Global Beautification of Layouts with Interactive Ambiguity Resolution

Pengfei Xu[1] **Hongbo Fu**[2] **Takeo Igarashi**[3] **Chiew-Lan Tai**[1]
[1]HKUST [2]City University of Hong Kong [3]The University of Tokyo

ABSTRACT

Automatic global beautification methods have been proposed for sketch-based interfaces, but they can lead to undesired results due to ambiguity in the user's input. To facilitate ambiguity resolution in layout beautification, we present a novel user interface for visualizing and editing inferred relationships. First, our interface provides a preview of the beautified layout with inferred constraints, without directly modifying the input layout. In this way, the user can easily keep refining beautification results by interactively repositioning and/or resizing elements in the input layout. Second, we present a gestural interface for editing automatically inferred constraints by directly interacting with the visualized constraints via simple gestures. Our efficient implementation of the beautification system provides the user instant feedback. Our user studies validate that our tool is capable of creating, editing and refining layouts of graphic elements and is significantly faster than the standard snap-dragging and command-based alignment tools.

Author Keywords

Global beautification; layout editing; snapping; alignment; ambiguity resolution; gestural interface

ACM Classification Keywords

H.5.2 Information Interfaces and Presentation (e.g. HCI): User Interfaces – Graphical user interfaces (GUI)

INTRODUCTION

Specifying precise relationships, such as alignment and equal-spacing between graphic elements, might be one of the most fundamental operations when creating or editing diagrams and other graphical documents. This is commonly achieved in commercial software packages, like Adobe Illustrator and Microsoft PowerPoint, by using command-based arrangement tools (e.g., issuing a command to equally space the selected elements horizontally) and/or direct positioning aided by snapping.

Snapping might be the simplest beautification technique. It first infers spatial relationships between an element being manipulated and each of the existing elements, and then provides snapping suggestions to interactively achieve desired relationships [5]. We classify snapping as a *local beautification* technique, since each time only a current element is being beautified while all existing elements are kept fixed, leading to an *element-by-element* beautification process. Due to its local nature, snapping itself is not very effective for designing constrained global patterns (e.g., equal-spacing patterns with the ending elements aligned, as shown in Figure 1). Therefore, snapping-based alignment tools are often used together with command-based tools. As we will discuss shortly, such traditional tools are rather tedious and require a carefully ordered set of manual operations to achieve a desired layout.

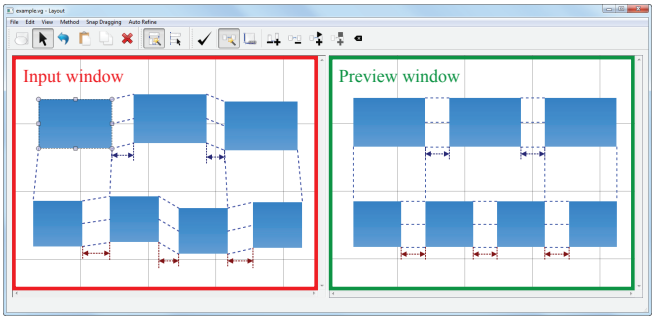

Figure 1. Our novel interface for global beautification of layouts of graphic elements with the power of interactive ambiguity resolution. Grids are visualized for reference purpose only but not for editing.

On the other hand, humans are able to unambiguously tell the desired layouts of graphic elements by viewing all elements as a whole. This motivates us to design a tool for *global beautification* of layouts of graphic elements, i.e., to first infer perceptually meaningful relationships among a set of roughly placed graphic elements and then refine their positions and sizes to get a well-aligned layout that involves only small changes to the input layout. While similar concepts of global beautification have been proposed for sketch-based user interfaces, the existing methods (e.g., [12, 22]) simply apply global beautification results directly to elements being edited, as often done for local beautification interfaces. Early beautification of elements being edited would prevent the user from placing them freely to form a global pattern, thus disturbing the process of layout design.

