Skippy: Single View 3D Curve Interactive Modeling

VOJTĚCH KRS, Purdue University
ERSIN YUMER, Adobe Research
NATHAN CARR, Adobe Research
BEDRICH BENES, Purdue University
RADOMÍR MĚCH, Adobe Research

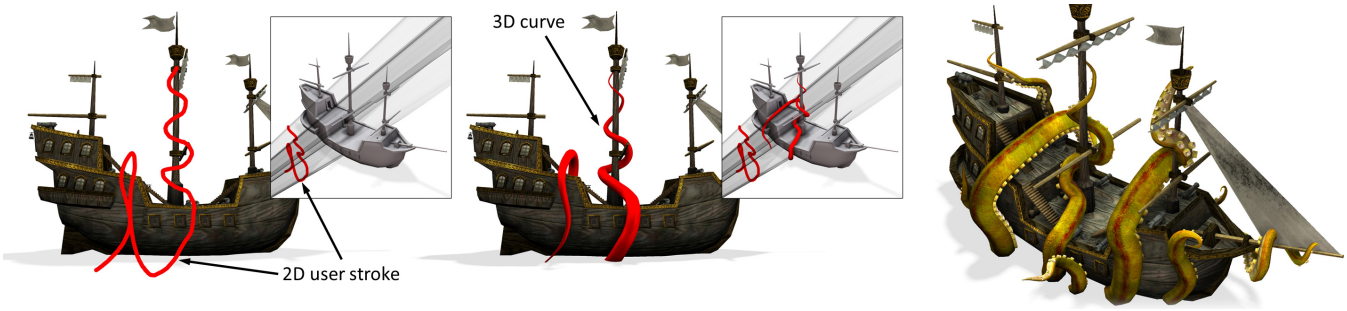


Fig. 1. The user draws a 2D stroke in front of the 3D model (left). The 2D stroke is converted into a 3D curve (middle). A complete 3D model from mutiple curves is generated within a few seconds (right).

We introduce Skippy, a novel algorithm for 3D interactive curve modeling from a single view. While positing curves in space can be a tedious task, our rapid sketching algorithm allows users to draw curves in and around existing geometry in a controllable manner. The key insight behind our system is to automatically infer the 3D curve coordinates by enumerating a large set of potential curve trajectories. More specifically, we partition 2D strokes into continuous segments that land both *on* and *off* the geometry, duplicating segments that could be placed in front or behind, to form a directed graph. We use distance fields to estimate 3D coordinates for our curve segments and solve for an optimally smooth path that follows the curvature of the scene geometry while avoiding intersections. Using our curve design framework we present a collection of novel editing operations allowing artists to rapidly explore and refine the combinatorial space of solutions. Furthermore, we include the quick placement of transient geometry to aid in guiding the 3D curve. Finally we demonstrate our interactive design curve system on a variety of applications including geometric modeling, and camera motion path planning.

CCS Concepts: • **Computing methodologies** → **Parametric curve and surface models**; • **Human-centered computing** → **Graphical user interfaces**;

Additional Key Words and Phrases: Single View, 3D Curve, Geometric Modeling

ACM Reference format:
Vojtěch Krs, Ersin Yumer, Nathan Carr, Bedrich Benes, and Radomír Měch. 2017. Skippy: Single View 3D Curve Interactive Modeling. *ACM Trans. Graph.* 36, 4, Article 128 (July 2017), 12 pages.
DOI: http://dx.doi.org/10.1145/3072959.3073603

This work has been sponsored by Adobe Research and by the National Science Foundation grant #1606396 *Haptic-Based Learning Experiences as Cognitive Mediators for Conceptual Understanding and Representational Competence in Engineering Education*
Author's addresses: Vojtěch Krs and Bedrich Benes, Computer Graphics Technology, 401 N Grant St, West Lafayette, IN 47907. Ersin Yumer, Nathan Carr and Radomír Měch, Adobe Research, 345 Park Ave, San Jose, CA 95110.

DOI: http://dx.doi.org/10.1145/3072959.3073603

1 INTRODUCTION

Computer graphics has achieved impressive results in areas such as rendering and computer animation. However, 3D modeling still poses many challenges; one of them is expressing user intent by simple means. User interaction, which is at the heart of modeling, is where most of the related complexity still exists. This is mainly due to the fact that most input and display devices currently in use are 2D, which is not intuitive to human interaction with the world, where we operate and think in 3D. To overcome the loss of depth, the user is usually forced to change the viewpoint, rotate the object, or use multiple viewports at once [Bae et al. 2008]; which can lead to a loss of efficiency.

A particularly difficult problem is drawing of 3D curves. These *space curves* are important to a variety of tasks such as planning of trajectories of dynamic objects, for example, particle systems or virtual cameras, design of curved surface patches, such as NURBS, or swept surfaces, or generalized cylinders. The main problem of drawing 3D curves in 2D is that there is an infinite number of possible configurations of the curve in the missing dimension, and, as explored by Schmidt et al. [2009a], even expert users have trouble with drawing 3D objects and curves. The foreshortening caused by perspective projection is especially difficult to get right and the resulting objects hardly match user's intent. While the use of shadows as visual depth cues has been shown to improve spatial

understanding [Cohen et al. 1999]; specifying the shape of curves when drawing in 3D poses an additional set of challenges.

Prior work has addressed this issue by using constraints and additional sources of information to infer the desired curve's shape. One of the common problems not addressed by the previous work is drawing behind occluding surfaces or drawing curves with high curvature and torsion, such as spirals. One of the underlying mechanisms that makes this such a hard task is that our prior knowledge of the object's shape alters our perception [Taylor and Mitchell 1997]. As noted by Matthews and Adams [2008], "people seem to draw what they know rather than what they see".

One way to make drawing in 3D easier is to impose assumptions about the underlying form, such as regularity, symmetry, planarity, and orthogonality [Bae et al. 2008; Kara and Shimada 2007; Schmidt et al. 2009b; Xu et al. 2014]. While these approaches work well in practice they are often restricted to a certain class of shapes, which may limit artistic expressiveness.

Reasoning about the 3D representation of the sketched curve using other 3D objects in the scene is a more practical and less disturbing approach from the user's point of view since it does not require any change of viewport at the time of drawing. De Paoli and Singh [2015] used this insight for modeling shapes around already existing 3D geometry. Their approach is limited to shorter and fully visible or partially visible symmetrical curve segments, which apply well to local shape modeling.

We present *Skippy*, a new method for sketching 3D curves from a single viewpoint using both existing or transient 3D geometry for shape inference. Our method enables the user to draw arbitrarily long, smooth curves that are placed at various depths between objects in the scene and that can also be partially occluded at authoring time (Figure 1). After being drawn the 3D curve can *skip* between different depths around the objects by clicking on the part of the curve that is visible or obstructed by an object. The user can also click on any surface and add transient guiding objects. Such temporary geometry is then also used to guide drawing of the curve without having to change the viewpoint. Furthermore, our system allows a quick intuitive way for users to guide and control the *skipping* behavior, allowing curves to be easily woven in a complex manner through the negative space surrounding any geometry. We do not place underlying assumptions about these curves (i.e., that they form surfaces, or represent regular geometric forms), and as such our curves can be used for a variety of applications from motion path planning, surface decoration, and even shape design. Finally we demonstrate a number of intuitive overdrawing mechanisms that enable iterative refinement of curve solutions interactively. Our main contributions are as follows:

- an intuitive approach for drawing 3D curves with varying depth using only 2D input by leveraging existing 3D shapes as guides,
- an efficient graph data structure that stores valid variations of the 3D curve for the input 2D stroke and enables real-time interaction with the curve, and
- a set of novel editing operations specifically for *skip* editing, re-drawing, and via anchoring through template 3D shapes.

2 RELATED WORK

Positioning 3D curves using a 2D interface poses numerous challenges, a number of which have been tackled in the sketch based modeling literature. We refer the reader to [Olsen et al. 2009] and a recent survey [Kazmi et al. 2014] for a more complete overview of this domain. To highlight the most relevant work in this space, we divide related work into a set of broad categories; each relying on different sets of underlying assumptions which ease the 3D curve drawing process.

Design Curve Modeling. The early work of Cohen et al. [1999] presented a single view interface for designing 3D space curves. The novel idea of this work was to rely on shadows as additional depth cues. This conceptual idea is orthogonal to our approach and we leverage a form of shadowing (i.e., real-time screen space ambient occlusion) in our interface to improve spatial understanding. The work of Cohen et al. [1999] did not directly address the tedious nature of specifying curve shape during the drawing process.

One way to make drawing in 3D easier is to choose an angle that minimizes the foreshortening. A popular approach, investigated for drawing 3D curves, is to rely on epipolar geometry. This allows sketching from a second viewpoint to find a unique solution to the curve's shape [Bae et al. 2008], or choosing two view orthogonal directions [Karpenko et al. 2004]. Such approaches either require a symmetric structure to be drawn, or consistent change of viewpoint; limiting artistic expressiveness.

