Example-Based Curve Synthesis

Paul Merrell [*] Dinesh Manocha
University of North Carolina at Chapel Hill

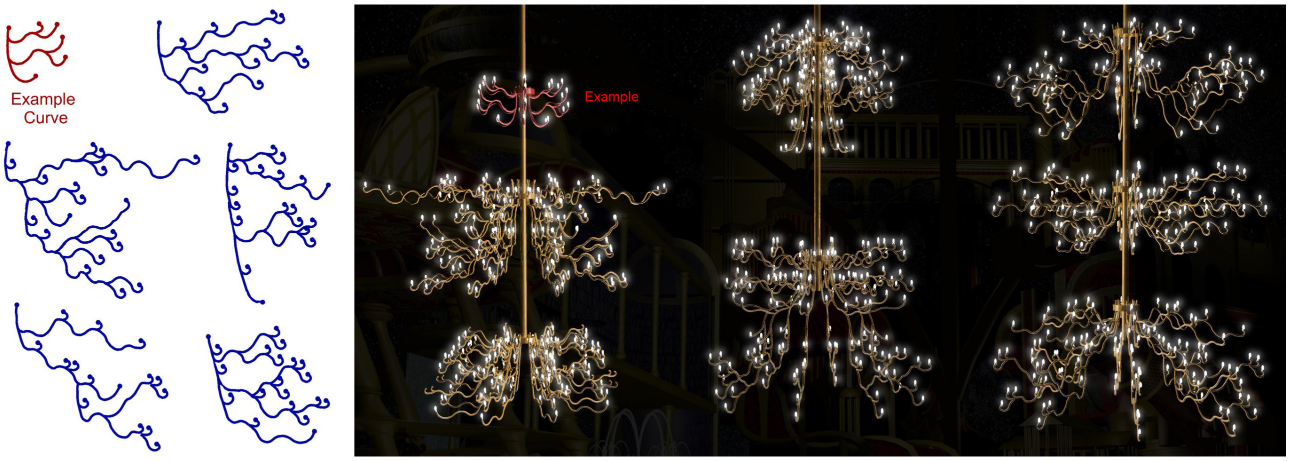

Figure 1: *Using the example curve sketched in red, several curves (blue) are generated automatically in the style of the example using our algorithm. The new curves have over 250 branches and all of them were created in less than three minutes. Five of the generated curves are shown on the left. The curves contain many branching points. The curves are directly used to create models of chandeliers by using them as generators for surfaces of revolution or lofting. The final models of the chandeliers are composed of about 200,000 Bezier patches, generated using extrusion operations.*

Abstract

We present a novel synthesis algorithm for procedurally generating complex curves. Our approach takes a simple example input curve specified by a user and automatically generates complex curves that resemble the input. The algorithm preserves many of the input shape features such as tangent directions, curvature, branch nodes, and closed loops. The overall approach is simple and can be used to generate varied curved 3D models in a few minutes. We demonstrate its application to generating complex, curved models of man-made objects including furniture pieces, chandeliers, glasses, and natural patterns such as river networks and lightning bolts.

1 Introduction

One of the key problems in computer graphics is to generate geometric models of complex shapes and structures. The main goal is to generate three-dimensional content for different domains such as computer games, movies, architectural modeling, urban planning and virtual reality. In this paper, we address the problem of automatically or semi-automatically generating complex shapes with curved boundaries. Curved artistic decorations with repeated patterns are an important part of the design of man-made objects. These include household items such as furniture, glasses, candlesticks, chandeliers, toys, etc.. Curved structures are also used in buildings and interior design. Moreover, many patterns in nature (e.g. terrain features or river network) or natural phenomena, such as lighting, also have a curved boundaries. As a result, we need simple and effective tools that can assist artists, designers, and modelers in designing elaborate curved objects and structures.

Most of the prior work in this area has been in procedural modeling, which generates 3D models with repeated patterns from a set of rules. These include L-systems, fractals and generative modeling techniques which can generate high-quality 3D models of plants, architectural models and city scenes. However, each of these methods is mainly limited to a special class of models. Instead our goal is to use example-based techniques, which are more general and generate complex models from a simple example shape [Aliaga et al. 2008; Merrell 2007; Merrell and Manocha 2008]. Some of these methods have been inspired by texture synthesis, but the current methods are limited primarily to complex polyhedral models with planar boundaries or layouts of city streets.

Main Result: In this paper, we present a new method for rapidly generating many curves based off an example. Our algorithm accepts a set of 2D curves as an input and rapidly generates many more complex curves in a similar style. The new curves bend and have branches like the example curves and have nearly the same local structure. Our approach is general and makes no assumptions about the shape or smoothness of the input curves. We perform local shape analysis based on the tangent vectors and curvature of the input curve, and generate output curves that tend to preserve these local features. This includes generating cusps, branches, and closed loops. We use a graph to maintain the topology of various curve segments and present automatic methods to incrementally refine the structure of the graph to generate the final curves. We also present an intuitive metric to evaluate the curves represented by the graph. Given a set of generated curves, we perform an extrusion operation or use the final curves as generators for surface of revolution to generate 3D models of curved objects. In practice, our algorithm takes a few minutes to generate 3D curved models.

