To appear in ACM TOG 34(4).

Lillicon: Using Transient Widgets to Create Scale Variations of Icons

Gilbert Louis Bernstein*
Stanford University

Wilmot Li†
Adobe Research

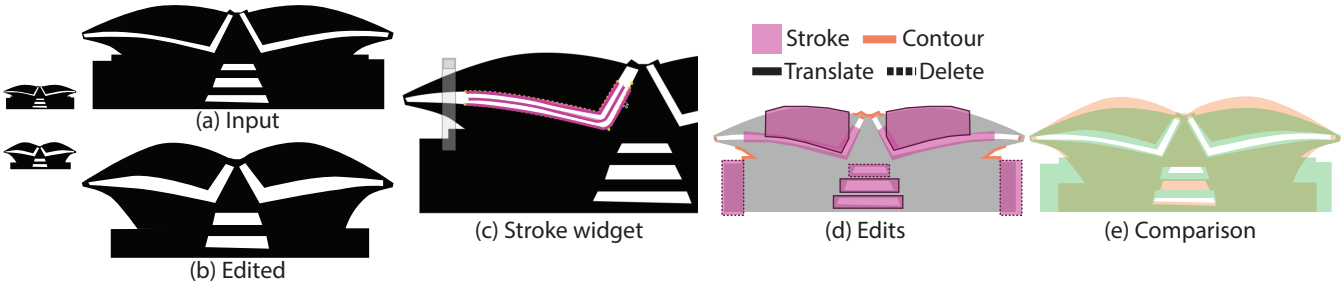


Figure 1: *Icon editing with Lillicon. Starting from an input icon of a building (a), we use our system, Lillicon, to create a scale variation (b) that is more legible at small sizes (left). With Lillicon, we apply transient widgets to select and manipulate visually apparent features, like the "stroke" in (c). Creating this result involves thickening, translating, deleting strokes and adjusting contours (d). In total, these edits took less than four minutes to perform in Lillicon. The result differs from the input in several ways (e) that improve its legibility at the target size.*

Abstract

Good icons are legible, and legible icons are scale-dependent. Experienced icon designers use a set of common strategies to create legible scale variations of icons, but executing those strategies with current tools can be challenging. In part, this is because many apparent objects, like hairlines formed by negative space, are not explicitly represented as objects in vector drawings. We present transient widgets as a mechanism for selecting and manipulating apparent objects that is independent of the underlying drawing representation. We implement transient widgets using a constraint-based editing framework; demonstrate their utility for performing the kinds of edits most common when producing scale variations of icons; and report qualitative feedback on the system from professional icon designers.

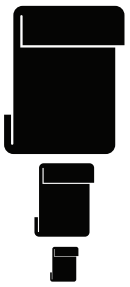
CR Categories: I.3.7 [Computer Graphics]—;

Keywords: downsampling, design tools, artist tools, vector graphics, constraint-based drawing, icons

1 Introduction

Icon design is a communication challenge. In part, this means choosing effective and appropriate symbols to convey an idea. But it also means making sure the depictions of those symbols are visually pleasing, consistent with other icons, and most of all *legible*. A significant component of this task is ensuring that all of these properties hold across the wide range of scales at which icons may be viewed. For example, an icon that represents a software application typically appears at several different sizes in different contexts — as a file icon, in toolbars, and in an online marketplace on a range of different platforms. In addition, icons may be viewed on devices with different display densities that induce different physical viewing sizes.

Simple uniform scaling of icons does not suffice to produce effective, legible icons at all viewing scales. In the inline scroll example, the hairlines that delineate the top and bottom folds of the script disappear as the icon is scaled down uniformly. A similar effect arises if you hold this paper far enough away or take off glasses you may be wearing. To prevent such problems, designers adjust the proportions, spacing, and level-of-detail of icons to produce multiple scale variations that are suited for display at different target sizes.



The need for non-uniform scale variations is fundamental, despite profound changes in display technology. Recently, the proliferation of display devices in an increasing range of sizes, aspect ratios and pixel densities has been driving designers to adopt new practices to cope with this diversity while preserving the legibility and aesthetic quality of their designs. Nonetheless, neither the switch from raster to vector formats, nor the increasing saturation of users' visual field by high-density displays will remove fundamental visual acuity limits (which vary with eyesight and viewing conditions). Nor will they eliminate the need to display icons in different parts of an application. Consequently, in carefully designed icon sets, designers will continue to hand craft scale variations to ensure legibility across all viewing situations.

While experienced icon designers are proficient at identifying the types of edits that are necessary to create effective scale variations, the process of executing such edits is often tedious. We observed two main difficulties. First, the relevant feature of the drawing that the designer wants to edit may not be conveniently exposed as a manipulable object. For example, in the script icon, the hairlines are defined as negative space outside of the filled black shape. There is no path, and thus no width parameter to edit. Second, any individual edit may require numerous secondary edits to preserve important properties of the drawing. These issues distract designers by focusing them on how to perform manipulations rather than which manipulations to perform. As a result, creating a single scale variation of one icon may require several minutes of manual editing. When creating variations for hundreds or thousands of icons used in an interface, these inefficiencies add up. For example, creating variations for an application with 500 icons at a rate of 3 minutes per edit results in 25 hours of expert designer work.

We propose *transient widgets* as a means of editing vector drawings

*e-mail: gilbert@gilbertbernstein.com
†e-mail: wilmotli@adobe.com

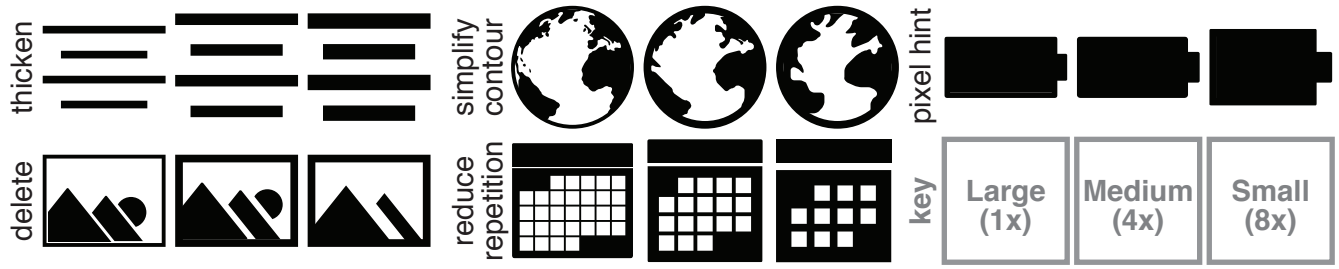


Figure 2: *Strategies used by expert designers to create scale variants in the Iconic responsive icon set.*

that addresses the challenges of creating scale variations (Figure 1). Transient widgets allow users to select and edit relevant features of drawings as if they were objects, but without worrying about the details of how the drawing is actually represented. If some portion of the drawing looks like a line or blob that the user wants to thicken or remove, then a transient widget helps them do that independently from how the drawing was constructed. In this sense, transient widgets enable users to interact with interpretations of a drawing in whichever way is most convenient for the edits they want to make. Our prototype system Lillicon implements transient widgets using a constraint-based drawing engine, providing three proof-of-concept widgets: blobs, strokes, and rectangles. Blobs and strokes in particular were chosen to capture common icon scaling strategies.

Contributions. In this paper, we identify, characterize and study the icon scaling task. We propose the concept of transient widgets, along with a means of implementation using a constraint-based editing engine. Finally, we perform an informal qualitative evaluation of our system with professional icon designers that both confirms major design decisions, and suggests that representation-independent editing tools resonate with designers.

2 The Icon Scaling Task

To better understand the task of creating icon scale variations, we conducted some formative research into (1) how icon scaling fits in the overall icon design process, (2) what strategies designers typically use to create scale variations, and (3) what obstacles arise when executing those strategies. We restrict our investigation to black-and-white icons represented as vector graphics. While many icons include color, such icons can often be expressed as colorings of underlying black-and-white designs, which suggests that our findings should generalize beyond this restriction.

