our algorithm on an extensive suite of data that includes CAD wireframes, sketched curves, and synthetic examples. The algorithm runs within a second for every input, and it produces desirable results for the majority of them. We also designed a interactive tool for prescribing degrees of non-manifold curves, viewing surface patches, and correcting poor cycles. The efficiency of our algorithm allows immediate feedback during these interactions.

2 Related Work

Here we attempt to categorize the literature on cycle discovery by the search strategies used, with an emphasis on the assumptions used by each strategy and the implication on their applicability. Note that many of the early methods take as input a 2D projection of the curve network without 3D coordinates. In the following, these methods are discussed together with those taking 3D inputs without special distinction.

Planar graph embedding If the curve network encodes a planar graph that is 3-connected, the cycles can be found efficiently by computing the (unique) planar embedding of the graph [Hanrahan 1982; Dutton and Brigham 1983]. For curve networks whose graphs are only 2-connected (but still planar), the planar embedding is not unique, and heuristic searches have been proposed that seek an optimal embedding guided by geometric criteria [Shpitalni and Lipson 1996; Inoue et al. 2003]. However, these methods require the input network to encode a planar graph and always produce cycles that define a genus-0 manifold surface. Extension of these methods to surfaces with non-zero genus is difficult, since there are no efficient algorithms for high-genus graph embedding. Also, it is not clear how the embedding approach can be extended to produce non-manifold outputs, which are often the case for conceptual designs (see Figure 2).

Decomposition Agarwal and Waggenspack [1992] proposed an approach that decomposes the solid object represented by a curve network into tetrahedra and merges the faces of those tetrahedra into cycles. The algorithm can be extended to also handle simple non-manifold configurations. However, since the algorithm relies on mostly topological information (e.g., degree at each junction), the method may easily create erroneous output on geometrically ambiguous models, as indicated in [Liu et al. 2002].

Cycle basis A cycle basis is a minimal set of cycles in a graph such that any cycle not in the basis can be constructed by the *ring sum* of some cycles in the basis. The algorithms of Ganter and Uicker [1983] and of Brewer and Courter [1986] start with an initial cycle basis, which can be efficiently computed (using spanning trees) but may contain undesirable cycles (such as cross-sectional ones), then perform a greedy search for a better basis. A slightly different approach is taken by Bagali and Waggenspack [1995] and, more recently, by Abbasinejad et al. [2011]. Starting from either the complete set of cycles or a pruned subset, they use a greedy algorithm to construct the optimal cycle basis.

Current cycle-basis methods have inherent difficulty in modeling closed, high-genus shapes. A cycle basis defines a non-manifold that does not bound any solid space. While additional cycles can be added to "close off" solids via heuristics, this remedy cannot be applied to surfaces with non-zero genus. In a network representing a closed manifold model with genus $g > 0$, a cycle basis has $2g - 1$ *more* cycles than the faces of the model. None of the existing methods address the deletion of redundant cycles, which has to be done manually by the user. Furthermore, while cycle-basis methods can produce non-manifold outputs, it is not clear how one could enforce specific curve degrees while constructing a basis.

Heuristic cycle search Another popular approach is to perform an exhaustive search over the space of all cycles, guided by some form of geometric cost and/or topological constraints (e.g., degree of each curve). To reduce the search complexity, methods in this class only identify cycles with a limited variety of geometry, such as planar cycles [Markowsky and Wesley 1980; Liu et al. 2002; Varley and Company 2010] or cycles bounding low-degree algebraic surfaces [Sakurai and Gossard 1983]. While cycles in CAD wireframe mostly fall into these types, curve sketches usually have more geometric variation (see Figure 10). Often times additional pruning criteria are used that are specific to 2D inputs, such as requiring a projected cycle to not self-intersect [Shpitalni and Lipson 1996; Liu and Lee 2001].

3 Overview

The input to our algorithm is a 3D curve network made up of *curves* joining at *vertices*. As mentioned earlier, we assume the curves are free of self-intersections. The *degree* of a vertex refers to the number of its incident curves. We refer to a *cycle* as a closed walk in the graph. A cycle is *simple* if no vertex or curve appears more than once in the walk. Given a set of cycles, the number of times that a curve is used in these cycles is called the *degree* of that curve.

If a non-manifold model is desired, our algorithm also requires the user to prescribe the degree of each non-manifold curve in the resulting cycles. We call the prescribed degree of a curve the curve's *capacity*. Capacity gives users exact control over the non-manifold topology of the output. Our algorithm assumes every curve has capacity 2 unless specified by the user. As a result, no user input is needed if the desired output is a closed, manifold model.

It is not always possible to find a set of cycles that meet a given set of capacity constraints. Even if such cycles exist, they may not all be simple. An example of a trivially non-simple cycle is one that visits the same curve twice in a row in opposite directions, like making a U-turn. Such cycle does not bound any meaningful surface. In Appendix A, we prove that the capacity constraints need to satisfy two mild conditions to guarantee the existence of a set of cycles that are free of U-turns: the sum of capacity of curves incident to a vertex v is even, and the capacity of any curve incident to v is no greater than the sum of the capacities of the remaining curves incident to v.

Given a curve network and capacities that meet the above conditions, our goal is to compute a set of cycles that define the most plausible surface patches while satisfying the capacity constraints. We first introduce an alternative representation of cycles, called a routing system (Section 4), that is equivalent to any set of cycles meeting the capacity constraints. With this representation, we give a new formulation of optimal cycles and develop an efficient search algorithm (Section 5). Finally we introduce an interactive tool that allows users to easily specify non-manifold capacity, correct cycles, and visualize surface patches (Section 6).