The overall algorithm is relatively simple, efficient and quite robust in practice. Our system also provides the capability to edit the generated curves and to create closed loops. We use our algorithm to generate complex 3D models of chandeliers, drinking glasses, candlesticks, river networks, lightning bolts, and cabinet handles with hundreds or thousands of curve segments or surface patches in a

[*]http://gamma.cs.unc.edu/synthesis

few minutes.

The rest of the paper is organized as follows. In Section 2, we discuss related work on procedural modeling and curve generation. In Section 3, we explain our curve generation algorithm. We show results and analysis our method in Section 4. We compare our approach to other methods in Section 5 and discuss ideas for future research in Section 6.

2 Related Work

A few techniques have been developed to transform curves sketched in one particular artistic style into a different artistic style [Hertzmann et al. 2002; Freeman et al. 2003]. The artistic styles are determined automatically from the example curves sketched by the user. Similar techniques have also been applied on meshes to transform the models [Bhat et al. 2004; Zelinka and Garland 2003] and also to tranform space-time curves for animation [Wu et al. 2008]. Simhon and Dudok [2004] use a hidden Markov model to add artistic details to sketches. Layouts of city streets have been creating using an interactive tool which uses tensor fields [Chen et al. 2008a]. Kalnins et al. [2002] developed a non-photorealistic rendering technique that draws 3D models in different artistic styles. It generates curves representing the strokes an artist might draw or paint in a particular style.

Example-based techniques are widely used to synthesize texture [Efros and Leung 1999; Kwatra et al. 2003] and similar techniques has been applied to vector data [Barla et al. 2006]. Three-dimensional closed polyhedral models can also be synthesized from example models [Merrell 2007; Merrell and Manocha 2008]. An example-based method has also been presented to synthesize the layout of city streets [Aliaga et al. 2008].

Procedural modeling techniques are widely used to generate different types of objects including urban environments [Müller et al. 2006; Wonka et al. 2003] and plants using L-systems [Měch and Prusinkiewicz 1996; Ijiri et al. 2005; Power et al. 1999]. Wong et al. [1998] have developed a procedural technique for designing decorative patterns, including floral patterns. Pottmann et al. [2007] have presented elegant algorithms to generate freeform shapes for architectural models.

Sketch-based interfaces have been developed as an intuitive way to model and deform meshes [Igarashi et al. 1999; Singh and Fiume 1998]. These methods complement our approach and can be used to transform 2D curves into full 3D models.

3 Curve Generation

In this section, we present our curve synthesis algorithm. We first introduce the notation and give an overview of our method. Next, we present techniques to connect and bend various segments in the input curve and use of the graph representation and modification to generate the final curves.

3.1 Overview

We assume that the input example curve is represented as a set of parametric curves $\{\mathbf{c}_1(t), \mathbf{c}_2(t), \ldots\}$, where $\mathbf{c}_i(t)$ is a 2D Bezier curve. In practice, our algorithm can handle any curve representation as long as it can be subdivided and we can evaluate bounds on its tangents and curvatures. The curves may or may not contain closed loops, branches, and cusps. Our algorithm generates a set of output curves that resemble the input curves and are composed of segments created from the set of input curves $s_i(t)$ as described in Section 3.2. The curves and the segments start at the value $t = 0$ and end at the value $t = 1$.

Our goal is not only to generate simple curves, but also to generate sets of curves with multiple branches and loops. We use a graph data structure to represent how the set of curves connect together (see Section 3.4). Each edge of the graph is a sequence of the curve segments $s_i(t)$. The graph is usually a simple connected component. It can have multiple components, but this generally discouraged. The user can edit the result by creating and moving adjustment points on the curves. In response, our algorithm is constantly generating new alternative versions of the graph by altering and adding segments $s_i(t)$. Each version is evaluated using an objective function that considers how well does the new set of curves adjust to the changes that the user makes, how well does it form the branches and closed loops that are desired in many cases, and how few self-intersections does it have. The algorithm is constantly searching for better solutions. While it searches, the best solution that has been found is displayed as part of our interactive interface. Section 3.4 describes how the set of curves in the graph is modified and evaluated. It is very unlikely that we can add segments together to produce curves that exactly touch the user's input points or form close loops. In order to accomplish these goals, the segments are bent and stretched as described in Section 3.3.

3.2 Creating and Connecting Segments

For the new set of curves to resemble the example curves, every local neighborhood of the new curve should closely resemble a neighborhood of the input example curve. We characterize a curve's local neighborhood in terms of its tangent angle $\theta = \text{atan2}(y', x')$ and signed curvature $k = \frac{x'y''-y'x''}{(x'^2+y'^2)^{3/2}}$. The tangent angles and curvatures are discretized into uniform bins $\hat{\theta} = \lfloor \frac{\theta}{\theta_b} \rfloor$ and $\hat{k} = \lfloor \frac{k}{k_b} \rfloor$ where θ_b and k_b are the size of the bins. In our algorithm, two curve segments can be connected if their tangent angles and curvatures are within the same bin.

The tangent angles and curvatures are used to subdivide the curves into segments s_i. The tangents and curvature may be undefined at some points such as cusps, since we do not assume C^1 or C^2 continuity in the input curves. All this means for the algorithm is that we never divide the curve at a cusp. The cusp is always inside one of the segments. It is never at the start or at the end of a segment.