We present a novel user interface for addressing a commonly known ambiguity problem in global layout beautification. As shown in Figure 1, our interface shows a preview of the beautification, without immediately modifying any input element. In our prototype, this preview is displayed in a separated window. While this interface is simple, it has the following benefits. First, the user can focus on the layout design by manipulating individual elements in the input window. Second, the user might refine the beautification results by slightly modifying the input elements. To facilitate a more direct con-

UIST 2014, October 5–8, 2014, Honolulu, HI, USA.

http://dx.doi.org/10.1145/2642918.2647398

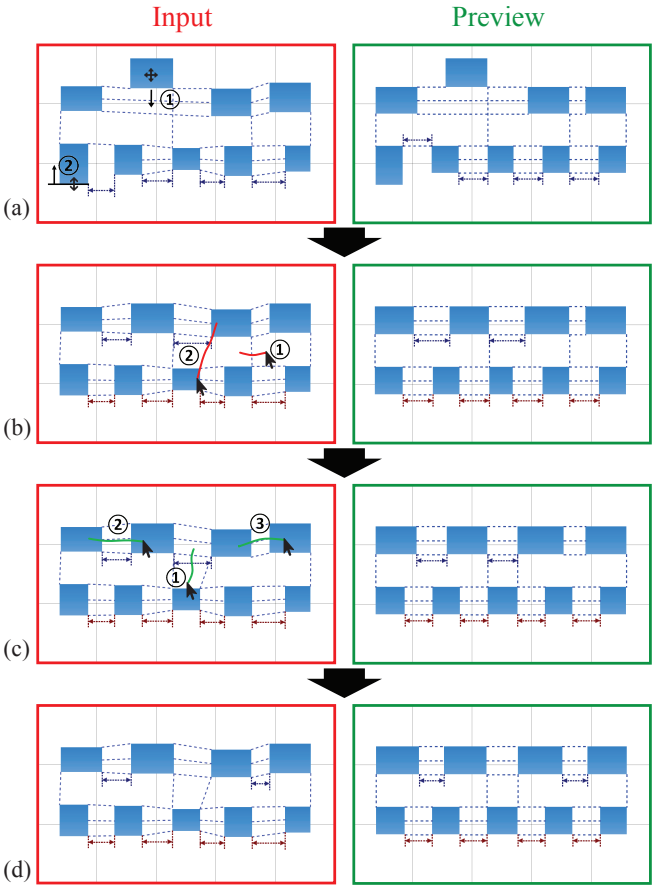

Figure 2. To refine a beautified layout, as previewed in the right column, the user may directly reposition (a-1) or resize (a-2) individual elements, or remove inferred (b-1) or add new (b-2) edge-alignment relationships, or remove inferred (c-1) or add new (c-2 and c-3) equal-spacing relationships.

trol of the beautification results, we also present a gestural interface, which allows easy editing of automatically inferred constraints, i.e., desired relationships between elements in the input window. See Figure 2 and the accompanying video for live demos.

We present an efficient and effective implementation of the proposed global layout beautification interface. Our tool allows interactive refinement of graphic elements and/or constraints, with instant beautification feedback. We conduct two studies to verify the effectiveness of our interface. First we run a study to evaluate the effectiveness of beautification preview. Second, we run a study to evaluate whether our tool could replace the standard snapping and alignment commands, omitting other layout helper features (e.g., equal-sizing, auto-align) which can peacefully coexist with our tool. Our results are promising and show that our tool provides a faster way for creating, editing, and refining the test layouts. The intuitiveness and ease-of-use of our interface are also confirmed by the user study participants.

RELATED WORK

It is hard to track down the history of shape alignment tools. Command-based alignment tools might be one of the most common ways to align objects [24]. They adopt a two-step procedure: first select a group of objects to be aligned and then issue a certain arrangement command (e.g., left-align or equally space the selected objects). Snapping is another widely used technique [4, 5]. It provides aligned positions by snapping an object to either the background grids, manually created guides, or other objects [11]. Snapping supports direct manipulation and is thus more intuitive to use, while command-based tools are more effective in aligning multiple objects simultaneously and have a better control of global alignment. Given their unique advantages, these traditional tools collectively are capable of creating very complex well-aligned layouts of graphic elements. However, even a simple arrangement task as shown in Figure 1 already demands a series of operations with the traditional alignment tools. The repeated use of operations like snapping, element selection, and command selection, is tedious and error-prone, especially for complex layouts. In addition, since different ordering of such operations might lead to different results, a user has to plan a series of operations beforehand and pay special attention to their order during editing. This somewhat diverts the user from the tasks of layout design and creation.