An important aspect that is often considered during object sketching are occlusions. Cordier and Seo [2007] construct self-occluding objects from free-form sketches using constrained optimization. Similarly, McCrae and Singh [2008; 2009] , use clothoid splines, to infer 3D road networks from sketches with self-crossings. McCann and Pollard [2009] order interactively 2D objects with local overlaps, which Igarashi and Mitani [2010] extended to 3D, and LayerPaint [Fu et al. 2010] allows users to paint on occluded surfaces using a multi-layer approach.

Another approach that helps with 3D modeling is leveraging existing geometry and environment. Coleman and Singh [2006] presented a method that adapts existing rough 3D curves to their surroundings. Turquin et al. [2007] allows users to sketch clothes on 3D mannequins from a single view by inferring the 3D position of sketched curves. Furthermore, 3D curves inferred from sketches have been used for hair design [Fu et al. 2007; Wither et al. 2007]. OverCoat [Schmid et al. 2011] uses proxy geometry to embed brush strokes in 3D space and enables sculpting of the underlying proxy geometry with brush strokes. Perhaps most closely aligned with our method is that of De Paoli and Singh [2015] who presented SecondSkin. Rather than allowing curves to be drawn directly *on* a surface, it allows artists to sketch design curves in the nearby shell offset space surrounding an object; providing many interesting design operations. In contrast, our system allows the user to simultaneously sketch over large collections of shapes, deals with dense occlusions, specifies curves that are further from the surface, all without changing view.

Organic Surface Modeling. The ambiguity of taking 2D sketch curves and producing 3D content can be reduced by assuming that the sketch curves represent some underlying *organic* form. Both the

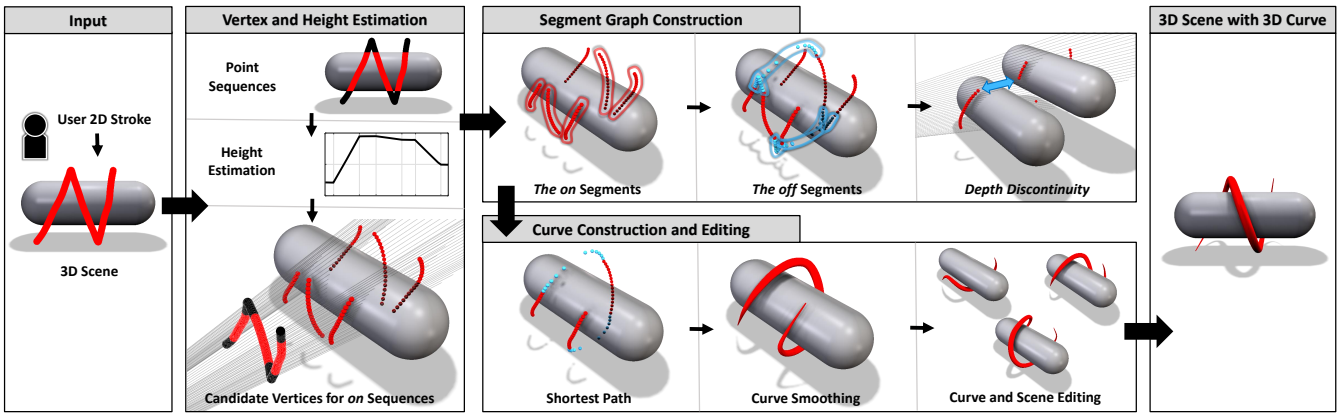


Fig. 2. Skippy overview. A 3D scene is displayed from a single view and the user draws series of 2D points that are converted into a 3D curve that passes through the scene. First, 3D candidate vertices are found for intersecting parts of the stroke. A segment graph is built from groups of the candidate vertices and is used to store all possible valid curves. The best curve is selected and converted into a full 3D curve. The user can quickly select different curves and skip from one solution to another.

seminal work of Teddy [Igarashi et al. 1999] and Fibermesh [Nealen et al. 2007], begin the design process by assuming the curves reside on silhouette edges of some inflated base shape. Karpenko et al. [2004] presented SmoothSketch which extended this notion by analyzing T-intersections and cusps in the drawing to enable the creation of more complex base shapes. Initial 3D base forms can be used to anchor more complex curve sketching operations. For example curves drawn directly *on* a surface can be pulled and tugged to deform the underlying shape. These surface curves can also be used as localized regions for extrusion [Igarashi et al. 1999; Nealen et al. 2007]. Both these works demonstrate that the presence of an existing 3D shape can bootstrap the 3D drawing process enabling the creation of more complex form. Recently, 2D curves have been used to construct 3D cartoon canvases in [Bessmeltsev et al. 2015]. We take inspiration from these works, however, we focus our attention on populating the empty space between shapes allowing our designers to bootstrap the 3D curve design with the types of complex 3D models that can easily be found on the web or in shape repositories.

Surface Modeling using Curve Networks. An alternative approach to aid users in creating 3D space curves is to assume the curves represent underlying man-made structure. The SKETCH interface [Zeleznik et al. 1996] starts by allowing the user to sketch box like forms which anchor additional 3D space curve sketching operations. Schmidt et al. [2009b] uses sketching on an initial ground plane to build a scaffold which acts as a set of visual constraints for sketching additional 3D curves. Photographs have also been used in conjunction with sketches to disambiguate form [Lau et al. 2010]. Xu et al. [2014] infer 3D curve networks from a given 2D sketched design by assuming implied regularities, such as planarity, curvature, symmetry, and parallelism. In contrast, we target our system at freeform space curves which may or may not directly participate in defining some underlying surface.

3D Drawing using Shape Priors. By restricting the class of target shapes to conform to some underlying model (i.e. procedural or otherwise), many drawing operations can be simplified. Just recently, deep neural networks were used to automatically infer 3D architectural procedural models from 2D user single view sketches [Nishida et al. 2016]. Chen et al. [2008] used Markov random fields with sketching to reconstruct 3D models of vegetation. Automatic character model reconstruction from 3D sketches has also been demonstrated [Buchanan et al. 2013]. Drawing assistance and recommendation can also be achieved using large pre-existing 3D shape collections. For example, shadows have been used to guide the users during drawing by leveraging a database of 3D template objects [Fan et al. 2013]. While the use of strong shape priors can greatly enhance 3D drawing, they can also potentially limit artistic freedom. Our system focuses more on freeform design and does not impose such restrictions.

3 METHOD OVERVIEW

Our goal is to infer a 3D curve shape from 2D strokes so that it follows the 3D scene geometry, with optional transient guide objects that can be added by the user. The user's 2D stroke is converted to a set of equidistant points in the view plane. Given these points and distances to geometric surfaces in the scene, we estimate 3D locations. This process may generate multiple candidate 3D locations for each 2D sample. Based on a curvature criterion we then find a combination of these 3D points that makes up the 3D curve.

Figure 2 shows an overview of our method. The input is a 3D scene that may also contain a set of transient objects. The 3D curves are input as sequences of strokes converted into sets of equidistant points in a 2D viewing plane. The output is a 3D scene that includes both the input scene and the added objects. The only requirement for the 3D scene representation is that it must be possible to find a distance field around it, since it is needed for fast distance calculations. After a stroke is drawn, the user can change the viewing direction and the curve is extended as the user continues to draw more strokes or modified when the user overdraws an existing part of the curve. We infer a 3D curve for each set of 2D strokes, so that the user can instantly see the modification during the sketching process.

The ***vertex and height estimation*** (Section 4) is the very first step of our pipeline. Note that we denote the 3D counterparts of 2D stroke points as vertices. The first step takes the input scene and the input 2D points and finds a set of candidate 3D vertices in 3D space for each 2D point and their distance from the geometry that we call the vertex height. The candidate vertices are the possible locations of the point in the 3D space. In order to create this set, we first project the set of 2D points into the 3D scene by constructing a set of rays, and classify the 2D points based on their intersection with the 3D objects as *on* and *off* points. We then use the non-intersecting rays to estimate the height for each sequence of *on* 3D vertices. Given the rays and the estimated heights, we can find a set of candidate vertex sequences for the *on* 2D points.

The ***segment graph construction*** (Section 5) step creates a graph of vertex segments at varying depths that is later used to find the optimal 3D curve. Segments are 3D polylines connecting given vertices. First, the segments corresponding to *on* points are constructed, forming the nodes of the segment graph. The edges of the segment graph represent the parts of the input stroke that did not intersect the geometry. These edges, which we call *off* segments, are constructed between individual *on* segments. An additional step is performed to handle depth discontinuities. This step adds nodes and edges to the segment graph that help to find a better solution.