Each segment s_i has a starting and ending tangent angle, $\hat{\theta}_i^s, \hat{\theta}_i^e$, and a starting and ending curvature $\hat{k}_i^s, \hat{k}_i^e$. Smooth segments are subdivided until their start and end are only one bin apart $|\hat{k}_i^s - \hat{k}_i^e| \leq 1$ and $|\hat{\theta}_i^s - \hat{\theta}_i^e| \leq 1$ or $|\hat{\theta}_i^s - \hat{\theta}_i^e| = \lfloor \frac{\pi}{\theta_b} \rfloor - 1$. The curve is never divided at a cusp since $\hat{\theta}$ and $\hat{k}$ are undefined. The cusp is always combined with a part of the curve immediately before and a part immediately after it into a single segment. This means that the cusp is integrated into a segment s_i whose beginning and end have well-defined tangents $\hat{\theta}_i^s, \hat{\theta}_i^e$.

We place all the curve segments cut from the example curve into a 2D array of bins shown in Figure 2. Most segments start and end in adjacent bins except for segments containing cusps which may extend across the array. The start of a segment s_j can be attached to the end of segment s_i if their endpoints are in the same bin $(\hat{\theta}_j^s, \hat{k}_j^s) = (\hat{\theta}_i^e, \hat{k}_i^e)$. Let A_i be the set of segments that may follow after s_i, $A_i = \{s_j | (\hat{\theta}_j^s, \hat{k}_j^s) = (\hat{\theta}_i^e, \hat{k}_i^e)\}$ and let B_i be the set of segments that may come before s_i. If for any i and j, $A_i = \{s_j\}$ and $b_j = \{s_i\}$, then the segment s_j is always attached to the end of s_i. We can combine these small segments into a larger unified seg-

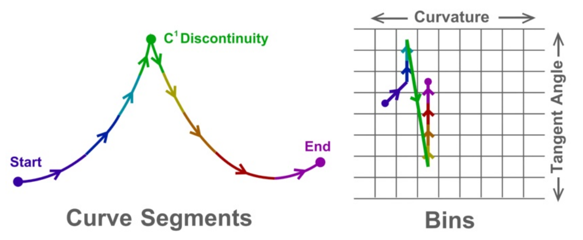

Figure 2: *The curve is decomposed into segments which have only small changes to their tangents and curvatures. The segments are placed into bins with bounds on tangent angles and curvatures.*

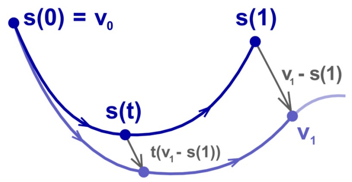

Figure 3: *The points along a parametric curve* $\mathbf{s}(t)$ *are bent and stretched so that the curve goes from the vertex* $\mathbf{v}_0$ *to* $\mathbf{v}_1$.

ment which improves the speed of the algorithm since there would be fewer segments to manage.

The input curves may form a closed loop by starting and ending at the same location $\mathbf{c}(0) = \mathbf{c}(1)$. In this case, there is nothing unusual about the segments at the start or end of the curve. Each segment has a segment before it and after it. But if the curve does not form a closed loop, then the curve must start or end with empty space coming before or after it or by branching off from another curve. In order to handle these special cases, we include special symbols in the sets A_i and B_i to indicate that before s_i there is a branching point or empty space.

3.3 Bending and Stretching the Curves

Our algorithm uses a graph to represent the topology of the output curve. The vertices of the graph are those points where the curve starts, ends, or branches as well as any point that the user selects as an adjustment point. The edges of the graph are sequences of segments. For each edge, we combine all the segments s_i of that edge into a single piecewise parametric curve $\mathbf{s}(t)$ (i.e. the composite curve). Let $\mathbf{s}(0)$ and $\mathbf{s}(1)$ be the start point and end point of the composite curve. Moreover, let $\mathbf{s}(t)$ be an arbitrary point on the curve. We estimate the arc length of $\mathbf{s}(t)$ using a piecewise linear approximation and compute an arc-length parameterization of $\mathbf{s}(t)$. In the rest of the paper, we assume that all the computed curves are represented using such an arc-length parameterization.

Let $\mathbf{v}_0$ and $\mathbf{v}_1$ be two vertices of the graph. Suppose that a particular curve should go from point $\mathbf{v}_0$ to $\mathbf{v}_1$. We position the curve $\mathbf{s}(t)$ such that $\mathbf{s}(0) = \mathbf{v}_0$. Ideally, $\mathbf{s}(1) - \mathbf{s}(0) = \mathbf{v}_1 - \mathbf{v}_0$, but otherwise we must stretch the curve so that it reaches $\mathbf{v}_1$ by performing a simple geometric deformation. We stretch the curve over its entire length as shown in Figure 3. We move the point $\mathbf{s}(1)$ by $\mathbf{v}_1 - \mathbf{s}(1)$ and every intermediate point $\mathbf{s}(t)$ by $t(\mathbf{v}_1 - \mathbf{s}(1))$. Long curves are distorted less than short curves when stretched a given distance since the stretching is distributed over a greater length. Our goal is to keep the amount that a curve stretches small especially for short curves.