Beyond the basic snapping and alignment commands, there exist many layout helper features, some of which have already been integrated into commercial editors. For example, many algorithms have been proposed for automatic or semi-automatic graph layout generation [25, 26, 29], e.g., the *auto-align* and *auto-space* tools in Microsoft Visio. However, they often significantly change the layout of the input elements, without attempting to infer and maintain the underlying patterns in the input layout. The equal-sizing feature in Microsoft Visio allows easy size-adjustment of selected elements. Several editors such as OmniGraffle and PowerPoint support easy creation of equally spaced elements when duplicating an element multiple times. These helper features are very efficient for only specific tasks and are orthogonal to our tool.

Our work is inspired by the previous efforts on beautification, which also aims to bring as little change as possible when improving content aesthetics. There are plenty of beautification techniques mainly developed for sketch-based user interfaces to tolerate errors from freehand input sketches. These existing techniques can be largely categorized into two groups [23]: *local beautification* and *global beautification*. Similar to snap-dragging [5], local beautification of freehand sketches beautifies a current input stroke based on either itself (often through sketch recognition [1, 2, 14, 21]) or its geometric relationship(s) to the existing elements (e.g., [18, 33, 34]). A typical user interface for local beautification achieves beautification progressively, stroke by stroke. Each stroke is replaced with its beautified version and kept fixed during the beautification of subsequent strokes. Due to the local nature, such interfaces are often easy to control. However, unfortunately, local beautification is not suitable for the beautification of layouts, which should be treated globally.

Compared to local beautification, global beautification has been insufficiently explored. Pavlidis and Van Wyk [22] present the first automatic beautifier for drawings and illus-

trations. Their algorithm focuses on inferring and enforcing relations among points and lines, and thus can be extended for our layout beautification problem by considering special requirements like how to preserve aspect ratios of elements. However, they do not give details about the user interface of their system. Bolz [6] presents for the first time a user interface for global beautification of drawings. It concentrates on how to deal with parameter settings (e.g., via a parameter menu). Due to possibly frequent parameter changes, the beautifier by default is manually activated each time, though it is discussed that automatic activation of the beautifier (like ours) is a desired feature. Bolz's system overlays the beautified drawing with an outline of the unmodified version for some seconds. A similar visualization of the original and beautified versions is adopted in [12], which however deliberately defers the beautifier's feedback. Otherwise, the user may easily get disturbed by beautification results. In contrast, our user interface allows instant feedback with minimal disturbance. Cheema et al. [7] present a novel sketch beautification algorithm but adopt a simple interface similar to Bolz's. None of the above methods has explored an interface for editing constraints.

Our optimization strategy for determining valid constraints bears some resemblance to the previous solutions for identifying geometric features towards sketch recognition. For example, Veselova and Davis [30] employ studies of human perception to determine which geometric features are the most important for symbol recognition. Hammond and Davis [15] present a graphical debugging tool for learning structural descriptions from automatically generated near-miss examples. Similar to earlier work [18, 22], we adopt an automatic, iterative approach to avoid dramatic changes to the input layout.

It is not always possible to infer desired relationships among roughly placed elements, especially when there are ambiguities. A possible solution for ambiguity resolution is to provide multiple alternatives as suggestions from which the user can then choose [17, 18, 20]. The effectiveness of such suggestion-based interfaces has been demonstrated for progressive beautification [18, 20]. It is unclear whether suggestions can be used to effectively resolve the ambiguity from global beautification, since a handful of ambiguous relationships may easily lead to a large set of suggestions. Instead of attempting to select the best suggestions from a big pool, we design a gestural interface for directly editing a limited number of constraints.