2.1 Icon Design Process

We asked several professional icon designers to describe their design process. Based on these discussions, we identified three basic stages in the creation of an interface icon. First, the designer brainstorms, consults with clients, and sketches out design ideas using paper, whiteboards, and other media appropriate for rapid iteration. Once she converges on the basic design, the designer switches to computer tools (e.g. a vector graphics editor) to turn sketches into a more polished, production-ready graphic. Finally, given a high-production-quality icon, the designer prepares the different asset versions required by clients, of which scale variations are the most common. At the time of writing, an iOS App icon is required [iOS] to support 1024^2px, 180^2px, 152^2px, 120^2px, and 76^2px versions. Guidelines also recommend the preparation of 120^2px, 80^2px and 40^2px versions to display with search results, and yet more variations for use in settings, the toolbar and tab bar. These requirements are often most easily satisfied by starting with an icon at the largest desired size (e.g., 128^2px) and editing it to produce a range of smaller scale variations (e.g. 96^2px, 64^2px, 48^2px, 32^2px, 16^2px). While we present three distinct stages, in practice designers may both revisit and re-iterate earlier stages as client demands change and anticipate later stages, building icons so that scale variations are easier to produce.

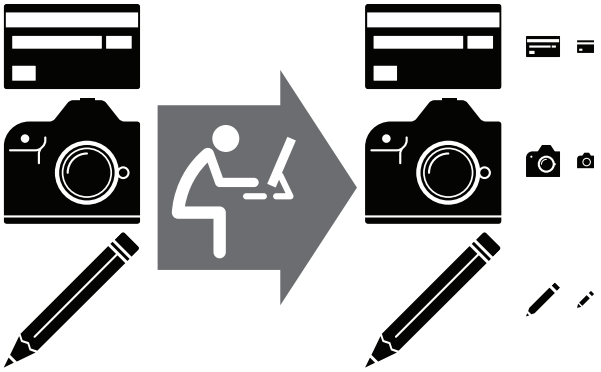


Figure 3: *The Icon Scaling Task. Given large detailed icons, produce versions suitable for display at a smaller size. (worker by Bart Laugs CC BY 3.0; all others Iconic)*

In this work, we address the third stage of the icon design process, which primarily involves editing of an existing icon design rather than creating the design from scratch. As a result, the tools we describe later in the paper focus on manipulating existing rather than drawing new geometry.

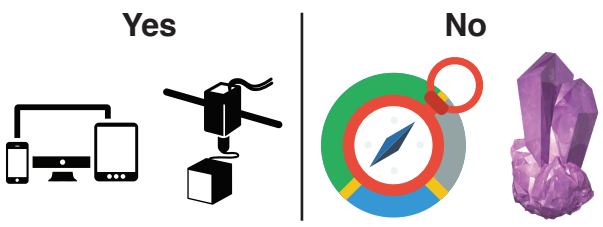


Figure 4: *To simplify our investigation and prototype, we focused on black-and-white vector icons, excluding color and raster. (3d printer by Freepik, compass by designmodo, crystal by aha-soft, all CC BY 3.0)*

2.2 Scale Variation Strategies

To identify effective strategies for scaling icons, we examined *Iconic*[Ico], a popular set of responsive vector icons. Iconic provides 3 standard sizes (lg, md, sm) for each of the 193 icons in their set. In addition to Iconic, we collected a set of around 5.5K vector icons from various online sites (dominantly flaticon.com and iconsolid.com) each of which came with only a single scale variation.

By applying zoom and downscaling operations to the icons in our dataset, we were able to observe when and how icons become illegible at small sizes. As icons shrink, several problems arise: hairlines (thin lines that are often only a pixel or two wide) and other small features (e.g., dots, stars) fade and disappear; complex silhouette and contour patterns become difficult to resolve; repeated elements blur and merge together. These scaling problems result in icons that appear shoddy/unprofessional and may cause eye strain for viewers. In some cases, the loss of a single critical hairline or important detail can have a significant impact on the recognizability of the icon.

To prevent these problems, the designers of Iconic employ a few key strategies. The most common technique is to make important hairlines and features thicker/larger to ensure that they remain visible as the icon shrinks. In conjunction, unessential hairlines and features are removed to reduce visual clutter. We observed thickening, enlarging or deleting of at least one feature in 65% of the Iconic dataset. Less common strategies include abstracting icon contours to exaggerate critical details and remove unnecessary ones (14%), and reducing the number of repetitions in icons with repeated elements so that individual elements appear larger (12%).

In addition to these adjustments, most of the smaller size Iconic icons are edited so that their contours line up as much as possible with the underlying pixel grid (71%). This type of "pixel hinting" for icons is analogous to font hinting [Shamir 2003] for type; the goal is to amplify contrast and legibility at small scales by maximizing the number of pixels that are either 100% "on" or "off." While many interface icons today are still pixel hinted, such optimizations are likely to become less important as consumers adopt high density screens, such as Apple's Retina Display, shifting focus back to fundamental visual acuity limits. Specifically, note that an average viewer with 20/20 eyesight has an angular resolution of about 1 arcminute in their fovea, which translates to a spatial resolution of approximately 300PPI when viewing a document at 12 in. The latest phone displays from Apple have a resolution of 400PPI, making it unlikely for viewers to resolve at the scale at individual pixels under normal viewing conditions. Beyond the scope of this paper, we should expect common wisdom among graphic designers to undergo heavy revision as high density displays become the de facto standard.

2.3 Creating Scale Variations with Existing Tools

To gain insight into the actual process of applying the aforementioned scale variation strategies, we attempted to manually scale down several icons using Adobe Illustrator as a representative vector graphics editing tool. Creating scale variations primarily involved editing paths using the Bezier curve (aka. *pen*) tool in conjunction with selection techniques. Occasionally the uniform scaling tool was helpful, though infrequently. Most editing consisted of dragging anchors and control points to reposition them.

In general, while identifying which editing strategies to apply was straightforward, executing those actions was tedious. That is, it was easy to point and say "that line should be widened," but it was relatively difficult to execute the desired action. Two major obstacles

Figure 5: *Visually apparent lines/features that a user may wish to edit are highlighted. Notice that these lines appear both as portions of objects (cyan) and in the negative space between other objects (magenta). (bus by Freepik CC BY 3.0)*

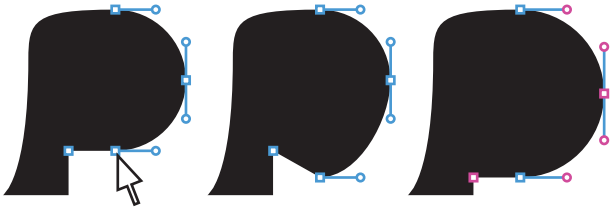


Figure 6: *Suppose a user drags the indicated control point (left) downwards. In most editors the result is to modify only that control point (middle). However, this means the user is likely going to need to drag an additional 6 control points (magenta, right) in order to restore apparent properties of the original drawing: a straight horizontal contour and semicircular contour.*

hampered us. First, a naive assumption was that we could simply select the path corresponding to a line and increase the width parameter. However, most lines in need of thickening were not represented as lines. They were simply regions between contour paths that arose in both the positive and negative space of an image. Second, as we resorted to dragging control points, we realized that each change tended to induce a series of requisite secondary edits/actions. For instance, semicircular endcaps are common, but if the base of the semicircle is widened, at least 5 control points now need individual repositioning—usually > 5 actions in order to tweak the object until it looks right again. At some point, this effort becomes great enough that reconstructing parts of the drawing from scratch starts to seem more attractive.