The ***curve construction*** (Section 6) step, finds the best path through the segment graph and constructs a smooth cubic spline. First, both the segment graph nodes and edges are scored using a curvature criterion. Then the best path through the segment graph is found and the segments along this path are concatenated into a single curve. Finally, the curve is re-sampled and iteratively smoothed. Additionally, editing operations such as change in depth or redrawing are facilitated.

4 VERTEX AND HEIGHT ESTIMATION

The input to our method is a 3D scene and a set of strokes that are sampled into sets of 2D points.

The first step of the pipeline takes the sequence of the input 2D points and the geometry and splits it into a sequence of *on* and *off* points that lie on the scene geometry (Figure 3). Afterwards, it generates the candidate vertices in 3D for the *on* sequences.

4.1 Point Sequences

The sequence of 2D input points is provided by the user in a single stroke. The input point sequence have varying distance, therefore we resample them so that the new sequence is equidistant. We denote the new point sequence by $P = (p_1, p_2, \ldots, p_{|P|}) \mid p_i \in \mathbb{R}^2$, where $|P|$ is the number of points.

We then divide the sequence of points into points that are *on* and *off* geometry after projection (Figure 3 and see also [De Paoli and Singh 2015, page 4]). We perform an initial projection of the 2D points p_i by casting a ray with direction r_i from the camera position c and finding intersections with the objects in the scene.

The rays that intersect the geometry will define *on* points while the non-intersection ones will determine *off* points. There is an implicit ordering of the rays r_i that is defined by the sequence of the input points p_i. Therefore, there is also an implicit ordering of the vertices v_i in the 3D space. Moreover, the points that are

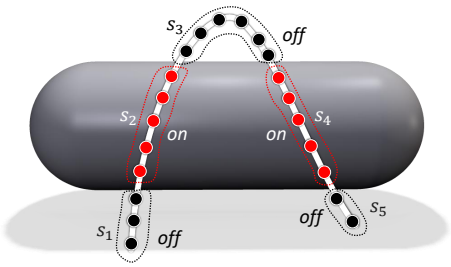


Fig. 3. The input 2D stroke is divided into on and off sequences.

off geometry will later be used to determine the distance of the final curve from the actual object. The points are then grouped into successive sequences $S = \langle s_1, s_2 \ldots, s_{|S|} \rangle$ with a flag whether it is *on* or *off* .

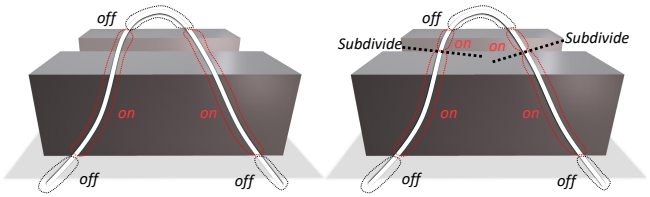


Fig. 4. The sequence of on 2D points with depth discontinuity (left) is divided into two on sequences (right).

A special care needs to be taken for sequences of *on* points with *depth discontinuity* such as in Figure 4. The situation indicates multiple obstructed surfaces and the *on* sequence needs to be split in two. In our implementation we parse all *on* sequences and we perform a check for each pair of subsequent points by comparing the distance of the ray intersections. If the intersections are far apart relative to the distance of their screen coordinates (three times or more in our implementation, assuming unit cube workspace and unit square screen), we check whether the slope between those two intersections is continuous. We cast an additional ray in between the successive points and check if the new intersection's distance is close to the average of the distance of the involved points (40% relative depth change or less). Otherwise if the distance of the intersection of the new ray is closer to either of the two vertices, the sequence is split in two.

4.2 Height Estimation

Our goal is to draw the curve at a certain distance from the geometry. Although it would be possible to ask the user for an explicit distance value input, we use a more intuitive way to infer the distance from the actual stroke. In particular, the distance of the curve from the geometry is derived from the distance of the rays defining the *off* points.

We call the distance of the final curve from the geometry its *height* and we denote it as h. Height is a function that returns the value for an input point p_j or a sequence s_i. All points in an *off* sequence s_i have their height $h(s_i)$ constant. It is found as the maximum distance of the rays that define the segment from the scene

geometry (Figure 5). The height of an *on* sequences is found by the linear interpolation of the heights of the neighboring *off* sequences (Figure 5). If the user starts or ends drawing on the object's surface the *on* sequence does not have two neighboring *off* sequences and we set the start or the end of the corresponding *on* sequence to zero.

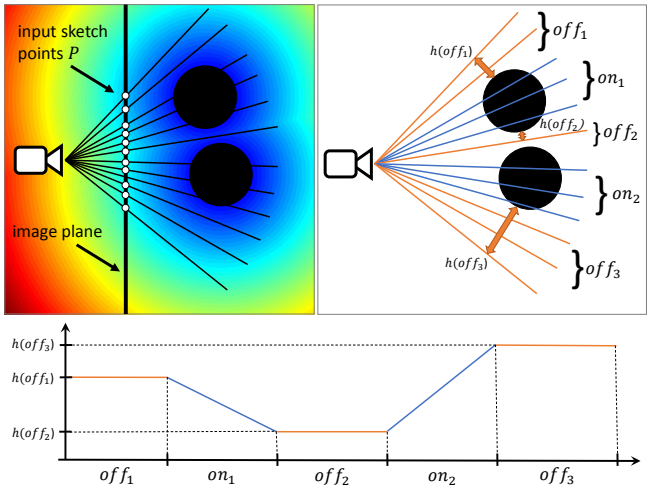


Fig. 5. Height of the off sequences is constant and given by the color-coded distance field (left). Height of the on sequences is interpolated from the neighbors.

After this step all *off* sequences have constant height $h(s_i)$ and all *on* sequences have their height interpolated.

The result of the distance estimation step is a mapping of the input points to the distance from the scene geometry.

4.2.1 Candidate Vertices for on Sequences. Next we find candidate vertices v but only for *on* sequences; the *off* sequences are processed differently in Section 5. We generate a distance field df for the entire scene by using the L_2 norm (Figure 5 top left). The distance field significantly speeds up the distance calculations.

In order to generate the candidate vertices, we again cast a ray r_i for each point from each *on* sequence. The candidate vertices are found as intersections with the isosurface at distance $df(h_i)$.

This step generates a large number of candidate vertices, some of which are unnecessary and can be removed. In particular, if we encounter two volumes in a row intersecting the ray, the space in-between them will be filled by two candidate intersections: one for the "after" the first object and one for "before" the second one. However, in practical experimentation the user usually does not need to have such a small level of refinement and one point is usually enough for curve editing. We choose to discard the point that is close to the back-face in our implementation, shown as "Discarded (in-between)" in Figure 6. The user may not expect the multiple points in the middle that may cause jumps in depth in the 3D curve construction. Moreover, this step prunes the amount of vertices and speeds up the computation. However, all discarded vertices are kept in the system and can be later accessed by editing operations.

The candidate vertices for *on* sequences are denoted v_i^j and are indexed in two ways. The lower index (Figure 6) corresponds to the index of the ray r_i that is also given by the index of the point p_i in the re-sampled input stroke. The upper index j is the ordering number of the intersection on the ray, with zero being closest to the camera. Moreover, we also assume all vertices in the first *on* sequence are not occluded i.e., the user starts drawing either off or in front of the geometry.

5 SEGMENT GRAPH CONSTRUCTION

The previous step created candidate vertices for *on* points and identified rays that do not intersect the geometry (potential *off* points). It also created the height function for all *off* and *on* sequences.

In order to generate the curve, we could consider individual combinations of all vertices. However, a ray $r_i, i = 1, 2, \ldots, n$ can generate multiple candidate vertices $\langle v_i^0, v_i^1, \ldots, v_i^k \rangle$ and the number of possible combinations greatly increases with each added ray and possible depths at which the candidate vertices can lie. The number of possible curves increases with $O(k^n)$. Therefore it is not feasible to calculate the curve for every combination. Fortunately, many of the combinations can be trivially rejected, for example, the connecting edge should not intersect geometry or they should have a similar distance from the camera (see details in Section 5.1).

We group candidate vertices into *segments* denoted by g. A segment is a 3D polyline that connects candidate vertices. We further classify the segments into *on* segments $\bar{g}$ and *off* segments $\hat{g}$.

After all segments were found we construct a *segment graph* that includes all possible valid (following constraints listed in Section 5.1) curves in the 3D space for the given stroke. The segment graph $\mathcal{G} = \{N, E\}$ has a set of nodes N that correspond to the *on* segments and the edges E correspond to the *off* segments (Figure 6 right). From the segment graph we automatically offer the best segment and let the user select a different one during the scene editing, for example, to skip part of the curve to a different depth.