Constraint-based systems have been extensively studied and a detailed review is beyond the scope of this paper. Most constraint-based systems require the user to specify constraints explicitly (e.g., [9, 13, 28]). Instead, our beautification interface focuses on automatic inference of geometric constraints, and thus only very few edits on constraints (after roughly placing elements) are needed to achieve a desired layout. Note that our current system performs once-off alignment only and does not maintain inferred constraints during subsequent editing [31]. Recently Zeidler et al. [32] introduce a novel layout preview but focus on the resizing behavior of a constraint-based layout during the design phase. Since constraint-based systems often need to deal with various linear and nonlinear constraints, possibly with complex interactions among them, significant effort has been devoted to the topic of constraint solving (e.g., [3, 16, 19, 27]). Such solvers can be potentially used for our beautification optimization problem.

USER INTERFACE

In this section we first describe our main interface for global beautification and then present our gestural interface for editing constraints. To make it simple, we first assume that the beautifier is already activated and applied to all input elements, and will discuss how our beautifier can coexist with other tools at the end of this section.

Global Beautifier Interface

Our global beautifier takes a set of roughly placed elements as input and returns a well-aligned layout as output. Since the desired relationships might not be immediately clear during editing, even to a human viewer, we design a user interface that does not directly change the location and size of elements in the input layout, but rather gives a preview of the refined layout in another window (Figure 1). In this way the user can focus on layout design itself by placing elements roughly, without being disturbed by the beautifier's feedback.

Since the input layout is not immediately replaced with the beautified version, the user may keep editing the current layout by translating or resizing individual elements, and our beautifier gives instant feedback in the preview window. This simple interface is suitable for creating, editing or refining layouts. At any point the user may confirm to apply the beautified layout to the input one.

Gestural Interface for Editing Constraints

Automatically inferring underlying relationships among elements plays a vital role in layout beautification. Since relationships are formulated as constraints in the beautification optimization, we will use the terms relationships and constraints interchangeably in the following discussion.

Automatically inferred relationships might not always be wholly desirable. This problem might be *indirectly* solved by interactive refinement of certain elements' position and/or size. Alternatively the user may *directly* add or remove relationships using a gestural interface as shown in Figure 2 (b) and (c). Our implemented beautifier currently supports two types of geometric relationships: edge alignment and equal-spacing.

At any time the user may edit constraints in the input window, where he or she may edit elements for interactive refinement. To facilitate the editing of constraints, we display all automatically inferred relationships using dashed lines and arrows in both the input and preview windows (Figure 1). The visualized constraints in the input window are editable via the gestural interface while those in the preview window are read-only for examining the currently achieved relationships. In real-world applications, constraints may be displayed on demand to avoid visual clutter, since in many cases only very few constraint edits often suffice.

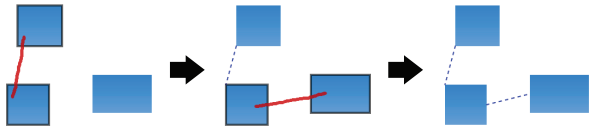

Figure 3. Two examples of our gestural interface for adding alignment constraints. An appropriate constraint is automatically determined by examining the relative location of two involved elements and the position of the stroke's ending points relative to the elements.

The user enters the mode for editing edge alignment constraints by clicking an "alignment constraint editing" button, which might be replaced with a gesture in the future implementation. In this mode, repositioning and resizing of individual elements are disabled. To remove an existing alignment constraint, the user performs a cutting gesture on a constraint of interest (Figure 2 (b-1)), i.e., drawing a stroke roughly perpendicular to the dashed line corresponding to the constraint. To add a new alignment constraint, the user simply draws a stroke connecting a pair of elements of interest (Figure 2 (b-2)). Our interface automatically determines whether a horizontal or vertical alignment constraint is needed by examining the relative location of the two elements. For example, the stroke in Figure 3 (a) is interpreted as an intention for vertical alignment while the stroke in Figure 3 (b) is for horizontal alignment. For horizontal alignment, the user may indicate the preference for top-, middle- or bottom-align by placing the ending points of the stroke into the respective parts of the elements. For this reason, the stroke in Figure 3 (b) is recognized for achieving middle alignment. A similar control exists for vertical alignment (see an example in Figure 3 (a)). To make the interface more user-friendly, we may put three control handles (e.g., small translucent circles) on each side of an element, indicating that the user can drag from one handle to another in order to set alignments. The mode for editing alignment constraints is deactivated when the user switches to other editing modes.