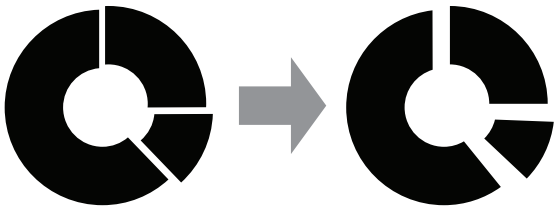


Figure 7: *As an extreme example of these principles, suppose we want to widen the gaps in this circular icon as indicated. Illustrating obstacle 1, the negative space cannot ordinarily be selected. Illustrating obstacle 2, the arc-ness of contours will not normally be preserved. (circular121 by Freepik CC BY 3.0)*

3 Transient Widgets

The idea for transient widgets comes from a subtle critique of vector/object-based drawing tools. In conventional object/vector-graphics tools like Adobe Illustrator, Microsoft PowerPoint, or Apple Keynote, drawings are represented as a composition of param-

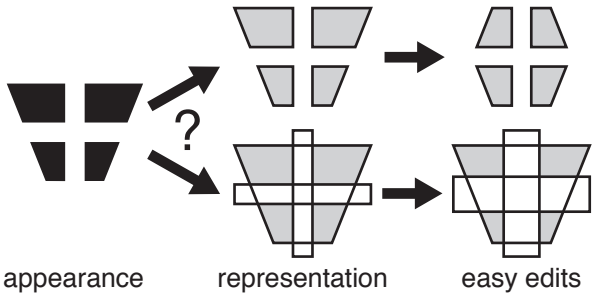


Figure 8: *Most vector/object-based editors subtly violate the WYSIWYG principle. This trapezoidal grid could be built either out of four disjoint or out of 3 layered quadrilaterals; the user cannot determine which without interacting. Depending on which representation was used, different manipulations are more readily afforded.*

eterized objects, layered in a given depth order. These parameters (e.g. Bezier control points, circle center and radius, etc.) are then exposed via sidebar sliders, textboxes or manipulators directly superimposed on the drawing. As a result, these systems implicitly identify the interpretation with the representation of a drawing. For the purposes of editing users are compelled to commit to a canonical interpretation of the drawing arising from the way they chose to construct it. Put another way, the affordances (means of control like sliders, handles, values) provided to the user are implicitly oriented towards this one canonical interpretation/representation. As illustrated in § 2.3, it is precisely the gap between a user's desire to manipulate an apparent object and the objects represented in a document that makes the execution of scaling strategies difficult.

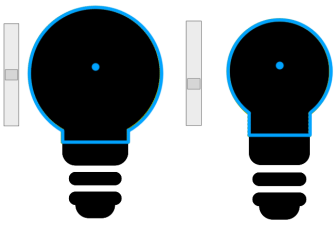
Transient widgets then, are an attempt to support an object-less editor, while also retaining and extending the benefits of object-based editing. A transient widget is temporary, rather than permanent; ancillary, rather than constitutive; an augmentation, rather than a building block; an interpretation rather than a representation. We define a **transient widget** as

> a *selection* of some portion of a drawing, which can reliably be interpreted to have (1) measurable *parameters* and (2) *methods* of being manipulated/modified.

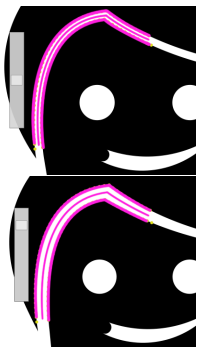
To throw this definition into relief, consider trying to apply it to standard multi-selection operations. For instance, a vector graphics editor that allows users to evenly, horizontally distribute a set of selected objects; a 3d modeling program that allows users to select arbitrary composite patches and apply free form deformations to them; visual analysis software that allows users to select a subset of data points to be re-plotted along different dimensions. All of these multi-selection tools allow users to modify selected elements, but none of them allow users to identify and manipulate selections that already exhibit specific structural characteristics. To take a specific example, suppose we want to create a transient distribution widget. The widget might interpret a selected set of elements as having an inter-element distance parameter that can be adjusted to change the horizontal spacing of the selection; depending on the design, we might do so by requiring elements to begin the manipulation evenly spaced, or by measuring the average inter-element spacing. The standard horizontal distribution tool applies an operation to arbitrary selections to make them be evenly horizontally distributed, rather than allowing manipulation according to existing, apparent structural characteristics—the core idea of transient widgets.

Transient Widgets in Lillicon

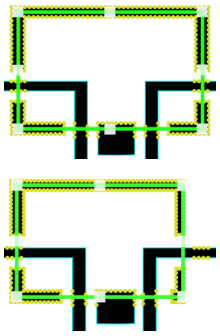
Lillicon supports three types of transient widgets, blobs, strokes and rectangles, each of which corresponds to a different interpretation of the drawing. To use a transient widget, the user selects a set of points on the contours of the drawing which separate black and white regions. Lillicon attempts to interpret the selection based on the widget type, and if successful, the system visualizes the interpretation and generates contextual manipulators in the UI. Here, we describe blob, stroke and rectangle widgets, detailing their parameters and manipulations.



Blob. A blob represents a roughly symmetric disk-like region with no particular anisotropic bias. The widget exposes a single parameter: an average **radius**. By manipulating the radius, blobs can be **scaled**. A blob can also be **deleted**, which flips the contained area from black to white or vice-versa. As shown on the right, Lillicon visualizes a blob as a highlighted disk with an indicated center point. A slider for manipulating the radius appears alongside the blob.



Stroke. A stroke represents a path-like region bounded on two sides. The "path" in question may be open or closed, straight or bent. The widget exposes an average **width** that allows the stroke to be **thickened** or **thinned**. As with blobs, a stroke can be **deleted**. As shown on the right, Lillicon visualizes a stroke as a highlighted region, with 3 paths traced over it: the two opposite sides and a middle path traveling between the two of them. A slider for manipulating the width appears beside the stroke.



Rectangle. A rectangle represents a collection of contour points that roughly approximate an axis aligned rectangle. Rectangles may possess rounded corners, gaps, or other abnormalities. The widget has four parameters: average **left**, **right**, **top**, and **bottom** coordinates. All of these parameters can be dragged as if manipulating a rectangle object in a conventional vector editing tool. As shown on the right, Lillicon visualizes rectangle widgets with a rectangular outline and 8 square handles.

4 Lillicon Implementation

Lillicon uses a constraint-based editing approach to (1) enforce an automatic set of useful constraints as the user edits a drawing, and (2) expose useful parameterizations of the underlying space of drawings. Before we discuss the implementation of transient widgets proper, we describe how this constraint engine works.

Input Format. Lillicon reads SVG files containing collections of cubic Bezier curve paths describing a region of the plane to color black (holes in regions are handled according to the standard).

If icons failed to conform to this representation, whether due to stroked paths, shape primitives or any other issue, we converted them into this format while preserving their rendered appearance. After converting all input to cubic Bezier curves, we automatically detect almost-arc curves and replace them with rational quadratic Bezier curves capable of representing arcs exactly. At this point, the icon is reduced to a set of closed contours, each consisting of a circular sequence of arc or (cubic) Bezier segments. For simplicity of exposition, we'll assume all segments are cubic Beziers in the following. Corrections for arcs are provided in the appendix.

4.1 Differential Manipulation

Differential manipulation[Gleicher and Witkin 1991a] is a technique for maintaining arbitrary (potentially non-linear) constraints on a drawing. Differential manipulation is particularly notable for framing constraint-based drawing as a constraint *maintenance*, rather than a constraint *satisfaction* problem. Numerically, this means posing a constrained integration problem, rather than a search problem. Since integration is still tractable in the presence of non-linear (e.g. geometric) constraints, this formulation of constraint-based drawing is well suited to our purposes.

We review our implementation here, but a detailed discussion may be found in Michael Gleicher's thesis[Gleicher 1994]. Let $\sigma \in \mathbb{R}^n$ be a state vector describing the drawing. Since our drawing is described by a collection of Bezier curves, σ is simply a concatenation of the 2d coordinates for each control point. Constraints are formulated as differentiable functions of the state $g_i(\sigma) = 0$. We require that a constraint can only be added if it holds for the current state. As a result, σ is always trivially a solution to $\forall i : g_i(\sigma) = 0$. In the differential manipulation framework, we instead define how to move between nearby states satisfying the constraints. That is, we define how to integrate on the manifold of admissable states.