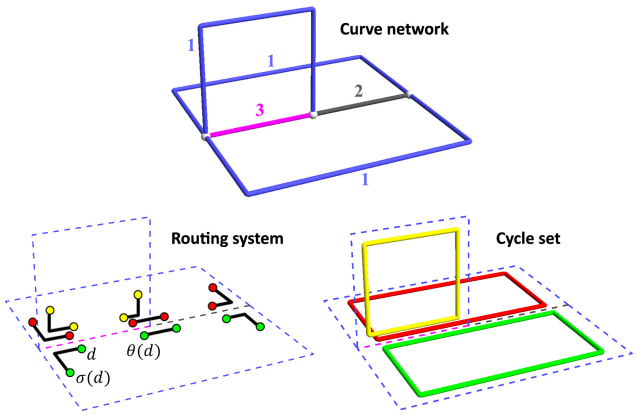

Figure 3: *Top: a curve network with capacities (curves are colored by their capacity). Bottom-left: a routing system (darts are drawn as circles, corner-mapped darts are connected by edges, and bridge-mapped darts share the same color). Bottom-right: the cycle set determined by the routing system.*

4 Routing system

We take a local approach to describe how curves are grouped into cycles. Informally, there are two types of groupings. Two curves $\{a, b\}$ incident to a common vertex form a *corner* if $\{a, b\}$ are consecutive in one cycle. Two corners $\{a, b\}, \{b, c\}$, one at each end of a curve b, form a *bridge* if $\{a, b, c\}$ are consecutive in one cycle. Note that the same curve may appear multiple times in a corner or a bridge. We call this new representation of cycles, consisting of corners and bridges, a *routing system*.

More formally, we associate each end of a curve with k instances of the curve, called *darts*, where k is the capacity of the curve. So there are a total of $2k$ darts for each curve. The corners and bridges can be written as mappings $\{\sigma, \theta\}$ that act on these darts and have the following properties:

1. σ (the corner mapping) maps a dart at a vertex to a different dart at the same vertex.
2. θ (the bridge mapping) maps a dart at one end of a curve to one of the darts at the other end of the same curve.
3. Both are involutions, i.e., $\sigma(\sigma(d)) = d$ and $\theta(\theta(d)) = d$.

Figure 3 shows an example of a routing system (bottom-left) for a curve network with capacity constraints (top). The darts are shown as round dots. A dart d is connected to its corner-mapped dart $\sigma(d)$ by a black segment, and shares the same color as its bridge-mapped dart $\theta(d)$.

A routing system uniquely determines a set of (possibly non-simple) cycles that meets the capacity constraints. Consider the graph whose nodes are the darts and whose edges connect darts that are mapped to each other by σ or θ. Since each node, representing a dart d, has exactly two edges, one to $\sigma(d)$ and one to $\theta(d)$, the graph is made up of disjoint circuits. Furthermore, since each curve with capacity k generates k pairs of darts, the curve appears in the circuits for exactly k times. Figure 3 bottom-right shows the cycle set determined by the routing system on the left. Conversely, any set of cycles uniquely determines a routing system, up to permutations of the darts at each end of a curve.

The routing system is a variation of the classical *rotation system* for graph embedding [Mohar and Thomassen 2001]. A rotation system is also defined by darts and mappings $\{\sigma, \theta\}$, except that

each graph edge creates only 2 darts, one at each end of the edge, and σ is a permutation among darts at a same vertex (rather than an involution). A rotation system determines a set of cycles that forms faces of a manifold, orientable surface. In contrast, a routing system can express surfaces that are neither manifold nor orientable.

5 Algorithm

We seek a routing system whose corresponding cycles bound plausible patches. The advantage of optimizing the routing system over directly optimizing the cycle set is two-fold. First, each variable in a routing system is local and has a much smaller range of choices than a variable in a cycle set. While the number of cycles grows exponentially with the size of the entire network, the number of possible corner mappings at a vertex (or possible bridge mappings at a curve) only depends on the degree at the vertex (or the capacity at the curve). Second, while any routing system automatically satisfies the capacity constraints, such constraints would have to be enforced when explicitly searching for cycles, which further increases the complexity of search.

We first discuss our choice of metrics to evaluate a routing system (Section 5.1). While we never explicitly construct a cycle, these metrics locally reflect both the quality of an individual cycle and the compatibility of adjacent cycles. We next describe an efficient algorithm for finding the optimal routing system by dynamic programming (Section 5.2). We then present a simple post-process procedure to rectify the results when our metrics fail to characterize the desired cycles (Section 5.3).

5.1 Cost metrics

Unlike most of the previously proposed metrics that evaluate an entire cycle, we need more localized metrics that can assess the likelihood of two curves (i.e., a corner) or three curves (i.e., a bridge) to be in a cycle. In our method we evaluate bridges instead of corners, since bridges involve a larger neighborhood and hence have more geometric information.

We consider two types of metrics. The first metric, the *intra-bridge* cost, acts on a single bridge and favors the bridge that borders a smooth and convex patch. The second metric, the *inter-bridge* cost, acts on a group of bridges and favors the group whose corresponding patches meet at a curve with small normal discontinuity. The inter-bridge cost is important for avoiding cycles that represent cross-sections rather than actual faces of the model, since surfaces bounded by cross-section cycles usually form sharp angles with surfaces bounded by surrounding cycles. The computation and rational of these costs are detailed below.

Intra-bridge cost Consider a bridge that involves a sequence of three curves $\{a, b, c\}$ (note that, as mentioned before, some of these curves could be identical). Since we desire simple cycles, we first test whether there is a path in the curve network connecting curves a and c without visiting b or its end vertices. If such a path does not exist, the intra-bridge cost is set to infinity. Otherwise, the cost is computed based on the geometry of the bridge, as follows.

Popular cost functions that have been used in the past include the length of a cycle (the shorter the better) and its planarity (the planar the better). However, computing length requires the knowledge of a complete cycle, which is exactly what our method is trying to avoid. Furthermore, cycles in a curve network are often far from being planar, and many planar cycles are not desirable cycles (e.g., those bounding cross-section faces).

Our cost functions are motivated by the following observations. First, designers typically place curves along *flow-lines* of the intended surface, which are strongly correlated with sharp features and lines of curvature [Bessmeltsev et al. 2012]. This implies that a desirable cycle should bound a surface that does not vary more than the curves in the cycle. Second, shorter cycles are often more "convex", that is, they bound patches that form small interior angles at the vertices. The correlation can be made clear in the common scenario when the cycle bounds a developable patch [Rose et al. 2007] and when each curve is a geodesic of the patch (e.g., a flow-line): the sum of interior angles over all vertices in the cycle is exactly $(n-2)\pi$ where n is the number of curves in the cycle.