For simplicity, the activation and deactivation of our gestural tool for editing equal-spacing constraints are achieved via mode-switching buttons. To remove an equal-spacing constraint the user performs a cutting gesture (Figure 2 (c-1)). To add a new equal-spacing constraint, the user invokes multiple (at least two) strokes, with each of them connecting a pair of elements (Figure 2 (c-2 and c-3)). Constraints for equal-spacing between the specified pairs of elements will then be added. Horizontal or vertical equal-spacing is automatically determined by looking into the relative position of pairs of elements.

Not all automatically inferred constraints can be enforced at the same time. Our interface only visualizes the constraints that can coexist. However, the user may introduce constraints that conflict with the existing ones. To capture the user's latest intention, any constraint that is in conflict with the newly added constraint will be removed. Thus in some cases the user may see one or more existing constraints removed due to the new constraints.

IMPLEMENTATION

Now we describe how we implement the proposed beautifier interface. Our implementation essentially follows the general beautification framework proposed by Pavlidis and Van Wyk [22]. We describe our implementation details for completeness.

Pattern Detection

We focus only on axis-aligned edge alignment and equal-spacing relationships, since they already enable all the functions of snapping and alignment commands. For illustration, we always use rectangles to represent graphic elements, since element alignment is often achieved at the bounding box level.

Inferring edge-alignment relationships. We will explain only the implementation for horizontal alignment. Detecting relationships for vertical alignment is done similarly. Following the snapping tools in existing editors like Microsoft Visio, we allow horizontal edge-alignment to happen only at the top edge denoted as s_t, bottom edge s_b and middle edge s_m (i.e., a horizontal segment passing through the center) of the bounding box of an element. Given a group of n graphic elements (Figure 4 (a)), we thus have $3n$ edges (Figure 4 (b)), from which our beautifier will infer desired alignment relationships (Figure 4 (d)), i.e., detect a set of horizontal lines along which the relevant elements get aligned (Figure 4 (c)). This is essentially a 1D clustering problem. Although there exist many general techniques for such clustering problems, most of them require indicating the scale of the cluster by parameters, which in most techniques are global. However, the alignment lines in a given layout may have different qualities. Some of them would be invalid if all alignment lines are detected under the same configuration. Therefore, the clustering algorithm should be adaptive to each alignment line.

Our solution is a variant of the classic *RANSAC* algorithm [10]. Since the number of edges is usually not big (from dozens to hundreds), it is possible to find globally optimal lines without randomness. Specifically, for every edge, we use the following iterative procedure to find a candidate alignment line: 1) the current edge as an initial inlier gives an initial fitted line. 2) All other edges are tested against the fitted line to find an updated set of inliers. 3) Refit the line to the updated set of inliers. Steps 2 and 3 are repeated till convergence. Each edge as the initial inlier gives a candidate alignment line, and we pick the best line as the determinate alignment line. The inliers (edges) of the picked line are then removed from the current set of edges (initially, $3n$ edges) and we repeat the above iterative procedure to find the next alignment line till no more lines can be found. See the appendix for more details of Steps 2 and 3.

Once we find a candidate alignment line for each edge, we select the best alignment line $\mathcal{P}_a$. Generally we prefer an alignment line with a larger group of inliers and a lower variance of the vertical coordinate among the inliers. All the inlier edges corresponding to the selected alignment line have edge-alignment relationships between each other (Figure 4 (d)).