The inputs to the segment graph construction algorithm are the scene geometry, the *on* candidate vertices, and the *off* rays. The segment graph construction is a three step process that is described in detail below. First, we connect candidate vertices for each *on* sequence on the corresponding side of the geometry and create *on* segments. Then we connect the *on* segments by generating *off* segments. There is a special case of the depth discontinuity (Figure 4) which we further discuss in Section 5.3.

5.1 The *on* Segments

We create the *on* segments by connecting all candidate vertices of *on* points. Each *on* segment is denoted $\bar{g}_{se}^j$, where j is the intersection order as above, and the lower index se denotes the start and then the end vertex. For example in Figure 6 we have $\bar{g}_{69}^1 = \langle v_6^1, v_7^1, v_8^1, v_9^1 \rangle$.

These segments will become the nodes N of the segment graph $\mathcal{G}$ (Figure 6 right). We create the *on* segments by connecting individual candidate vertices of consecutive *on* points and then group them into the longest segments that can be found. Individual connections of candidate vertices are tested against three criteria: 1) the connection does not intersect the geometry (example in Figure 6: "Discarded (no connection)"), 2) the vertices lie at similar distance from the

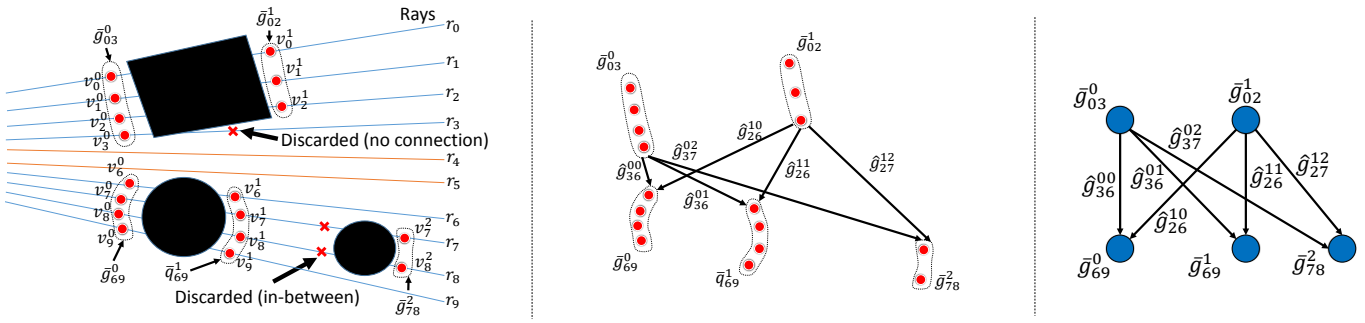


Fig. 6. Segment creation and indexing from the candidate vertices for on segments (left), the possible connections of the on segments (middle), and the corresponding segment graph (right).

camera, and 3) the gradient of the distance field is similar at the vertex position. We then find all segments that start at a candidate vertex with no inbound connections and end at a candidate vertex with no outgoing connections.

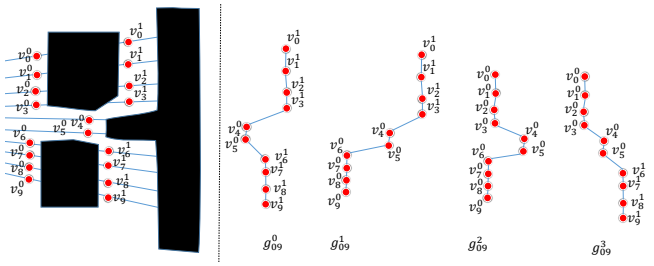
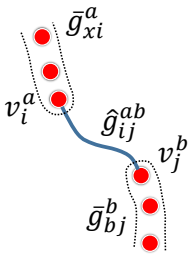


Fig. 7. Depth discontinuity of an on segment (left) will lead to multiple on segments (right).

While this construction is simple for a short sequence of vertices on a single side of geometry, it can be more complicated in cases of depth discontinuity (Figure 4). Each ray can generate multiple candidate vertices at different distances (vertices with varying upper index in Figure 7), but the discontinuity will tend to merge and split the segments that would generate large zig-zag steps in the 3D curve. By applying the above-described construction we obtain an acyclic oriented graph (the direction of the stroke defines the orientation). From this we extract all segments that start from the candidate vertex with the lowest index and end in the candidate vertex with the highest one ($start = \{v_0^0, v_0^1\}$, $end = \{v_9^0, v_9^1\}$ in Figure 7). Because of the varying depth of the vertices in each segment, the upper index of the *on* segment is given by the order in which it was extracted ($g_{09}^0, g_{09}^1, \ldots, g_{09}^3$ in the example in Figure 7). We store *all* those segments as nodes of the segment graph, because they provide alternatives for the final curve construction. A key idea behind our approach is that we can select the subset of curve segments that lead to a desired outcome (e.g., a final curve that is smooth with a monotonic curvature).

5.2 The *off* Segments

After all *on* segments are constructed we can connect them by constructing the *off* segments. Because there are multiple combinations on how the segments can be connected, we use the graph $\mathcal{G}$ where the *on* segments are its nodes and *off* segments correspond to the graph edges as shown in an example in Figure 6 middle. Each *on* segment (left) is connected to all segments that are accessible by skipping a sequence of *off* rays. Every such connection is an edge in the segment graph (right).



The *off* segment $\hat{g}_{ij}^{ab}$ corresponds to an edge in $\mathcal{G}$ that connects two *on* segments $\bar{g}_{xi}^{a}$ and $\bar{g}_{jy}^{b}$. More precisely, the *off* segment $\hat{g}_{ij}$ connects the last vertex v_i^a from the first *on* segment with the first vertex of the second segment v_j^b. Note that we need the upper indices for an *off* segment, because the vertices v_i and v_j can be at different depths that would cause multiple *off* edges with the same lower indices.

To find the *off* vertices of the *off* segment $\hat{g}_{ij}^{ab} = \langle w_{i+1}, \ldots, w_{j-1} \rangle$, we interpolate the depth of v_i^a and v_j^b. In other words, we interpolate between $d_i = \|c - v_i^a\|$ and $d_j = \|c - v_j^b\|$, where c is the position of the camera.

The input sketch of the *off* segment defines a surface passing the rays $\langle r_{i+1}, \ldots, r_{j-1} \rangle$ over which we interpolate the depth. Since the user is free to draw virtually any shape, such as loops, zig-zags, etc., we cannot always guarantee the *off* segment to be smooth. For most cases, we use linear interpolation of d_i and d_j to calculate the vertices (Figure 9 bottom). This produces a reasonably smooth segment, especially if the input sketch is also smooth. However, the linear interpolation fails when there is a sharp corner in the input sketch. We detect the corner by using the algorithm from [Ben-Haim et al. 2010] and use sigomoidal interpolation that achieves a smoother result that is closer to a *circular arc* (Figure 9 top)

$$d(t) = d_i(1 - \pi(t)) + d_j\pi(t),$$

where $\pi(t)$

$$\pi(t) = \begin{cases} -\frac{1}{2}\sqrt{1-(2t)^2} + \frac{1}{2} & \text{if } t \in [0, 0.5] \\ \frac{1}{2}\sqrt{1-(2t-2)^2} + \frac{1}{2} & \text{if } t \in (0.5, 1]\,. \end{cases} \quad (1)$$

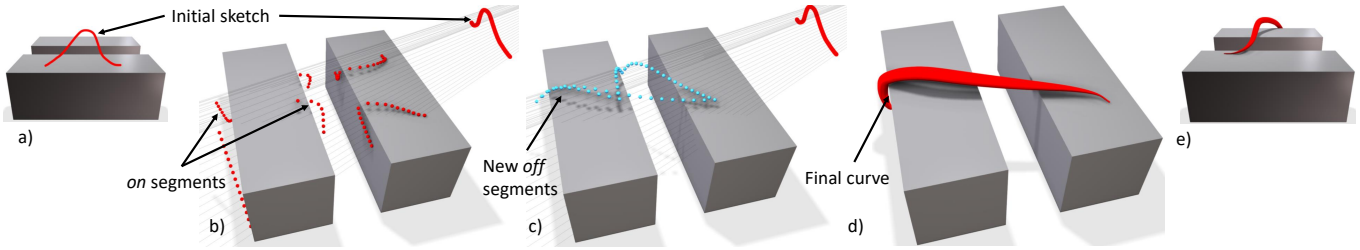


Fig. 8. When depth discontinuity occurs (a), we break the conflicting on segments (b) and insert new off segments that bridge the discontinuity gap (c) and allow for smooth curve generation (d-e).