We now have a simple method to stretch a curve given its two endpoints. The endpoints are the vertices of the graph. The locations of some of the vertices may be specified by the user, but any remaining vertices are treated as unknown free variables in the plane. The goal is to determine the locations of the free vertices that would minimize the extent of the stretching. The stretching should be minimized because it distorts the segments s_i so that they look less like the original segments in the input. The amount each curve stretches is equal to the length of the vector $\mathbf{v}_1 - \mathbf{s}(1) + \mathbf{v}_0 - \mathbf{s}(0)$ where $\mathbf{v}_0$ and $\mathbf{v}_1$ are the vertices where the curve ends when after it is stretched and $\mathbf{s}(1) - \mathbf{s}(0)$ are the unstretched endpoints. This equation gives the amount of stretching for each edge of the graph. We minimize the amount of stretching in the least squares sense. The values of $\mathbf{s}(1) - \mathbf{s}(0)$ are known as well as any vertex locations the user has specified. We solve for the remaining vertex locations. We use a weighted least squares optimization that weights each equation according to the reciprocal of the length of each curve to ensure that the short curve segments stretch less than the long segments.

Algorithm 1 Method for incrementally modifying the structure of the graph.

```
 1: Pick a region to modify.
 2: Cut out the segments in the region.
 3: for each cut segment do
 4:     Create a curve by adding segments using $A_i$.
 5:     $d_i$ = the shortest distance found from the extended curve to
        any of the targets
 6:     if $d_i < d_{\max}$ then
 7:         Stretch the curve to the closest target.
 8:     else
 9:         Add the curve as a leaf node.
10:     end if
11: end for
12: Evaluate the modified graph.
13: if the modified graph is worse than the original then
14:     Revert back to the original.
15: end if
```

3.4 Choosing the Sequence of Curve Segments

Our method is designed to constantly refine and improve the current solution as the user makes adjustments to it. Our algorithm constantly computes possible modifications to the graph including topological changes such as adding edges for branches or loops as well as changes to the path of each edge of the graph. These modifications are randomly generated and many of them do not improve the result, but since our algorithm is able to evaluate hundreds of small modifications per second, we can generate many random modifications and select appropriate ones quickly.

Algorithm 1 gives an overview of the algorithm for modifying the graph. First, we pick part of the graph to modify and cut curves out of the graph. We attach a new curve to where the original curve was cut and then evaluate the modified graph in terms of several criteria explained in Section 3.5 such as how much the original segments need to be stretched and how many new branches appear. We keep the modified graph if it improves the result based on the evaluation function proposed in Section 3.5.

First, we select a portion of the graph to modify. We modify a circular region chosen at random. Its center is a randomly selected point on the graph. Within the region, we remove the curve segments cutting them from the graph as shown in Figure 4(b). This leaves several cut segments which need to be repaired by attaching new segments to them. The cuts are repaired by creating a new curve with one of its ends attached to the cut and the other end unattached

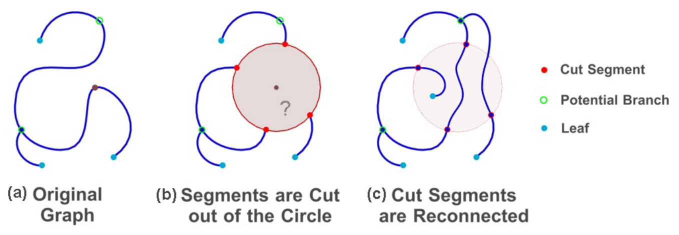
Cut Segment
Potential Branch
Leaf
(a) Original Graph
(b) Segments are Cut out of the Circle
(c) Cut Segments are Reconnected
?

Figure 4: *The graph is modified by removing segments from part of the graph and then reconnecting them.*

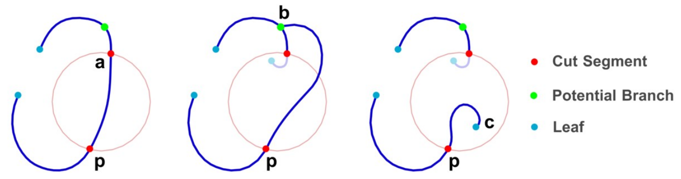

Figure 5: *The curve starting at point **p** could end up at several possible locations. It could connect to the cut segment at point **a** or as a branch to point **b** or created an unattached leaf at point **c**.*

or attached to a cut or a branching point. These three possibilities are shown in Figure 5(a-c). Branches can only be attached to curved segments which had branches in the input set of curves. The ends of the new curve can only be attached to a few possible locations. The simplest option is to leave one of the ends unattached, but it is not necessarily the best option since it may cause parts of the graph may become disconnected.

For each segment s_i, there is a set of segments A_i that may follow after it as described in Section 3.2. The strategy is to assemble together segments chosen at random using the sets A_i. Even though this is a random process, we can produce good results by rapidly exploring and testing many possible options. We would like to evaluate the new curve as it is being generated. Ideally, the new curve would end at one of several target locations. The target locations are either a cut segment (Figure 5(a)) or a branching point (Figure 5(b)). Furthermore, the end of the new curve must have a particular tangent and curvature so that it can be attached properly to the target location. We could evaluate each curve while it is being generated by measuring the distance from the end of the curve to each target location and taking the minimum distance among all the targets. But this evaluation ignores two important facts. First, the curve must have the proper tangent and curvature when it reaches the target to maintain continuity. Second, we are evaluating curves that are unfinished which we plan to add segments onto. A better evaluation would first extend the curve slightly by finding the shortest possible extension to the curve that ends with the proper tangent and curvature. We compute the shortest extension for each target location and measure the distance from the end of each extended curve to its target. The distance to the closest target is used when there are multiple targets.

