

Interactive 3D Sketching in Virtual Reality

Emilie Yu



Kongens Lyngby 2020

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

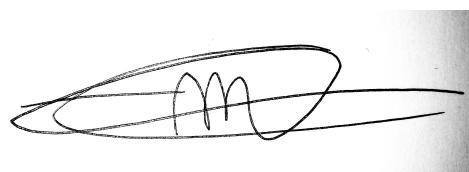
Abstract

While traditional design sketches provide a quick way for designers to express a 3D concept, sketches are inherently 2D, meaning that the artist must perform a projection of their 3D idea and that the viewer must decode it. The idea to express one-self using 3D input is not new, as sketching in immersive environments has been prototyped since more than 2 decades. However, users face issues intrinsic to 3D sketching, which hamper their creation process. One issue is the difficulty to perform precise sketching motions in 3D, necessary to sketch correctly. Another issue is with visualisation of a 3D sketch: while a 2D sketch has a fixed viewpoint, a 3D sketch must be made to look correct from any viewpoint. We set out to address these challenges with an approach based on automatic stroke beautification, and on inferring the surfaces that the user envisions while they sketch, in order to provide occlusion cues that help understand the sketch. Our prototype and the user study show that this method offers promising improvements over free-hand 3D sketching.

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Digital Media Engineering.

Lyngby, 03-August-2020



Emilie Yu

Acknowledgements

I would like to thank my advisor Andreas, for supporting this project from the very start, while always providing great advice. I would also like to thank my collaborators Adrien, Tibor, Rahul and Karan, for investing themselves in this project and always bringing fresh ideas, feedback and support. Last but not least, thanks to my colleagues at INRIA and more particularly thanks to David, for inspiring me and distracting me from lockdown insanity during this special semester.

Contents

Abstract	i
Preface	iii
Acknowledgements	v
1 Introduction	1
1.1 Motivation	1
1.1.1 Sketching: a traditional design tool	2
1.1.2 Using VR as an immersive creation interface	2
1.2 Goal	4
2 Related work	5
2.1 Immersive sketching	5
2.2 2D sketch beautification and sketch depth inference	7
2.3 Surfacing of 3D curve networks	9
3 Method	13
3.1 User workflow	14
3.2 Stroke beautification	15
3.2.1 Principle	15
3.2.2 Stroke pre-processing	16
3.2.3 Detecting geometric constraints	17
3.2.4 Enforcing constraints while preserving user input	20
3.3 Surface inference	27
3.3.1 Curve network from intersecting strokes	28
3.3.2 Automatic surface patch detection	29
3.3.3 User guided patch creation	35
3.3.4 Surface patch geometry	35

3.3.5 Drawing on surfaces	37
4 Results	41
4.1 User interface	41
4.1.1 Draw and delete	42
4.1.2 Navigation	42
4.1.3 Helper objects	43
4.2 Evaluation	46
4.2.1 Sketches and analysis	46
4.2.2 User study	48
5 Conclusion	57
A Beautifying line segments	61
A.1 Choosing which constraints to apply	61
A.2 Applying constraints	62
B Surface patches update mechanism	63
C User study video tutorial	65
D Controller cheatsheets	67
E Project plan	69
Bibliography	73

CHAPTER 1

Introduction

1.1 Motivation

A growing body of work on the subject of algorithmically understanding sketches provides us with data and understanding of how designers use the medium of 2D sketching to represent the 3D shape of an object [21, 56]. Meanwhile, the hardware technology to track headset and hand-held controllers precisely in 3D is mature and commercially available [15, 23]. This technology enables artists to draw in mid-air by simply moving their hand in space, creating 3D strokes that can be directly interpreted as such in an immersive environment, without the need for projection on a 2D screen. Bypassing that step of projection of a 3D concept to 2D, inherent to 2D sketching, could lead to interesting new ways to create. Externalizing 3D concepts directly in 3D could provide a more immediate creation experience, and enable designers to avoid one of the challenges of sketching: making sure the observers can infer the correct depth information from their sketch.

1.1.1 Sketching: a traditional design tool

Designers use sketching early on in the design process, as a way to both externalize their mental concept of a product and develop the ideas further, evaluating and precising them as they sketch [46].

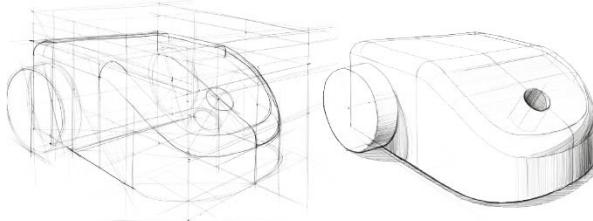


Figure 1.1: Designer sketches from OpenSketch dataset [21]

By looking at examples of designer sketches [13, 21], and the definition of a sketch as described by Buxton in the context of user experience design [8], we can extract a few core properties that designers seem to value in an exploratory sketch:

- Sketching is *quick* and *inexpensive*.
- A sketch should reveal a *minimal* level of detail, and *suggest* solutions rather than confirm one.

While both pen and paper and digital tools are now used commonly for this creation process, these input modalities are inherently 2D. We seek to propose a 3D sketching interface, for which the above desirable properties would be satisfied while "3D sketching".

1.1.2 Using VR as an immersive creation interface

As virtual reality gains popularity, this technology is being used by artists as a new creative medium that allows them to draw or model while immersed in their creation, in a virtual 3D space. In recent years, commercial tools have been released to enable artists to sketch free-hand in mid-air [17] and to create 3D design concepts by sculpting digital clay [14] or through an interface similar to desktop CAD modelling [18, 20] (Fig. 1.2).



Figure 1.2: Commercial tools for VR creation. From left to right: TiltBrush [17], Oculus Medium [14], Gravity Sketch [20]

Despite the success of these tools among a nascent community of artists, experimental studies have shown that users face new challenges while sketching in VR, as they are unable to draw precisely in all arbitrary scales and orientations that a VR environment could afford. Difficulty to accurately judge distances in the depth dimension, motor limitations and lack of a supporting surface are all factors contributing to less precise strokes [2, 34].

While TiltBrush [17] has its main interaction as free-hand sketching, experienced artists manage to create beautiful pieces by painting dense accumulation of strokes (see Fig. 1.3), where lack of precision matters less than in the tasks devised in previously mentioned usability studies. Such densely painted 3D sketches also provide occlusion cues that help understand the objects by looking at solid surfaces rather than at visually overlapping sparse contour and feature strokes. However, the 3D paintings created in TiltBrush - because of their high stroke density - put more emphasis on rendering the detailed look or texture of a surface, rather than the overall 3D shape, as a design sketch should do. Design sketches are composed of a sparse set of strokes that infer the surface (see Fig. 1.1).



Figure 1.3: TiltBrush artworks. Left: by Estella Tse, right: by Tristan Eaton.

Modeling tools such as Gravity Sketch and Google Blocks can be used to create efficient 3D models, but they use interaction metaphors close to desktop-based 3D modelling tools such as Blender, where the underlying mathematical representation of the geometry such as mesh vertices and curve control points are exposed. While very efficient in the hands of a trained user, they lack the immediacy and accessibility of sketching.

1.2 Goal

In this context, we set out to propose novel interactive solutions to use VR sketching as a 3D concept design tool. We explore ways to help artists realize their 3D visions by sketching a sparse set of strokes mid-air. We choose mid-air sketching as the main interaction metaphor for creation, then augment it with automatic stroke beautification and surface inference such that users can focus on shape exploration through sketching rather than on fighting the aforementioned difficulties of VR sketching.

Interactive stroke beautification automatically corrects user imprecision, while respecting the intent of their input. This allows them to focus on the main shape of their strokes, rather than on trying to precisely reach target positions in 3D space, which is known to be prone to errors and detrimental to stroke quality [2]. With stroke beautification, intended stroke intersections are enforced, so it becomes easy to draw a well-connected network of strokes. Assuming that the user means to represent a solid object with a sparse set of strokes, we can use the curve network obtained by beautification to infer the surface that the strokes bound. These surfaces provide adequate occlusion cues, which helps the user understand their sketch from any viewpoint.

We implement a VR application capable of interactive beautification of 3D strokes and surface inference from those strokes. We then conduct a user study to find out whether such a sketching interface can enable artists to explore 3D design concepts better than with free-hand 3D sketching. Finally, we produce a variety of 3D sketches with our tool, to demonstrate its versatility.

CHAPTER 2

Related work

The work we present in this thesis is related to previous work on immersive sketching, as we use similar interaction metaphors and face the same challenges. It is also related to work on inferring artist intent from 2D sketch, from which we take inspiration to implement our 3D stroke beautification method. Finally, an important part of this thesis is about inferring surfaces from 3D curves, therefore we will give an overview of some work on surfacing of 3D curve networks.

2.1 Immersive sketching

Since early immersive environments such as the *Responsive Workbench* [31] the *CAVE* [9] and head-mounted-displays took off - if not commercially, at least in many research groups - researchers have investigated the possibility of using these new types of display and input as an interface for artistic creation. In 1995, *HoloSketch* presented a novel "WYSIWYG" interface to create 3D objects, by using direct 3D position input from a tracked hand-held device to create primitive objects or free-form tubes that follow the hand motion of the user. *Surface Drawing* enabled users to create and deform surfaces in mid-air with a sweep of the hand (Fig. 2.1 left) [42]. In *FreeDrawer*, the 3D model was created on the *Responsive Workbench* by sketching a sparse network of feature curves, from which the system inferred surfaces that could later be deformed

(Fig. 2.1 right) [55]. While this body of work concerns interfaces that no longer exist, their ideas are still very relevant to modern immersive sketching applications. TiltBrush and many other creative applications employ a mid-air sketching interaction similar to the ideas presented in *HoloSketch* and *Surface Drawing*. Our work follows the same philosophy as *FreeDrawer*, where the 3D form is defined by a sparse set of curves. While they only snap nearby curves to form a network, we support more beautification constraints. We also propose automatically detected surface patches, while they rely on the user to indicate all patches that should be surfaced.

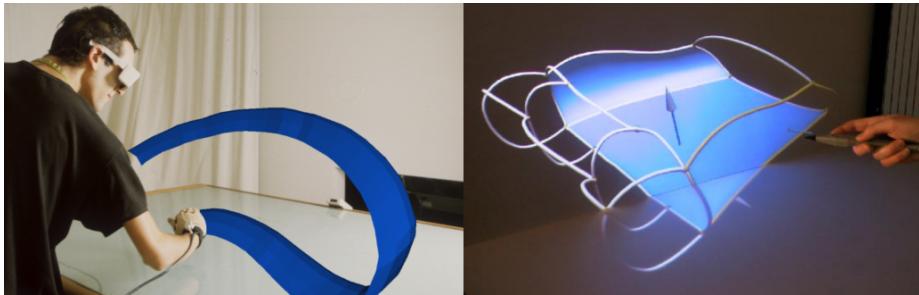


Figure 2.1: Left: *Surface Drawing* [42], right: *FreeDrawer* [55].

Supporting our hypothesis that VR sketching could be of interest for designers, Israel et al. [26] reported that designers have a strong interest in using an immersive sketching interface. Moreover Israel et al. [26] claim that despite early technical limitations, 3D sketching in a *CAVE* presents promising qualities to help them externalize ideas and foster creativity.

However, new challenges arise in VR sketching interfaces, compared to traditional sketching. When tasking users with sketching simple strokes like a straight line, a circle or perpendicular lines, Arora et al. [2] and Machuca et al. [34] have measured that the accuracy of the result compared to the target is quite poor. The lack of a supporting surface induces the need for better motor control in a space with more degrees of freedom than traditional 2D sketching. Adding to that, low spatial ability can inhibit users, making them more prone to errors when positioning strokes relative to each other.

To overcome these challenges in immersive sketching, a variety of interfaces have been proposed. One idea is to reduce the degrees of freedom when drawing a stroke mid-air. *3-Draw* [41] did so by decoupling the act of drawing the curve to define its shape from indicating its position and orientation in the overall drawing. Keefe et al. [29] use haptic feedback from a Phantom device and a 2-hand interaction metaphor inspired by tape drawing [6] to separate drawing the curve from indicating its tangent direction (Fig. 2.2a). Other methods avoid

direct 3D freehand sketching in the creation process. Jackson and Keefe [27] proposed to use curves from a 2D sketch as a basis for VR creation, while Arora et al. [3] use a 2D tablet on which the artist can sketch precise strokes that were mapped to a proxy 3D surface defined by a few freehand 3D strokes (Fig. 2.2b). In a similar spirit, Kim et al. [30] capture hand motion to describe 3D scaffolding surfaces on which to draw. Contrary to these methods, we tackle head-on the challenge of imprecision in free-hand sketching. We keep free-hand sketching as the main interaction metaphor for our system, and rely purely on an algorithmic process to disambiguate imprecise strokes.

Finally, some work has been done on beautification of freehand 3D strokes, which is directly related to our method. Machuca et al. [33] beautify the user’s strokes and suggest snapping targets by automatically detecting potential geometric relationships between new and existing strokes (Fig. 2.2c). In contrast to our approach, they only support planar strokes and they limit snapping to the endpoints of strokes. On a high level, Machuca et al. [33] focused on enabling the user to sketch strokes with precise geometric relationships (such as perpendicular lines or offset circles). In contrast, the goal for our beautification method is to create connected free-form curve networks.

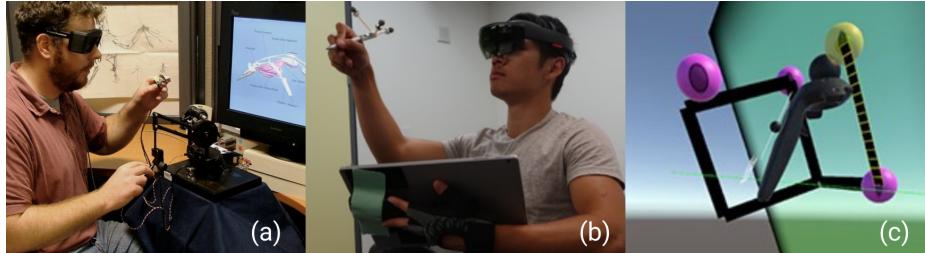


Figure 2.2: (a) *Drawing On Air* [29]. (b) *SymbiosisSketch* [3]. (c) *Multiplanes* [33].

2.2 2D sketch beautification and sketch depth inference

A number of methods have been developed to perform what is commonly called "sketch beautification". Their usefulness comes from the common need to create diagrams or drawings that satisfy specific geometric constraints. Without sketch beautification, it is difficult to do so by hand, especially with inputs such as a computer mouse or trackpad.

Pavlidis and Van Wyk [39] introduced one of the first such systems, that can take a line drawing as an input and output a drawing close to the input while satisfying geometric constraints. However, they focused on simple drawings composed of a small number of line segments. While this system was applied as a post-process to the drawing once it was finished, Igarashi et al. [25] proposed an interactive beautification method, that treats each new stroke as it is sketched, while keeping the rest of the sketch as is. This approach enabled them to consider more complex sketches and geometric relationships between strokes. The interactivity of the system also gives more control to the user by letting them choose between multiple possible beautified results. More recently, Fišer et al. [16] proposed an interactive beautification method that supports general Bézier curves as input, and an exhaustive number of geometric constraints, enabling users to create very complex beautified drawings (Fig. 2.3a). While all of these methods work with 2D sketches, we took inspiration from their approaches to design our interactive 3D beautification method. While we support a smaller set of geometric constraints, we also work on Bézier curve as in *ShipShape* [16] and we draw inspiration from the way they select a set of geometric constraints to apply to a stroke.

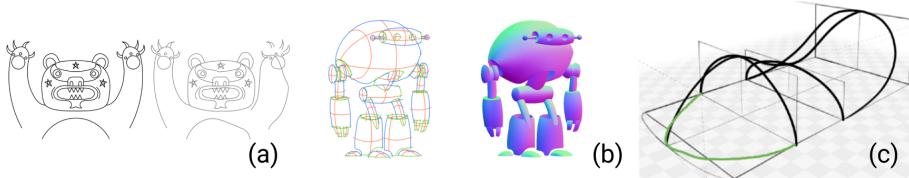


Figure 2.3: (a) *ShipShape* [16]. (b) *CrossShade* [47]. (c) *Analytic Drawing of 3D Scaffolds* [44].