Furthermore, an *off* segment can begin on the first ray or end on the last ray. In this case we do not use interpolation. For an *off* segment $\hat{g}^{xa}_{0i}$ beginning on the first ray we calculate the average change in depth $\overline{\Delta d}$ of several vertices of the following *on* segment $\bar{g}^{b}_{iy}$. (Up to four vertices were used to calculate $\overline{\Delta d}$ in our implementation.) We then extrapolate the vertices of $\hat{g}_{0b}$ as follows

$$w_q = c + (d_i + (i - q)\overline{\Delta d})r_q. \tag{2}$$

The case of *off* segment ending on last ray, $\hat{g}^{ay}_{i|P|}$ is analogous and the *on* segment used is the previous one $\bar{g}^{a}_{xi}$. To connect these outer *off* segments and maintain the graph structure, we add a node $\bar{g}^{0}_{00}$ or $\bar{g}^{0}_{|P||P|}$ at the beginning or end, respectively, that have zero length.

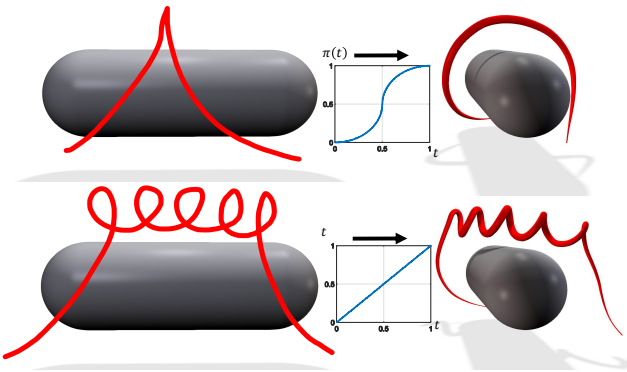


Fig. 9. Off segment interpolation. An off segment with a single corner in initial sketch (top) is interpolated by Equation 1. Otherwise linear interpolation is used (bottom).

5.3 Depth Discontinuity

The last case is a treatment of depth discontinuity (Figure 8). The direct connection of the consecutive *on* segment would generate sharp corners with large depth jumps. To avoid this situation we generate new *off* segments that bridge the *on* segments that participate in depth discontinuity as shown in Figure 8. The *off* segments are added between all *on* segments that are accessible by skipping an *on* sequence that includes a depth discontinuity. Note the new *off* segment is in fact parallel to the *on* segment but connects the *on* segment that is behind the second object. When multiple depth discontinuities occur, the curve generation algorithm described in Section 6 will select the smoothest path in the graph that corresponds to the curve passing behind the last object. This is also the typical intuitive choice of the users, but it can be overridden if needed.

6 CURVE CONSTRUCTION AND EDITING

The previous step created a graph $\mathcal{G}$ that contains interconnected *on* and *off* segments (Figure 2). A number of curves can be generated by finding paths in $\mathcal{G}$ that go through all the rays of the initial sketch. However, not all curves have good visual properties. We use the intuition that the curve should be smooth and follow the curvature of the underlying geometry. Figure 10 shows that if the geometry is round the curve will likely go behind the object.

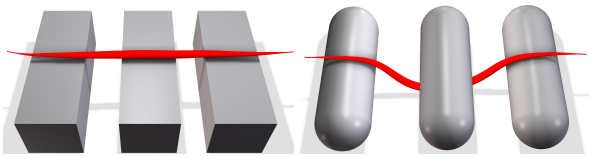


Fig. 10. The path selection attempts to keep the curvature of the off segment constant. In this way the underlying geometry navigates the direction of the curve. The curves were sketched from top-down viewpoint.

Therefore, to select the best path, we define a weight for the nodes and edges of $\mathcal{G}$ that is based on curvature of the individual segments. When a best path through the graph is found, we connect the traversed segments into a single curve. Finally, the curve is resampled and iteratively smoothed, while making sure that the final curve does not deviate from the initial sketch and the height of the curve is preserved.

6.1 Optimal Path

We are not aware of any method to determine the *best* curve based on the above-described criteria. A common approach is to select a curve that does not deviate in depth but our objective is to create curves with varying depth. Another way to select the curve is by considering its fairness as explored by Levien and Séquin [2009], where the authors noted that one of the indicators of a fair curve is its monotonic curvature.

Since we stitch the curve together from segments, we need a criterion that can be evaluated independently for each segment and

reused for different final curves. We chose the criterion to be *integral of absolute change in curvature* that we denote K:

$$K = \int |\kappa(s)'| ds, \kappa(s) = \left\| T'(s) \right\|, \tag{3}$$

where s is the arc length parameter of the curve and $\kappa(s)$ is the curvature defined by using the unit tangent vector T.

The minimization of this criterion favors curves with monotonic curvature. Whenever the underlying geometry causes the curve to turn, this criterion prefers the curve to keep turning, which often results in the curve wrapping around the object (Figure 10).

We estimate the discrete curvature for each *on* segment $\bar{g}^a_{ij}$ from its vertices $\left\langle v^a_i, v^a_{i+1}, \ldots, v^a_j \right\rangle$. The curvature of the *off* segments is calculated from its vertices but also from the last and first vertices of the connecting *on* segments (inset Figure in Section 5.2). This makes sure that any sharp connection between *on* and *off* segments is penalized.

The curvature is estimated differently for input strokes that form a loop. If the first p_0 and the last point $p_{|P|}$ of the input 2D stroke are within a small distance, we merge the first and last segments and the curvature is estimated for this merged segment.

The previous step assigned the weights to nodes and edges of the graph. In this step we find the path through the graph that will represent the final curve. Such a path has to start with a segment that includes the vertex for the first ray, i.e., any segment $\bar{g}^a_{0x}$, and end with a segment that includes the vertex for the last ray, i.e., any segment $\bar{g}^b_{y|P|}$. In the case of beginning or ending the stroke *off* the geometry, recall that these segments can be zero length ($\bar{g}^0_{00}$ or $\bar{g}^0_{|P||P|}$). The graph is implicitly topologically sorted, therefore we simply do a depth first traversal from the nodes starting at first ray and perform edge relaxation, noting the best predecessors at each node. To construct the curve we simply retrace the best predecessors from all nodes ending at $|P|$ and concatenate the segments. In our implementation we use Catmull-Rom splines to construct the final curve geometry.

6.2 Curve Smoothing

The previous steps generate a 3D curve that follows the geometry but may have some sharp turns. To improve its quality it is resampled and iteratively smoothed.

Recall that the 2D points are equidistant in 2D (Section 4.1). However, when projected to 3D, the distance between successive vertices of the 3D curve is not constant so we further resample the curve in 3D so that the distance between vertices is constant.

We use the active contours approach [Kass et al. 1988] to smooth the 3D curve with two additional constraints. First, similar to [Kara and Shimada 2007], we make sure that the final curve's projection to the sketching viewpoint is similar to the sketched 2D curve. Second, we preserve curve height h that was defined in Section 4.2. To smooth the curve we minimize the energy of the curve E by using the gradient descent:

$$E = \int_0^1 \left(E_{internal}(s) + E_{external}(s)\right) ds. \tag{4}$$

The internal energy is the sum of continuity and smoothness energies:

$$E_{internal}(s) = \alpha \left\| \frac{dv(s)}{ds} \right\|^2 + \beta \left\| \frac{d^2 v(s)}{ds^2} \right\|^2, \tag{5}$$

where $v(s)$ is the position of the curve at arc length parameter $s \in (0, 1)$. The external energy is defined as:

$$E_{external}(s) = \gamma |r(v(s)) \cdot \Gamma(s)| + \delta |d(v(s)) - h(s)|, \tag{6}$$

where $r(v(s))$ is the direction of the ray from the camera to $v(s)$, $\Gamma(s)$ is the direction of the ray from the initial sketch at s, $d(v(s))$ is the distance of $v(s)$ the geometry, and $h(s)$ is the height of the curve at s. The α, β, γ, δ are respective weights of individual terms and $\alpha + \beta + \gamma + \delta = 1$. We use $\alpha = 0.0039$, $\beta = 0.011$, $\gamma = 0.982$, and $\delta = 0.0019$ in our implementation, which prefers smoothness over equidistance of vertices and penalizes even a small deviation from the input sketch.

6.3 Curve and Scene Editing

Skippy offers by default a smooth curve that keeps the distance from the surface as defined by the user strokes and follows the surface geometry (please see the accompanying video). However, this may not always be the user's preferred choice and we can easily provide alternative curves that are stored in the segment graph $\mathcal{G}$.

Stroke modifiers allow the user to change the selected curve while drawing. Using a bidirectional gesture such as a mouse wheel, the user can change the depth of the last inferred segment. The depth is only a suggestion, as a further stroke points can change the configuration of the curve. Furthermore, a modal gesture, like a key being pressed, can disable intersections. This is particularly useful in dense scenes, such as in Figure 14, since otherwise the system automatically assumes that any geometry intersecting the stroke will have effect on the final curve.