The curves can be evaluated very quickly. We only need to add up a set of vectors that describe the relative positions of the start and end of each segment and their extensions. The extensions are computed from a lookup table. The lookup table is only computed once and stores the shortest paths by extending the curve so that it ends with a particular tangent and curvature. It is computed using Dijkstra's shortest-path algorithm.

We generate the curves by assembling segments together and evaluate the curves by determining how close they come to their targets after they are extended. The ends of the curves may come close to their targets, but they almost never exactly touch their targets. In order to touch precisely, the curves are stretched slightly as described in Section 3.3. The end of the new curve also might not get close to any of the target locations. In this case, the one end of the curve is left unattached creating a leaf in the graph, i.e. a vertex with a single edge incident to it.

3.5 Evaluating the Modified Graph

The circular region shown in Figure 4(b) may cut many different segments. Every cut segment is attached to a new curve by using the above method multiple times if necessary. The new curves are evaluated individually based on how close they get to their targets before they are stretched. However, after every individual curve is created, we need to evaluate the collection or graph of curves as a whole. The entire graph is evaluated and compared to the graph as it was before any modifications based on the amount of stretching and the number of leaves, branches, and self-intersections. If the modified graph evaluates better than the unmodified graph, the modification is kept. Otherwise, it is discarded. The graph is evaluated based on a set of penalties described in the equation

$$p = \sum_{i=1}^{n} w_i d_i + w_L L - w_R R + w_D D + w_S S \qquad (1)$$

The terms in the equation are defined as:

d_i: The distance of the i-th curve from its target
L: The number of leaves in the graph
R: The number of branches in the graph
D: The number of disjoint unconnected parts of the graph
S: The number of self-intersections in the graph
w_i, w_L, w_R, w_D, w_S: Different weights applied to each penalty

The $w_i d_i$ term is a stretching penalty based on how much the i-th curve is stretched to reach its target. It is especially important that small curves are not stretched large distances, so the weight w_i is inversely proportional to the length of the curve. The stretching penalty has some unintended consequences since there is no need to stretch those curves that connect to leaf nodes as shown in Figure 5(c). So these curves do not have a stretching penalty. Consequently, the stretching penalty alone would cause an excessive number of leaf nodes to be produce. To compensate for this a penalty is added for each leaf node in the graph. Also, we would like to encourage branches to be generated, so for each branch we subtract a value from the penalties. It is possible for parts of the graph to become disconnected as shown in Figure 5(c). To strongly discourage such a result, we count the number of disjoint parts of the graph and add a heavy penalty w_D for each. We also strongly discourage self-intersections, by counting the number of self-intersections S and adding a heavy penalty w_S for each. By adding up all of these different penalties we evaluate the graph as a whole and use the modified graph only if it improves the result.

Most of the penalties in Equation 1 can be computed very quickly. The most expensive computation is to count the number of self-intersections in the set of curves. As a result, we only perform that computation after we have established that the new graph is a good candidate based on the other factors in the equation.

Algorithm 1 shows how to improve the graph over time and to respond to changes the user makes. The user can move points on the graph to stretch or bend the curves and then Algorithm 1 will improve the curves so they are stretched less. The initial graph is either a closed loop or a single open curve. Branches might be added

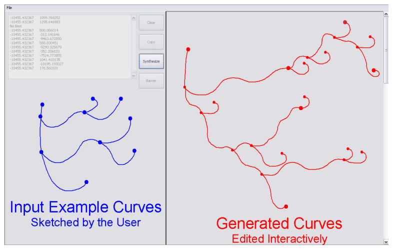

Figure 6: *The user interface. The user begins by sketching the example curves on the left. Then new curves are automatically generated on the right and can be edited interactively.*

	Num. Output	Input Seg.	Output Seg.	Input Branch	Output Branch
Chandelier	8	105	5,686	4	56
Cabinet	9	38	9,787	0	0
Streams	5	203	9,900	5	82
Lightning	4	633	12,509	3	149
Glasses	13	42	2,749	0	0
Candlestick	8	53	1,080	0	0

Table 1: *Table of the number of output curves generated, input segments, output segments, input branches, and output branches.*

to the graph if they improve the result which they often according to Equation 1.

4 Results and Analysis

Our method was tested on example curves corresponding to different kinds of natural and man-made objects. It was used to design the branches of many chandeliers in Figure 1. Automatically generated curves were used to design candlesticks (Figure 10) and drinking glasses (Figure 12) by revolving the curves around an axis. It was also used to model fancy cabinet handles in Figure 9, bolts of lightning (Figure 7), and river systems (Figure 8). The output curve computed by our system is a piecewise Bezier curve.

The following weights were used to produce each result: $w_L = 10, w_R = 80, w_d = 2000$, and $w_s = 1000$. The values of w_i, d_i, and k_B are computed using distances measured in pixels. The bins were set to $\theta_B = 12^\circ$ and $k_B = 0.02$. The same values were used for all the results. They were not tweaked for any particular result, but they could be adjusted for example to increase or decrease the number of branches by changing w_R.

The results in each figure were produced with some brief user interaction. The user first sketches a set of example curves. Then a new set of curves is generated. The user moves points on this set and in response the set grows or contracts in real-time. The entire process is very brief. The individual curves in the figures were generated in seconds. All of the curves in each figure were created in only a few minutes.