Another body of work focuses on inferring depth from 2D sketches, in order to enable artists to create 3D curve networks, 3D objects, or a 3D appearance from a 2D sketch [10, 43, 44, 47, 56]. While these methods take input that is a lot more ambiguous than in our 3D sketching case, their core idea of lifting ambiguities in the input while preserving the artist’s intent is identical to our goal. One approach to lift the depth ambiguity from a sketch is to leverage domain specific knowledge on how artists draw. Shao et al. [47] observed that concept sketches often contain cross-section curves, which help the viewers to perceive the curvature of a drawn object. By expressing these perceptual cues as geometric constraints, they derived a method to infer the surface normals, and proposed an automatic shading technique for concept sketches (Fig. 2.3b). Similarly, Xu et al. [56] leveraged global perceptual cues from concept sketches in order to infer depth for each sketched curve, effectively creating a 3D curve network from a 2D sketch. While these algorithms are applied to a finished sketch, other methods allow the users to create 3D curves interactively on a

2D interface such as a pen tablet. Taking inspiration from traditional drawing practices, Schmidt et al. [44] proposed an interactive method to infer depth of 3D curves sketched in 2D by using construction lines of the drawing as cues (Fig. 2.3c). In a different application domain, De Paoli and Singh [10] use a 3D model as a base to draw on, and proposed a method for inferring depth by recognizing the relations between the sketched 2D strokes and the model. Our interface adopts a similar workflow, as we interactively lift ambiguities in an input stroke by using information from the previously drawn strokes.

2.3 Surfacing of 3D curve networks

Following the rising availability of 3D curve network data, coming either from 2D sketches and the methods mentioned above (Sec. 2.2), from direct capture with mobile devices [49], or from sketching interfaces [4], methods to generate a closed surface that extrapolate those curves have been developed. In an artistic context, the interest of doing so is mainly for visualisation, as the surfaces provide a clearer representation of the shape, with occlusion cues. A curve network created by a designer is a compact yet representative description of a 3D shape [19], that a human observer can easily interpret (see Fig. 2.4 left for examples of curve networks). While this is natural to do for humans, there are actually two distinct steps in this process. First, we must recognize which cycles of the curve network bound solid surface patches, as opposed to the cycles that bound cross-sections or boundary of the model for example. Secondly, we have to infer the curvature of each surface patch, based on the sparse curve network. In the following paragraphs, we summarize previous work on these two aspects. We draw inspiration from these methods to implement an interactive surfacing of curves (Sec. 3.3).

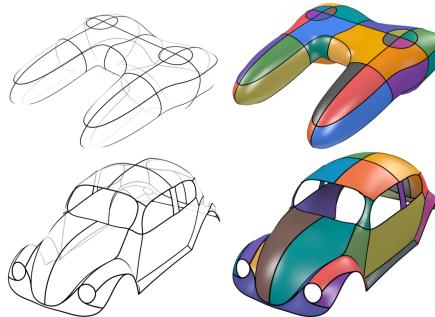


Figure 2.4: Curve networks and the envisioned surfaces, reconstructed with the method by Pan et al. [37].

To recognize cycles that bound surface patches algorithmically is not an easy task. The search space of all possible cycles in the graph of a typical 3D curve network from a sketch is big, and the validity or quality of one cycle can not be determined locally, but rather in the context of the other cycles. Abbasinejad et al. [1] proposed to construct an optimal cycle basis with respect to weighing of cycles defined by simple heuristics. Their algorithm favors cycles that are short (in number of edges), close to being flat and that do not separate the graph. Combined with a user interface to add a closing patch (the cycle basis does not close off the surface) and eventually correct erroneous patches, they achieved good results on the *ILoveSketch* dataset [4]. To support shapes of higher genus and non manifold surfaces, Zhuang et al. [57] introduced a more general approach by reformulating the search of an optimal set of cycles as a search for the optimal mappings at each vertex and curve that describe how incident curves at a vertex form cycles. The cost metrics they use favor patches that are smooth, convex and if multiple patches share a curve, they should be uniformly distributed around the curve. By searching for these optimal local mappings with a dynamic programming algorithm, combined with an effective pruning strategy, they can find a solution for a variety of complex 3D curve models in under a second. The approach we have on this problem is different from these methods, as the curve network that we treat is constructed interactively. We leverage that fact by detecting cycles on the network locally, where a new curve is added. In the context of the interactive VR application, it is also possible for the user to indicate any missing surface patches, so we rely on that to disambiguate some cases (see Section 3.3.2).

Once the cycles bounding patches are found, one must construct a geometric representation of the surface patch that matches human expectation (see Fig. 2.4 right). This geometric representation is typically a mesh, for which one must define both topology (connectivity) and geometry (vertex positions). Zou et al. [58] proposed an efficient algorithm to triangulate multiple closed 3D curves, by applying a dynamic programming algorithm to select the optimal triangles with respect to a given metric (such as total triangle area or total dihedral angles) among a reduced set of possible triangles obtained from the Delaunay tetrahedralization of the input curves. This method can provide an initial triangle mesh for each surface patch, but its geometry may not be aligned with what an observer would envision for the curve network. With additional normal information at the curves, Stanko et al. [48] guide traditional variational methods by providing curvature information through propagated normals, to obtain more realistic shape curvature. Based on the observation that artist-created 3D curve networks are largely composed of representative flow-line curves that are mostly aligned with principal curvature directions, Bessmeltsev et al. [7] surface a curve network with a quad-mesh by finding pairings between curve segments that bound regions of homogeneous flow-lines, then use the pairings to define the surface in-between as a smooth interpolation between those curve segments.

Driven by similar perception-based heuristics, Pan et al. [37] proposed a method to generate a surface with curvature lines well aligned with the flow-line curves. They improve curvature by iteratively refining the mesh and an associated cross field. The goal is to have the cross field directions smoothly interpolate the directions of the flow-lines, and the principal curvature directions of the mesh match the cross field. We choose to apply directly the efficient triangulation algorithm by Zou et al. [58] to obtain a triangle mesh from a boundary curve (see Section 3.3.4 for more details). While applying the fairing methods by Stanko et al. [48] or Pan et al. [37] would have significantly improved our surface geometry, we leave this for future work. The surfaces we obtain are mainly used for occlusion purposes, so their geometry is not an essential aspect.

CHAPTER 3

Method

In this chapter we present the components of our interactive stroke beautification and surfacing VR application. First we showcase the high-level workflow to create a 3D sketch in our application (Section 3.1). Then we give an overview of the stroke beautification process (Fig. 3.1b). It takes as input a stroke drawn by the user mid-air, and should output a stroke that is close to the input while satisfying as many regularity constraints as possible, with respect to the existing strokes in the sketch. Finally, we explain how we can use the strokes to infer the envisioned surface (Fig. 3.1c) and further use this surface to constrain some strokes to lie on it.

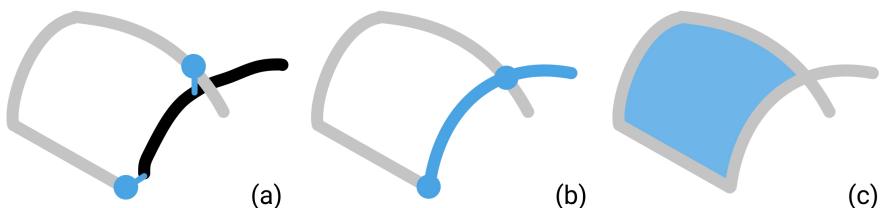


Figure 3.1: Method overview. (a) Existing sketch (light grey), newly sketched stroke (black) and constraints (blue). (b) Beautified stroke. (c) Inferred surface.

3.1 User workflow

Let's illustrate the workflow with the example of drawing a computer mouse (Figure 3.2).

- (a, b) The user creates the base of the mouse by sketching multiple short strokes that are automatically linked into one long continuous stroke. The strokes are also neatened to be planar.
- (c) When the user closes off the loop, a surface patch appears.
- (d) The user adds a section stroke on the mouse, which is neatened to intersect the base. The surface is updated to take the section shape into account.
- (e) After adding a few more strokes, the rough shape of the mouse is defined.
- (f) The user refines the surface by adding more strokes.
- (g) Finally, the user adds some detail, the strokes are projected to lie on the surface.
- (h) Result.

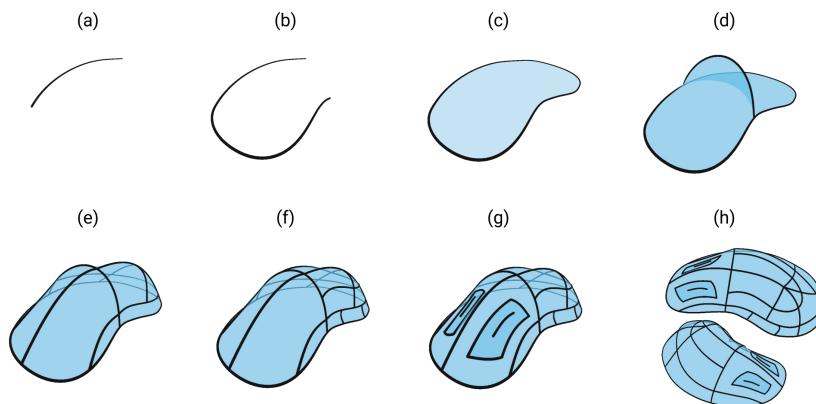


Figure 3.2: User workflow of a computer mouse sketch.

3.2 Stroke beautification

3.2.1 Principle

The need for stroke beautification in 3D sketching arises from the broadly observed lack of precision that users face with this medium [2, 33, 34]. In the context of 3D design sketches, we can formulate the hypothesis that some stroke properties are highly desirable for the user (see Section 3.2.3). For example, in Figure 3.3a, it seems likely that the strokes forming sections of the tubular shape should intersect the strokes from the sides, like in Figure 3.3b.

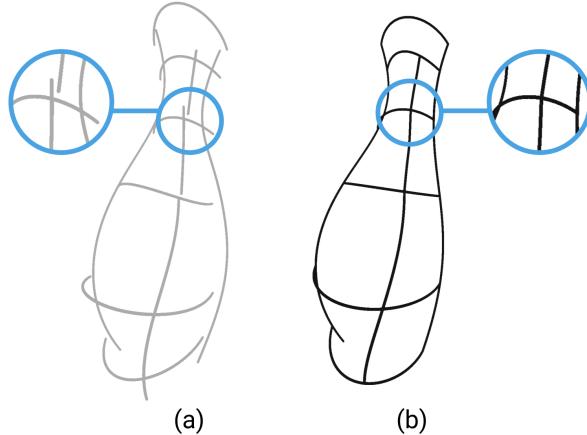


Figure 3.3: (a) Freehand sketch, (b) Beautified sketch.

Each time the user draws a new stroke, we first pre-process the stroke by fitting a smooth line or curve to the input samples (Sec. 3.2.2). Then we beautify it and replace the raw input stroke (Fig. 3.3a) by a beautified stroke (Fig. 3.3b). The process can be broken down into two main steps. First we must infer the user intent to determine which constraints should be applied to the stroke (Sec. 3.2.3). We look for potential constraints in the sketching environment, composed both of an orthogonal 3D grid and the previously drawn strokes. Secondly, we minimally reshape the input stroke to satisfy the constraints (Sec. 3.2.4). We devise a general method based on optimisation to apply geometric constraints on sketched strokes, while ensuring that we preserve the user input as much as possible.

3.2.2 Stroke pre-processing

Before beautifying the stroke, we pre-process it to remove unintended noise, then fit a smooth curve to the input samples. We choose to use lines and cubic Poly-Bézier curves as parametric representations of the strokes. Poly-Bézier curves - compared to other representations of curves such as Catmull-Rom curves - comport a smaller number of degrees of freedom, with a relatively small number of control points. They also strike a good balance between expressivity and smoothness, so they are well-suited for design sketches [56].

Hooks removal We notice that a common unintended pattern in VR sketching is the presence of "hooks" at the beginning and end of a stroke (Fig. 3.4). This is caused by a slight involuntary hand motion when pressing and releasing the trigger to draw.

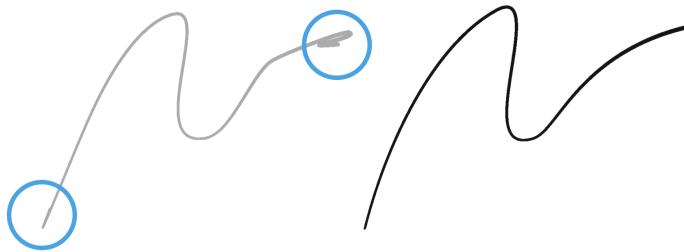


Figure 3.4: Left: stroke with hooks. Right: hooks removed.

We use the same method as Liu et al. [32], and cut the hooks by detecting C^1 discontinuities near the start and end of the stroke.

Line fitting We detect whether an input stroke is close to being a line, and in that case replace the stroke by a line. We look at the following criteria to decide:

- The total length of the input stroke L_c is close to the distance between the stroke endpoints L_l , as in *ShipShape* [16].
- The average drawing speed of the stroke is above a threshold.

Having a condition on the drawing speed enables users to deliberately draw quasi-linear strokes without them snapping to lines by drawing slowly, to indicate their intent to be more precise.

Bezier curve fitting In the general case, we fit a poly-Bézier curve to the input stroke. We first simplify the sampling of the input stroke by applying *Ramer-Douglas-Peucker* algorithm [12] to the initial list of samples gathered during the time the trigger was pressed. Then we fit a poly-Bézier curve with G^1 continuity using a least-square fitting approach as described by Schneider [45]. We obtain a succession of cubic Bézier curves, that minimally smooth the input stroke. We parameterize the complete poly-Bézier curve as: $B(t)_{t \in [0,1]}$.

3.2.3 Detecting geometric constraints

The beautification approach relies on first detecting which geometric constraints are applicable to the input stroke [25]. We search for constraints by looking at how the new stroke relates to previous strokes in the sketch. We also use an orthogonal 3D grid that spans the drawing volume to constrain the strokes, just as one would look for orthogonal alignments, or use a grid as support in 2D sketches [16, 25, 44]. In this section, we first explain how we define distance and angular thresholds for detection, then we list the 3 types of constraints that we detect. Finally, we explain the difference between the constraints that we want to enforce exactly, and those that we can settle for approximating.

3.2.3.1 Distance and angular thresholds

To detect constraints, but also to later apply them on the strokes, we need to define some threshold values that we rely on to decide whether a distance or an angle is small.

We decide to define a single distance threshold for the whole application: δ , which is the diameter of the sphere corresponding to the brush tip. Initially, when the drawing is of scale $s = 1$, $\delta^1 = 0.02$ in the units of the *Unity* scene.

Having a single distance threshold enables us to smoothly handle drawing volume scaling. As the user zooms in, we update this application wide threshold

such that for a scale $s \geq 1$, δ is appropriately scaled:

$$\delta = \frac{\delta^1}{s} \quad (3.1)$$

This enables the user to gain more precise control over their strokes by zooming in the drawing volume.

For the angular threshold, we define $\theta = \frac{\pi}{6}$, which empirically seems to strike a good balance between adequate snapping and expressivity.

Defining those threshold values to be appropriate for different users and sketches is one of the main challenges we face in this project. While making δ dependent on drawing scale is an intuitive way to provide some control to the user, it would be necessary to make these thresholds customizable in a real application.

3.2.3.2 Geometric constraints

Intersections. We search for intersections between the input stroke and an existing stroke (Fig. 3.5), or between the input stroke and a grid point. We also detect self-intersections near the endpoints, to form closed loops.