After the stroke has been finished, the user can **modify the curve** by redrawing its part or changing the depth of certain segments. The redrawing can be done from any viewpoint and is performed by inferring a new curve from a redraw stroke. This new curve has its end vertices fixed to match the closest vertices of the original curve and is used to replace the part of the original curve that is being redrawn. Furthermore, the depth of individual parts of the curve (the *on* segments) can be selected manually, again by using a bidirectional gesture such as the mouse wheel as shown in example in Figure 11 and in the accompanying video.

Fig. 11. Once the segment graph has been calculated the curve can be modified by simply clicking on the 2D stroke or the geometry covering the stroke to change its depth.

We found that **transient geometry** is a powerful way to model the curves. Transient objects work as scaffolds for the 3D curves. We allow the user to add a transient object, defined as a mesh,

by clicking on a surface of an existing one. The object is placed such that its vertical axis is perpendicular to the surface and the mouse wheel controls its scale. If there is no object in the scene, the transient object is placed to the origin. Once the transient object is not needed it can be removed by shift click. Similarly, any new 3D geometry created around the space curves generated by Skippy can act as a transient geometry and can be removed at any stage. In Figure 18, several transient objects were used to produce snakes that stay farther from the head. The process in Figure 13 is similar, except that the initial transient sphere was placed at the origin.

7 IMPLEMENTATION AND RESULTS

Our system was **implemented** in C++ with OpenGL and glm and we tested our results on an Intel-based desktop computer with Intel Xeon E5-1630 @ 3.7GHz, 12GB of DDR4 RAM and NVIDIA GeForce GTX TITAN X.

Timing of our application depends on several aspects. The most important is the number n of the resampled points p_i and the number of the triangles of the input geometry. The speed of the application also depends on the number of ray intersections, i.e., multiple occlusions. This generates multiple *on* segments that add to the number of combinations of the *off* segments (edges of the segment graph). The speed of the application also depends on various variables such as the distance field resolution (we use an octree of depth seven) and curve discretization (we use screen distance of 5-15 pixels). The distance field needs to be modified when temporal geometry is added or deleted. To speed up the calculation we store individual distance field for each object, thus the recalculation depends only on the added geometry.

We report the timing of the individual steps of our application for all results in Table 1. The timing is for the last step of the model creation, when all the 2D strokes are present in the scene and the scene is most complex. In order to get to the most complex case, the user goes through a sequence of simpler scenes as shown in example in Figure 12 that reports timing of the design process during a model creation for two different objects. It can be seen that the number of multiple rays (holes) greatly affects the calculation time.

The *authoring* column of Table 1 reports the time necessary for the design of each model, when the user may undo some actions, erase wrong parts etc. Overall, the object creation was in order in seconds.

Smoothing is performed both while the user is drawing and after the stroke is finished. Only a few iterations (32) is performed while drawing. Once the stroke is finished, we smooth the curve using 32 iterations per frame until we reach no substantial change in position of the vertices or we reach a maximum limit (we typically use 512).

An example in Figure 13 and in the accompanying video shows usage of the **transient geometry**. A sphere is used as an object that carries the initial curve that defines the overall shape of the final object Figure 13 a). The sphere is deleted and the first curve becomes a part of the scene (Figure 13 b)).

Results. Figure 18 shows an example of Medusa that heavily uses transient objects (see also the video). The input scene is a model of the head without hair. The user starts by placing several transient objects that help to position the first snakes in 3D. The snakes become a part of the scene and the user draws further snakes around

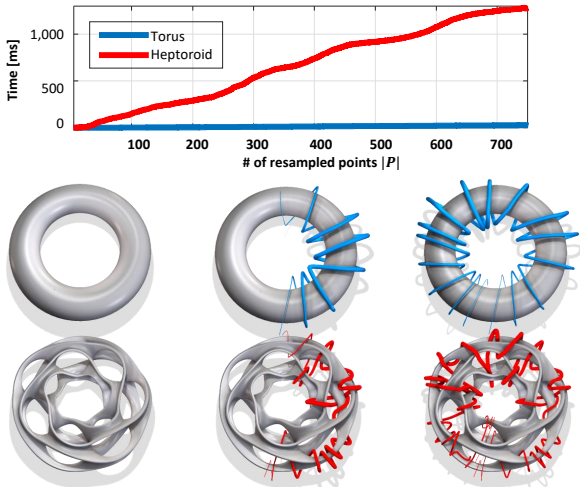


Fig. 12. The scene generation time increases with the complexity of the scene and the number of intersections for each point.

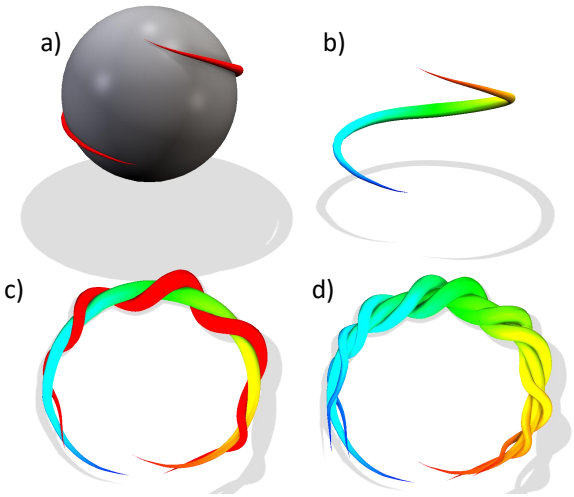


Fig. 13. An example of usage of transient geometry. Sphere is used as a shape-defining object and after a first sketch it is deleted a). The first curve becomes part of the scene and is used to wrap several additional curves around c)-d). The overall look of the resulting geometry is defined by the initial transient sphere.

them. This was a time demanding scene to complete and the overall authoring time was 16 minutes. The final scene has 36 curves with the total of 1, 167 vertices made from 1, 810 input points. The complete scene calculation from the 2D points was 2.1 seconds.

Figure 1 (and the accompanying video) shows an example of Kraken that has been generated by multiple strokes from a single view. It is interesting to observe that the individual strokes actually correspond to wrapping the tentacles of the Kraken around the ship. This example had 606 input points that generated 724 3D vertices. The time to generate the complete model from the 2D strokes was 98 ms and the overall authoring time was a little bit over one minute.

Figure 14 shows an example of a dense scene (tree) that is enhanced by cords with lights. This example includes multiple occlusions of tiny objects that is a difficult case for our framework, because it causes frequent depth skipping during the curve drawing. This input scene had 1,273 input points that generated 1,653 vertices and it took nearly four seconds to generate it. Authoring of this scene took about 19 minutes.

Fig. 14. A dense scene with tiny geometry is a difficult case because of frequent skipping during curve drawing.

Figure 15 shows an example of a path control in a scene of a city that could be used in an animation for camera planning. In order to provide flyover the center of the city the user created transient geometry and wrapped the curve around it. Authoring of this scene took 11 seconds and the curve generation was 16 ms.

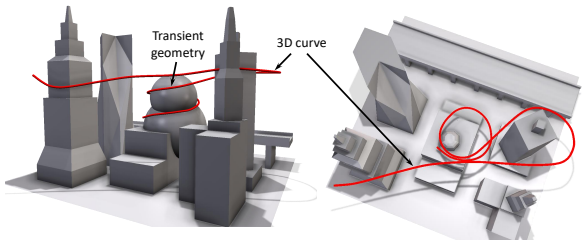


Fig. 15. Camera path through the city can be sketched very quickly with the help of transient geometry. The curve was sketched from the viewpoint on the left.

The cup with snake in Figure 16 shows generation of wrapping of a 3D object with a single stroke made up of 615 points. Authoring of this scene was around 1.5 minutes and the curve was generated in 75 ms.

User interaction. We engaged novice users (see Figure 16) as well as professional artists to refine and test the system. Although the users had the option to change the depth of the last segment while drawing, they noted that the automatic prediction helped them to draw more efficiently.

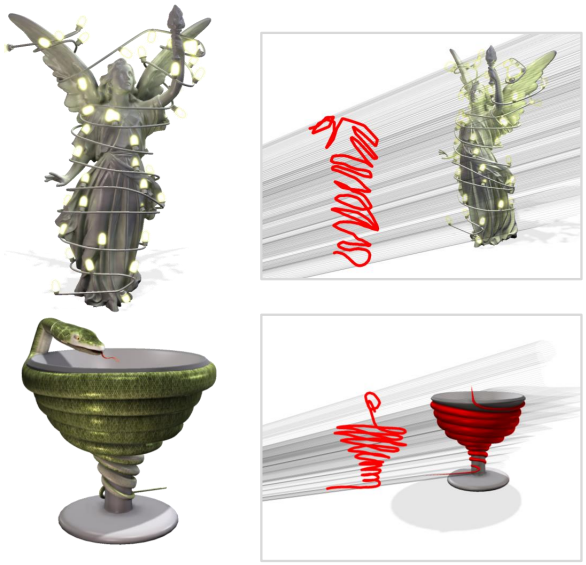


Fig. 16. Two examples of wrapping an object by a single stroke.