Analysis: The final shapes produced by our algorithm depend on the set of example curves and the adjustment points added by the user as part of the editing. Our underlying algorithm ensures that the local features of the output curves, in terms of tangent vectors and curvature, are similar to the features of the example curves. However, the stretching deformation can alter these features. The running time of the algorithm varies as a function of number input segments, curve degree and the number of branch points in the final curve. In practice, our algorithm is very fast and can generate new curves at interactive rates on a desktop PC for curves with many

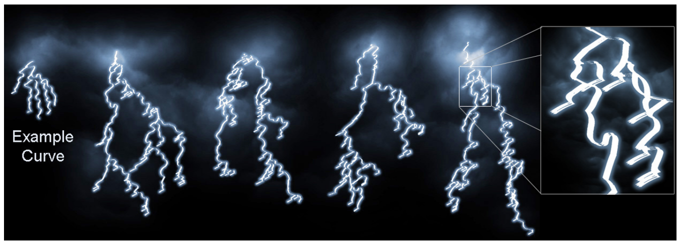

Figure 7: *From a sketch of a few bolts of lightning, several more complex lightning patterns are generated in under two minutes. The output curves contain about 150 branches and over 12,000 segments.*

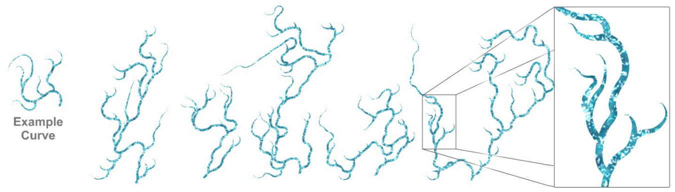

Figure 8: *From a sketch of a river system, several larger and more complex river systems are automatically generated in under two minutes. The output curves each have an average of about 80 branches and 10,000 segments.*

segments.

If the input curves contain branches, the output will also as shown in Figure 1, 7, and 8. Likewise, if the input contains cusps, the output will also as shown in Figures 7, 9, 11 and 12.

5 Comparison and Limitations

Much of the prior work in procedural modeling has focused on modeling plants using L-systems. Our method has rules similar to L-systems. For example, the acceptable sequences of segments described in Section 3.2 could be generated using a complex L-system. Overall, we expect that prior algorithms based on L-systems would generate higher quality models of plants and trees. In constrast, our curve synthesis algorithm is faster and more

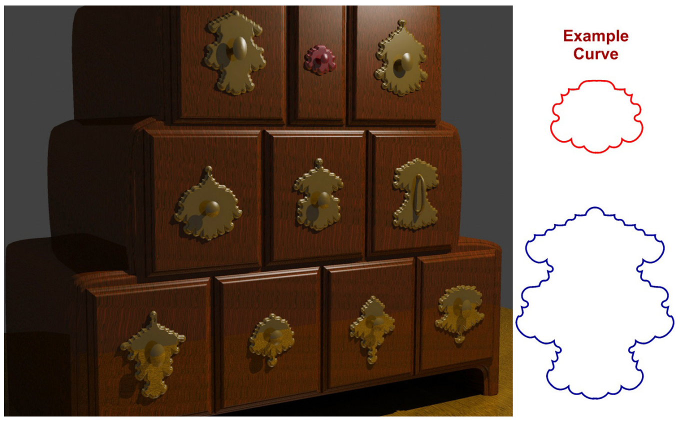

Figure 9: *From a simple sketch of the handle of a cabinet, several complex cabinet handles are designed automatically in the same style. The example handle is the top center red-tinted handle.*

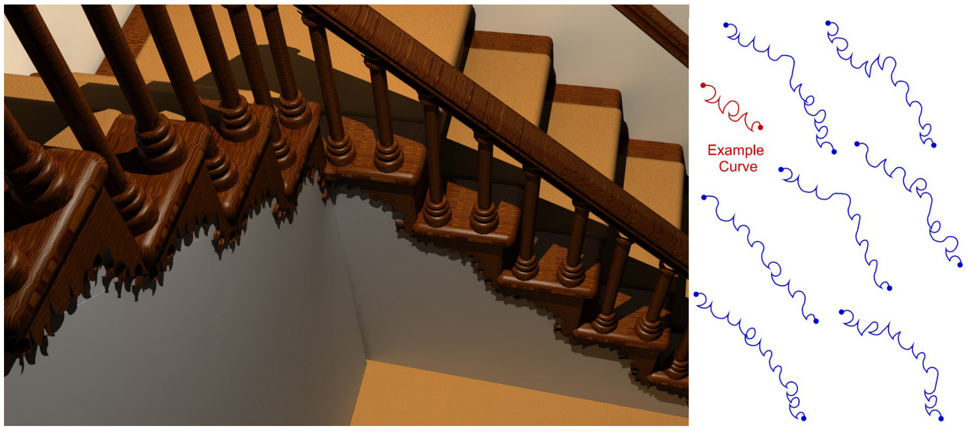

Figure 11: *Many curves are generated to add decoration beneath every step of a staircase. The generated curves are shown on the right. The curves were generated in under two minutes.*