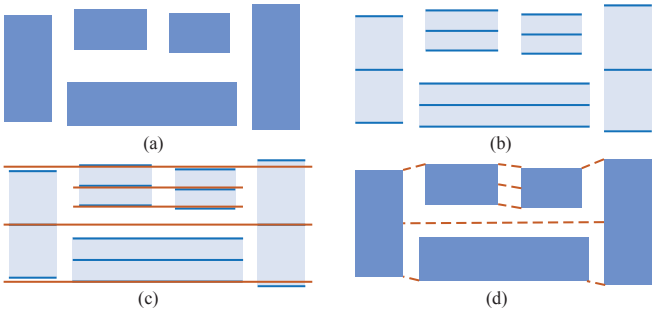

Figure 4. Illustration for inferring relationships (dashed lines) for horizontal alignment.

Inferring equal-spacing relationships. In our implementation we allow the existence of equal-spacing relationships only in a group of graphic elements that have edge-alignment relationships. We use an iterative bottom-up clustering approach to detect equal-spacing relationships. The spacing is defined as the distance between every pair of *adjacent* elements. Initially, each spacing cluster consists of only one spacing. Each iteration groups two clusters with the smallest distance. The distance between two spacings d_1 and d_2 is defined as $|d_1 - d_2|/(d_1 + d_2)$. For two spacing clusters, it is defined as the difference of the average spacing between two clusters. The clustering process continues until the distance of the best pair of spacing groups is larger than a predefined threshold (we set as 0.15). Note that our gestural interface allows interactively adding equal-spacing constraints to non-adjacent pairs of elements (see an example in Figure 2 (c)).

Layout Refinement by Optimization

Following the previous beautification works [22], we use an optimization-based approach to refine the layout to satisfy the inferred relationships while retaining as much as possible the original layout, in terms of both the position and size of each graphic element.

Objective. Let s_t, s_b, s_m, s_l, s_r, s_c denote the top, bottom, (horizontal) middle, left, right, (vertical) center edges of a graphic element. The objective of bringing the least change to the original layout can be achieved by preserving the coordinates $p(\cdot)$ of the edges s_* of each graphic element $g \in \mathcal{G}$, where $\mathcal{G}$ is the set of input graphic elements. Specifically, for vertical edges, s_l, s_c and s_r, $p(\cdot)$ are their horizontal coordinates, and for horizontal edges, s_t, s_m and s_b, $p(\cdot)$ are their vertical coordinates. To retain the original layout we minimize the following objective function:

$$E = \sum_{g \in \mathcal{G}} \sum_{*} (p'(s_*^g) - p(s_*^g))^2, \tag{1}$$

where $p'(\cdot)$ denotes the changed coordinates of the edges after refinement.

Constraints. The above minimization is subject to a set of constraints. We first identify the intrinsic constraints of each graphic element g, which are $p'(s_l^g) + p'(s_r^g) = 2p'(s_c^g)$ and $p'(s_t^g) + p'(s_b^g) = 2p'(s_m^g)$. If the aspect ratio a^g needs to be retained during the refinement (e.g., for images), an extra constraint is added: $p'(s_r^g) - p'(s_l^g) = a^g(p'(s_t^g) - p'(s_b^g))$.

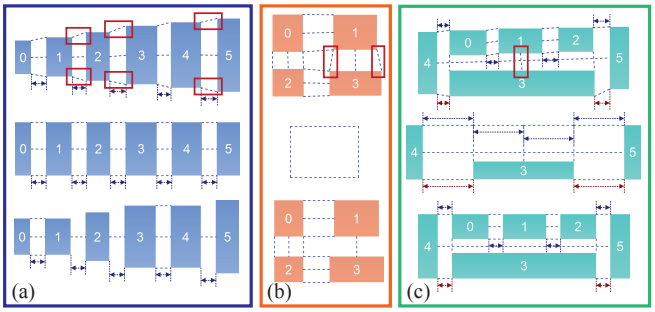

Figure 5. Top: input layouts. Middle: results by simultaneously enforcing all detected constraints, leading to degenerating cases in (b) and (c). Bottom: our results, by automatically rejecting the highlighted constraints. Note that the constraint of keeping the aspect ratios of elements is enforced in (b) and (c) but not (a).