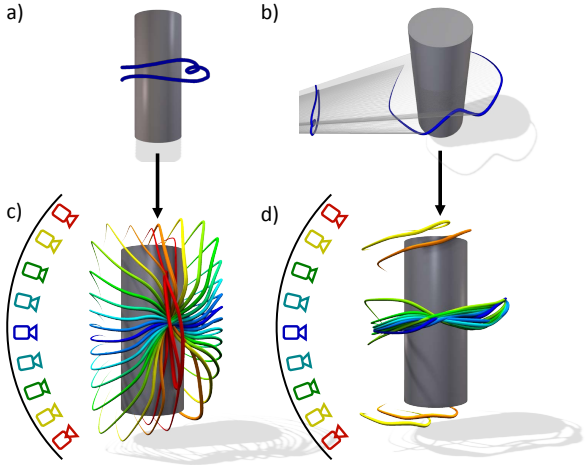


Fig. 17. Influence of the viewpoint on the final curve. Curve defined by a stroke (a) is inferred (b). By applying the same stroke from different viewpoints (c), the shape of the curve changes depending on the orientation. In (d) we generate the curve from (b) by using its projection to a different viewpoint, which gets progressively harder due to foreshortening and for more oblique views it is hard to obtain the same curve. Blue to red transition signifies the increasing absolute elevation angle of the camera.

The majority of the curves were created with a single stroke without any editing operations. Whenever a change was needed, the curve was usually removed and sketched again from scratch. In more complicated cases, such as Figures 1, 14 and 18, redrawing of parts of the curve and manual depth changes were utilized more often.

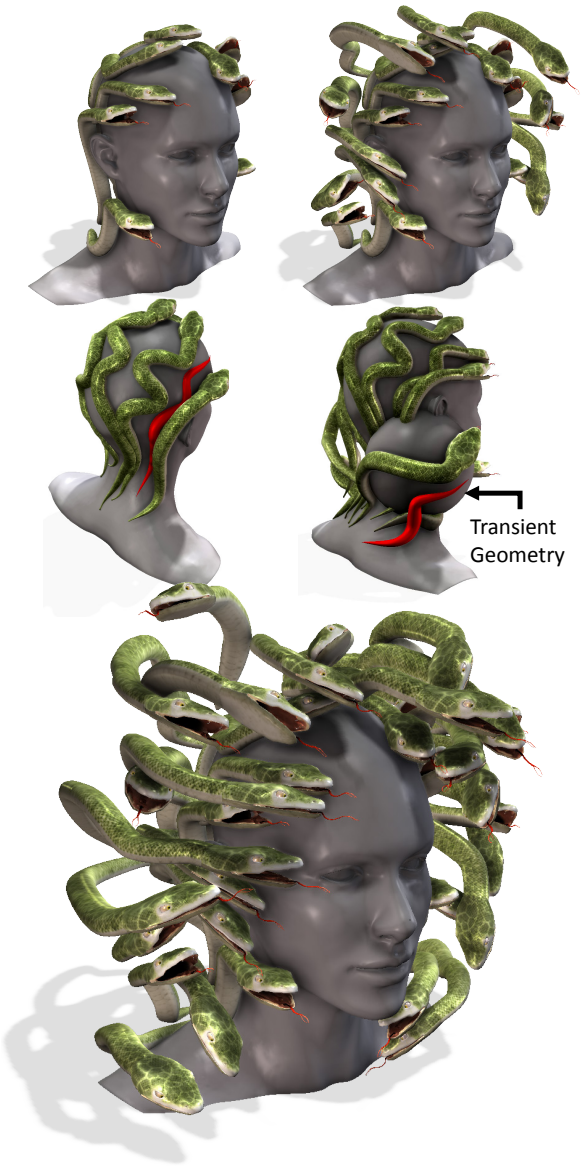


Fig. 18. Two frames from the creation of Medusa (top left and right) show usage of transient geometry that keeps the snakes away from the head. Some snakes eventually become part of the geometry and are used as supporting objects as well.

Even though every curve was sketched from a single viewpoint, a change of viewpoint was not entirely eliminated. The most common perspective changes were rotating the camera to inspect the resulting curve and choosing a perspective that minimized foreshortening, since the resulting curve may differ depending on chosen viewpoint as shown in Figure 17.

8 CONCLUSION

We presented Skippy, a novel algorithm for 3D curves generation from single view. The user draws a 2D stroke and the algorithm divides it into *on* and *off* sequences. The distance of the 3D curve from the object (its height) is estimated from the user stroke. The sequences are converted to *on* segments in 3D and so called segment graph encodes the *on* segments as its nodes and all possible connections of the *off* segments as segment graph edges. The optimal path is generated that follows the object geometry and has monotone curvature. The user can quickly edit the curve by selecting alternative options that are all encoded in the segment graph. Moreover, we include the concept of transient geometry that is used to scaffold the curve creation. We show Skippy on a number of examples ranging from simple wrapping of curves around objects to complex scenes with intertwined geometries.

Limitations. One of the limitations arising from input geometries in the form of triangle soups, is that arbitrarily small elements can be in place. Skippy will not be able to treat these small and noisy geometries before any post-processing. Intersection with such structures cause jumping between different options. Although we solve this partially by allowing a variable threshold of rays that can be ignored, it shifts the problem to higher frequencies or causes unwanted geometries to be ignored. Also, Skippy does not eliminate the need for changing the viewpoint. In case of a complex guiding object or scene the user may have to draw the curve in several parts and reposition the view between each part to reduce the foreshortening of the guiding objects. Still, the number of necessary viewpoint changes is small compared to traditional approaches.

There are several possible avenues for **future work**. One of them is to allow branching or more complex network topologies on the generated curves. Similarly to our redrawing approach, the curve end points could be restricted, for example to lie on an existing curve structure, or we could enforce orthogonality or parallelity to existing curves. Another area of future work could address the shape of the curve. We assume that a visually plausible curve is a smooth one, but it would be interesting to allow different options, such as corners and sharp features. One of the ways to achieve this would be to use Manhattan or Chebyshev distance instead of Euclidean as the basis of the distance field or explicitly detect sharp features and modify the curve accordingly. We have experimented with the way the curve is controlled by the object surface. So far we only consider the distance of the stroke that defines the height of the curve. This concept could generalize to considering different properties such as salient features that could attract or repulse the curve, or the texture on the surface could further control the curve shape. Finally, we believe our method has strong potential application in both Virtual and Augmented Reality (VR, AR) modeling systems, allowing artists to rapidly decorate and populate large or distant spaces (either virtual or real) that might not otherwise be easily accessible. In other words, Skippy may allow artists to draw content that extends beyond their physical reach; such a system is particularly relevant to AR where the world cannot be scaled down. For this reason we find it exciting to explore the use of Skippy combined with 6-DOF tracking for design tasks in immersive and virtual environments.

ACKNOWLEDGEMENTS

This work has been sponsored by Adobe Research and by the National Science Foundation grant #1606396 *Haptic-Based Learning Experiences as Cognitive Mediators for Conceptual Understanding and*

Table 1. Time Performance. Mesh triangles shown are for the final scene including meshes generated by curves. The authoring time refers to the time spent by the user, the total time is the time the system spent on generating the curve.

	Input		Time					Output	
Model	**Mesh Triangles**	**Input Points**	**Vertex Est. [ms]**	**Segment Graph Constr. [ms]**	**Curve Const. [ms]**	**Total [ms]**	**Authoring [s]**	**# of Curves**	**# of 3D Vertices**
Kraken (Fig 1)	72,236	606	71	9	18	98	84	8	724
Curve art (Fig 13)	41,328	348	15	9	10	34	41	3	290
Medusa (Fig 18)	477,280	1,810	1,921	176	11	2108	984	36	1167
Tree (Fig 14)	81,132	3689	319	53	57	429	1140	35	851
City (Fig 15)	3,588	192	12	4	2	16	11	1	67
Lucy (Fig 16 top)	139,940	665	118	305	715	1138	210	1	417
Snake cup (Fig 16 bottom)	10,240	615	49	9	15	73	95	1	715

Representational Competence in Engineering Education. We would like to thank Darius Bigbee and Suren Deepak Rajasekaran for help with the modeling of various examples. Furthermore, we would like to thank to Alexander Spivak for providing the Medieval Ship model, kaneflame3d for providing the Female Head model, Standford 3D Scanning Repository for providing the Armadillo and Lucy models and Carlo H. Séquin for providing the Heptoroid model.

REFERENCES

Seok-Hyung Bae, Ravin Balakrishnan, and Karan Singh. 2008. ILoveSketch: as-natural-as-possible sketching system for creating 3d curve models. In *Proc. of User interface software and technology*. ACM, 151–160.