Our intra-bridge cost measures the smoothness (as changes in surface normal) and convexity (as sum of interior angles) of the most smooth and convex patch that has the bridge as its border. While it is expensive to explicitly construct a patch, we are only concerned with the local shape of the patch at the bridge curves. Hence we implicitly represent a "patch" by a family of normal vectors, one associated with each line segment of curves $\{a, b, c\}$. To limit our search space to plausible patches, we consider normals that are generated by parallel-transporting [Wang et al. 2008] an initial normal vector at the first line segment of a. Such normals are ideal for our purpose as they are twist-minimizing, implying that they do not vary any more than the curves themselves. Two parallel-transport families of normals are shown in Figure 4 top for a simple, planar bridge (the initial normal vector in each family is thickened).

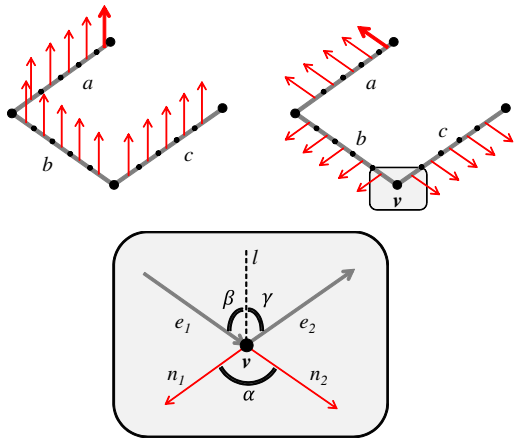

Figure 4: *Top: two families of normals generated by parallel-transporting two initial normals (thickened) along the same bridge. Bottom: the bending angle (α) and interior angle ($\beta+\gamma$) at a bridge vertex v and notions used in their calculation.*

Given a family of normals, we evaluate the smoothness and convexity of the hypothetical patch having these normals as follows. The evaluation is done at each of the two bridge vertices (i.e., the two ends of b). Let e_1, e_2 be the vector for the two incident line segments of the vertex, oriented consistently along the bridge, and n_1, n_2 be the normal on each segment (see Figure 4 bottom). Smoothness is measured by the *bending angle*, which is simply the angle formed by n_1 and n_2. Convexity is measured by the *interior angle*, which is the sum of the two angles formed by e_i $(i = 1, 2)$ and the intersecting line l of the two planes whose normals are respectively n_i $(i = 1, 2)$. Here we use $l = n_1 \times (e_1 + e_2)$, which is stable even when $n_1 = n_2$. The cost for the normal family is given by the sum of bending angles and interior angles at the two bridge vertices.

The intra-bridge cost is the minimum cost among all parallel-transport normal families on the bridge. For efficiency we restrict

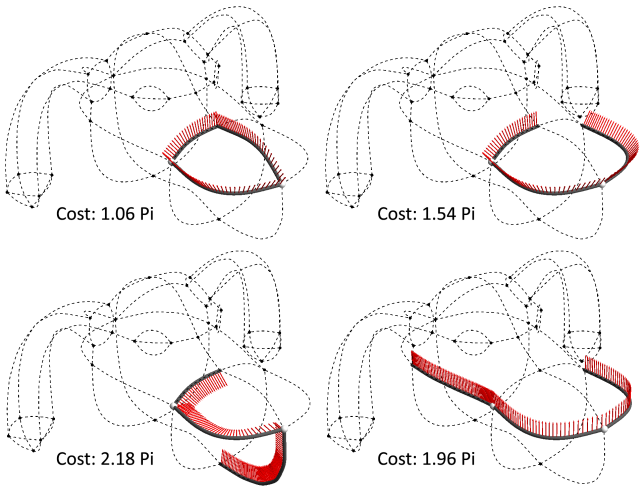

Figure 5: *Intra-bridge costs and families of parallel transport normals on selected bridges of a curve network (Dog Head).*

our search to a small number (e.g., 30) of families whose initial normals are equiangularly spaced on the first line segment of a. Figure 5 shows several bridges on a sketched curve network. The intra-bridge cost is noted for each bridge, and the normal family that attains the minimal cost is visualized along the bridge. Note that the bridge surrounding a desirable face (top-left) has the lowest cost, because it defines a smooth patch that forms small interior angles at the bridge vertices (light gray spheres).

Inter-bridge cost Consider the set of all bridges $\{a_i, b, c_i\}$ in a routing system for $i = 1, \ldots, k$ over a curve b with capacity k. We would like the patches that are bounded by these bridges to meet as smooth as possible along b. In the best configuration, these patches should partition the space around b in a uniform manner. That is, if we order the patches radially around curve b, the dihedral angles between consecutive patches should be identically $2\pi/k$. In the worst configuration, all k patches overlap.

Let n_i be the normal on the first line segment of b in the normal family that attains the intra-bridge cost for the bridge $\{a_i, b, c_i\}$. Assuming that n_i are sorted radially around that line segment, and let α_i be the angle formed by normals n_i, n_{i+1}. The inter-bridge cost is defined as

$$k\sqrt{\frac{\sum_{i=1}^{k}(\pi - \alpha_i)^2}{k}} - (k-2)\pi$$

The cost is bounded between 0 in the best situation ($\alpha_i = 2\pi/k$ for all i) and 2π in the worse situation ($\alpha_i = 0$ for all i).

5.2 Optimization

With cost metrics defined, we would like to find a routing system that minimizes the sum of intra-bridge costs over all bridges and inter-bridge costs over all curves. We shall first re-state the task as a graph search problem. We will then present an algorithm capable of finding the optimal solution, followed by pruning strategies for achieving practical performance.

Graph formulation Consider a graph $\mathcal{G}$ where each node represents a possible corner mapping at a vertex, and two nodes at the ends of a curve are connected by an edge. The weight of the edge is the minimal cost for bridging the two corner mappings represented by the two nodes. Specifically, for an edge that connects two nodes at the ends of curve b, we enumerate all possible bridge mappings on b and find one with the lowest sum of intra-bridge and inter-bridge costs. The edge is then associated with this lowest cost as well as the bridge mapping that attains the cost. Figure 6 shows a portion of this graph including the nodes and edges at two vertices v_1, v_2 of a curve b.