These three constraints hold for every graphic element and must be satisfied by the optimal solution.

The rest of the constraints are derived from the detected relationships. Let $\mathcal{S}_a$ denote a set of inliers that will be aligned to the same line $\mathcal{P}_a$. If the edges s_i and s_j have edge-alignment relationships between them, we have $p'(s_i) = p'(s_j)$. Each pair of neighboring graphic elements in $\mathcal{P}_a$ gives rise to one such equality constraint, resulting in a total of $|\mathcal{S}_a| - 1$ constraints, where $|\mathcal{S}_a|$ is the cardinality of $\mathcal{S}_a$. Let $\mathcal{S}_s$ be a set of element pairs with the equal-spacing relationship detected for every element pair. Similarly, this introduces another $|\mathcal{S}_s| - 1$ equality constraints.

Optimization. In general, only a subset of the constraints can be enforced during the optimization. This is because simultaneously enforcing some constraints may cause the refined layout to deviate too much from the original layout (Figure 5 (a)). Worse, the layout may become degenerate in that the sizes of some graphic elements may become zero or negative (Figure 5 (b) and (c)).

Similar to [18, 22], we use an iterative approach to determine valid constraints. Specifically, we iteratively check the validity of each of our alignment and spacing equality constraints by examining the optimized layout. After adding each constraint to the system, we check the current optimal solution and drop the constraint if the layout becomes degenerate or changes too much. Since all the constraints are in the linear equality form, and the constraints are added incrementally, this strategy works efficiently. The alignment constraints are added in the order in which the corresponding alignment lines are detected, i.e., the constraints are preserved in a higher priority if the corresponding alignment line is detected earlier. Within the $|\mathcal{P}_a| - 1$ edge-alignment constraints corresponding to a specific alignment line, we rank them by their alignment error. This means that a constraint $p'(s_i) = p'(s_j)$ has a higher priority if $||p(s_i) - p(s_j)||$ is smaller. After considering all the constraints of an edge-alignment pattern, we add the detected equal-spacing relationships from this alignment pattern. All constraints of an equal-spacing pattern are simultaneously added, and they are all rejected if the new layout becomes degenerate or deviates too much from the original layout, since equal-spacing often exists as a global pattern. We use the CHOLMOD sparse linear system solver [8] to in-

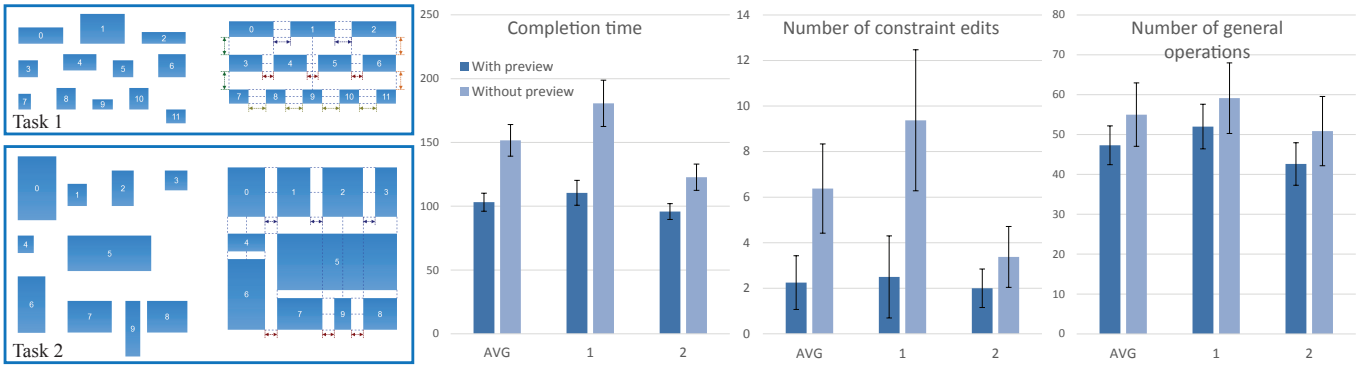

Figure 6. Two layout refinement tasks used in study 1 and the resulting statistics. Error bars represent standard error of the mean.

crementally solve the resulting constraint optimization problem after each constraint is added. It provides fast update for the linear system, without repeatedly solving it from scratch, and thus is able to solve the incremental constraint optimization problem efficiently.