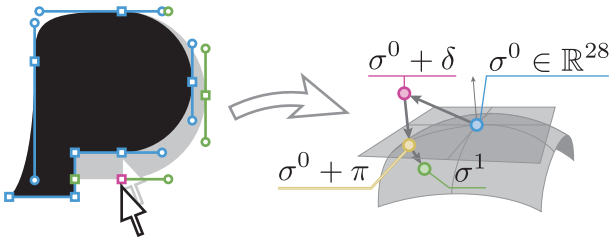


Figure 9: *A visual explanation of differential manipulation. This drawing can be encoded via coordinates for* 14 *control points as* $\sigma^0 \in \mathbb{R}^{28}$ *(blue). The user proposes an edit* δ *by dragging (red). This edit is projected into* π *in the tangent space of the constraint manifold (yellow). Finally, gradient descent finds a point on the constraint manifold* σ^1 *to use as the next drawing state (green).*

Consider a user dragging a control point from position (x_i^0, y_i^0) to position (x_i^1, y_i^1). This edit can be described with the vector $\delta = (0, \ldots, x_i^1 - x_i^0, y_i^1 - y_i^1, \ldots, 0)$. Generally, let $\delta \in \mathbb{R}^n$ be a vector based at some initial state σ^0, so that δ describes the desired direction to move in. A priori, δ may not be tangent to the manifold of admissable states (e.g., we cannot move the control point alone without violating some constraint). Consequently, we first project δ into the tangent space via a linear system solve. Let $J_{ij} = \frac{\partial}{\partial \sigma_j} g_i(\sigma^0)$ be the constraint Jacobian at σ^0. Then let $\pi \in \mathbb{R}^n$ be the solution to the underdetermined system $J\pi = 0$ that minimizes $||\delta - \pi||_2$. We solve for Lagrange multipliers via the normal equations $JJ^T\lambda = J\sigma^0$, using a conjugate gradient solver. Given the Lagrange multipliers, we construct $\pi = \delta - J^T\lambda$. Once projected, the point $\sigma^0 + \pi$ is much closer to the admissable state manifold, but probably lies some distance away due to non-linearities. To snap back onto the manifold, we apply naive gradient descent (with 50 iterations) to the energy $||g(x)||_2^2$. This solution scheme follows the original paper[Gleicher and Witkin 1991a] but could be replaced by more sophisticated integrators.

Inferred Constraints. Part of our motivation for using a constraint engine was to reduce the number of low-level secondary edits required of users. In order to do this, we automatically detect and enforce a set of standard constraints on the drawing: straight lines remain straight; vertical and horizontal straight lines remain vertical and horizontal; groups of horizontal and vertical lines at the same x or y coordinate remain grouped; and C1 and G1 continuity is enforced at anchor points between bezier curves. Details are provided in the appendix.

4.2 Extending Manipulation with Measurements

Differential manipulation allows us to edit subject to constraints, but it doesn't help us figure out how to translate actions (thicken stroke, scale blob, etc.) into edit vectors δ. To do that, we need a way to expose different useful parameterizations of the space of drawings. Gleicher developed a sophisticated automatic differentiation scheme (what we might call a domain-specific language today) called snap-together-mathematics[Gleicher and Witkin 1991b] to solve this problem. Rather than reimplement his approach, which involves constructing and maintaining arithmetic circuits/functions of the underlying state vector, we simply augment the state vector with new variables and constrain them to behave as reparameterizations of the existing variables. While our method, which we call *measure-and-reify*, does not have any more expressive power than snap-together-mathematics, we found it to be an expedient approach for linking transient widget parameters to the constraint system

A **measurement** is any differentiable function of the state vector $\phi(\sigma) \in \mathbb{R}$. Given a measurement function, we can **reify** that measurement by introducing a new variable x, initializing it to value $\phi(\sigma)$ using the current state, and adding a constraint function $g(\sigma, x) = \phi(\sigma) - x$. For example, we could introduce a measurement function that computes the distance between two points and then reify the measurement by creating a distance variable, initializing its value to the current distance, and adding the appropriate constraint function to the system. Importantly, reified measurements will never increase or decrease the underlying degrees of freedom in the system. Rather, these state augmentations just expose different parameterizations of the same underlying space. As a result, it becomes trivial to (1) apply differential manipulation to adjust the measured quantities, (2) take further measurements from existing measurements, (3) add constraints that involve measured quantities.

Polygonal Proxy. The first and most basic set of measurements we take defines polygonal proxies for the underlying contour curves. Measurements allow our state vector to link and simultaneously maintain these two redundant representations of our drawing (polygonal proxies and Bezier curves). First, we compute a uniform distribution of points along each contour curve segment. Each of these points has a constant parameter value t_i, from which the point's coordinates can be computed using the standard Bezier curve function, which is a linear interpolation of the control points. Consequently, we can measure and reify the point coordinates $(x_i, y_i) = B(P_i, t_i)$. While the purpose of the polygonal proxy is to expose a representation for the transient widgets to attach to, one immediate benefit is to allows users to edit the drawing by dragging any of these (densely) sampled contour points of the drawing.

4.3 Transient Widget Implementations

The implementation of transient widgets involves two steps. First, the system analyzes the user-specified selection to determine whether the relevant interpretation is valid. For valid selections, the system then reifies the appropriate measurements to support differential manipulation of those parameters.

Before describing the details of these two steps for each widget, we note that Lillicon allows users to form selections by dragging out rectangular axis-aligned marquees. Any point of the polygonal proxy within the rectangle is added to the selection. Holding a modifier key allows points to be added/removed from the current selection so that users can build arbitrary selections. Once a selection is formed, it becomes a widget proposal.

Blob

Interpretation. Since a blob must represent a genus 0 region of the drawing, (excluding any internal contours) interpretation starts by connecting the selected contour points together into a loop. To begin, if an entire contour loop is selected, either we're done finding a loop, or we've found multiple loops (and therefore reject the selection). If no entire contour loop has been selected, then the first step is to run the loop closure algorithm from the appendix to produce a candidate loop. Lillicon then puts this loop through a series of filters to determine whether or not the proposed loop actually represents a valid region of the drawing: that the loop does not intersect itself, intersect unselected contours, or contain any other contours within itself. Finally, since blobs are intended to be relatively isotropic, we run PCA on the selected points and reject the selection if the two eigenvalues differ by more than a ratio of $3:1$.

Manipulation. We measure the blob's center (cx, cy) as the average of all the points in the selection, and the blob's radius r as the average distance between this center and each selection point. Dragging the slider widget scales the blob by dragging this exposed radius parameter.

Deletion. To delete a blob, we first add to the drawing any segments of the selected loop that were not already edges of the polygonal proxy contours. (This may involve splitting the underlying Bezier curves.) Then, we remove all other segments of the selected loop. These operations effectively flip the blob region from black to white or vice versa. Whenever a deletion operation is performed, Lillicon re-infers constraints for the entire drawing and reconstructs a new polygonal proxy.

Stroke

Interpretation. Stroke interpretation begins the same way as blob interpretation: forming a loop, except strokes allow for up to two loops to be extracted. In the case of two loops, we check to make sure they are nested: i.e. represent a genus 1 region. Given a genus 0 or genus 1 region, Lillicon then triangulates the region using Triangle[Shewchuk 1996]. If the region is genus 0, then the dual-graph of this triangulation is guaranteed to be a tree; if genus 1, then it's guaranteed to contain exactly one cycle. If the region is genus 0, then we extract the maximum length path through the tree; if genus 1, we extract the unique cyclic path. Given a dual path through the triangulation, we can now assign two "sides" to the selection, and for each triangle define its apex vertex and base vertices. In the case of genus 0, we can also assign two ends to the selection: the two edges of the last triangle on each end of the path that don't connect to another triangle on the path. As a final filter, we check that the extents of the two ends of the stroke are within a $3:1$ ratio of each other, and likewise for the lengths of the two sides.