To detect potential intersections with existing strokes, we use Unity’s built-in *Physics Colliders* to detect when the brush tip comes within a radius $r_{detect} = 2 * \delta$ of an existing stroke while sketching. For the regular grid, we simply check whether the brush comes within a radius r_{detect} of one of the grid points (see Section 4.1.3 for more details about the grid). We assume that the user wants to draw a closed loop if the endpoints of the stroke are within a distance r_{detect} of each other.

Tangent alignment. At an intersection, we compare the tangents between both strokes. If the angle between both tangents T_{new} and T_{old} is under the threshold θ , we add a constraint on the tangent of the new stroke at the intersection point (Fig. 3.6).

Planarity. We compute the best-fit plane to the control points by least squares. If the distance from the plane to the farthest control point is below the threshold r_{detect} , we add a planarity constraint to the stroke (Fig. 3.7). We also test whether the plane normal \vec{n} is close to one of the orthogonal directions of the

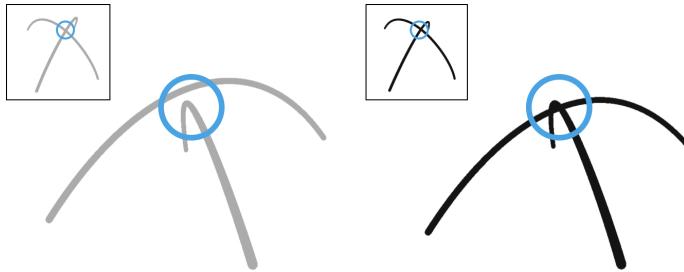


Figure 3.5: Left: strokes with near intersection. Right: beautified strokes with intersection constraint. Insets: other viewpoint.

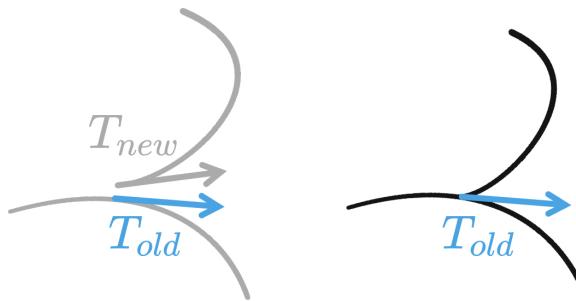


Figure 3.6: Left: strokes with near aligned tangents. Right: beautified stroke with tangent alignment constraint.

grid, and if it is close within an angle θ , we snap the plane to be an orthogonal plane with respect to the grid.

3.2.3.3 Exact constraints and fuzzy constraints

We decide to separate the geometric constraints between those that we want to enforce exactly (hard constraints), and those that we will simply try to reach, by minimization of a quantity (fuzzy, or soft constraints).

An intersection constraint enforces network connectivity, so we want to either completely satisfy it or reject it. Therefore we formulate it as a hard constraint. We formulate both planarity and tangent alignment constraints as soft constraints, because they correspond to aesthetic properties that are desirable but that we can settle for approximating.

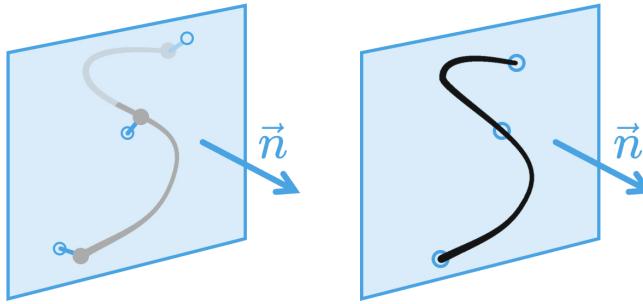


Figure 3.7: Left: non-planar stroke and best fit plane. Right: beautified planar stroke.

Having these constraints as soft terms gives us more flexibility in applying sets of constraints that would not lead to a feasible exact solution. For example we can apply a planarity constraint on a stroke with non perfectly planar intersection constraints, and still get a solution that enforces the intersection perfectly while being as planar as possible.

3.2.4 Enforcing constraints while preserving user input

Our goal is to enforce an arbitrary number of constraints on the input stroke, while preserving the user intention as much as possible. We assume that the input motion performed by the user is reliable to some extent, and we want to prevent deforming it, while making it respect as many constraints as possible. This problem is similar to what Xu et al. [56] describe for lifting a 2D sketch in 3D. Taking inspiration from their method, we formulate this problem as solving a constrained optimization.

We describe fidelity to the input stroke as an energy function with respect to the control point positions (Section 3.2.4.1), and pose the geometric constraints as hard and soft equality constraints on these positions (Section 3.2.4.2). We further explain how we can select an optimal subset of constraints from the set of all detected constraints, in order to avoid overly-deforming the input (Section 3.2.4.4).

In the following sections we will provide details about how we beautify strokes that are represented by cubic poly-Bézier curves. For line segments, due to their low degrees of freedom, it is enough to use some simple heuristics to select which constraints to apply. We then beautify the lines by modifying the endpoints

position, within reasonable thresholds (see Appendix A).

3.2.4.1 Fidelity energy

The objective function of the optimization problem describes our goal of staying as close to the user input as possible. Our formulation extends the *Projection accuracy* energy described by Xu et al. [56], to compare 3D curves, instead of 2D curves.

We want to minimize both variation in absolute position of the control points P_i^k from the input positions \bar{P}_i^k and variation in the slope of Bézier polygon edges $e_{ik} = P_{i+1}^k - P_i^k$ (see Fig. 3.8). This leads us to define the *fidelity energy* as:

$$E_{fidelity} = \frac{1}{|P_i^k| * L^2} * \sum_{i,k} \|P_i^k - \bar{P}_i^k\|^2 + \frac{1}{|e_{ik}|} * \sum_{i,k} \frac{\|(P_{i+1}^k - P_i^k) - (\bar{P}_{i+1}^k - \bar{P}_i^k)\|^2}{\|\bar{P}_{i+1}^k - \bar{P}_i^k\|^2} \quad (3.2)$$

We normalize each term by, respectively, the total number of control points $|P_i^k| = 3*K + 1$ and the total number of Bézier control polygon edges $|e_{ik}| = 3 * K$, K being the number of cubic Béziers. As our Bézier curves are all arbitrarily subdivided, depending on both input stroke curvature variations and the number of intersection constraints applied (see Sec. 3.2.4.2), we normalize these terms so that we can later compare fidelity energy between different candidate results (see Sec. 3.2.4.4). To the same end, we normalize the first term by a scale factor $L = \max_i(\|\bar{P}_0^i - P_0^0\|)$ (an approximation of the span of the stroke) and the second term by the initial lengths of the control polygon edges length $\|\bar{P}_{i+1}^k - \bar{P}_i^k\|$.

3.2.4.2 Geometric constraints formulation

In this section we provide details on how we express each of the geometric constraints mentioned in Section 3.2.3, in order to apply them in the optimization problem.

As explained in Section 3.2.3.3, we differentiate between hard and soft constraints. We note c the hard constraints, which are quantities that we constrain to be 0. We note E the soft constraints, which are quantities to minimize.

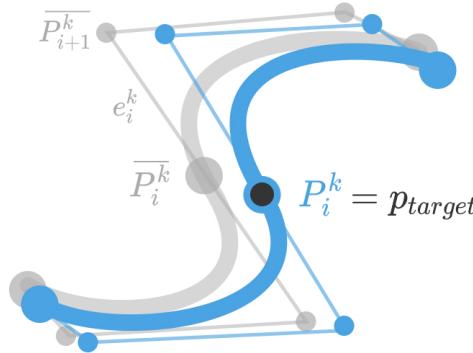


Figure 3.8: Input stroke (grey) and optimized stroke (blue) to match the intersection constraint p_{target}

Additionally to the geometric constraints that we detect (Sec. 3.2.3), we also enforce G^1 continuity between the different Béziers of the cubic poly-Bézier curve as a hard constraint. This is necessary to obtain smooth beautified curves.

Intersection constraint To constrain the poly-Bézier curve $B(t) \in [0,1]$ to intersect a point p_{target} , we decide to first compute the closest point on the curve from p_{target} :

$$p^* = B(t^*) = \min_t (\|B(t) - p_{target}\|) \quad (3.3)$$

Then, if we are close from an existing control point P_0^κ , such that: $\|P_0^\kappa - p^*\| < \delta$, we will constrain that control point. Otherwise, we split the input curve at t^* using de Casteljau's algorithm. This yields a new control point on the curve $P_0^\kappa = p^*$. In both cases, we express the hard constraint c as:

$$c = P_0^\kappa - p_{target} = 0 \quad (3.4)$$

We decide to split the input curve to allow for sets of intersections that may not have been feasible with the original poly-Bézier curve, for lack of degrees of freedom. We rely on the fidelity energy and the procedure described in Section 3.2.4.4 to reject constraints that would deform the input too much, and to select which constraint to apply at a control point if there are more than one.

Closed curve constraint We express the constraint that enables the creation of closed loops the same way as an intersection constraint between the endpoints of the curve: P_0^0 the first control point and P_3^{K-1} the last (K being the total number of cubic Béziers in the poly-Bézier curve).

$$c = P_0^0 - P_3^{K-1} = 0 \quad (3.5)$$

G^1 continuity constraint We want the poly-Bézier curve to have G^1 continuity, even after the beautification process, meaning that we want to satisfy some control point equality and tangent alignment between successive Béziers.

$$c = P_3^{k-1} - P_0^k = 0, \quad \text{for } k \in [1, K-1] \quad (3.6)$$

$$c = \frac{(P_3^{k-1} - P_2^{k-1})}{\|P_3^{k-1} - P_2^{k-1}\|} - \frac{(P_1^k - P_0^k)}{\|P_1^k - P_0^k\|} = 0, \quad \text{for } k \in [1, K-1] \quad (3.7)$$

For the C^0 constraint (Equation 3.6), we simply remove all control points variables P_0^k , for $k \in [1, K-1]$ from the optimization.

For the G^1 constraint (Equation 3.7), we linearize this constraint by approximating the norms of the control polygon edges by the initial norms: $\|P_i^k - P_{i-1}^k\| \approx \|\bar{P}_i^k - \bar{P}_{i-1}^k\|$. Making this constraint linear enables us to solve the optimization efficiently (see Section 3.2.4.3) and gives reasonable results in our use case.

Tangent alignment constraint We can constrain the tangent at a control point on the curve P_0^k to align with a target direction T_{target} :

$$E = \|(P_1^k - P_0^k) \times T_{target}\|^2 \quad (3.8)$$

Planarity constraint We can constrain all control points to lie in a plane of normal \vec{n} :

$$E = \sum_{i \in \{0, 1, 2\}, k \in [0, K-1]} \|(P_{i+1}^k - P_i^k) \cdot \vec{n}\|^2 \quad (3.9)$$

3.2.4.3 Solving the optimization problem

The complete optimization problem that we solve, for the control points position variables $\{P_i^k\}$, $i \in [0, 3]$, $k \in [0, K - 1]$, the m_h hard constraints and the m_s soft constraints is:

$$\begin{aligned} \min_{\{P_i^k\}} \quad & E_{fidelity}(P_i^k) + \sum_{j \in [0, m_s - 1]} E_j(P_i^k) \\ \text{st. } \quad & c_j(P_i^k) = 0, \text{ for } j \in [0, m_h - 1] \end{aligned} \quad (3.10)$$

The soft constraints energy functions E_j , and the hard constraints equations c_j are as defined in Section 3.2.4.2.

For each hard constraint $j \in [0, m_h - 1]$, we have n linear equations with k variables, such that we can write the constraint as: $c_j = C_j * P_{|k} = 0$, C_j a $n \times k$ matrix and $P_{|k}$ a $k \times 1$ vector that corresponds to a subset of the control points variables.

The objective function has a linear gradient, and all hard constraints are linear so we can find the solution to this optimization problem by solving a linear system (Equation 3.11), using the Lagrange multipliers method [36].

$$\begin{bmatrix} A & C^T \\ C & 0 \end{bmatrix} * \begin{bmatrix} P \\ \Lambda \end{bmatrix} = \begin{bmatrix} B \\ 0 \end{bmatrix} \quad (3.11)$$

We build A and B such that $A * P - B = \frac{\partial}{\partial P} (E_{fidelity} + \sum E_j)$. The matrix C corresponds to the stacked C_j blocks of the hard constraints. P are the control points position variables, it is a vector with $3 * (K * 3 + 1)$ lines, one line per control point coordinate. Λ is the vector of Lagrange multipliers.

3.2.4.4 Intersection constraints rejection

Prevent over-deformations. If we enforce all intersection constraints that are detected (Sec. 3.2.3), nothing guarantees that we wouldn't deform the input stroke into something very different from what the user drew (see Fig. 3.9b).

In the set of m detected intersection constraints $S = \{C_i\}_{i \in [1, m]}$ there is a subset $S^* \subseteq S$ of constraints that will not overly deform the input stroke when

applied (Fig. 3.9b). To find this subset, we can solve for all the possible subsets to obtain different candidates for the beautified result. We can then compare the candidates and choose the best one, according to some metric. A similar approach where multiple beautified candidates are generated from subsets of geometric constraints is used by Igarashi et al. [25] and Fišer et al. [16] in 2D beautification, and by Schmidt et al. [44] to infer depth from scaffolding strokes.

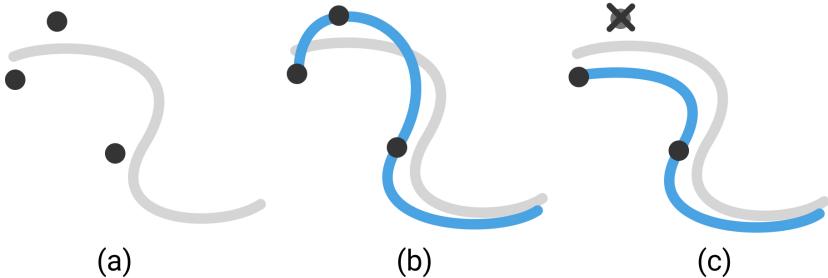


Figure 3.9: (a) An input stroke with all detected intersection constraints. (b) Applying all constraints results in a very distorted stroke. (c) Applying a subset of the constraints results in a stroke close to the input.

Evaluate candidates. We need an appropriate metric to evaluate the candidate results obtained from each subset of constraints. The fidelity energy is a good measure of how much the candidate deviates from the input. However, the subset that results in the optimal fidelity energy would always be the empty subset $S^* = \emptyset$, as the output would be identical to the input stroke. We need to balance our fidelity objective with a second objective. That is that we strive to satisfy as many intersection constraints as possible, to beautify the stroke and make it more regular.

Thus we define a *regularity energy*, to express this objective. Empirically, we want this energy to make it costly to remove a constraint and also that it becomes increasingly costly to remove more constraints. Like Schmidt et al. [44], we observe that some constraints are more desirable than others, so for different kind of intersections we define a different cost c per constraint (see Fig. 3.10). We then compute the total energy of a subset $S^* \subseteq S$ of intersection constraints as:

$$E_{\text{regularity}}(S^*) = e^{-(\frac{c_{S^*}}{c_S})^2} \quad (3.12)$$

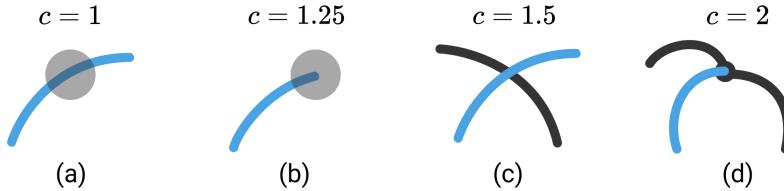


Figure 3.10: Intersection constraint cost in different cases. (a) At a grid point. (b) At an endpoint of the input stroke. (c) Intersection with an existing stroke (black) (d) At an existing intersection in the sketch.

with the subset cost $C_S^* = \sum_{i \in S^*} c_i$, and the full set cost C_S computed similarly on all detected intersection constraints. The cost by constraint c_i is defined as in Figure 3.10.