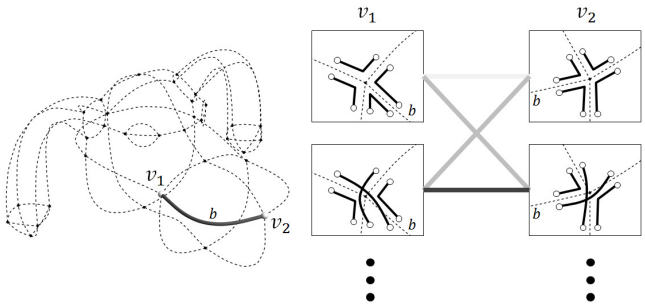

Figure 6: *A portion of the search graph (left) for the Dog Head (right). The graph has multiple nodes for a vertex (e.g., v_1), each corresponding to a corner mapping at that vertex (darts are drawn as white circles). Nodes at two ends of a curve (e.g., b) are connected by edges, each weighted by the optimal bridging cost (lighter gray has lower weight).*

With the graph defined, our problem becomes searching for a minimum-weight subgraph of $\mathcal{G}$ that covers all vertices. Here *covering* means that the subgraph contains exactly one node of $\mathcal{G}$ at each vertex of the network.

Optimal solution We can use a dynamic programming algorithm to find the optimal solution. Starting from a single vertex of the curve network, the algorithm finds the minimum-weight subgraph of $\mathcal{G}$ that covers an increasing larger set of vertices. Specifically, we maintain a vertex set V which starts off as an empty set and gets expanded by one vertex at a time. We call vertices in V that are connected to some vertex not in V the *exterior vertices*, noted as ∂V. After an expansion, we compute, for each possible combination of nodes that cover ∂V (called a *state*), the minimum-weight subgraph of $\mathcal{G}$ that covers the entire V. These subgraphs can be updated from those computed in the previous iteration. The update is made possible by the fact that the optimality of the nodes at interior vertices of V is not affected by the expansion of V. The algorithm terminates when V contains all vertices.

Pruning strategies While capable of finding the optimal solution, the algorithm has a high computational cost. We can reduce the search complexity to be linear in the total number of vertices by restricting both the number of nodes at each vertex and the number of states expanded at each iteration to be small constants. Specifically,

- When constructing $\mathcal{G}$, we create nodes for only a few promising corner mappings at each vertex v. To do so, we assign a cost to each pair of curves $\{a, b\}$ incident to v that assesses its likelihood in forming low-cost bridges with neighboring curves. The cost is the minimum inter-bridge cost among all bridges $\{a, b, c\}$ for some curve c plus the minimum inter-bridge cost among all bridges $\{d, a, b\}$ for some curve d. We then search for the K_1 corner mappings at v with the smallest total costs.
- During the dynamic programming search, we only maintain

the K_2 states with the smallest subgraph weights at each iteration. To reduce the chance of missing out good solutions as a result of this pruning, we expand the vertex set V in a predefined order that minimizes the number of states what would have been expanded at each iteration without pruning. The order is computed using a greedy expansion before dynamic programming.

Here, both K_1, K_2 are user-defined constants. We observed in our tests that the algorithm can find the optimal solution even with small values of these constants (we use $K_1 = 10$ and $K_2 = 100$).

The only component of the algorithm that can still have a high complexity is searching for the lowest-cost corner mappings at a vertex v. In the special case where every incident curve to v has capacity 2, the search reduces to the traveling salesman problem in a graph whose nodes are the incident curves to v and whose edge weights are the corner costs. To tame the complexity, we reduce the search space by only considering a pair of curves $\{a, b\}$ a possible corner if the cost of the pair (as described above) is among the $k_a + K_3$ smallest in all curve pairs involving a or among the $k_b + K_3$ smallest in all curve pairs involving b, where k_a and k_b are the capacity of a and b and K_3 is another user-defined constant. We find that setting $K_3 = 1$ is sufficient to include the desirable corners in most of our examples while allowing efficient computation.

In summary, we compute the optimal routing system in three steps. First, we pre-compute the intra-bridge costs for each possible bridge in the network. Second, we build $\mathcal{G}$ by searching for the most promising corner mappings at each vertex and computing the cost of the optimal bridge mappings between them. Finally, we use dynamic programming with a constant bandwidth to find the optimal corner mapping at each vertex.

5.3 Breaking cycles

Our cost metrics prefer cycles bounding patches that are smooth, making small interior angles at vertices, and having small normal discontinuity across the curves. While most desirable cycles in CAD wireframes or sketched curves fit into this profile, some cycles can be highly twisty, making large obtuse angles at vertices, or meeting at sharp angles with other cycles. In these cases, the optimal routing system computed by our algorithm may not be completely correct.

We found that such atypical cycles are usually few in number in a curve network, and hence only a few vertices and curves will have incorrect corner and bridge mappings. Furthermore, these wrong mappings often manifest themselves as repeated vertices and curves in a cycle. An example is shown in Figure 7, where all cycles computed by our algorithm are correct except for 3 non-simple cycles, one of which is shown in (b). These cycles are located in the narrow space between the four fingers of the hand, where the desirable cycles (faces of adjacent fingers) meet at very sharp corners.

We can break a non-simple cycle into several disjoint ones by adjusting the corner mapping σ at a repeated vertex v. Let $d_1, d_2, \ldots, d_{2m-1}, d_{2m}$ ($m \geq 2$) be the sequence of darts at v visited by the cycle, such that $\sigma(d_{2i-1}) = d_{2i}$ for $i = 1, \ldots, m$. By modifying σ so that $\sigma(d_{2i}) = d_{2i+1}$ for all i (d_{2m+1} refers to d_1), the original cycle will be broken into m cycles. The modification does not affect any other cycles. Note that the modification is only made if no pair of darts $\{d_{2i}, d_{2i+1}\}$ belongs to a same curve.

To break as many cycles as possible, we process each non-simple cycle in turn. For each cycle, we order its repeating vertices by their number of repetitions, and go through these vertices until we can modify the corner mapping σ as described above. If any newly created cycle is still not simple, it is added to the list of non-simple cycles and the process continues.