Manipulation. We measure the width of the stroke in two steps. First, we measure and reify the height of each triangle using the base/apex distinction. Then, we measure and reify the average height of the triangles. We call this average height the width and translate dragging on the slider into dragging this parameter of the drawing.

Deletion. Stroke deletion is identical to blob deletion.

Rectangle

Interpretation. The rectangle widget's selection mechanism is a rectangular annulus, whose thickness is controlled with a sidebar slider. This annulus partitions the points selected into 4 overlapping groups: the left points, the right points, the top points, and the bottom points. If any of the four groups are empty, the selection is rejected as a rectangle.

Manipulation. To manipulate the selected rectangle, we first compute an average x coordinate for the left and right sets of points, and an average y coordinate for the top and bottom sets of points. These coordinates are reified as four individual measurements. As the user drags the corner or side handles of the rectangle widget, we apply differential manipulation to adjust the corresponding rectangle parameter(s).

5 Qualitative Evaluation

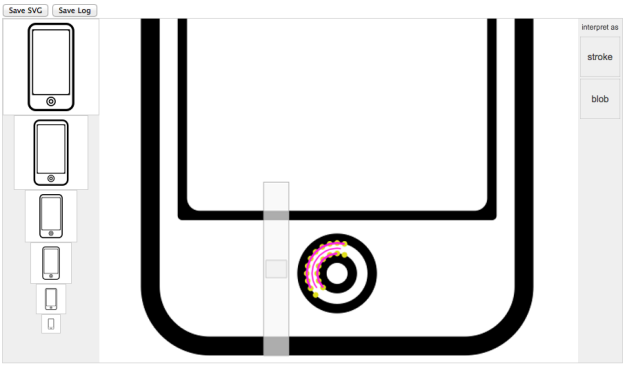


Figure 10: *The Prototype Interface. The main canvas (center) is where the user edits the icon by dragging contours, and selecting points to form transient widgets. Here, a stroke widget has been formed, exposing a slider to manipulate the width. The miniviews (left) allow the designer to instantly observe the icon at different scales. The widget-mode buttons (top right) let the designer constrain which widgets the system will try to form from a selection.*

To evaluate whether transient widgets are useful for creating scale variations, we recruited 4 professional designers (3 male, 1 female) who had no previous knowledge of our project to participate in an exploratory study. All the designers either create icons currently as part of their work, or have done so in the recent past. We brought each participant into the lab, introduced the features of Lillicon, and after a brief warmup task, we asked them to create smaller scale versions of two icons taken from Iconic (Figure 11g,h). We chose simple icons to prevent participant exhaustion. For each icon, participants first made the edit using Lillicon[1] and then performed the same edit again using their preferred vector editing tool; one designer used Adobe Fireworks, one used the vector tools in Adobe

[1]Given our small number of participants, we chose this order so that learning effects worked against Lillicon.

Photoshop, and two used Bohemian Sketch. Before each editing task, designers examined the small scale Iconic version of the icon to familiarize themselves with the desired modifications. At the end of the session, we conducted an exit interview where we asked participants to compare their experiences with Lillicon and their preferred tool.

Figure 10 shows the Lillicon interface that participants used for the study. In addition to the features described in the caption, the main canvas supports zoom and pan. The entire system also supports an unlimited undo feature. If the user does not explicitly select one of the widget-mode buttons on the top right, Lillicon tries to interprets each selection first as a stroke, and then as a blob, if the stroke is rejected. The version of Lillicon that we used for the study did not include the rectangle widget, since it was not available at the time.

5.1 Findings

Not surprisingly, we observed very different editing strategies for the two conditions. With Lillicon, all participants made extensive use of the stroke and blob widgets to adjust icon proportions. In some situations, the designers used the widgets in very consistent ways; they manipulated the circular heads of the Music Notes as blobs and the right-angled hairline near the top of the Script as a stroke. However, there was more diversity in how the designers chose to edit other parts of the drawings. For example, participants thickened the top beam and two vertical stems of the Music Notes using a variety of strategies: manipulating the parts independently as three separate transient strokes; thickening them as a single stroke; thickening the two vertical stems as stroke and then directly dragging the top and bottom contours of the beam to adjust its weight. In the Script icon, participants increased the negative space at the bottom left of the scroll either by treating the white region as a transient stroke and thickening it, or by directly dragging the contours of the black region to create more negative space.

With their preferred tool, participants primarily resorted to manipulating Bezier curve control points. As professional designers, they were all very familiar with Bezier curve representations. Still, for every task, participants had to carefully inspect the control point layout and often do some initial trial-and-error modifications to determine a specific strategy to pursue. Moreover, even though they are all clearly proficient with their preferred tools, participants appeared to have some difficulty executing certain edits using Beziers. Two of them resorted to redrawing portions of the icon rather than make secondary edits to control point handles (e.g., to preserve arcs). The other two designers used Bezier editing exclusively to complete the tasks, but they often performed a significant amount of trial-and-error editing to determine which specific subsets of control points to manipulate to achieve the desired adjustments.

There are a few common themes in the feedback from the exit interviews. The consensus was that transient widgets made it easier to perform editing operations compared to existing vector graphics tools. Three participants elaborated by saying that they liked how transient widgets enable them to treat different selected regions as either strokes or blobs without having to understand or rely on the underlying representation of the drawing. By contrast, editing with existing tools forced them to consider the specific configuration of Bezier control points or whether an apparent stroke is in fact represented with an explicit stroke primitive. On the other hand, all the designers mentioned that they liked having the option of performing the type of precise edits that they are able to achieve with their preferred existing tools. To this end, they felt Lillicon would greatly benefit from features like shortcut keys for quickly zooming the view, a pixel grid, and units of measurement for transient widget parameters (e.g., this stroke should be 5px wide). All participants felt that the editing features in Lillicon would be useful in the icon scaling process, perhaps as part of a more full-featured vector graphics tool that also supports more precise, low-level editing.

Two participants mentioned that they were sometimes surprised by how the system interpreted their selections. There were some *false positives* where designers were surprised that Lillicon formed a widget for a given selection. For instance, when selecting a set of contour points on the top of the Music Note to collectively drag, the user was surprised to see Lillicon form a stroke widget. There were also cases where the user did not explicitly select the widget-mode and Lillicon formed a stroke widget when a blob was expected. In both of these cases our widget visualizations seemed to help participants identify the interpretation error, confirming our design goal in providing visual feedback. On the other hand, participants did not receive any feedback for *false negatives* where Lillicon rejected a selection that the designer expected to form a widget. For instance, the ratio filter (§4) on stroke endcaps sometimes prevented seemingly reasonable stroke selections from forming. False negatives were a more common source of frustration than false positives.

While we did not design our study to allow for direct comparisons between the task completion times for the two conditions (the order of conditions was fixed, so learning effects for each icon may have come into play) we can report that participants were 1.5–5 times faster using Lillicon in half the tasks and roughly 1.2 times faster using their preferred existing tool in the other half, with median completion times of 134 seconds for Lillicon and 210 seconds for the existing tool. We find these completion time results encouraging for two reasons. First, Lillicon lacks many common features like keyboard shortcuts and text entry fields for setting parameters that can have a significant impact on user efficiency for editing tasks. Moreover, there was a massive gap between the two conditions in terms of familiarity; while participants were asked to use Lillicon for the first time, they all have an extremely high level of expertise with their preferred tools.

6 Results

In addition to soliciting feedback from icon designers, we used Lillicon to generate scale variations of several icons from our dataset to get a sense for the generality and limitations of our approach. Figures 1 and 11 show some of our results, and our submission video includes editing sessions for additional icons. Creating these scale variations involved 2–10 transient widgets and a few contour dragging operations. The edits for each result took between 1 and 5 minutes to perform.

Overall, we found transient widgets especially useful for editing icons with compound shapes composed of several apparent parts. For example, the Music Notes (Figure 11g) are composed of two circular blobs at the ends of two vertical stems connected by a curved bar across the top, and the Car (Figure 11d) has a thin hairline around the windshield that merges into the body. Since such parts are not typically represented as separate, discrete objects, they are hard to manipulate independently with existing tools. With transient widgets, we can select and edit each part separately to adjust their relative proportions.