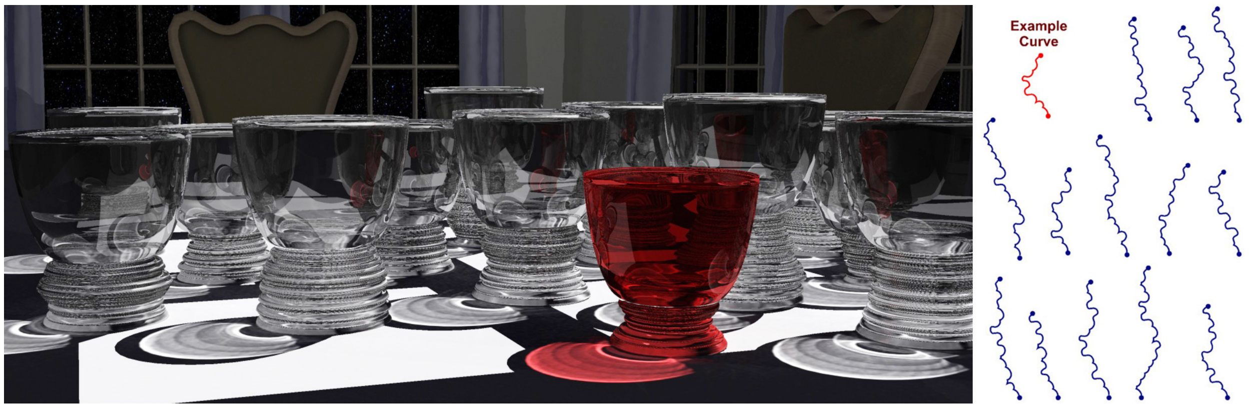

Figure 12: *The profile of a drinking glass is sketched (red curve) and used to design many glasses in a similar style (blue curves). The glasses are modeled by revolving the curves around an axis. The surface of revolution obtained from the example input curve is shown as red tinted glass.*

amenable to user-driven changes and modifications. The main benefit of our approach is in generating models of curved man-made objects (e.g. decorative objects) such as furniture and household items. We are not aware of any prior procedural methods for generating such curved objects. As a result, tools that are general and that can model a wide variety of objects are highly valuable. Our method can model a wide variety of objects because it is example-based. The user can easily switch between creating two very dissimilar types of objects by sketching a new example curve.

Our method has some similarities with the curve analogies algorithm [Hertzmann et al. 2002] since both methods are based on using example curves. However, our method is designed with a different application in mind. The curve analogies method is designed to take curves drawn in one style and draw them in a different style. Curves have both a local and a large-scale structure. Our method and curve analogies both attempt to reproduce the local structure of a set of example curves. The difference is that in curve analogies the large-scale structure of the new curves is based on a user's sketch while in our method it is automatically generated. Our method is designed for cases where the user has not determined the large-scale structure or would like to rapidly generate many different sets of curves which vary widely on a large scale. Also, our method is designed to generate set of curves that form exact closed loops and branches with a minimal amount of stretching.

Another method that is based on using example is model synthesis, but it is not well-suited for generated curved shape since curved object must either be created on a grid [Merrell 2007] or would consume excessive time and memory [Merrell and Manocha 2008].

Limitations: As explained in Section 3.4, the curves are generated by making small incremental changes and testing if the changes improve a cost function. In some ways, our algorithm is performing an optimization in the design space and there is a risk that the current solution may fall within a local minima of the cost function meaning that small incremental changes would not improve the solution and large changes would be needed. The shape of the final object depends on the input curve and where the user moves the adjustment points. In some cases, our approach can result in unnatural shapes. Our algorithm can produce self-intersections, and we need to explicitly check for them.

6 Conclusion and Future Work

We have presented a method for taking a set of curves sketched by a user and automatically generating a new set of curves that resemble the input. Parts of each new curve closely resemble parts of the example curve because they have the same geometric characterization including tangent vectors, curvature and branches. The example curve is divided into segments. These segments are bent

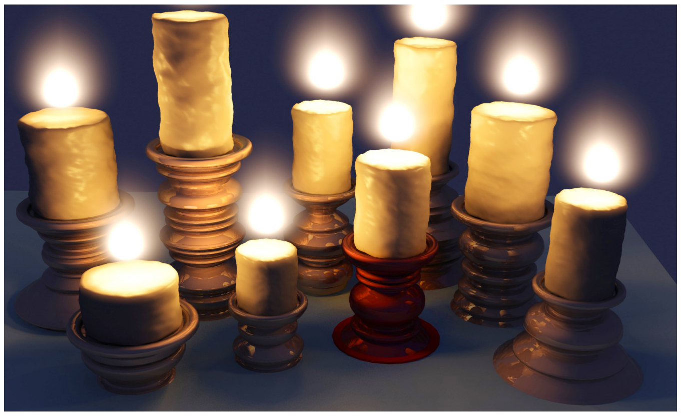

Figure 10: *The profile of a candlestick is sketched and used to design many similar candlesticks in under one minute. The curves are revolved around an axis to produce these candlesticks. The example candlestick is tinted red.*

and stretched and rearranged to fit a cost function. The cost function gives poor scores to curves that follow the user's control points, that do not self-intersect, and that have many branches. We have applied the algorithm to different input curves and used to generate curved models of different man-made objects and some natural patterns.