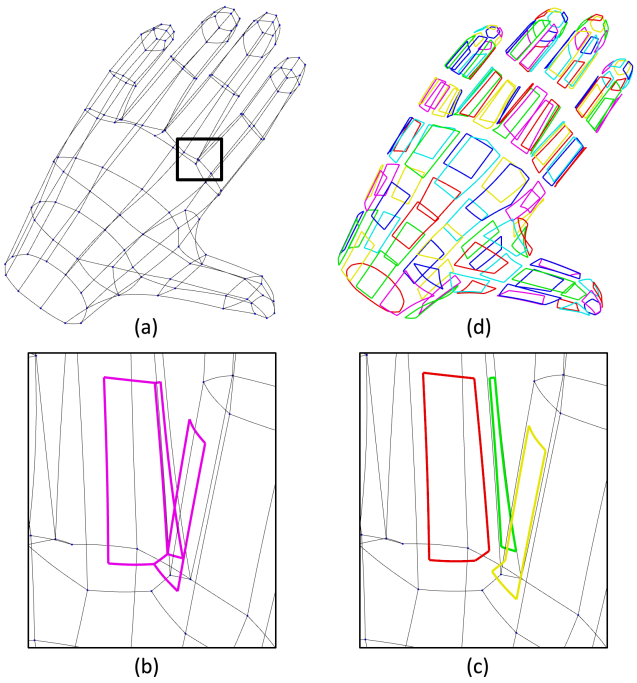

Figure 7: *Curve network of a hand (a), a non-simple cycle produced by optimization (b), cycles after breaking (c), and the complete set of cycles (d).*

We observed in our tests that this procedure is highly successful in correcting the occasional mistakes made by our optimization algorithm. For example, the non-simple cycle in Figure 7 (c) is broken into three simple (and more plausible) cycles in Figure 7 (d) after modifying the mapping at only one vertex.

6 Interactive tool

We developed an interactive tool for users to provide capacity information for non-manifold curves. The tool also creates surface patches for visualization, and offers convenient means for users to modify the results. We next detail these features.

Changing capacity If the network contains non-manifold or boundary curves, our tool allows the user to change the capacity of these curves (which is 2 by default). The interface is designed to eliminate the need for tedious button-clicking: as the user moves the mouse cursor across the screen, the nearest curve is automatically highlighted, and the user can increase or decrease the curve's capacity simply by scrolling up or down the mouse wheel (see a demo in the accompanied video). The program also highlights the vertices where the capacity conditions (see Section 3) are not met, prompting the user to provide further inputs. An example is shown in Figure 8. After changing the capacity of one curve to 1 (i.e., a boundary curve), its two end vertices are highlighted by "X", because the sum of curve capacities at each vertex is not even. After changing the capacity of another curve to 3 (i.e., a non-manifold junction), the parity condition is met at both vertices and the "X" signs disappear.

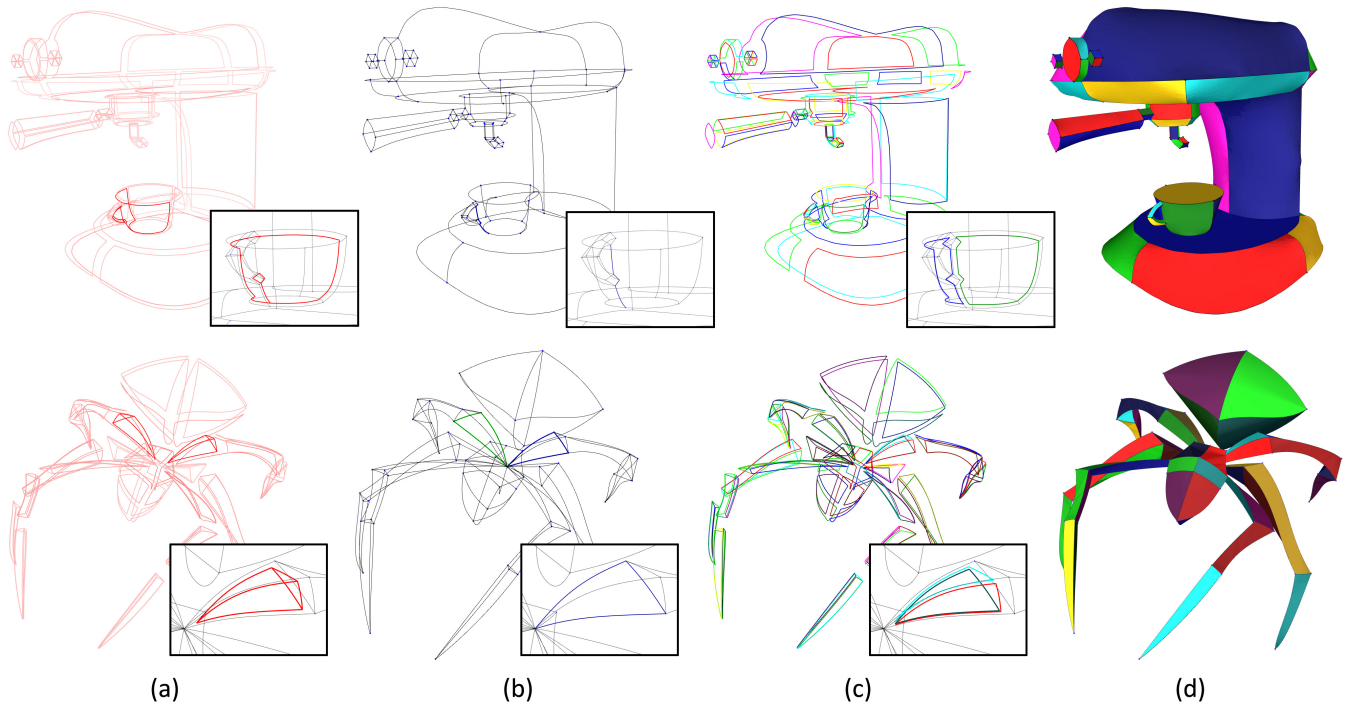

Figure 9: *Imposing constraints. (a): results of automatic algorithm on Espresso and Spider where cycles are colored by their costs (showing close-ups of problematic cycles). (b): constraining curve sequences provided by the user (highlighted in blue). (c,d): the resulting (correct) cycles and surface visualization.*

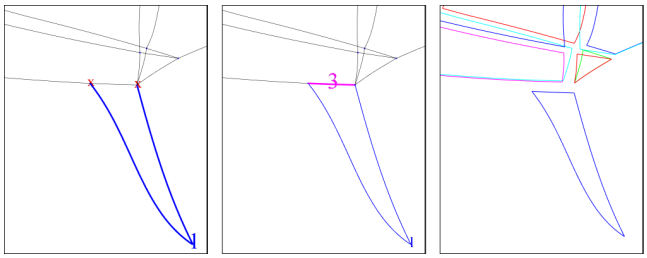

Figure 8: *Left and middle: two steps in changing the curve capacity at the bottom of the Boat example in Figure 2. Right: the resulting cycles.*

Viewing patches To help visualize the results, the tool can optionally create one triangulated patch for each computed cycle. To get better-looking patches, we utilize the parallel-transport normal families that attains the intra-bridge cost for each curve on the cycle. We then use a dynamic programming algorithm [Zou et al. 2013] to compute a triangulation that matches these normals as best as possible while minimizing the average dihedral angle between triangles. This initial triangulation is then refined and smoothed while still maintaining the normal constraints [Andrews et al. 2011]. While this method generates plausible surface geometry for most of our examples, the results may be improved using more specialized surfacing algorithms such as [Bessmeltsev et al. 2012].