Transient widgets are also very helpful for manipulating negative space. In several of the icons in Figure 11, the gaps between object contours form visually apparent strokes and blobs that are not represented as objects. Modifying such gaps with existing tools requires a significant amount of low-level editing of the contours themselves. In contrast, transient widgets allow us to select negative hairlines and blobs and then modify their parameters.

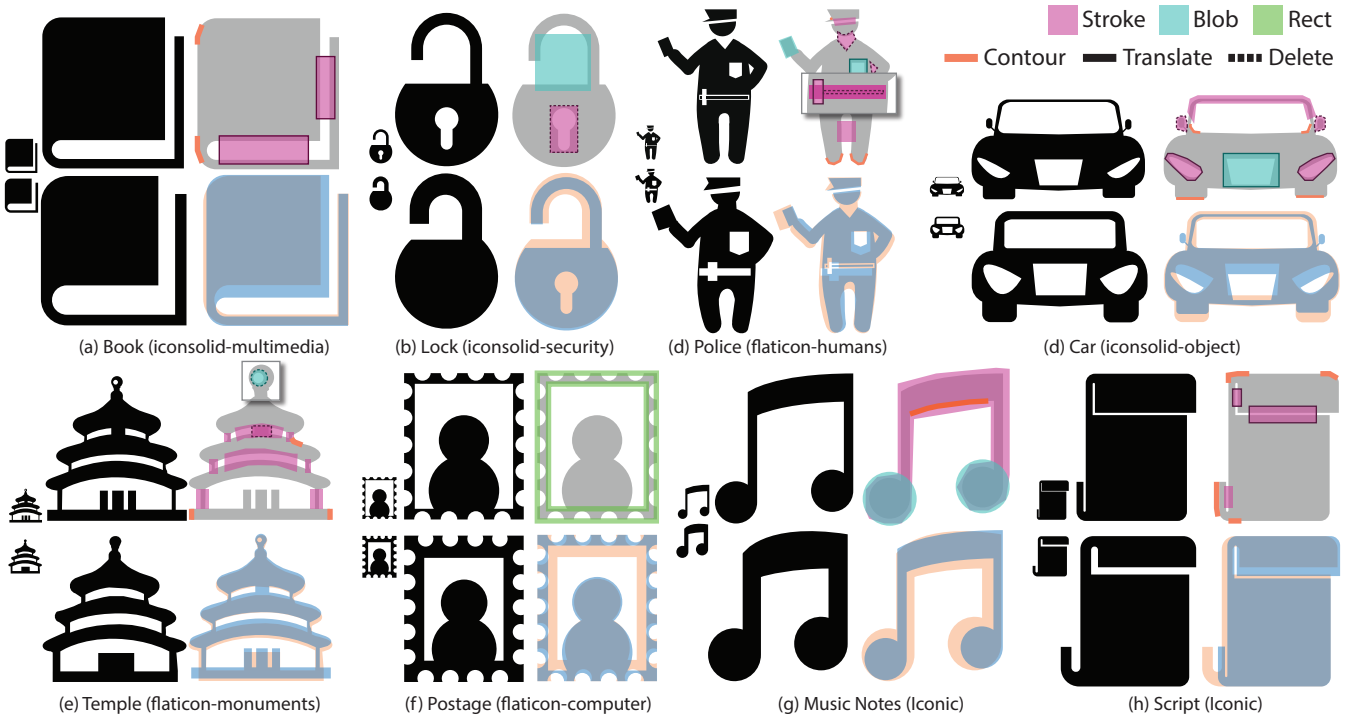


Figure 11: *Results generated with Lillicon. These eight icons were edited using Lillicon to produce scale variations suitable for display at a small size. (Music Note and Script were used in our study.) For each result, the first two columns show the original (top) and edited (bottom) icon at the target size and enlarged. In the top right is a visualization of which Lillicon editing tools were applied. As noted in the legend, the colors indicate the type of widget, and the border of each colored region indicates how the widget was used. If the widget has no border or a solid border then the widget parameters were also adjusted. We use orange lines to indicate where contours were directly dragged. Finally, the bottom right image superimposes the input and output drawings to make the differences easier to see.*

While we mainly used the stroke and blob widgets, the rectangle widget was useful for manipulating "irregular" rectangular regions, such as the notched frame of the Postage icon (Figure 11f). Here, we applied two rectangle widgets: one to shrink the inner boundary and one to expand the outer boundary. As we expand the outer boundary, the notches retain their semi-circular shape.

7 Related Work

7.1 Icons and Scaling

While most classical work on image scaling (i.e. signal-processing theory) tries to preserve proportionality across different scales, there is also a substantial amount of work on scale-sensitive variation. For instance, retargeting methods (i.e. changing aspect-ratio) have been proposed for image processing[Rubinstein et al. 2010; Huang et al. 2009] and even 3d geometry[Kraevoy et al. 2008]. While much of this body of work is less applicable to the problem of uniform rescaling, it does emphasize the importance of preserving certain features and/or proprtions. For instance, Setlur et al.[2005] created retargetable vector animations by relying on the hierchical structure of a scene graph/SVG file, along with user-provided importance annotations. Content-adaptive image downscaling[Kopf et al. 2013] provides an automated method for uniform downscaling, but was expressly not designed to handle legibility objectives like font hinting.

The body of work on automatic font-hinting[Hersch and Betrisey 1991; Zongker et al. 2000; Shamir 2003] is probably closest in spirit to our focus on the legibility of small scale images. In contrast to the font hinting line of work, we aim to provide interactive tools that help designers create scale variations. While automation may ultimately be possible, our investigation suggests that experienced designers make many subjective judgements (based on their expertise) to create high quality small-scale images. Moreover, many of the designers we spoke with preferred to retain authorial control over the process.

Most academic work on icons focuses on which qualities make icons effective, rather than the design process itself. For instance, Kineticons[Harrison et al. 2011] explores the design space of animated icons. A smaller set of work also focuses on the autogeneration of memorable[Lewis et al. 2004] or semantically appropriate[Setlur and Mackinlay 2014] icons.

7.2 Transient Widget Precursors

Transient widgets are hardly the first attempt to resolve representational issues in drawing and modeling tools. Skeletal Strokes [Hsu et al. 1993; Hsu and Lee 1994] described an authoring paradigm where stroke objects are used as a 2d rigging system for creating drawings and animations. Unlike Skeletal Strokes, which are a prescription for how to build and represent drawings, transient stroke widgets are a way to interpret parts of already existing drawings. Reverse engineering methods like GlobFit[Li et al. 2011] do work with existing objects, decomposing shapes into Boolean combinations of parametric primitives. However because they must infer one "true" representation, these methods are ill-posed for accomodating the multiple interpretations a user may form. Lillicon can be seen as supporting a specialized form of a planar map represen-

tation[Baudelaire and Gangnet 1989; Asente et al. 2007; Dalstein et al. 2014], which are also focused on resolving subtle violations of the WYSIWYG principle. Transient widgets are a natural means of recovering objects that remains consistent with the philosophy of this approach.

Transient widgets are reminiscent of Eric Saund's work on perceptually-based editing of free-form sketches[Saund and Moran 1994; Saund et al. 2003], which proposed a what-you-perceive-is-what-you-get tool for pen-based computing. Saund also abandoned the traditional object/vector-based representation of drawings, but was primarily focused on issues of gestalt perception; so his work was focused on selection and grouping mechanisms. Transient widgets are instead (a) primarily about manipulating "objects" and (b) defined by use rather than perception; if you can edit it like a stroke, then it's a stroke as far as we care. However, Saund's work and Lazy Select[Xu et al. 2012] could be very useful for improving Lillicons selection mechanisms.