Then, finding the best candidate subset boils down to finding a subset of constraints S^* such that the resulting stroke control points P_i^k (from solving Equation 3.11) and the cost of the subset C_S^* minimize the following energy:

$$E = \lambda * E_{fidelity}(P_i^k) + (1 - \lambda) * E_{regularity}(S^*) \quad (3.13)$$

The parameter λ expresses how important fidelity to the input is relatively to regularity enforced by the intersection constraints. We find that $\lambda = 0.85$ works well, after using the system extensively.

Greedy search on subsets. We decide to find a solution using a greedy approach. Solving the linear system of Equation 3.11 is the most costly part of the algorithm, and there is no way to pre-factor the matrices as they vary depending on which intersection constraints are applied, and how the poly-Bézier curve is subdivided. Therefore, we can't afford to test every subset of S ($2^{|S|}$ subsets).

We know that the set of intersection constraints that we initially have should contain mostly valid constraints, as we only register a constraint if the input stroke actually passes near the constraint position. What we really want to do is *reject* the constraints that are superfluous, or incompatible with other constraints. This case can arise for example in a zone with very high stroke density, causing a lot of possible intersections to be detected (Fig. 3.11).

We find a suitable subset while minimizing the number of times we need to solve the linear system by starting from the full set of constraints S and iteratively testing subsets from S formed by removing one element, until we don't see any improvement in the energy E (Equation 3.13). For example if we have m constraints in a set S_m , we solve while applying all constraints and compute the energy E_m for this set. Then we solve for all m subsets of S_m formed by removing one constraint. If the best of these subsets S_{m-1} has an energy E_{m-1} lower than E_m , we continue looking at smaller subsets of S_{m-1} . Otherwise we stop and keep the subset S_m .

While this method doesn't guarantee that we find the optimal subset, in practice it gives good result in a reasonable amount of time, as there are usually not many constraints that should be rejected (see examples on Figure 3.11).

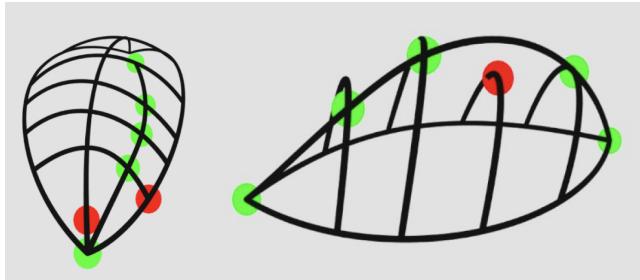


Figure 3.11: Cases where too many constraints are detected. This is susceptible to happen in areas with a lot of strokes (left), or in cases where the input stroke is smooth but the potential constraints do not form a smooth path (right). Some constraints are applied (green), while some are rejected (red).

3.3 Surface inference

Similarly to how an artist defines a complex shape with only a few strokes on a 2D sketch, one can draw 3D strokes that represent only sparse features of an envisioned 3D solid shape. However, the free view-point navigation that a 3D sketch offers - contrary to a 2D sketch - poses additional challenges to the artist, as their work may be hard to read from some viewpoints (see Fig. 3.12). In this section, we propose a method to address that challenge by automatically inferring the surfaces that a sparse set of strokes represent, thus providing valuable occlusion cues to help viewers understand the 3D shape.

To achieve this, we can first rely on our beautification algorithm (Sec. 3.2) to

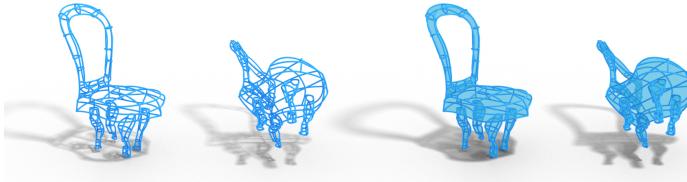


Figure 3.12: Sketch of a chair, without and with surface patches

form a well-connected curve network from the user’s input strokes. From the intersection constraints that we apply on the strokes, we extract connectivity information to build a curve network (Sec. 3.3.1). On the curve network we need to infer which cycles should bound a surface patch, and which should not (Sec. 3.3.2). In ambiguous cases, we will rely on user input to guide the process (3.3.3). Finally, we generate geometry for the surface patch, so that we can render it in the drawing (Sec. 3.3.4). We can further consider the surface patches as a scaffolding entity in the sketch, on which users can draw strokes to add small details (Sec. 3.3.5).

3.3.1 Curve network from intersecting strokes

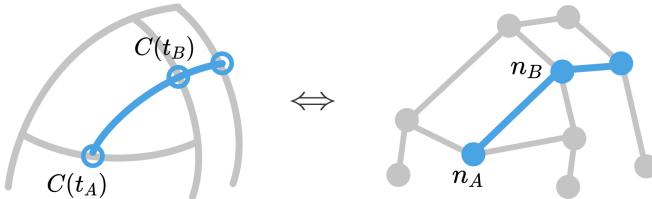


Figure 3.13: Geometric representation of the sketch with strokes, and equivalent graph representation of the sketch with segments and nodes.

We define a graph representation of the sketch, that we update after each stroke addition or deletion operation. The graph contains all connectivity information about the strokes. It is composed of segments and nodes:

- A node n_i has a position in space. It also maintains a list of all its neighboring segments.
- A segment has two endpoint nodes n_A and n_B . It is the restriction of a

parameterized curve $C(t)_{t \in [0,1]}$ to an interval $I_s = [t_A, t_B]$ bound by the endpoint nodes n_A and n_B (see Fig. 3.13).

When a stroke is added to the sketch, we look at which intersection constraints are applied by the beautification algorithm. These intersection constraints contain a reference to the intersected stroke, from which we can find the corresponding segment in the graph. Then we can update the graph by appropriately splitting segments, and adding nodes.

Similarly, when a stroke is deleted we remove the corresponding segments from the graph and merge the neighboring segments if necessary.

3.3.2 Automatic surface patch detection

In the graph representation of the sketch, we must decide which cycles most probably bound surface patches that the user envisions. While there exists methods to do so on a complete curve network [1, 57], in our case the curve network is iteratively created by adding strokes to the sketch. We decide to choose a local approach to trigger patch creation and deletion after the user sketches or deletes a stroke.

First, we need an algorithm to walk the graph, searching for the most probable user-intended cycles that contain a given segment. We then combine that algorithm with an update mechanism to decide on which segments we should run the algorithm after each user action on the sketch (see Appendix B).

3.3.2.1 Local cycle search algorithm.

Inspiration We adapt the method that Stanko et al. [48] apply on a 3D curve network with normal information to our case where we do not have normals on the curves as input. The intuition behind their method is that if the surface represented by the curve network is smooth and manifold, at each node it is possible to sort the segments around the normal vector at the node, so that at a (Node, Segment) pair, we can always determine which segment comes next in a cycle lying on the surface. This is similar to what can be done on a 2D curve network (see Figure 3.14).

This method finds cycles by considering only local choices at nodes of the graph, therefore we found it to be well-suited to our use case. It is indeed very easy to

implement in our context of frequent local updates on the graph, that may only break and create cycles among the neighboring segments. Compared to other approaches that rely on evaluating curve pairings [57] or cycles [1] according to some metric, the method we adopt doesn't require us to list and compute scores for such high-level objects. We progress along the graph according to local geometric properties that we can compute on the fly.

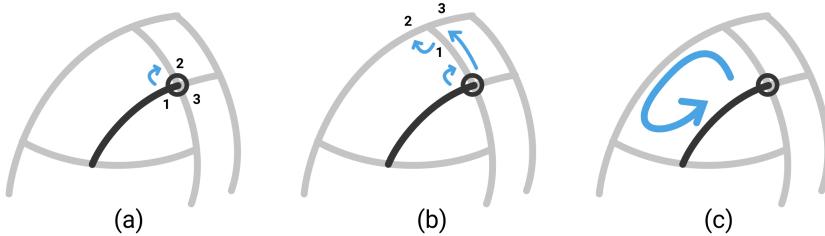


Figure 3.14: Finding a cycle on a 2D curve network. (a) At a node, we can sort the segments, and we take the next segment in clock-wise order. (b) We go to the opposite node and do the same thing. (c) When we reach the initial node again, we have found a cycle.

Defining normals at nodes The particularity of our case is that normals are not known on the curve network. We do not exploit controller orientation information, as we observed that users do not necessarily orient their hand appropriately to convey the correct normal orientation at a stroke, as it would require them to twist their wrist or move around the sketch a lot. However, each node with 2 or more non collinear segments attached gives us a good approximation of the tangent plane of the surface at this point, from which we can compute the normal.

At each node with 2 or more non collinear segments, we compute a normal by fitting a plane to the tangent vectors of the strokes at this point. Assuming that the strokes represent a smooth surface, and that there are enough strokes to correctly define it, this gives us a good approximation of normal directions at each node. Still, an ambiguity remains in the orientation of the normals, as we don't know a priori which are pointing outwards or inwards with respect to the surface that we are trying to find.

We lift this ambiguity locally, by comparing normal orientations in-between neighboring nodes. Assuming that a curve segment lies on the envisioned surface without significant geodesic torsion, parallel-transporting the normal \vec{N}_A from endpoint A to endpoint B should give a vector $\vec{N}_{A\parallel B}$ which approximately

matches the direction of normal \vec{N}_B [37]. By transporting a node normal along a curve segment [22], we can compare orientations of both normals from the segment endpoint nodes. If they are consistently oriented, we can apply the method explained before exactly. If they are oriented in opposite directions, we should simply reverse the order in which we go around this node, by going counter-clockwise instead of clockwise (see Fig. 3.15).

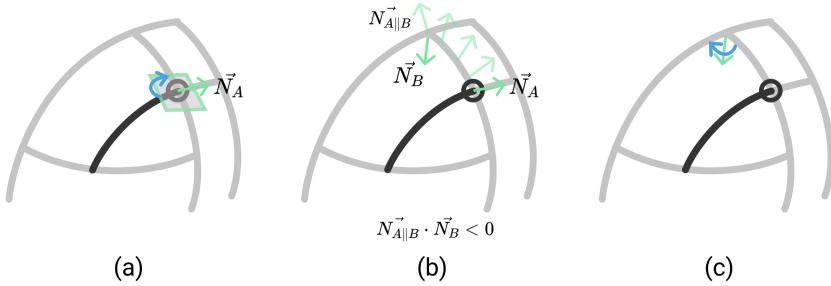


Figure 3.15: Finding a cycle on a 3D curve network. (a) At a node A , we can approximate the tangent plane of the surface (with normal \vec{N}_A) and sort the segments in this plane. We take the next segment in clockwise order (b) We go to the opposite node, parallel transport the node normal \vec{N}_A to the node B , then compare orientations between $\vec{N}_{A||B}$ and \vec{N}_B . (c) As the normals are oriented in opposite directions, we take the next segment in counter-clockwise order around the node normal \vec{N}_B .

3.3.2.2 Limitations

The simplicity of the method we use induces a number of limitations. Indeed, we make some hypotheses on the properties of the surface that we seek, which happen to not always be true in a sketch. Here we present those limitations and some alternatives, and in Section 3.3.3 we will present a user-guided fallback to overcome failures to detect some surface patches.

Sharp features. The algorithm works reliably if we can assume that the strokes describe a smooth surface, such that the normals that we compute at each node correspond to the normal of the intended surface at this point. If that's not the case, for example if the neighboring strokes of a node define sharp features, there is no guarantee that we can compute a sensible tangent plane and normal, and in consequence, the segments can't be reliably sorted at this

node.

We strive to limit the failures caused by sharp features, which we found to be very frequent in the sketches, by using an alternative heuristic at such nodes (see Fig. 3.16). We detect when there is no consistent tangent plane at a node by looking at the error from the plane fitting operation we do to compute the normal. If it is above a threshold that we empirically define at $\epsilon_{sharp} = 0.3$, then we label the node as being sharp.

At such a node B , we know that we can't rely on sorting nodes around the node normal \vec{N}_B to decide which should be the next segment (Fig. 3.16a). Instead, we will rely purely on the transported normal from the previous node $\vec{N}_{A\parallel B}$. First we exclude the segments that are not close to being in the plane defined by $\vec{N}_{A\parallel B}$, and sort the rest around $\vec{N}_{A\parallel B}$. Then we choose the next segment, in the same order (clockwise or counter-clockwise) as before (Fig. 3.16b). This heuristic is based on the intuition that the parallel-transported normal from the previous node should be a good guess of the normal in the vicinity of the sharp node, on the side of the patch that we are trying to find. Therefore we only look at segments that approximately lie in the tangent plane defined by the transported normal.

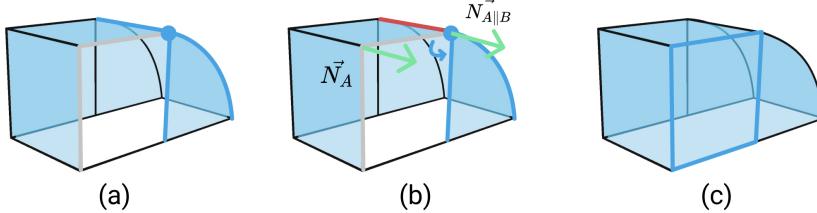


Figure 3.16: (a) At a sharp node B (blue), the node normal is ill-defined. We cannot use it to determine which of the neighboring segments (blue) should be the next segment in the cycle (grey). (b) We exclude the segment (red) that is not in the plane defined by the parallel-transported normal $\vec{N}_{A\parallel B}$ and choose the next segment in clockwise order around $\vec{N}_{A\parallel B}$ among the remaining options (blue). (c) The cycle is detected successfully.

However, the limitation remains in cases where a sharp feature is defined with not enough strokes to form clear sharp intersections (see Fig. 3.17). In this case, adding more strokes can fix the issue.



Figure 3.17: In some cases there are not enough intersections in the sketch to successfully detect cycles. Adding strokes provides more cues to the algorithm, which has more chances of succeeding.

Trim curves. Our method to disambiguate the normal orientations by using parallel-transported frames (or Bishop frames) along the curve relies on the hypothesis that the curve lies on the intended surface without significant geodesic torsion. Curves on a surface with no geodesic torsion are aligned with curvature lines of the surface, and have been found to be frequent in 2D sketches [24, 37]. We argue that this is a good hint that most curves in the 3D sketch should present that property too. However, there are cases where the curves instead represent trim curves of the surface and are not aligned with curvature lines (Fig. 3.18a). Along such a curve, the transported normal $\vec{N}_{A||B}$ may be very different from the actual surface normal at the node \vec{N}_B (see Fig. 3.18b). Therefore, comparing those normals will not yield meaningful information on their respective inwards or outwards orientation with respect to the surface, as the directions between them may widely vary.

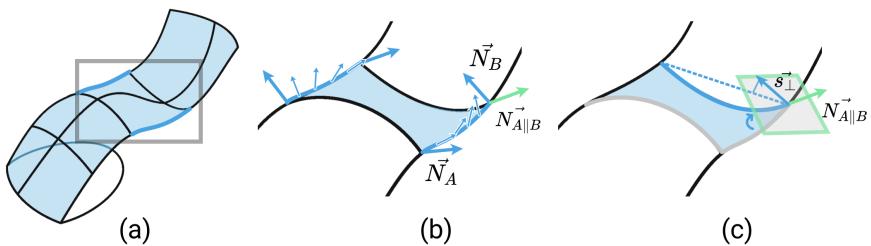


Figure 3.18: (a) A chair sketch that comports trim curves (blue). (b) We see that the surface normals (blue) do not correspond to parallel transported families of vectors along the trim curves. Therefore, $\vec{N}_{A||B}$ does not match the direction of \vec{N}_B , as it would on a curvature line. (c) We disambiguate this case to find the next segment in the grey cycle by looking at the projections of segment directions s_\perp in the plane defined by $\vec{N}_{A||B}$.