There are many avenues for future work. We can improve the user interface and use more sophisticated physically-based deformation algorithms for stretching the curves. We would like to use our algorithm to generate more kind of models include architectural structures and outdoor scenes. The set of input curves can also include subdivision curves, and our approach can be extended to generate non-planar 3D curves as well as 4D space-time curves for animation. Each segment in the output curve is a translated copy of a segment in the input curve. We would like to extend our method to allow rotated copies also. This is useful for objects that are not orientation-sensitive such as the river systems in Figure 8. This is in contrast to the chandeliers in Figure 1 which must have a certain orientation to hold up the candles. Finally, we would like to extend our approach to procedurally generate 3D surface models composed of freeform surfaces.

References

ALIAGA, D. G., VANEGAS, C. A., AND BENEŠ, B. 2008. Interactive example-based urban layout synthesis. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, ACM, New York, NY, USA, 1–10.

BARLA, P., BRESLAV, S., THOLLOT, J., SILLION, F. X., AND MARKOSIAN, L. 2006. Stroke Pattern Analysis and Synthesis. 663–671.

BHAT, P., INGRAM, S., AND TURK, G. 2004. Geometric texture synthesis by example. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, ACM Press, New York, NY, USA, 41–44.

CHEN, G., ESCH, G., WONKA, P., MÜLLER, P., AND ZHANG, E. 2008. Interactive procedural street modeling. In *Proc. Of ACM SIGGRAPH '08*, ACM, New York, NY, USA, 1–10.

CHEN, X., NEUBERT, B., XU, Y.-Q., DEUSSEN, O., AND KANG, S. B. 2008. Sketch-based tree modeling using markov random field. *ACM Trans. Graph. 27*, 5, 1–9.

EFROS, A. A., AND LEUNG, T. K. 1999. Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision*, 1033–1038.

FREEMAN, W. T., TENENBAUM, J. B., AND PASZTOR, E. C. 2003. Learning style translation for the lines of a drawing. *ACM Trans. Graph. 22*, 1, 33–46.

HERTZMANN, A., OLIVER, N., CURLESS, B., AND SEITZ, S. M. 2002. Curve analogies. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, Eurographics Association, 233–246.

IGARASHI, T., MATSUOKA, S., AND TANAKA, H. 1999. Teddy: a sketching interface for 3d freeform design. In *Proc. Of ACM SIGGRAPH '99*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 409–416.

IJIRI, T., OWADA, S., OKABE, M., AND IGARASHI, T. 2005. Floral diagrams and inflorescences: interactive flower modeling using botanical structural constraints. In *Proc. Of ACM SIGGRAPH '05*, ACM Press, New York, NY, USA, 720–726.

KALNINS, R. D., MARKOSIAN, L., MEIER, B. J., KOWALSKI, M. A., LEE, J. C., DAVIDSON, P. L., WEBB, M., HUGHES, J. F., AND FINKELSTEIN, A. 2002. WYSIWYG NPR: Drawing strokes directly on 3D models. *Proc. Of ACM SIGGRAPH '02 21*, 3 (July), 755–762.

KWATRA, V., SCHDL, A., ESSA, I., TURK, G., AND BOBICK, A. 2003. Graphcut textures: Image and video synthesis using graph cuts. *Proc. Of ACM SIGGRAPH '03*, 277–286.

MERRELL, P., AND MANOCHA, D. 2008. Continuous model synthesis. *ACM Trans. Graph. 27*, 5, 1–7.

MERRELL, P. 2007. Example-based model synthesis. In *I3D '07: Symposium on Interactive 3D graphics and games*, ACM Press, 105–112.

MĚCH, R., AND PRUSINKIEWICZ, P. 1996. Visual models of plants interacting with their environment. In *Proc. Of ACM SIGGRAPH '96*, 397–410.

MÜLLER, P., WONKA, P., HAEGLER, S., ULMER, A., AND GOOL, L. V. 2006. Procedural modeling of buildings. *ACM Trans. Graph. 25*, 3, 614–623.

POTTMANN, H., LIU, Y., WALLNER, J., BOBENKO, A., AND WANG, W. 2007. Geometry of multi-layer freeform structures for architecture. In *Proc. of ACM SIGGRAPH '07*.

POWER, J. L., BRUSH, A. J. B., PRUSINKIEWICZ, P., AND SALESIN, D. H. 1999. Interactive arrangement of botanical l-system models. In *I3D '99*, ACM, New York, NY, USA, 175–182.

SIMHON, S., AND DUDEK, G. 2004. Sketch Interpretation and Refinement Using Statistical Models. In *Proceedings of Eurographics Symposium on Rendering 2004* , EUROGRAPHICS Association, A. Keller and H. W. Jensen, Eds., 23–32, 406.

SINGH, K., AND FIUME, E. 1998. Wires: a geometric deformation technique. In *Proc. Of ACM SIGGRAPH '98*, ACM, New York, NY, USA, 405–414.

WONG, M. T., ZONGKER, D. E., AND SALESIN, D. H. 1998. Computer-generated floral ornament. In *Proc. Of ACM SIGGRAPH '98*, ACM, New York, NY, USA, 423–434.

WONKA, P., WIMMER, M., SILLION, F., AND RIBARSKY, W. 2003. Instant architecture. In *Proc. Of ACM SIGGRAPH '03*, 669–677.

WU, Y., ZHANG, H., SONG, C., AND BAO, H. 2008. Space-time curve analogies for motion editing. In *GMP*, 437–449.

ZELINKA, S., AND GARLAND, M. 2003. Mesh modelling with curve analogies. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches & Applications*, ACM, New York, NY, USA, 1–1.