Modifying cycles If the user is not happy with the results, she can adjust the cycles by clicking on a sequence of curves that belongs to a desired cycle (the sequence does not have to form a complete cycle). These sequences act as additional constraints to the algorithm. To enforce these constraints, the algorithm simply needs to make sure that each pair (resp. triple) of consecutive curves in a constraining sequence appears in the corner (resp. bridge) mappings when constructing the graph $\mathcal{G}$.

We found that it can be very difficult for a user to spot poor cycles, even with surface visualization. Since poor cycles are usually more twisty than desirable cycles, we offer a visualization mode where each cycle is colored by the total intra-bridge costs over all triples of consecutive curves on the cycle. The color varies in shades of red where lighter red indicates a lower cost. Two examples are shown in Figure 9 (a). In both cases, the highest-cost cycle is a bad one, prompting the user to specify a constraint there (shown in (b)), which in turns results in a desirable result (shown in (c)).

7 Results

We tested our method on a variety of inputs that include CAD wireframes, curves created by sketching tools [Bae et al. 2008; Schmidt et al. 2009], and synthetic examples extracted from existing surface models. The shapes represented by these inputs include both closed manifold (with possibly high genus) and non-manifold surfaces. For networks representing non-manifold surfaces, we use our interactive tool to provide the capacity information on each non-manifold curve. All experiments were done on a 6-core 3.0GHz workstation with 12GB memory.

Our algorithm (optimization followed by cycle breaking) successfully found correct cycle sets for the vast majority of the test data. Figure 10 shows a gallery of some of the results (the complete set can be found in the supplementary material). Only a few inputs require additional constraints, and two of them are shown in Figure 9. Such failure usually occurs when there are several nearby desirable cycles with high costs. For example, in the Espresso machine, the exterior faces of the cup make large obtuse angles at four vertices at the ends of the cup handle. In the Spider, a segment of the leg forms a very flat tetrahedron whose faces meet at sharp an-

gles along two adjacent curves. In these cases, the optimal set of cycles under our cost metrics may contain more than one incorrect cycles at those vertices or curves, which cannot be corrected by cycle breaking. However, these mistakes can be easily corrected with a few constraints provided by the user.

As shown in Table 1, our algorithm takes less than a second to run even for our largest test made up of hundreds of curves. In contrast, existing methods usually take much longer to run on smaller inputs. Our method achieves an order of magnitude improvement in speed over the most recent method by Abbasinejad et al. [Abbasinejad et al. 2011]; it takes our algorithm at most hundreds of milliseconds to run the same models that took their method at least seconds to run (compare Table 1 in their paper for common examples such as Jetfighter, Speaker, Phoenix, Roadster, Boat, Spider, and various Spacecrafts). More importantly, the complexity of our algorithm scales roughly linearly with the size of the input graph, due to the local formulation of the problem and the efficient search procedure, whereas most existing approaches (e.g., cycle basis methods) scale exponentially.

The interaction time for specifying capacity takes from a few seconds to a few minutes using our tool. Since we did not create most of these curve networks, the majority of this time was spent on "reverse-engineering" the intent of the designer, looking for non-manifold curves and figuring out their degrees. We expect the interaction time to be significantly shorter if it was done by the designer of the curve networks, and better yet, during the design process.

8 Conclusions

We introduce a novel algorithm for finding boundary cycles of patches in a curve network created by CAD or sketching tools. The major departure from previous methods is that we consider an alternative representation, called routing system, which implicitly encodes a set of cycles by local variables at each vertex and curve of the network. By optimizing cost metrics designed for these variables, we are able to compute correct cycles more efficiently and handle inputs with more complex topology and geometry than previous methods.

While our algorithm requires users to supply capacity information, it would be ideal if the program can automatically identify possibly non-manifold curves and suggest their capacity. This feature would be particularly necessary if our algorithm is to be incorporated into a curve-sketching system for providing surface visualization during the sketching process. Such capability is not present in current systems, partly due to the inefficiency of the cycle-discovery step. While our algorithm is fast, without identifying "boundaries" of a partial sketch, it will always produce a closed object. There is some initial research into detecting simple non-manifold elements [Sun and Lee 2004]. However, identifying complex non-manifold configurations is still a challenging, and often subjective, task.

Acknowledgements We thank the authors of [Bae et al. 2008; Schmidt et al. 2009] for providing the curve sketches, WiZ WORX for the wireframe of *Gehaeuse*, Open CASCADE shape factory for *misc2* and FIRST CAD library for *kk35*. The work is supported in part by NSF grants (IIS-0846072, IIS-1302142) and a gift from Adobe. The work of the first author was done at Washington University with the support from China Scholarship Council.

Model	Curves	Non-mani. curves	CPU time (ms)	Constraints
Gehaeuse	258	0	195	0
kk35	358	0	386	0
misc2	410	0	447	0
Ship	16	10	17	0
Building D2	20	0	17	0
Pheonix	28	16	59	0
Plane	31	0	31	4
Jetfighter	50	4	44	0
Cup	50	0	63	0
Fighter	55	0	67	0
Speaker	64	0	52	0
Spacecraft49	64	12	77	6
Roadster	66	11	56	0
Hawk	81	0	88	0
Torso	92	2	109	0
Submarine	84	0	108	0
Car	98	0	87	0
Boat	102	10	102	0
House	106	10	107	0
Spacecraft87	110	22	124	12
Spacecraft12	112	28	149	6
Taxi	119	8	132	0
Spider	130	0	184	6
Building A2	149	17	230	0
Espresso	190	0	147	4
Frame	32	0	32	0
Mug	42	0	69	0
Genus	48	0	71	0
Twisty	48	0	55	0
Pulley	48	0	73	0
Block	64	0	91	0
Fertility	83	0	122	0
Bug	132	0	98	0
Enterprise	227	0	299	0
Hand	281	0	324	0

Table 1: *Statistics of test data and results. The data is divided into CAD wireframes (top), sketched curves (middle), and synthetic tests (bottom).*

References

ABBASINEJAD, F., JOSHI, P., AND AMENTA, N. 2011. Surface patches from unorganized space curves. *Comput. Graph. Forum 30*, 5, 1379–1387.