In such cases, at a node B where we detect that the segment between A and B is a trim curve ($\vec{N}_{A\parallel B}$ and \vec{N}_B have very different directions), we make an attempt at choosing whether to go in clockwise or counter-clockwise order around B by projecting main segment directions on the plane defined by the parallel transported normal $\vec{N}_{A\parallel B}$ (see Fig. 3.18c). This amounts to look at the main shape of the patch that would be formed by choosing each one of the segments. This is more reliable than looking at the local tangent plane at B , where we cannot disambiguate normal orientation. This is quite hazardous, for one it is not clear what we should define as "main direction" for a segment. We take the line between the endpoint nodes of the segment, but for a very long and curved segment this can be quite misleading. However, it does manage to sort out a number of cases like the one in Fig. 3.18.

Non-manifold objects. In the automatic cycle detection algorithm, we assume that each curve can't bound more than two cycles. This works well if the surface that is intended is a manifold surface. In sketches, it is relatively common that one wishes to create non-manifold surfaces, for example to add the wing on a plane (see Fig. 3.19). While the final visible surface of the plane may be a manifold surface, while the user is sketching the wing, they may be temporarily in a non-manifold configuration. To alleviate the frustration that these cases can raise, we add the possibility for the user to enforce the creation of a non-manifold surface (see Section 3.3.3).

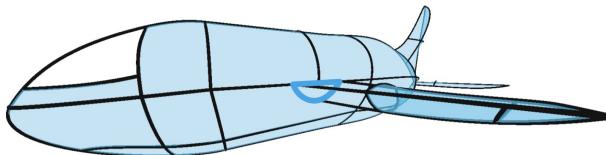


Figure 3.19: The wing of the plane is not surfaced correctly near the base because the automatic algorithm forbids the curves at the base (blue) to bound 3 patches.

Reliance on tangent directions at intersections This method relies quite heavily on how well tangent directions on curves at intersections describe the underlying surface curvature. This will cause issues if the tangent directions are susceptible to be badly distorted near intersections, for example near stroke endpoints. This dependency made implementing hooks removal as a pre-process

(Section 3.2.2) highly necessary for the patch detection algorithm to work reliably, as hooks typically distort the stroke curvature near the endpoints.

3.3.3 User guided patch creation

To overcome the limitations of our patch detection algorithm mentioned in Section 3.3.2.2), we rely on the interactive nature of our system to let the user guide the patch formation process.

We let the user trigger patch creation by positioning their controller near the center of the patch that they envision, and clicking a button.

We then use this position information to find the closest stroke segment, then walk along the graph by always choosing the segment that goes most towards the direction of the input position. We do not limit our search to manifold surfaces, so we also look at segments that already bound two patches.

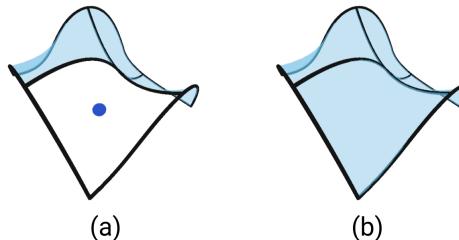


Figure 3.20: (a) The user positions their dominant hand (blue dot) near the center of the patch that is missing and clicks. (b) The patch is detected.

This rather simple algorithm works in most cases, provided that the patch is small enough such that its center is defined without ambiguity with respect to the neighboring strokes. The users can always add more strokes to the sketch in case the patch that they envision cannot be created in this way.

3.3.4 Surface patch geometry

Once we know which cycles bound patches, we can generate a triangle mesh that fills that cycle. First, we triangulate the closed 3D curve formed by the

cycle segments to get a triangle mesh topology and initial vertex positions, then we remesh and smooth the surface.

We choose to use existing methods for which we have available source code in order to get a result efficiently, without having to delve in depth into the implementation.

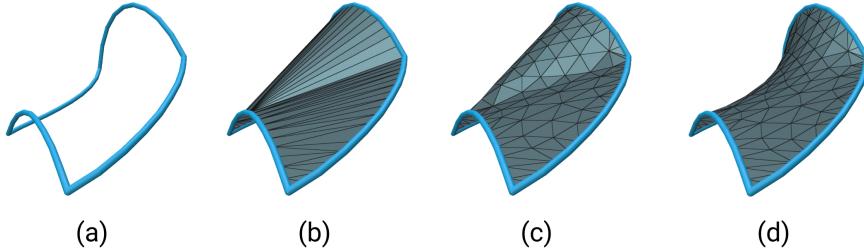


Figure 3.21: (a) Boundary curve. (b) Boundary curve triangulation. (c) Isotropic remeshing. (d) Smoothed mesh.

Triangle mesh from patch boundary We directly apply the algorithm by Zou et al. [58], for which the C++ source code is provided. We call the triangulation source code from the run-time application as a *Unity* native plug-in [50] by providing a C interface to native code and compiling it as a shared library.

We give as input a list of vertices on the boundary of the cycle to the triangulation plug-in (Fig. 3.21a), and get as output a triangle mesh with vertices and face indices (Fig. 3.21b). However, before sending the mesh back to the runtime application, we process it to make both the mesh and the shape smoother.

Mesh processing When the patch is triangulated, we refine it in 2 steps, using the C++ CGAL geometry processing library [51]:

- We isotropically remesh the interior while constraining the boundary, so that all edges become close to a target edge length (Fig. 3.21c).
- We smooth the shape using CGAL’s implementation of the curvature flow algorithm [11, 28] (Fig. 3.21d).

While this process provides us with a smooth mesh, the shape of the mesh does not necessarily match user expectation. Indeed, the smoothing operation

minimizes membrane energy which makes the surfaces tend towards minimal surfaces. This is why they look like "soap bubble films" (see Fig. 3.22a). To obtain more intuitive results for the surfaces, we could apply methods to optimize the mesh curvature to match curvature of the input curves, such as the work of Pan et al. [37] or Stanko et al. [48]. Using the algorithm of Stanko et al. [48] by giving as input normals on curves extrapolated from the normals at nodes of the curve network, gives better results than our smoothed surfaces (see Fig. 3.22). However, for lack of time, we didn't fully integrate their method in our runtime application. The main challenge is to compute correct normal information for each boundary curve. The normals that we compute at each node have ambiguous orientation, so it is not trivial to extrapolate them on the curve. Moreover, our surface patches are computed one at a time, so we must be careful to always consistently orient normals across patches that share a boundary curve.

We settle for these "soap bubble" surfaces, which still fulfill the initial goal of providing valuable occlusion cues. The user can always define the surface more precisely by adding strokes to stretch the surface, as if they were adding support rods to a tent structure.

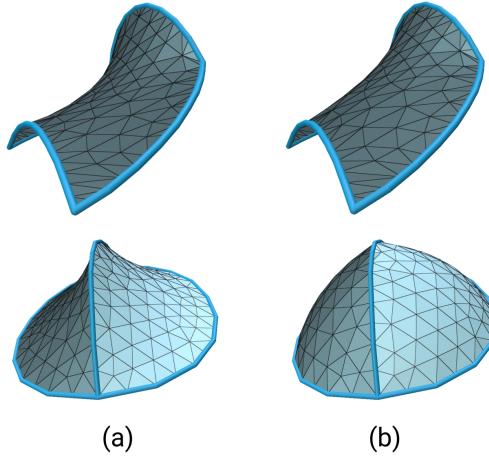


Figure 3.22: (a) Our result. (b) Surface is better aligned with user expectation using [48].

3.3.5 Drawing on surfaces

We enable users to use the surfaces as solid scaffolds to sketch on. They can thus create small details or pieces with a base lying on an existing surface in

the sketch (see Fig. 3.23). To do so, we first automatically detect which input strokes are intended to lie on a surface patch. Then we project the stroke so that it appears to lie on the surface.

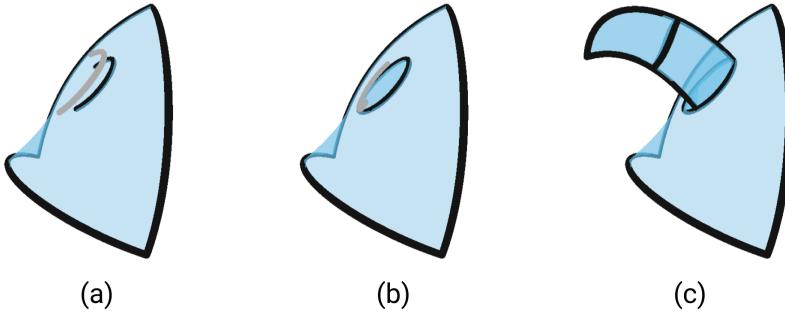


Figure 3.23: (a, b) The input stroke (grey) is neatened to lie on the surface patch. (c) The user draws a horn attached to the patch.

Detecting on-surface strokes. When the user sketches a new stroke, we classify it as being either a standard stroke or a stroke intended to be drawn precisely on a surface patch. The criteria to decide whether the stroke is an on-surface stroke on a surface patch S_i are:

- *Proximity.* All points on the input stroke are within a distance $2 * \delta$ of S_i .
- *Alignment.* The start and end tangents of the stroke are not within an angular threshold θ of the normal of S_i at the nearest point from stroke start and end points.
- *Non-breaking stroke.* The stroke does not break the cycle that bounds the patch S_i .

See section 3.2.3.1 about the threshold values δ and θ mentioned here.

We allow both stroke endpoints to be neatened to lie on one or two different surface patches if they follow the *proximity* criteria. This can be useful to draw a handle on this basket for example (Fig. 3.24).

Neatening on-surface strokes. To correct the input stroke such that it seems to lie on a surface patch, we choose to simply project its control points

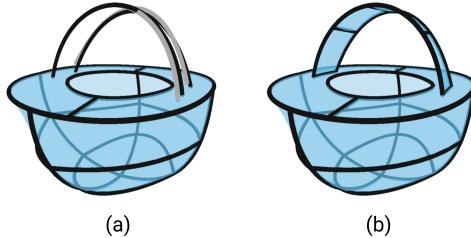


Figure 3.24: (a) The input strokes (grey) are neatened so that the endpoints lie on the patches representing the top of the basket. (b) The user continues to sketch the handle.

on the patch. This does not rigorously make the entire stroke lie on the patch, as Bézier curve points are weighted averages of the control points and taking a linear combination of points on a surface does not necessarily yield a point on the surface except on a plane. However, it looks correct in most cases, by rendering all surface patches slightly further in the depth direction than the strokes.

A better approach to do that could be to compute the correct on-surface positions for the points of the Bézier curves by computing weighted averages on the surface [38].

For simplicity, we choose to apply this on-surface projection as a post-process to the rest of the beautification process (Sec. 3.2). A more unified approach would be to integrate the on-surface constraint as a geometric constraint in the optimization problem that we solve for beautification. This could prevent over-deformation of input by enforcing fidelity, as we do for the other constraints.

Limitations. We observe that it is not easy to draw consistently in the proximity of a surface, especially for longer strokes or on very curved surfaces. To limit the precision and effort required to draw strokes on surfaces successfully, we could have a specific sketching mode where strokes would be projected on a surface patch selected by the user. This could be similar to the interaction in *EverybodyLovesSketch* [5] where the user can select a surface bound by strokes to define it as a temporary sketching canvas where the strokes will be projected.

Having a specific mode to sketch on the surface could also lift the ambiguity between strokes that should connect to the network that defines the surface, and strokes that should simply lie on the surface. It is indeed quite difficult to sketch in the interior of a patch without accidentally getting close to the strokes

that bound the patch, which may trigger undesired beautification.

Finally, an interesting yet unresolved aspect of having strokes lying on surfaces is a way to gracefully deal with the strokes when the underlying surface is deleted, or modified. In the current prototype, we simply leave the strokes hanging mid-air, but we could imagine trying to project them on the nearest remaining surface for example.

CHAPTER 4

Results

We build a prototype virtual reality application combining all the ideas exposed in the previous chapter. Here we present the user interface for this application (Sec. 4.1), and how we evaluate it. To evaluate this project, we create a variety of 3D sketches with it to demonstrate its versatility and expressivity (Sec. 4.2.1). Then we conduct a user study to evaluate how well this interface can help users explore 3D designs in virtual reality (Sec. 4.2.2).

4.1 User interface

The user interacts with the application through two tracked VR controllers, one in each hand. They wear a VR headset to view the scene from different viewpoints, in stereo. The only restriction we impose on hardware is that it should be compatible with SteamVR [53], which is the case for the most common headsets (HTC Vive, Oculus headsets and Valve Index). The user interface is also showcased in the video created for the user study (Appendix C).

To get an idea of how the different actions we mention in this section are mapped to a VR controller, please refer to Appendix D to see action to button mappings for the SteamVR supported controllers.

4.1.1 Draw and delete

The main interactions are to draw strokes and to delete elements (either a stroke or a surface).

Drawing is done by pulling the trigger button on the dominant hand controller, which is similar to the drawing interaction in other VR creation applications such as TiltBrush [17] and GravitySketch [20].

Deleting an object is done by bringing the dominant hand close to the object that needs to be deleted, and clicking a button. When the dominant hand is close enough to the object such that a click will trigger deletion, we shade the object differently with a red color, to signify that it is selected for deletion (see Fig. 4.1).

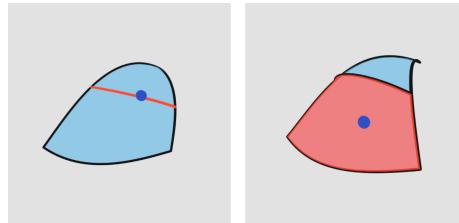


Figure 4.1: When a stroke or surface patch is selected by the user by putting their dominant hand (blue dot) in its proximity, the object becomes red.

4.1.2 Navigation

To enable the user to navigate in the scene as they draw without having to physically move around, we adopt a "grab and move" interaction metaphor. We also allow the user to scale up or down the drawing volume with a two-handed gesture. Both of these interaction metaphors are found in other VR applications [17, 20] and should therefore feel familiar to experienced VR users. For novices, they still quite natural as they mimick real-life 3D manipulation (for grab and move) or usual 2D touch-screen interactions (for zoom).

Grab and move. By pressing a button, the user starts grabbing the whole drawing volume. They can then see their hand motion mapped to a translation and rotation of the drawing volume in real-time. We display the grabbing hand

in a yellow color to signify grabbing. When they release the button, the drawing volume is fixed again in the position they left it on release. We implement this interaction following the method described by Robinett and Holloway [40].

Scaling. By pressing a button and moving both hands further apart or closer together, the user sees the scene scale up or down around them. We display the current zoom level and a line between both hands, as a cue that scaling is happening (Fig. 4.2). When the user releases the button, the scene scale becomes fixed again. We use scaling as a way for the user to signify their intent to be more or less precise: when the scene is zoomed in, the trigger zone around an element to select it for deletion becomes smaller, which makes it easier to select something in a cluttered area of the sketch. Moreover, as mentioned in Section 3.2.3.1, the distance threshold that rules over the beautification algorithm becomes smaller when the drawing is zoomed in, making it possible to avoid some undesirable snapping of strokes, which is also very convenient when the stroke density gets higher. We implement scaling as described by Robinett and Holloway [40], such that the dominant hand is the origin of the scale operation.

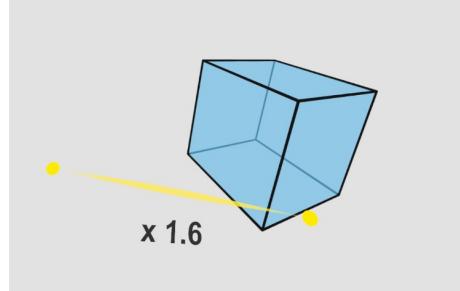


Figure 4.2: The user is scaling the drawing.

4.1.3 Helper objects

Finally, we add to the interface a few helper objects to assist the user in the creation process.

3D grid. We display a 3D grid composed of points and thin lines, that covers the whole drawing volume. The grid points also serve as intersection targets in the beautification process (see Sec. 3.2.3).