7.3 Constraint-based Editing

In the last decade, geometry researchers explored a wide variety of deformation criteria[Botsch and Sorkine 2008], but mostly did not address the problem of editing with constraints. iWires[Gal et al. 2009] is the notable exception. Like iWires, we found that a small set of easy to detect features yielded a large benefit during editing. However, we chose to use Gleicher's systematic approach rather than the ad-hoc solver described in iWires.

There is a long history of constraint-based editing going back to Sutherland's Sketchpad [Sutherland 1964]. We directly leverage Gleicher's thesis work[Gleicher 1994], which was used to build the constraint-driven object/vector-based drawing system Briar[Gleicher 1992]. Though Lillicon is built on the same technical framework, it explores a different space of editing ideas and issues. Briar focused on constraint-maintenance and snap-dragging interactions, while Lillicon hides constraint specification and maintenance from the user. Briar retains the layered primitive parametric object representation, while Lillicon relies on transient widgets. Finally, we note that while we only rely on the most basic integration scheme proposed by Gleicher[1991a], he later proposed more sophisticated numerical techniques, including support for inequality constraints.

8 Limitations

The designers who evaluated Lillicon did not feel comfortable giving up access to the existing, precise, low-level tools that they were accustomed to (e.g., Bezier curve tools). While it was not practical for us to implement all such features in Lillicon, our system design approach of using multiple constraint-linked representations makes it possible to integrate transient widgets into existing, fully-featured vector graphics tools.

Designers also complained about unexpected behavior when trying to form widgets. We could probably improve our widget interpretation algorithms to better anticipate the user's intention. However, the results of our study indicate that providing better visual feedback is a more effective and necessary mechanism for improving predictability and reducing frustration.

Finally, in our constraint engine, we use an L_2 metric over the entire state vector, including measured parameters, when updating the drawing. This can sometimes lead to unintuitive behavior where portions of the drawing shift in unexpected ways to minimize energy associated with invisible parameters. One solution is to impose the metric only on the part of the state corresponding to the polygonal proxy. However, rather than further mimicing Gleicher's original work, we believe there is an interesting opportunity to merge constraint-based editing with the last decade of work on surface deformation energy models. Perhaps an L_1 or harmonic metric would result in better constrained editing behavior.

9 Conclusion & Future Work

In this research, we conducted a detailed investigation of an important design task: creating icon scale variations. By framing this problem as a question of designer augmentation rather than replacement, we arrived at the idea of *transient widgets* as a flexible mechanism that enables common scaling operations. Building on this foundation, we see a number of promising avenues for future work.

One immediate opportunity is to consider smarter selection mechanisms for forming transient widgets. As discussed in §8, better widget interpretation algorithms and visual feedback would be helpful. However, extending Lillicon to support more complicated selections of multiple widgets at once may have an even bigger impact.

Another interesting question to consider is whether we can further abstract and automate the icon scaling task. Even if the results aren't perfect, functionality to automatically produce scale variations of an icon and/or an entire icon set would be very useful. However, extending the spirit of the approach we took to that level of task abstraction is far from trivial. How do we keep the designer in control? How do we deal with the inevitable failure of automatic mechanisms? How can the user understand how and why the system behaves the way it does?

Finally, while we chose to focus on the icon scaling stage of icon design, there are also significant opportunities to help designers during earlier stages of icon design, or to apply these tools to other tasks. In those contexts tools oriented towards exploring unexpected variations on a candidate design could be tremendously useful. Investigating this new set of task requirements would likely yield further insights. Meanwhile, trying to extend a tool like Lillicon to support such constructive and exploratory tasks seems likely to produce further exciting tools for object creation and manipulation.

Acknowledgements

We would like to thank Isabelle Landthaler and Shawn Cheris for explaining various aspects of the icon design workflow, Nathan Carr, Radomír Měch and Jovan Popović for helpful technical discussions, our study participants, and the anonymous reviewers for their thoughtful feedback and suggestions that improved this paper.

References

ASENTE, P., SCHUSTER, M., AND PETTIT, T. 2007. Dynamic planar map illustration. *ACM Trans. Graph. 26*, 3 (July).

BAUDELAIRE, P., AND GANGNET, M. 1989. Planar maps: An interaction paradigm for graphic design. *SIGCHI Bull. 20*, SI (Mar.), 313–318.

BOTSCH, M., AND SORKINE, O. 2008. On linear variational surface deformation methods. *IEEE Transactions on Visualization and Computer Graphics 14*, 1 (Jan.), 213–230.

DALSTEIN, B., RONFARD, R., AND VAN DE PANNE, M. 2014. Vector graphics complexes. *ACM Trans. Graph. 33*, 4 (July), 133:1–133:12.

GAL, R., SORKINE, O., MITRA, N. J., AND COHEN-OR, D. 2009. iwires: An analyze-and-edit approach to shape manipulation. *ACM Trans. Graph. 28*, 3 (July), 33:1–33:10.

GLEICHER, M., AND WITKIN, A. 1991. Differential manipulation. In *Proceedings of Graphics Interface '91*, 61–67.

GLEICHER, M., AND WITKIN, A. 1991. Snap together mathematics. In *Advances in Object-Oriented Graphics I*, E. Blake and P. Wisskirchen, Eds., EurographicSeminars. Springer Berlin Heidelberg, 21–34.

GLEICHER, M. 1992. Briar: A constraint-based drawing program. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, New York, NY, USA, CHI '92, 661–662.

GLEICHER, M. 1994. *A Differential Approach to Graphical Interaction*. PhD thesis, Carnegie Mellon University.

HARRISON, C., HSIEH, G., WILLIS, K. D., FORLIZZI, J., AND HUDSON, S. E. 2011. Kineticons: Using iconographic motion in graphical user interface design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, New York, NY, USA, CHI '11, 1999–2008.

HERSCH, R. D., AND BETRISEY, C. 1991. Model-based matching and hinting of fonts. *SIGGRAPH Comput. Graph. 25*, 4 (July), 71–80.

HSU, S. C., AND LEE, I. H. H. 1994. Drawing and animation using skeletal strokes. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '94, 109–118.

HSU, S. C., LEE, I. H. H., AND WISEMAN, N. E. 1993. Skeletal strokes. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology*, ACM, New York, NY, USA, UIST '93, 197–206.

HUANG, Q.-X., MECH, R., AND CARR, N. A. 2009. Optimizing structure preserving embedded deformation for resizing images and vector art. *Computer Graphics Forum 28*, 7, 1887–1896.

Iconic. `http://useiconic.com/`. Accessed: 2015-01-12.

iOS human interface guidelines. `https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/IconMatrix.html`. Accessed: 2015-01-12.

KOPF, J., SHAMIR, A., AND PEERS, P. 2013. Content-adaptive image downscaling. *ACM Trans. Graph. 32*, 6 (Nov.), 173:1–173:8.

KRAEVOY, V., SHEFFER, A., SHAMIR, A., AND COHEN-OR, D. 2008. Non-homogeneous resizing of complex models. *ACM Trans. Graph. 27*, 5 (Dec.), 111:1–111:9.

LEWIS, J. P., ROSENHOLTZ, R., FONG, N., AND NEUMANN, U. 2004. Visualids: Automatic distinctive icons for desktop interfaces. *ACM Trans. Graph. 23*, 3 (Aug.), 416–423.

LI, Y., WU, X., CHRYSATHOU, Y., SHARF, A., COHEN-OR, D., AND MITRA, N. J. 2011. Globfit: Consistently fitting primitives by discovering global relations. *ACM Trans. Graph. 30*, 4 (July), 52:1–52:12.

RUBINSTEIN, M., GUTIERREZ, D., SORKINE, O., AND SHAMIR, A. 2010. A comparative study of image retargeting. *ACM Trans. Graph. 29*, 6 (Dec.), 160:1–160:10.

SAUND, E., AND MORAN, T. P. 1994. A perceptually-supported sketch editor. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology*, ACM, New York, NY, USA, UIST '94, 175–184.

SAUND, E., FLEET, D., LARNER, D., AND MAHONEY, J. 2003. Perceptually-supported image editing of text and graphics. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology*, ACM, New York, NY, USA, UIST '03, 183–192.