AGARWAL, S. C., AND WAGGENSPACK, W. N. 1992. Decomposition method for extracting face topologies from wireframe models. *Computer-Aided Design 24*, 3, 123 – 140.

ANDREWS, J., JOSHI, P., AND CARR, N. A. 2011. A linear variational system for modeling from curves. *Comput. Graph. Forum 30*, 6, 1850–1861.

BAE, S.-H., BALAKRISHNAN, R., AND SINGH, K. 2008. Ilovesketch: as-natural-as-possible sketching system for creating 3D curve models. In *Proceedings of the 21st annual ACM Symposium on User Interface Software and Technology*, ACM, New York, NY, USA, UIST '08, ACM, 151–160.

BAGALI, S., AND WAGGENSPACK, JR., W. N. 1995. A shortest path approach to wireframe to solid model conversion. In *Proceedings of the third ACM Symposium on Solid modeling and applications*, ACM, New York, NY, USA, SMA '95, ACM, 339–350.

BESSMELTSEV, M., WANG, C., SHEFFER, A., AND SINGH, K. 2012. Design-driven quadrangulation of closed 3D curves. *Transactions on Graphics (Proc. SIGGRAPH ASIA 2012) 31*, 5.

BIONDI, E., DIVIETI, L., AND GUARDABASSI, G. 1970. Counting paths, circuits, chains, and cycles in graphs: a unified approach. *Canad. J. Math. 22*, 22–35.

BREWER, III, J. A., AND COURTER, S. M. 1986. Automated conversion of curvilinear wire-frame models to surface boundary models; a topological approach. In *Proceedings of of ACM SIGGRAPH 1986*, ACM, New York, NY, USA, SIGGRAPH '86, ACM, 171–178.

DUTTON, R. D., AND BRIGHAM, R. C. 1983. Efficiently identifying the faces of a solid. *Computers & Graphics in Mechanical Engineering 7*, 2, 143 – 147.

GANTER, M. A., AND UICKER, J. J. 1983. From wire-frame to solid geometric: Automated conversion of data representations. *Computers in Mechanical Engineering 2* (sept), 40–45.

GRIMM, C., AND JOSHI, P. 2012. Just DrawIt: a 3D sketching system. In *Proceedings of the International Symposium on Sketch-Based Interfaces and Modeling*, SBIM '12, ACM, 121–130.

HANRAHAN, P. M. 1982. Creating volume models from edge-vertex graphs. In *Proceedings of ACM SIGGRAPH 1982*, ACM, New York, NY, USA, SIGGRAPH '82, ACM, 77–84.

INOUE, K., SHIMADA, K., AND CHILAKA, K. 2003. Solid model reconstruction of wireframe cad models based on topological embeddings of planar graphs. *Journal of Mechanical Design 125*, 3, 434–442.

LIU, J., AND LEE, Y. T. 2001. Graph-based method for face identification from a single 2D line drawing. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 23*, 10 (oct), 1106 –1119.

LIU, J., LEE, Y. T., AND CHAM, W.-K. 2002. Identifying faces in a 2D line drawing representing a manifold object. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 24*, 12 (dec), 1579 – 1593.

MARKOWSKY, G., AND WESLEY, M. A. 1980. Fleshing out wire frames. *IBM J. Res. Dev. 24* (September), 582–597.

MOHAR, B., AND THOMASSEN, C. 2001. *Graphs on Surfaces*. The Johns Hopkins University Press.

ROSE, K., SHEFFER, A., WITHER, J., CANI, M.-P., AND THIBERT, B. 2007. Developable surfaces from arbitrary sketched boundaries. In *Proceedings of the fifth Eurographics symposium on Geometry processing*, SGP '07, ACM, 163–172.

SAKURAI, H., AND GOSSARD, D. C. 1983. Solid model input through orthographic views. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '83, ACM, 243–252.

SCHMIDT, R., KHAN, A., SINGH, K., AND KURTENBACH, G. 2009. Analytic drawing of 3D scaffolds. In *Proceedings of ACM SIGGRAPH Asia 2009*, ACM, New York, NY, USA, SIGGRAPH Asia '09, ACM, 149:1–149:10.

SHPITALNI, M., AND LIPSON, H. 1996. Identification of faces in a 2D line drawing projection of a wireframe object. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 18*, 10 (oct), 1000 –1012.

SUN, Y., AND LEE, Y. T. 2004. Topological analysis of a single line drawing for 3D shape recovery. In *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '04, ACM, 167–172.

VARLEY, P. A., AND COMPANY, P. P. 2010. A new algorithm for finding faces in wireframes. *Computer-Aided Design 42*, 4, 279 – 309.

WANG, W., JÜTTLER, B., ZHENG, D., AND LIU, Y. 2008. Computation of rotation minimizing frames. *ACM Trans. Graph. 27*, 1 (Mar.), 2:1–2:18.

WESCHE, G., AND SEIDEL, H.-P. 2001. Freedrawer: a free-form sketching system on the responsive workbench. In *Proceedings of the ACM Symposium on Virtual reality software and technology*, ACM, New York, NY, USA, VRST '01, ACM, 167–174.

ZOU, M., JU, T., AND CARR, N. 2013. An algorithm for triangulating multiple 3D polygons. *Computer Graphics Forum 32*, 5, 157–166.

A Existence of cycles

Proposition A.1 *There exists a set of cycles, free of U-turns, whose curve degrees equal the capacities if and only if the following conditions hold at every vertex v with a set of incident curves C_v:*

1. *The sum of capacities of all curves in C_v is even.*
2. *No curve in C_v has a greater capacity than the sum of capacities of remaining curves in C_v.*

Proof: While the necessity follows in a straight-forward manner, we will focus on sufficiency. We will show that a routing system exists given capacities meeting these conditions. Since bridge mapping always exists, we will only need to construct a corner mapping at v. To prevent U-turns, no two darts belonging to a same curve can be mapped to each other.