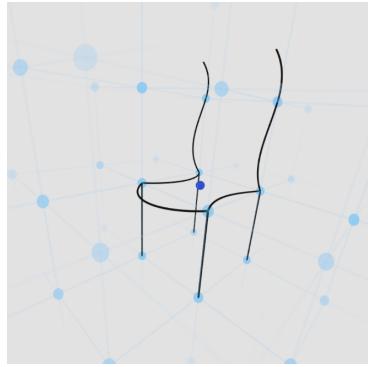


Figure 4.3: The grid can be used as a scaffold to define the proportions of this chair in the early stage of the sketch.

The 3D grid has two purposes. First it serves as a scaffold structure that the user can rely on to build their 3D sketch (Fig. 4.3). It is inspired by previous work on 2D sketching interfaces [5, 44] that use scaffolding elements to help users define shape and position in space of their strokes. Moreover, in the context of VR 3D sketching, it helps the user perceive scale in the depth direction correctly, which has been found to be difficult [2, 34]. Machuca et al. [34] also advised to use visual guides to provide depth cues, which the regular 3D grid achieves.

To prevent cluttering the user’s field of view, we display the grid only in the vicinity of their dominant hand, and gradually fade its visibility away from it. The grid lines help the user perceive the main orthogonal directions of the drawing space, which can serve as beautification targets for lines, or for planar strokes (Sec. 3.2.3).

We also link the grid resolution to the scaling factor, such that when the scaling of the drawing volume reaches double the initial scale, we double the grid resolution. This enables users to get more precise measurements while zoomed in. A further step would be to make the grid resolution customizable, as in *EverybodyLovesSketch* [5].

The grid is procedurally generated by combining a vertex shader and a geometry shader. The vertex shader generates the regular grid point positions, given a resolution, the number of desired points and a position for the grid origin. Then the geometry shader generates vertices for view-aligned circles and lines, to render the grid points and lines. It is therefore easy to edit the grid resolution in real-time.

Mirror plane. We add a planar mirror to the drawing scene, to help the user draw symmetric objects. The user can draw a stroke on one side of the mirror, and its symmetric will automatically appear on the other side of it.

To enable the mirrored strokes to serve in further patch creation, we must put some care in updating the stroke graph correctly, for example connecting strokes that have one endpoint on the mirror to their symmetric. We also assist the user in using the mirror by adding points on the mirror plane that come close to the stroke as potential intersection constraints (Sec. 3.2.3). In this way it becomes easy to draw a closed path by drawing only one side of it, ending the stroke in proximity of the mirror (Fig. 4.4). We also automatically project strokes that lie close to the mirror to perfectly lie on it, as it is usually what the user intends.

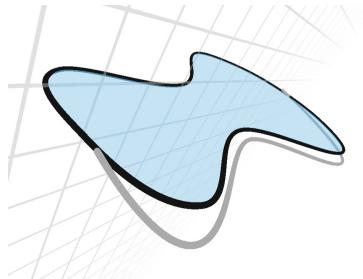


Figure 4.4: The user draws on only one side of the mirror (grey stroke) and the result is a closed path obtained by constraining the stroke endpoints to the mirror and creating the symmetric stroke.

We mirror all stroke creation and deletion operations. However, for lack of time, we do not leverage the mirror for the patch creation and deletion process. To find surface patches, we simply apply the same method on new strokes on either side of the mirror. Nevertheless, we mirror user requested patch addition (Sec. 3.3.3), to alleviate the user’s work in adding missing patches.

We display the mirror as a planar grid with thin grey lines, to help users perceive where it stands in space (see Fig. 4.4).

Intersections visualization. Stroke intersections play a crucial role in the correct formation of surface patches. We provide the user with cues that an intersection was detected correctly, in order to facilitate their understanding of the underlying network, which can help them diagnose why some patches are not correctly inferred. We display all intersections as small black dots with low opacity (see Fig. 4.5).

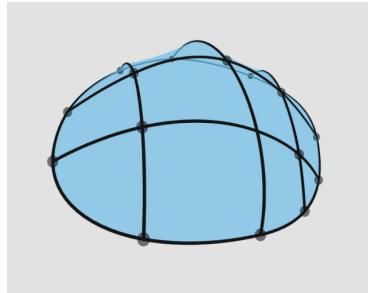


Figure 4.5: The intersections are displayed as little black dots. Here it is easy to see which intersection was not enforced in the sketch.

4.2 Evaluation

We evaluate whether our system can help users create 3D concepts by sketching in VR. We were initially inspired by traditional sketching, which is a way for designers to express their ideas in a quick and inexpensive way, with a minimal amount of details to leave room for idea exploration rather than give a precise specification of the shape (Sec. 1.1.1). At the same time, we proposed to alleviate the imprecision challenge of 3D sketching and provide adequate visualization of the sketch as it is created (Sec. 1.1.2). We seek to verify whether our prototype fulfills some of these qualities.

First, we show a variety of 3D sketches created by an expert user of the application, to demonstrate its potential and analyze performance of the automatic surface detection method (Sec. 4.2.1). Then we conduct a user study with 4 participants to test whether the system is intuitive to learn and use for novice users, and provide them with some benefits compared with free-hand 3D sketching (Sec. 4.2.2).

4.2.1 Sketches and analysis

In this section we present a few 3D sketches created by someone familiar with the system. All sketches were done in a short amount of time, which includes time needed for exploring multiple possibilities for each sketch (see Table 4.1). The mirror plane was used for symmetric shapes such as the vacuum cleaner or the sewing machine. The artist often based the 3D sketch on a 2D photo reference, that they would examine before starting the sketch in VR. No existing 3D models were used however.

Qualitatively, we confirm that the system permits a great level of versatility in the shapes that can be created. We observe that the system allows for the quick creation of free-form surfaces, such as the folds of the dress, or the architectural structure, which can be first roughly defined with a few strokes, then refined locally to achieve the desired complex curvature.

By looking at the results we can see that the surfaces greatly help in understanding the 3D shape that is represented by the strokes, from multiple viewpoints.

We observe that the "draw on surface" feature is mostly used to draw small attachments on the surfaces, such as the feet of the chair, the buttons on the sewing machine and on the mouse or the spikes on the shield. We do not see much interest in drawing textural details on the surface, which is coherent with the idea of keeping the sketch at a low detail stage, to focus on exploring the form.

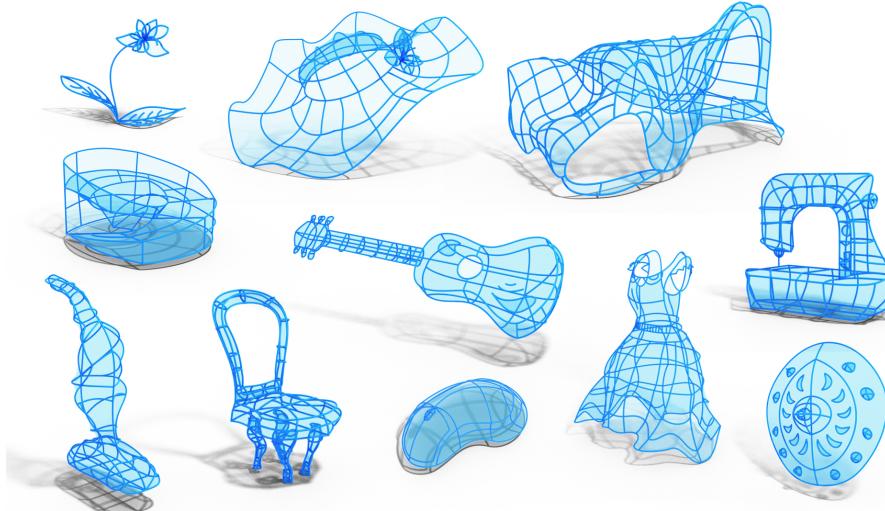


Figure 4.6: 3D sketches created by an expert user with the prototype application.

We also measure the number of patches detected for each sketch, both those created automatically by the cycle detection algorithm (Sec. 3.3.2) and those created by the semi-automatic procedure (Sec. 3.3.3). These quantities (see Figure 4.7), give us an overview of the performance of the cycle detection algorithm. The numbers may be slightly skewed in favor of the algorithm because we don't count the false negative patches (detected patches that were not correct and had to be deleted by the user), however this case happens rarely, as it is more often a case of missing a few patches.

Table 4.1: Completion times for all models (min)

Computer mouse	6:40
Flower	8:43
Shield	4:57
Guitar	15:18
Vacuum	7:14
Chair	9:54
Dress	9:10
Sewing machine	12:34
Hat	10:17
Architecture concept 1	11:43
Architecture concept 2	18:50

We see that the amount of manual input for patch detection depends on the sketch. For example, the guitar sketch is the one that required the greatest amount of manual input to find all surfaces correctly (83 patches found following a manual "add patch" input). This is coherent with our discussion in Section 3.3.2.2, where we showed that the automatic cycle detection will be more or less successful depending on the type of surface that the user is trying to represent. The guitar is typically an object with many sharp edges, which explains the poor performance of the algorithm. Similarly, the chair also has a rather high count of manual patch detection (41 patches), which can be explained by the presence of edges that bound more than 2 cycles at the junction between the back rest and the seat. This local non-manifoldness is also one of the known limitations. On the other hand, we see that the automatic cycle detection performs very well on smooth surfaces such as the free-form roof (Architecture 2), with only 5 missed patches.

While the amount of manual input necessary to find the surfaces in the worse case seems quite high, we should keep in mind that the user only had to do about half of these actions in most cases. When the mirror is used (as is the case for the guitar and the chair), such inputs were mirrored too, making it easy for the user to add missing patches on both sides of the mirror.

4.2.2 User study

To evaluate how our system can be learnt by novice users and help them in 3D sketching, we conduct a user study among a small group of people. The goals of this study are to test the core functionalities of the system, beautification and automatic surfacing, and observe whether such a workflow helps users in creat-

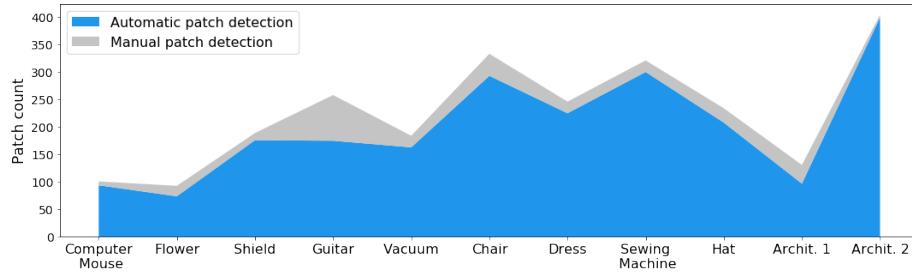


Figure 4.7: Surface patch detection performance. We measure the amount of surface patches detected automatically and manually for a variety of sketches by a user familiar with the system.

ing 3D concepts. We will compare our complete system (called *Patch* system) against two degraded versions:

- *Armature* system: without automatic surface inference (the
- *Freehand* system: without surface inference nor beautification, which is the standard 3D sketching condition.

4.2.2.1 Protocol

Participants. We recruited 4 participants (age 25-50, 1 female) for this study. Participants were professional artists (P1, P4) with prior experience in sketching for product design or researchers in Computer Graphics or HCI (P2, P3). All participants were familiar with VR, having been using it for at least a year, some participants (P1, P4) using it once a week or daily. P1, P2 and P4 also indicated being familiar with CAD or 3D modelling software, such as Blender, ZBrush and MasterpieceVR. All participants were compensated for their time.

Procedure. The study was conducted remotely, at the participant’s office or home and with their own VR hardware. We were present during the study through voice and video chat, with the possibility to see their screen and provide appropriate guidance and troubleshooting. The study lasted around 1h15 to 1h30.

One day prior to the study, we sent to all participants a short prompt explaining the task that they would have to complete. The prompt explained what objects

the participant would have to sketch and showcased a few wireframe objects from *True2Form* [56], to give an example of the sketching style that they would have to adopt.

The participants started the study by watching a 10 minutes video tutorial, where we explained what they would have to do during the study (see Appendix C to watch the video). We started by explaining the different features that each of the 3 systems (*Freehand*, *Armature* and *Patch*) present. Then we explained some study specific aspects, such as the full plan for the study and the different models that they would have to draw. Finally we gave some high-level tips to overcome known limitations of the system, and to reduce muscular fatigue. After watching the video, the participant could start the study, which is composed of a tutorial phase, then a study phase.

Each participant had a different order in which they were presented with the 3 systems, to limit effects such as learning to affect the results. This order was applied to both the tutorial phase and the study phase. Each task, both in tutorial and in study phase was to sketch a single model with one of the systems.

During the tutorial phase, the participants had to sketch a computer mouse model, with each of the 3 systems. They had the possibility to use the mirror plane, in order to draw only half of the object.

During the study phase, the participants had to sketch 2 models with one of the 3 systems (each sketch lasting about 5 minutes), then they had a break and filled-out a short form about the system they used. Then they would do the same with another one of the systems.

The 2 models we asked them to sketch are:

- A desk lamp
- A running shoe

For each of the models (mouse, lamp and shoe) we provided the participant with a rough box volume that indicates the scale and position at which they should sketch the object. This was necessary to encourage the participants to sketch at a scale that is appropriate, with respect to the beautification threshold distance.

Additionally, we provided the participants with a cheatsheet with all the controls available in the application, and displayed it in VR so that they could always refer to it, even once the study started (see Appendix D).

When all tasks were done, the participants could quit the application, send us the data collected during the study and fill-out a post-study questionnaire.

4.2.2.2 Results

We present the results obtained from the user study and provide a qualitative analysis of user's creation, by discussing the differences between the sketches created with each of the 3 systems. We also measure how well the beautification method satisfies user intent, and provide the same metric as in the previous section to evaluate automatic surfacing. Finally, we summarize qualitative feedback from the users.

Difference between sketches from different systems. We showcase the participants' creations in Figure 4.8. In this section we will discuss qualitative differences between the sketches, trying to find common points of variation between different users.

We observe that multiple users (P1, P2 and P4) seem to have adopted a very different sketching style between the *Freehand* system, and the other 2 systems. In *Freehand*, (first 2 columns Fig. 4.8), they sketch with a style that presents a lot more surface details, like small notches (P2) or details such as the laces (P1) on the shoes, small elements like springs on the desk lamp (P1). P4 goes even further and creates very dense sketches, where the strokes almost intend to form a continuous surface. To quantify this tendency, we can look at the counts of strokes sketched for each system and model by the participants (Fig. 4.9). We see that P2 and P4 draw more strokes in *Freehand*, compared to the other 2 systems, but P2 also deletes more strokes in *Freehand*, which makes the end count difference not significant. Similarly for P1, the difference is visible but not significant and the effect is absent completely for P3, which seems to sketch almost the same thing for each system. Other metrics may be more useful in exploring this difference that we observe, such as stroke length for example.

This observation could lead to the following interpretations. First, the beautification of the *Armature* system discourages users from drawing too small details, as the distance threshold is chosen to work well when sparse network of strokes are sketched, not in very dense zones of strokes. Secondly, in the *Armature* system, users are empowered to sketch well-connected networks of curves that form the armature of the object. Therefore, we hypothesize that this lifts the need for them to suggest ill-defined positions in 3D space by sketching more textured details. The precision of their neatened strokes may encourage them

to build the overall shape of the object, rather than precising the sketch with little details.

Finally, when the *Patch* system is used, this should further relax the necessity to draw as many strokes as in the other systems to define the shape of the object. We do not observe significant variations in number of strokes used however.

While we see qualitative differences in user behavior and results when using the different systems, the fixed tasks, limited time and limited number of both tasks and participants makes it difficult to obtain clear conclusions from the quantitative data that we analyzed. In future work, we plan to extend the study to more participants and analyze the data more in-depth, for example looking at how the constraints are used throughout a sketching session in *Armature* or *Patch* system, and at the timeline of interactions during a sketch.