SETLUR, V., AND MACKINLAY, J. D. 2014. Automatic generation of semantic icon encodings for visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, New York, NY, USA, CHI '14, 541–550.

SETLUR, V., XU, Y., CHEN, X., AND GOOCH, B. 2005. Retargeting vector animation for small displays. In *Proceedings of the 4th International Conference on Mobile and Ubiquitous Multimedia*, ACM, New York, NY, USA, MUM '05, 69–77.

SHAMIR, A. 2003. Constraint-based approach for automatic hinting of digital typefaces. *ACM Trans. Graph. 22*, 2 (Apr.), 131–151.

SHEWCHUK, J. R. 1996. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *Selected papers from the Workshop on Applied Computational Geormetry, Towards Geometric Engineering*, Springer-Verlag, London, UK, UK, FCRC '96/WACG '96, 203–222.

SUTHERLAND, I. E. 1964. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE Design Automation Workshop*, ACM, New York, NY, USA, DAC '64, 6.329–6.346.

XU, P., FU, H., AU, O. K.-C., AND TAI, C.-L. 2012. Lazy selection: A scribble-based tool for smart shape elements selection. *ACM Trans. Graph. 31*, 6 (Nov.), 142:1–142:9.

ZONGKER, D. E., WADE, G., AND SALESIN, D. H. 2000. Example-based hinting of true type fonts. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '00, 411–416.

A Arcs and Constraints

Arcs via rational quadratic Bezier curves. Let P_0, P_1, P_2 be the three control points of a rational quadratic Bezier curve and $w_0 = 1, w_1, w_2 = 1$ be the three weights on those control points. For reference, a point at parameter value t is given by the formula $A(t) = \sum_i \binom{2}{i} \frac{w_i}{4} P_i / \sum_i \binom{2}{i} \frac{w_i}{4}$. Aside from constraining w_0 and w_2, further enforce the constraint that P_1 always lies on the bisector of P_0 and P_2. Now, given the 3 control points, and the constraint that the curve must be an arc, it's known that w_1 is uniquely determined to be $\sin \frac{\theta}{2}$ where $\theta = \angle P_0 P_1 P_2$.

When adding arcs to the state vector, w_1 is added along with the control point positions. Two special constraints are added for each arc segment. One of these is a bisector constraint. The other enforces the value of w_1, and can be derived from the sin equality.

$$\begin{aligned} \text{bisect}(P_i) &= \langle P_1 - P_0, P_2 - P_0 \rangle - \langle P_2 - P_1, P_2 - P_0 \rangle \\ \text{arcw}(P_i, w_1) &= ||P_2 - P_0|| - w_1(||P_2 - P_1|| + ||P_0 - P_1||) \end{aligned}$$

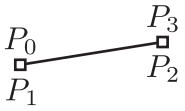
When tangent points of an arc are needed to determine geometric continuity at an anchor, P_1 is always used.

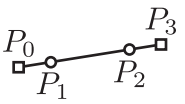
Constraints. Define the following constraint functions:

$$\begin{aligned} \text{colinear}(A,B,C) &= \det \begin{vmatrix} A_x - B_x & C_x - B_x \\ A_y - B_y & C_y - B_y \end{vmatrix} \\ \text{eq}(a,b) &= a - b \\ \text{eqpt}(A,B) &= A - B \\ \text{midpt}(A,B,C) &= 2B - A - C \end{aligned}$$

Constraints are applied to the underlying drawing using the following heuristics

No Handle. For each cubic Bezier, if $P_0 = P_1$ then $\text{eqpt}(P_0, P_1)$ is enforced; likewise for P_2 and P_3.



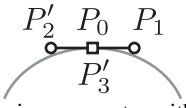


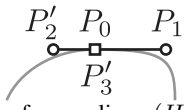
Straight Line. Cubic Beziers that represent straight lines are kept straight. Specifically $\text{colinear}(P_0, P_1, P_3)$ and $\text{colinear}(P_0, P_2, P_3)$ are enforced.

Horizontal & Vertical. If a straight line is horizontal, then $\text{eq}(P_{0y}, P_{3y})$; similarly for vertical lines.

Horizontal & Vertical Groups. If multiple horizontal lines have the same y-value, then their y-values are constrained to be equal; similarly for vertical lines.

C1 Continuity. If two successive cubic Beziers in a path have symmetric handles around an anchor point, maintain the symmetry using $\text{midpt}(P_2', P_3' = P_0, P_1)$, where P_i' and P_i are the control points of successive Bezier segments, with shared anchor $P_3' = P_0$.





G1 Continuity. If C1 continuity does not hold at a shared anchor point, but the handles on either side $H_{in} = P_2'$ and $H_{out} = P_1$ are colinear with the anchor $A = P_3' = P_0$, then enforce $\text{colinear}(H_{in}, A, H_{out})$. If either curve is a straight line redefine the associated handle as $H_{in} = P_0'$ or $H_{out} = P_3$; if it's an arc, then $H_{in} = P_1'$, $H_{out} = P_1$.

Contiguous Curvature. Each arc in the drawing is fit with a circle. If an arc is found to lie on its neighbor's circle, then we assume the two arcs form one contiguous arc of a common circle. In this case, we estimate the common circle's center and radius, add those 3 variables to the state vector and constrain the polygonization samples from the arcs to lie on the newly represented circle. This leads to redundant constraints, which the solver handles fine. Note that it is very important that arcs be represented exactly as rational quadratic Bezier curves in order for adding this constraint to be safe.

B Ad-hoc Loop Closure Algorithm

Input/output. The input is a set of "runs", each of which is an ordered list of points. Call the first point in a run the **tail** and the last point the **head**. The output is a single cyclic sequence ordering all of the input runs.

Definitions. We call any cyclically ordered sequence of runs a **loop**, with the interpretation that the head of each run is connected to the tail of the subsequent run. We call the extra line segment introduced by this connection a **gap**. The length of each gap is measured using Euclidean distance, and the **cost** of the loop is the sum of its gap lengths.

Algorithm. First, convert each input run into a trivial loop by connecting the head of the run to its own tail. Then, we repeat the following until only one loop remains: find the pair of loops with a minimal cost **splice**, and perform that splice. A splice is specified by identifying a gap in each of the loops being spliced, cutting the loops at those gaps and reconnecting them into one loop. (Because loops are directed, the operation is unambiguous once gaps have been specified.) The cost of a splice is the cost of the resulting loop minus the sum cost of the two original loops. Caching is used to save wasted computation in the search for a minimal cost splice.

Loop Filtering

Loop Closure produces a cyclic sequence, but doesn't guarantee non-intersection or other desirable properties. We filter the output for desirable properties and fail to form transient widgets when these filters are not passed.

No Isolated Points. All runs fed into loop closure must have at least two points in them. (This simplifies our deletion code by preventing the introduction of non-manifold points in our curve representation)

No Self Intersections. If the loop intersects itself, then reject it.

No Contour Intersections. If the loop output crosses an edge of the drawing's polygonization that wasn't part of a run (i.e. if any of the new gap edges cross an edge of the polygonal proxy) then reject the loop.

Two loops max. Only allow two loops maximum, including selected closed loops and the loop produced by the preceding algorithm.

One loop forms a disk. If there is one loop, then that loop should not enclose any other edges from the polygonization of the drawing.

Two loops form an annulus. If there are two loops, then one of the loops should be enclosed by the other, and any edges from the polygonization of the drawing should either be enclosed in both loops or enclosed in neither. (i.e. the area between the two loops doesn't contain anything from the polygonization)

Warning: This loop closure algorithm produced the obvious, desired closure in all situations we observed, so long as there was an obvious way to close a sequence of runs into a loop. In other cases, the filters ensured safety for downstream processing. However, this algorithm was developed to get our prototype working, not because it was particularly elegant or has any kind of robustness guarantees. Future implementers would likely do just as well with another ad-hoc method of their own.