The construction is iterative. At each iteration, we pair a dart of the curve in C_v having the most un-paired darts with a dart of the curve having the second-most un-paired darts. If multiple curves have the same number of un-paired darts, an arbitrary curve is chosen.

We first show that the two conditions in the proposition hold for the un-paired darts at each iteration. It is evident that the total number of un-paired darts remain even in the process. Suppose, after the i-th iteration, some curve $c \in C_v$ has more un-paired darts than the union of the remaining un-paired darts. We will show this is impossible, by induction. First, this curve cannot be the one that we just picked for pairing. To see this, let n_c^i be the number of un-paired darts of c before the i-th iteration, and n^i be the total number of un-pair darts at v at that time. Since the conditions hold before, we have $n_c^i \leq n^i - n_c^i$. If we had picked a dart of c to pair during the i-th iteration, and since $n_c^{i+1} = n_c^i - 1 \leq (n^i - 2) - (n_c^i - 1) = n^{i+1} - n_c^{i+1}$, the number of un-paired darts of c will still be no greater than the number of remaining un-paired darts. Now suppose we picked darts from two other curves, a and b, during the i-th iteration. Hence we have $n_a^i \geq n_c^i$, $n_b^i \geq n_c^i$, $n_a^{i+1} = n_a^i - 1$, $n_b^{i+1} = n_b^i - 1$, and $n_c^{i+1} = n_c^i$. If, after the iteration, c has more un-paired darts than the union of all remaining un-paired darts, then $n_c^{i+1} > n_a^{i+1} + n_b^{i+1}$. These inequalities lead to $n_c^{i+1} < 2$. Hence $n_c^{i+1} = 1$, and no other curve has any un-paired darts after the i-th iteration, which contradicts the fact that the total number of un-paired darts should be even.

Next, we show that the iterative construction would finish with no remaining un-paired darts. The only situation otherwise would be that there is a single curve with multiple un-paired darts. However, this would contradict to the fact we just showed above that no curve has more un-paired darts than the union of remaining un-paired darts. □

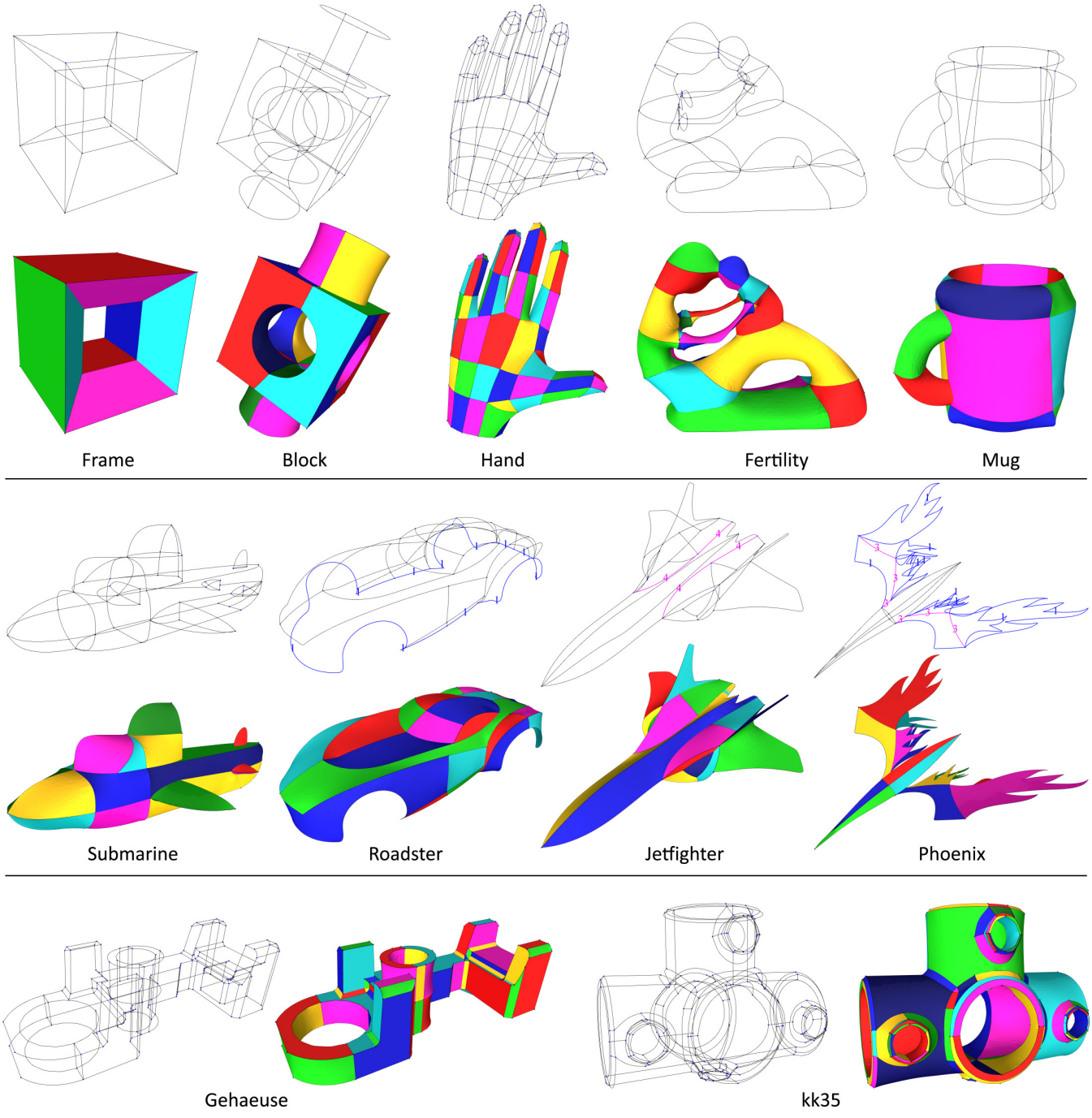

Figure 10: *A gallery of results automatically generated by the algorithm: synthetic examples (top), sketched curves (middle), and CAD wireframes (bottom).*