David Ben-Haim, Gur Harary, and Ayellet Tal. 2010. Piecewise 3D Euler Spirals. In *Proceedings of the 14th ACM Symposium on Solid and Physical Modeling (SPM '10)*. ACM, New York, NY, USA, 201–206. DOI: https://doi.org/10.1145/1839778.1839810

Mikhail Bessmeltsev, Will Chang, Nicholas Vining, Alla Sheffer, and Karan Singh. 2015. Modeling Character Canvases from Cartoon Drawings. *ACM Trans. Graph.* 34, 5, Article 162 (Nov. 2015), 16 pages. DOI: https://doi.org/10.1145/2801134

Philip Buchanan, Ramakrishnan Mukundan, and Michael Doggett. 2013. Automatic single-view character model reconstruction. In *Proceedings of the International Symposium on Sketch-Based Interfaces and Modeling*. ACM, 5–14.

Xuejin Chen, Boris Neubert, Ying-Qing Xu, Oliver Deussen, and Sing Bing Kang. 2008. Sketch-based Tree Modeling Using Markov Random Field. *ACM Trans. Graph.* 27, 5, Article 109 (Dec. 2008), 9 pages. DOI: https://doi.org/10.1145/1409060.1409062

Jonathan M. Cohen, Lee Markosian, Robert C. Zeleznik, John F. Hughes, and Ronen Barzel. 1999. An Interface for Sketching 3D Curves. In *Proc. of I3D (I3D '99)*. ACM, New York, NY, USA, 17–21. DOI: https://doi.org/10.1145/300523.300655

Patrick Coleman and Karan Singh. 2006. Cords: Geometric Curve Primitives for Modeling Contact. *IEEE Comput. Graph. Appl.* 26, 3 (May 2006), 72–79. DOI: https://doi.org/10.1109/MCG.2006.54

Frederic Cordier and Hyewon Seo. 2007. Free-Form Sketching of Self-Occluding Objects. *IEEE Comput. Graph. Appl.* 27, 1 (Jan. 2007), 50–59. DOI: https://doi.org/10.1109/MCG.2007.8

Chris De Paoli and Karan Singh. 2015. SecondSkin: Sketch-based Construction of Layered 3D Models. *ACM Trans. Graph.* 34, 4, Article 126 (July 2015), 10 pages. DOI: https://doi.org/10.1145/2766948

Lubin Fan, Ruimin Wang, Linlin Xu, Jiansong Deng, and Ligang Liu. 2013. Modeling by drawing with shadow guidance. In *Comp. Graph. Forum*, Vol. 32. Wiley Online Library, 157–166.

Chi-Wing Fu, Jiazhi Xia, and Ying He. 2010. LayerPaint: A Multi-layer Interactive 3D Painting Interface. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 811–820. DOI: https://doi.org/10.1145/1753326.1753445

Hongbo Fu, Yichen Wei, Chiew-Lan Tai, and Long Quan. 2007. Sketching Hairstyles. In *Proceedings of the 4th Eurographics Workshop on Sketch-based Interfaces and Modeling (SBIM '07)*. ACM, New York, NY, USA, 31–36. DOI: https://doi.org/10.1145/1384429.1384439

Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. 1999. Teddy: A Sketching Interface for 3D Freeform Design. In *Proc. of SIGGRAPH (SIGGRAPH '99)*. 409–416. DOI: https://doi.org/10.1145/311535.311602

Takeo Igarashi and Jun Mitani. 2010. Apparent Layer Operations for the Manipulation of Deformable Objects. *ACM Trans. Graph.* 29, 4, Article 110 (July 2010), 7 pages. DOI: https://doi.org/10.1145/1778765.1778847

Levent Burak Kara and Kenji Shimada. 2007. Sketch-based 3D-shape creation for industrial styling design. *IEEE Comp. Graph. and Applications* 27, 1 (2007), 60–71.

Olga Karpenko, John F. Hughes, and Ramesh Raskar. 2004. Epipolar Methods for Multi-view Sketching. In *Proc. on SBIM (SBM'04)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 167–173. DOI: https://doi.org/10.2312/SBM/SBM04/167-173

Michael Kass, Andrew Witkin, and Demetri Terzopoulos. 1988. Snakes: Active contour models. *International Journal of Computer Vision* 1, 4 (1988), 321–331.

Ismail Khalid Kazmi, Lihua You, and Jian Jun Zhang. 2014. A Survey of Sketch Based Modeling Systems. In *Proc. CGIV*. 27–36. DOI: https://doi.org/10.1109/CGiV.2014.27

Manfred Lau, Greg Saul, Jun Mitani, and Takeo Igarashi. 2010. Modeling-in-context: user design of complementary objects with a single photo. In *Proc. of SBIM*. Eurographics Association, 17–24.

Raph Levien and Carlo H Séquin. 2009. Interpolating Splines: Which is the fairest of them all? *Computer-Aided Design and Applications* 6, 1 (2009), 91–102.

William J Matthews and Amy Adams. 2008. Another reason why adults find it hard to draw accurately. *Perception* 37, 4 (2008), 628–630.

James McCann and Nancy Pollard. 2009. Local Layering. *ACM Trans. Graph.* 28, 3, Article 84 (July 2009), 7 pages. DOI: https://doi.org/10.1145/1531326.1531390

James McCrae and Karan Singh. 2008. Sketching Piecewise Clothoid Curves. In *Proceedings of the Fifth Eurographics Conference on Sketch-Based Interfaces and Modeling (SBM'08)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 1–8. DOI: https://doi.org/10.2312/SBM/SBM08/001-008

James McCrae and Karan Singh. 2009. Sketch-based Path Design. In *Proceedings of Graphics Interface 2009 (GI '09)*. Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 95–102. http://dl.acm.org/citation.cfm?id=1555880.1555906

Andrew Nealen, Takeo Igarashi, Olga Sorkine, and Marc Alexa. 2007. FiberMesh: Designing Freeform Surfaces with 3D Curves. *ACM Trans. Graph.* 26, 3, Article 41 (2007). DOI: https://doi.org/10.1145/1276377.1276429

Gen Nishida, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Adrien Bousseau. 2016. Interactive Sketching of Urban Procedural Models. *ACM Trans. Graph.* 35, 4, Article 130 (2016), 11 pages. DOI: https://doi.org/10.1145/2897824.2925951

Luke Olsen, Faramarz F Samavati, Mario Costa Sousa, and Joaquim A Jorge. 2009. Sketch-based modeling: A survey. *Computers & Graphics* 33, 1 (2009), 85–103.

Johannes Schmid, Martin Sebastian Senn, Markus Gross, and Robert W. Sumner. 2011. OverCoat: An Implicit Canvas for 3D Painting. *ACM Trans. Graph.* 30, 4, Article 28 (July 2011), 10 pages. DOI: https://doi.org/10.1145/2010324.1964923

Ryan Schmidt, Azam Khan, Gord Kurtenbach, and Karan Singh. 2009a. On Expert Performance in 3D Curve-drawing Tasks. In *Proc. of SBIM (SBIM '09)*. ACM, New York, NY, USA, 133–140. DOI: https://doi.org/10.1145/1572741.1572765

Ryan Schmidt, Azam Khan, Karan Singh, and Gord Kurtenbach. 2009b. Analytic Drawing of 3D Scaffolds. *ACM Trans. Graph.* 28, 5, Article 149 (Dec. 2009), 10 pages. DOI: https://doi.org/10.1145/1618452.1618495

Laura M Taylor and Peter Mitchell. 1997. Judgments of apparent shape contaminated by knowledge of reality: Viewing circles obliquely. *British Journal of Psychology* 88, 4 (1997), 653–670.

Emmanuel Turquin, Jamie Wither, Laurence Boissieux, Marie-Paule Cani, and John F. Hughes. 2007. A Sketch-Based Interface for Clothing Virtual Characters. *IEEE Comput. Graph. Appl.* 27, 1 (Jan. 2007), 72–81. DOI: https://doi.org/10.1109/MCG.2007.1

Jamie Wither, Florence Bertails, and Marie-Paule Cani. 2007. Realistic Hair from a Sketch. In *Shape Modeling and Applications, 2007. SMI '07. IEEE International Conference on.* 33–42. DOI: https://doi.org/10.1109/SMI.2007.31

Baoxuan Xu, William Chang, Alla Sheffer, Adrien Bousseau, James McCrae, and Karan Singh. 2014. True2Form: 3D Curve Networks from 2D Sketches via Selective Regularization. *ACM Trans. Graph.* 33, 4, Article 131 (July 2014), 13 pages. DOI: https://doi.org/10.1145/2601097.2601128

Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes. 1996. SKETCH: An Interface for Sketching 3D Scenes. In *Proc. of SIGGRAPH (SIGGRAPH '96)*. ACM, 163–170. DOI: https://doi.org/10.1145/237170.237238