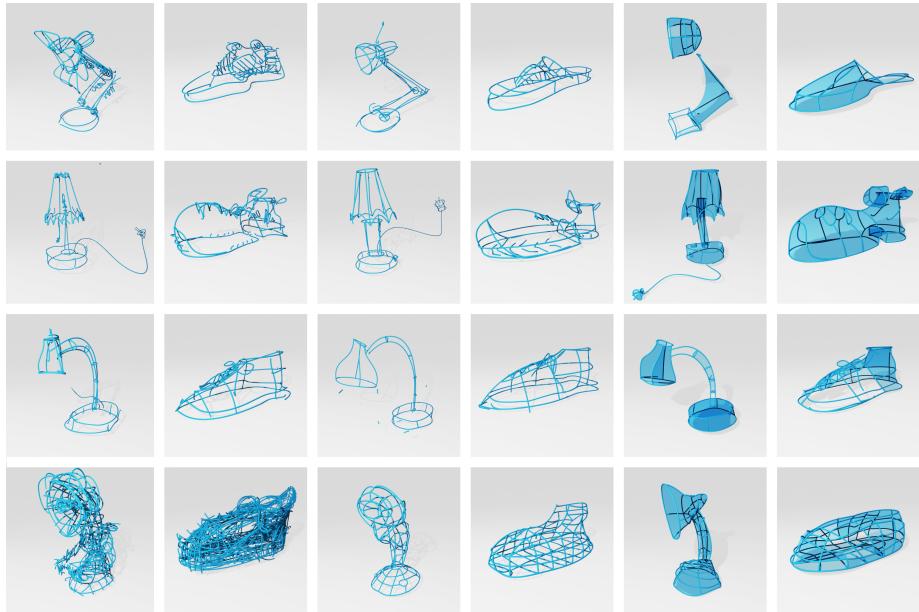


Figure 4.8: User sketches from the study. Each row is one user (P1 to P4, from top to bottom). Each 2 columns correspond to one system, in order: Freehand, Armature, Patch.

Measuring satisfaction regarding beautification results. We measure whether the participants are satisfied by the beautification results generated in the *Armature* system by measuring the relative life time of a stroke in their sketch. We compute the stroke relative life time L as a function of its creation

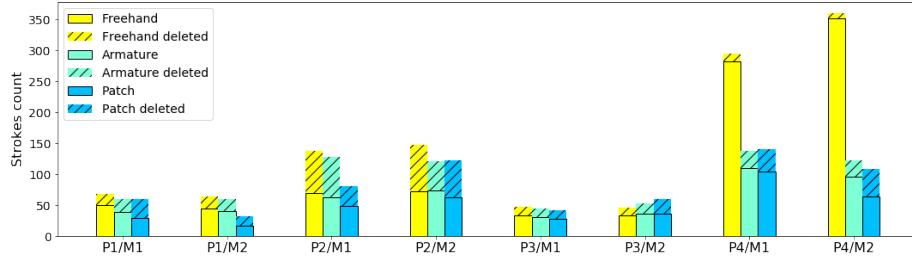


Figure 4.9: Number of strokes sketched and deleted in every system, for each participant and model. M1 is the desk lamp and M2 is the running shoe.

time t_{add} , its deletion time t_{del} , and the time at which the drawing is completed T_{sketch} :

$$L = 100 * \frac{t_{del} - t_{add}}{T_{sketch} - t_{add}} \quad (4.1)$$

A stroke that ended up not being deleted will live until the end of the sketch. It is a stroke that the user is satisfied with, and it will have $L = 100\%$. A stroke that gets deleted a very short time after its creation will have a small L , and it is a good indicator that the user is not satisfied with it.

We plot the strokes life time for the desk lamp sketch in the *Armature* system for the 4 users (Figure 4.10). We observe that for all users we have a majority of strokes that have a relative lifetime above 20% of their expected life time, which means that the user was satisfied with the result. It is not a large majority, especially in the case of P2. We think that this raises an interesting point concerning the rather stiff nature of our sketching and beautification framework, which does not allow for further editing of a stroke once it is created. It leaves the user with only the binary choice to keep or delete the entire stroke. This can be quite frustrating when only a small part of the stroke needs editing.

Surfacing performance. We measure the performance of the automatic surfacing algorithm in the same way as in Section 4.2.1.

While the number of patches that the users had to manually indicate varies, we see that it stays relatively small compared to the amount of patches detected automatically during the sketch. There are also cases where the user failed to surface a patch, even with the manual method. We can see the missing patches

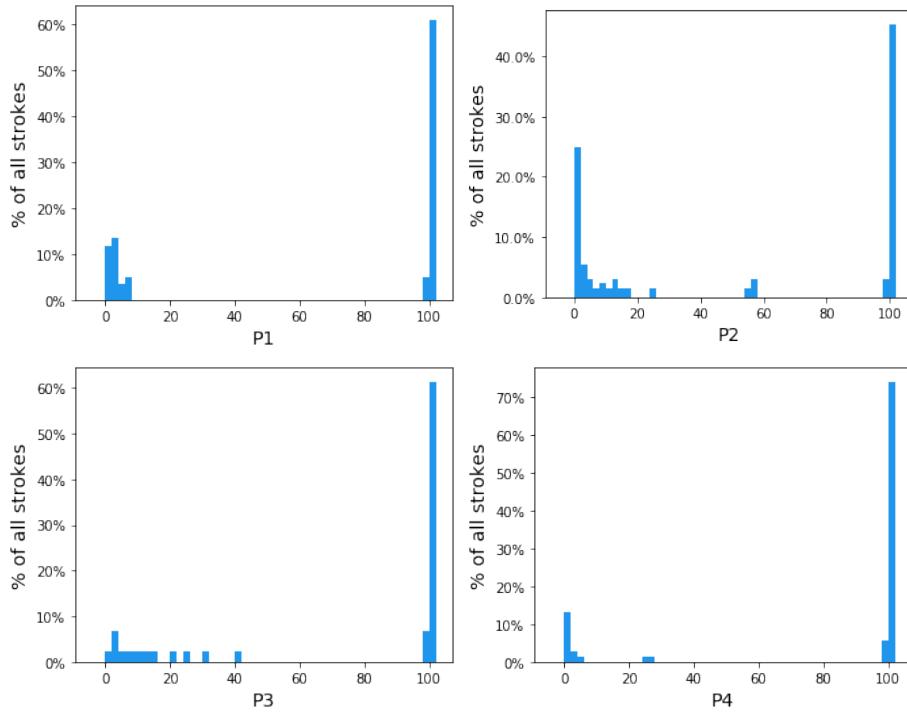


Figure 4.10: Stroke relative lifetime (as a percent of potential lifetime) for 4 users drawing the desk lamp with the Armature system. Stroke lifetime, or the time before a stroke is deleted, can show how satisfied the user is with his result. A very short lifetime indicates that the user deleted the stroke right away, because it did not satisfy them.

on some of the results (Fig. 4.8). This can happen when the curve network is not correctly connected, with some intersections missing, or when the users add a lot of strokes that define details such as the laces, contrary to the usual structural strokes that the algorithm expects (see the shoe of P3 in the last column, Fig. 4.8).

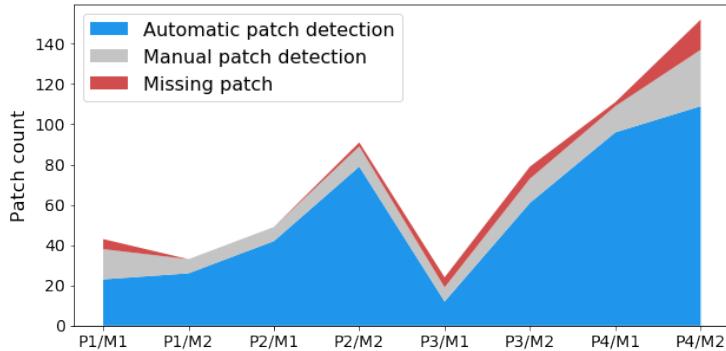


Figure 4.11: Surface patch detection performance. We measure the amount of surface patches detected automatically and manually during the sketches with *Patch* system by all users. We manually count the patches that seem to be missing from the final sketch, to account for places where both automatic and user-guided surfacing failed (or the user forgot to add a patch). M1 is the desk lamp and M2 is the running shoe.

Qualitative feedback. All participants showed great interest and enthusiasm towards the *Patch* system. They validate our hypothesis that surface patches greatly improve visualization of the 3D sketch.

P5: I think it helped in visualising the end result before I get there

P2: It allows users to define surface without many lines which can clutter a design and possibly de-emphasise important features that need line definition.

One participant expressed that the *Armature* system didn't help them in expressing an idea, as it was too constraining. This is in part because of the strong smoothing applied on strokes, as they do not support having sharp corners. This is not an issue inherent to our method, so we could remedy to it in the future.

Multiple participants expressed that they would have liked an option to disable beautification temporarily, or cancel it on a stroke, in order to avoid failure cases where they would draw strokes really close to each other without intending them to touch. One particular failure case that multiple users encountered is when trying to draw close-by parallel strokes, it is common for the strokes to be beautified such that they lie on top of each other.

One participant expressed that they would like the option to hide all or parts of the strokes that define the surface patches, as they would like less lines in the final design.

Finally, a participant stated that they would like to be able to further edit the curve network shape, by pulling on the strokes, as if the sketch was a wire sculpture. This idea is similar to what Nealen et al. [35] apply in a 2D sketching application, which was adapted to VR inputs by Verhoeven and Sorkine-Hornung [54]. It could be an interesting avenue to explore, to provide more ways for the user to edit strokes after they are sketched, rather than having to delete them and restart.

CHAPTER 5

Conclusion

In this thesis we present a complete method to create 3D sketches in VR, based on a sparse set of strokes that automatically connect to form an armature on top of which the intended surface is displayed.

The user study provides us with some qualitative evidence that automatic surfacing combined with stroke beautification provide value to users, by enabling them to focus on creating the overall form of the object rather than its details. We validate that our prototype is usable for novices, and that experienced users can achieve satisfying results on a variety of sketches.

However, the user study surfaces a number of limitations with opportunities for future improvements.

Lack of flexibility. As of now, the sketching workflow is quite stiff, as there is no way to edit or even delete some parts of a stroke, it is only possible to delete the whole stroke and redraw it. This can lead to user frustration, as it is quite annoying to have no way to correct one's small mistakes. One promising avenue of improvement that continues to leverage sketching as the main interaction could be to allow over-sketching a stroke to edit it, as in *ILoveSketch* [4].

Beautification weaknesses. We observed during the user study that the success of the beautification method is very dependent on finding good threshold values for a particular user. One solution would be to let the threshold values be customizable. However, it also seems like finding an appropriate threshold value that will work for a whole sketch is a hard task in itself. Maybe a more sophisticated approach would be to find other ways to make the thresholds dependent on context, as we did with the zoom dependent distance threshold. One idea could be to look at user sketching speed, drawing fast being associated with a more care-free gesture which we should beautify stronger than a very slow and meticulous stroke, similar to what Thiel et al. [52] proposed in 2D.

Another weakness of our beautification approach is that it does not take stroke similarity into account. We observe multiple cases where users do not wish their stroke to intersect another stroke, whereas they wish their stroke to be an offset curve from the other nearby stroke. By measuring stroke similarity, we could both detect and enforce these constraints as in *ShipShape* [16], but also forbid the beautification to make similar strokes completely overlap, as it happens sometimes currently.

Automatic surfacing. While our initial idea for automatic surfacing was very simple, we encountered many edge cases that needed to be taken care of, which led the overall algorithm to be not always reliable. Another option would've been to use an existing method that works on a complete curve network such as the cycle detection by Zhuang et al. [57]. However this may have been more difficult to optimize in order to maintain an interactive framerate. We could also have relied on more explicit user input, such as asking the user to specify when there stroke closes off a surface patch, by using a specific tool or sketching mode.

The surface update mechanism ruled by stroke addition and deletion is hard to control in a way that doesn't surprise the user. Indeed, some patches may be broken involuntarily when the user adds a stroke in a neighboring area. It is difficult to judge whether a stroke was meant to break and replace a particular patch or not. Similarly, some patches that the user explicitly deleted previously may be detected back again automatically. Making this process more conform to user expectations with regards to preserving desired patches consistently is difficult to achieve automatically. A user-guided approach could be to ask the user to "solidify" the patches with which they are happy, by marking them in the interface in some way, so that they won't be affected by further sketching.

Finally, the geometry of the surface patches could be optimized to better match the intent defined by the strokes, so that less strokes are required to define a

satisfying surface.

Drawing on surfaces. Drawing strokes projected on the surfaces is something that seemed to be of interest to our participants in the study, but that still needs work to be fully usable. A set of interesting possibilities and new questions can be raised by looking more in-depth into how the strokes that lie on the surface should behave, for example maybe a closed-loop could define holes in a surface patch. However, we also observe that it is inherently difficult for users to draw precisely on or near a surface that lies in 3D space. The best way to do so may be to use 2D input, as in *SymbiosisSketch* [3].

APPENDIX A

Beautifying line segments

A.1 Choosing which constraints to apply

When beautifying a line segment, we can't apply more than 2 intersection constraints, as the line segment does not have more degrees of freedom. Therefore in cases where more than 2 intersection constraints are detected, we must choose only 2.

We use a simple heuristic based on the intuition that users would not draw a stroke longer than it needs to be, so they wouldn't encounter constraints near the extremities of the stroke unless they intended to reach them. We always choose the 2 constraints that are closest to each of the endpoints of the line segments.

We use the angular threshold θ (Sec. 3.2.3.1) to determine whether a line segment is close enough to an orthogonal direction to be constrained to it.

A.2 Applying constraints

To constrain a line segment to one or two intersection constraints, we simply update one or both endpoints such that the line segment passes through the constraint(s).

If the constraint is close enough to one of the endpoints (within a distance $2 * \delta$, with δ from Sec. 3.2.3.1), we set the endpoint to be at the intersection constraint position.

If there are 2 intersection constraints and they are not near the endpoints, we project both endpoints on the line formed by the intersection constraints. This yields the beautified line segment.

If there is 1 intersection constraint that is not near one of the endpoints, we translate the line segment so that it passes through the constraint.

APPENDIX B

Surface patches update mechanism

In this Appendix we provide a high-level explanation of the patch update mechanism we use to automatically delete and create patches, on stroke creation or deletion.

Each stroke creation or deletion automatically triggers the necessary local cycle searches and patch deletion. To achieve this, we need to know which changes are susceptible to form a cycle, or break an existing one.

During the graph update following a user action (addition/deletion of stroke), we store in a cache:

- Segments: the segments on which a new cycle may be formed.
- Cycles: the cycles that may be broken due to a segment crossing them (see Fig. B.1a).

Adding a stroke to the sketch will trigger:

- Adding all the new segments from the stroke to the segments cache

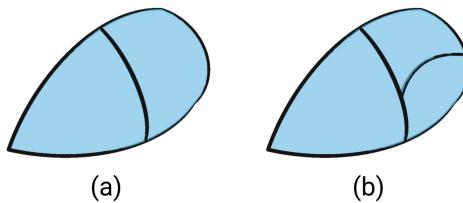


Figure B.1: (a) Before adding the stroke. (b) After adding the stroke, the patch that is crossed by the new stroke is automatically removed and replaced by two smaller patches on either side of the new stroke.

- Adding the cycles from the intersected segments to the cycle cache

Stroke deletion triggers:

- Deleting all cycles bound by this stroke
- Adding the neighboring (non-deleted) segments from the stroke to the segment cache

After the action is done, we look through the caches. We first examine each cycle, and check whether it is crossed by any new segment, in which case we delete the cycle, but add one of its segments to the segments cache to trigger a new search for cycles there. Then we examine each new segment and run a local cycle search from both of its endpoint nodes (Section 3.3.2).

After a new cycle is found, we check whether there already exists a cycle bound by the same segments, if it is the case, we do not accept the new cycle, to avoid duplicates.

This yields a list of new cycles and deleted cycles, which we can use to appropriately create or delete surface patches from the *Unity* scene.

APPENDIX C

User study video tutorial

Here we provide a link to the video tutorial that we gave to the participants before the study, to familiarize them with the system.

[Please click here to see the video.](#)

APPENDIX D

Controller cheatsheets

In this Appendix we provide the controllers mappings for each controller type supported by SteamVR. We gave these cheatsheets to the participants during the study, to help them remember the controls. The blue dot stands for the dominant hand, while the grey dot stands for the non-dominant hand. Each participant could adjust which hand was their dominant hand in the executable for the study.

The mappings of actions on the controllers are quite disputable, they were defined in a way that was most practical for quick implementation purposes where no on-screen UI or buttons were necessary, but they certainly lack usability.

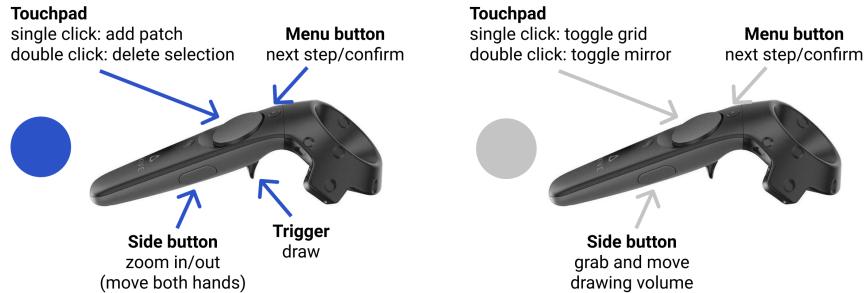


Figure D.1: HTC Vive controllers



Figure D.2: Oculus Touch controllers

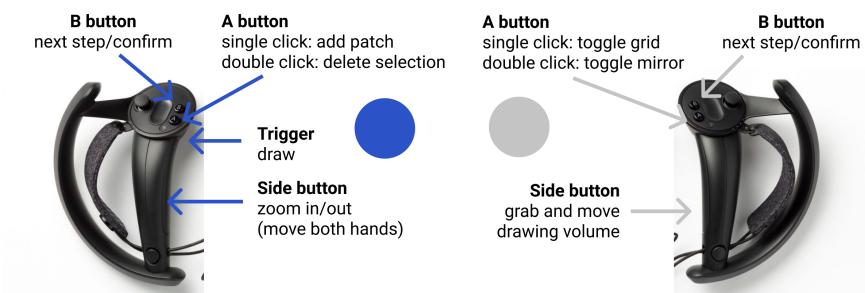


Figure D.3: Valve Index controllers

APPENDIX E

Project plan

Here we present an outline of the timeline followed during this project. We followed closely the list of different ideas that we planned for, at the beginning of the project. At the time, we did not set up a provisional planning with deadlines for each step, due to the large uncertainty we had on the feasibility of some parts, or on how long each step would take.

In retrospect, not setting up fixed dates to start on some parts such as the user study impacted on them in a way that could have been avoided, if we had stuck to a fixed plan. Indeed, we would have needed more time to run the study completely, as it is difficult to schedule participants remotely and there are many potential issues with running the study in this way.

January

We started the thesis by a 3-week project course (02507), in order to lay the technical foundations for the thesis project.

- Implemented a prototype VR sketching application in Unity.

- Experimented with simple beautification techniques, such as constraining endpoints of strokes
- Conducted a small user study to pinpoint what difficulties users would encounter while sketching in VR.

February

- Explored the existing litterature on sketch beautification and depth-inference, as well as sketching interfaces both in immersive environments and in 2D.
- Precised the scope of the project to focus on the ideas that seemed the most promising and novel: stroke beautification and later automatic surfacing of curve networks.
- Continued working on the prototype application and built the basic bricks of the constraint detection for beautification.
- Realised that enforcing constraints in a local manner without seeking to preserve input shape was not going to work.

March

- Experimented with, and finally implemented the optimisation approach for beautification. This was done by using artificial sketch data in a Python code base (in part due to inaccessibility of VR equipment during lockdown in France).

April

- Ported the beautification method from Python to the Unity C# scripts, and adapted it to the existing application.
- Informally validated the performance of the beautification method by doing a serie of sketches.
- Started to think of how to find the intended surface from the strokes, by reading previous work on the subject.

May

- Implemented the graph structure and the procedures to update the graph while the user adds strokes and intersections.
- Implemented the basic cycle detection algorithm, and discovered the main edge cases.
- Adapted the source code from Zou et al. [58] to our use case, and compiled it as a native plugin for the runtime Unity application.

June

- Added remeshing and smoothing to the native plugin, to get nicer surfaces.
- Implemented all alternative heuristics to help the cycle detection algorithm deal with a greater variety of cases.
- Implemented "drawing on surfaces".
- Implemented the user-guided surfacing interaction and logic.
- Implemented drawing volume scaling interaction, and adaptive beautification thresholds.
- Defined the protocol for the user study.
- Wrote the *Introduction* of the thesis, and created a number of explanatory figures for the *Method* section.

July

- Implemented all necessary helper features and data logging utilities for the remote user study.
- Worked on fixing bugs and improving usability.
- Ran a few small pilot studies to spot usability issues.
- Prepared the study material (video tutorial, forms).
- Ran the study remotely with 4 participants.
- Analyzed the results.
- Wrote most of the thesis.

Bibliography

- [1] Fatemeh Abbasinejad, Pushkar Joshi, and Nina Amenta. Surface patches from unorganized space curves. In *Computer Graphics Forum*, volume 30, pages 1379–1387. Wiley Online Library, 2011.
- [2] Rahul Arora, Rubaiat Habib Kazi, Fraser Anderson, Tovi Grossman, Karan Singh, and George W Fitzmaurice. Experimental evaluation of sketching on surfaces in vr. In *CHI*, volume 17, pages 5643–5654, 2017.
- [3] Rahul Arora, Rubaiat Habib Kazi, Tovi Grossman, George Fitzmaurice, and Karan Singh. Symbiosissketch: Combining 2d & 3d sketching for designing detailed 3d objects in situ. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2018.
- [4] Seok-Hyung Bae, Ravin Balakrishnan, and Karan Singh. Illovesketch: as-natural-as-possible sketching system for creating 3d curve models. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 151–160, 2008.
- [5] Seok-Hyung Bae, Ravin Balakrishnan, and Karan Singh. Everybodylovessketch: 3d sketching for a broader audience. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 59–68, 2009.
- [6] Ravin Balakrishnan, George Fitzmaurice, Gordon Kurtenbach, and William Buxton. Digital tape drawing. In *Proceedings of the 12th annual ACM symposium on User interface software and technology*, pages 161–169, 1999.

- [7] Mikhail Bessmeltsev, Caoyu Wang, Alla Sheffer, and Karan Singh. Design-driven quadrangulation of closed 3d curves. *ACM Transactions on Graphics (TOG)*, 31(6):1–11, 2012.
- [8] Bill Buxton. *Sketching user experiences: getting the design right and the right design*. Morgan kaufmann, 2010.
- [9] Carolina Cruz-Neira, Daniel J Sandin, Thomas A DeFanti, Robert V Kenyon, and John C Hart. The cave: audio visual experience automatic virtual environment. *Communications of the ACM*, 35(6):64–73, 1992.
- [10] Chris De Paoli and Karan Singh. Secondskin: sketch-based construction of layered 3d models. *ACM Transactions on Graphics (TOG)*, 34(4):1–10, 2015.
- [11] Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 317–324, 1999.
- [12] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization*, 10(2):112–122, 1973.
- [13] K Eissen and R Steur. Sketching (5th printing): Drawing techniques for product designers. *Art and Design Series. Bis*, 2008.
- [14] Facebook. Oculus medium. <https://www.oculus.com/medium/>, 2015.
- [15] Facebook. Oculus rift. <https://www.oculus.com/rift/>, 2016.
- [16] Jakub Fišer, Paul Asente, and Daniel Sýkora. Shipshape: a drawing beautification assistant. In *Proceedings of the workshop on Sketch-Based Interfaces and Modeling*, pages 49–57, 2015.
- [17] Google. Tilt brush. <https://www.tiltbrush.com>, 2016.
- [18] Google. Google blocks. <https://arvr.google.com/blocks/>, 2017.
- [19] Giorgio Gori, Alla Sheffer, Nicholas Vining, Enrique Rosales, Nathan Carr, and Tao Ju. Flowrep: Descriptive curve networks for free-form design shapes. *ACM Transactions on Graphics (TOG)*, 36(4):1–14, 2017.
- [20] GravitySketch. Gravity sketch. <https://www.gravitysketch.com/>, 2017.
- [21] Yulia Gryaditskaya, Mark Sypesteyn, Jan Willem Hoftijzer, Sylvia Pont, Fredo Durand, and Adrien Bousseau. Opensketch: A richly-annotated dataset of product design sketches. *ACM Transactions on Graphics (TOG)*, 38(6):232, 2019.

- [22] Andrew J Hanson and Hui Ma. Parallel transport approach to curve framing. *Indiana University, Techreports-TR425*, 11:3–7, 1995.
- [23] HTC. Htc vive. <https://www.vive.com>, 2016.
- [24] Emmanuel Iarussi, David Bommes, and Adrien Bousseau. Bendfields: Regularized curvature fields from rough concept sketches. *ACM Transactions on Graphics (TOG)*, 34(3):1–16, 2015.
- [25] Takeo Igarashi, Satoshi Matsuoka, Sachiko Kawachiya, and Hidehiko Tanaka. Interactive beautification: a technique for rapid geometric design. In *ACM SIGGRAPH 2007 courses*, pages 18–es. 2007.
- [26] Johann Habakuk Israel, Eva Wiese, Magdalena Mateescu, Christian Zöllner, and Rainer Stark. Investigating three-dimensional sketching for early conceptual design—results from expert discussions and user studies. *Computers & Graphics*, 33(4):462–473, 2009.
- [27] Bret Jackson and Daniel F Keefe. Lift-off: Using reference imagery and freehand sketching to create 3d models in vr. *IEEE transactions on visualization and computer graphics*, 22(4):1442–1451, 2016.
- [28] Michael Kazhdan, Jake Solomon, and Mirela Ben-Chen. Can mean-curvature flow be modified to be non-singular? In *Computer Graphics Forum*, volume 31, pages 1745–1754. Wiley Online Library, 2012.
- [29] Daniel Keefe, Robert Zeleznik, and David Laidlaw. Drawing on air: Input techniques for controlled 3d line illustration. *IEEE Transactions on Visualization and Computer Graphics*, 13(5):1067–1081, 2007.
- [30] Yongkwan Kim, Sang-Gyun An, Joon Hyub Lee, and Seok-Hyung Bae. Agile 3d sketching with air scaffolding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2018.
- [31] Wolfgang Kruger, C-A Bohn, Bernd Frohlich, Heinrich Schuth, Wolfgang Strauss, and Gerold Wesche. The responsive workbench: A virtual work environment. *Computer*, 28(7):42–48, 1995.
- [32] Chenxi Liu, Enrique Rosales, and Alla Sheffer. Strokeaggregator: Consolidating raw sketches into artist-intended curve drawings. *ACM Transactions on Graphics (TOG)*, 37(4):1–15, 2018.
- [33] Mayra D Barrera Machuca, Paul Asente, Wolfgang Stuerzlinger, Jingwan Lu, and Byungmoon Kim. Multiplanes: Assisted freehand vr sketching. In *Proceedings of the Symposium on Spatial User Interaction*, pages 36–47, 2018.

- [34] Mayra Donaji Barrera Machuca, Wolfgang Stuerzlinger, and Paul Asente. The effect of spatial ability on immersive 3d drawing. In *Proceedings of the ACM Conference on Creativity & Cognition (C&C'19)*. <https://doi.org/10.1145/3325480.3325489>, 2019.
- [35] Andrew Nealen, Takeo Igarashi, Olga Sorkine, and Marc Alexa. Fibermesh: designing freeform surfaces with 3d curves. In *ACM SIGGRAPH 2007 papers*, pages 41–es. 2007.
- [36] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.
- [37] Hao Pan, Yang Liu, Alla Sheffer, Nicholas Vining, Chang-Jian Li, and Wenping Wang. Flow aligned surfacing of curve networks. *ACM Transactions on Graphics (TOG)*, 34(4):1–10, 2015.
- [38] Daniele Panozzo, Ilya Baran, Olga Diamanti, and Olga Sorkine-Hornung. Weighted averages on surfaces. *ACM Transactions on Graphics (TOG)*, 32(4):1–12, 2013.
- [39] Theo Pavlidis and Christopher J Van Wyk. An automatic beautifier for drawings and illustrations. *ACM SIGGRAPH Computer Graphics*, 19(3):225–234, 1985.
- [40] Warren Robinett and Richard Holloway. Implementation of flying, scaling and grabbing in virtual worlds. In *Proceedings of the 1992 symposium on Interactive 3D graphics*, pages 189–192, 1992.
- [41] Emanuel Sachs, Andrew Roberts, and David Stoops. 3-draw: A tool for designing 3d shapes. *IEEE Computer Graphics and Applications*, (6):18–26, 1991.
- [42] Steven Schkolne, Michael Pruett, and Peter Schröder. Surface drawing: creating organic 3d shapes with the hand and tangible tools. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 261–268, 2001.
- [43] Johannes Schmid, Martin Sebastian Senn, Markus Gross, and Robert W Sumner. Overcoat: an implicit canvas for 3d painting. In *ACM SIGGRAPH 2011 papers*, pages 1–10. 2011.
- [44] Ryan Schmidt, Azam Khan, Karan Singh, and Gord Kurtenbach. Analytic drawing of 3d scaffolds. *ACM transactions on graphics (TOG)*, 28(5):1–10, 2009.
- [45] Philip J Schneider. An algorithm for automatically fitting digitized curves. In *Graphics gems*, pages 612–626. Academic Press Professional, Inc., 1990.

- [46] Donald Schön. The reflective practitioner. *New York*, 1083, 1938.
- [47] Cloud Shao, Adrien Bousseau, Alla Sheffer, and Karan Singh. Crossshade: shading concept sketches using cross-section curves. *ACM Transactions on Graphics (TOG)*, 31(4):1–11, 2012.
- [48] Tibor Stanko, Stefanie Hahmann, Georges-Pierre Bonneau, and Nathalie Saguin-Sprynski. Smooth interpolation of curve networks with surface normals. 2016.
- [49] Tibor Stanko, Stefanie Hahmann, Georges-Pierre Bonneau, and Nathalie Saguin-Sprynski. Shape from sensors: Curve networks on surfaces from 3d orientations. *Computers & Graphics*, 66:74–84, 2017.
- [50] Unity Technologies. Native plug-ins. <https://docs.unity3d.com/Manual/NativePlugins.html>, 2020.
- [51] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 5.0.2 edition, 2020. URL <https://doc.cgal.org/5.0.2/Manual/packages.html>.
- [52] Yannick Thiel, Karan Singh, and Ravin Balakrishnan. Elasticurves: exploiting stroke dynamics and inertia for the real-time neatening of sketched 2d curves. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 383–392, 2011.
- [53] Valve. Steamvr unity plugin. https://github.com/ValveSoftware/steamvr_unity_plugin, 2018.
- [54] Floor Verhoeven and Olga Sorkine-Hornung. Rodmesh: Two-handed 3d surface modeling in virtual reality. 2019.
- [55] Gerold Wesche and Hans-Peter Seidel. Freedrawer: a free-form sketching system on the responsive workbench. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 167–174, 2001.
- [56] Baoxuan Xu, William Chang, Alla Sheffer, Adrien Bousseau, James McCrae, and Karan Singh. True2form: 3d curve networks from 2d sketches via selective regularization. 2014.
- [57] Yixin Zhuang, Ming Zou, Nathan Carr, and Tao Ju. A general and efficient method for finding cycles in 3d curve networks. *ACM Transactions on Graphics (TOG)*, 32(6):1–10, 2013.
- [58] Ming Zou, Tao Ju, and Nathan Carr. An algorithm for triangulating multiple 3d polygons. In *Computer Graphics Forum*, volume 32, pages 157–166. Wiley Online Library, 2013.