While our simple strategy generally works well and efficiently, we are aware of another possible solution which determines constraints by adding linear inequality constraints to restrict the solution domain. For example, the refined size of a graphic element should be within a certain range. We can then incrementally add a constraint and retain it only if the feasible region of this constraint problem is not empty. For this solution, we may resort to existing constraint solvers like QOCA [19] and Gecode [27].

USER STUDIES

We have extensively tested our technique on layouts of various patterns, most of which are time consuming to produce when snapping and alignment commands are used. See some of the tested layouts in Figures 5–8. Our tool is also applicable to layouts with nested elements (See Figure 7). Note that a fixed set of parameter values were always used throughout our experimentation. Two user studies were conducted to evaluate the effectiveness of our our technique.

Study 1: Evaluation of Beautification Preview

We first evaluated the performance of the beautifier with and without a preview window. Eight university students helped with the evaluation. All of them had extensive experience in using the traditional alignment tools, e.g., in Microsoft PowerPoint.

Apparatus. The study was conducted on an ordinary PC (DELL Optiplex 960), with 3.00 GHz Intel Core 2 Duo CPU and 4.00 GB RAM. Two LCD display monitors (20-inch and 17-inch) were connected, one for the beautifier interface (with the preview window if used), and the other only for displaying a static target layout. Our beautifier achieved real-time performance for moderately complex layouts. For example, for a typical layout containing a dozen of elements like those shown in Figure 8, it took our technique less than 0.01 seconds for alignment relationships inference and less than 0.02 seconds for layout refinement.

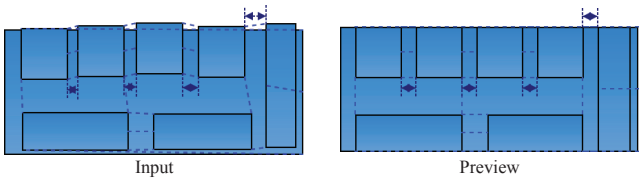

Figure 7. The application of our tool to layouts with nested elements.

Tasks. We asked participants to perform layout refinement. For each task, each participant was asked to refine a source layout, which already resembled a target layout with visualized alignment and equal-spacing relationships (Figure 6), displayed on a separate monitor. The source and target layouts were consistently numbered so that the participants knew the correspondence between the elements. Common operations, such as element translation, resizing and selection (by single click or rectangle selection tool), were allowed. But we disabled element creation and removal.

Two tasks (Figure 6) were tested and had different layout complexity. Task 1 was more challenging, since it involved more potential ambiguities during refinement. Each participant was required to complete each of the two tasks, with two tools: our beautifier with and without a preview window. For the latter, the beautified result was instantly applied to the input layout, as similarly done in the previous works [6, 12]. See the accompanying video for such an interface in action. In total we had 8 (participants) $\times$ 2 (tools) $\times$ 2 (tasks) = 32 trials.

With each of the two tools (i.e., the beautifier with and without a preview), the participants were asked to quickly reproduce each of the two target layouts by achieving as many visualized target relationships as possible. However, they were allowed to proceed to the next task without fully reproducing the target layout of the current task. To help the participants better track their progress, the alignment and equal-spacing relationships of the layout were visualized and shown in the preview window or the input window in the case of no preview window. The order of the tasks and the tools in each task were counter-balanced across participants. Before the study, the participants were introduced to these two tools, and they practiced in a short warm-up session until they felt comfortable. The whole study lasted less than 20 minutes on average for each participant.

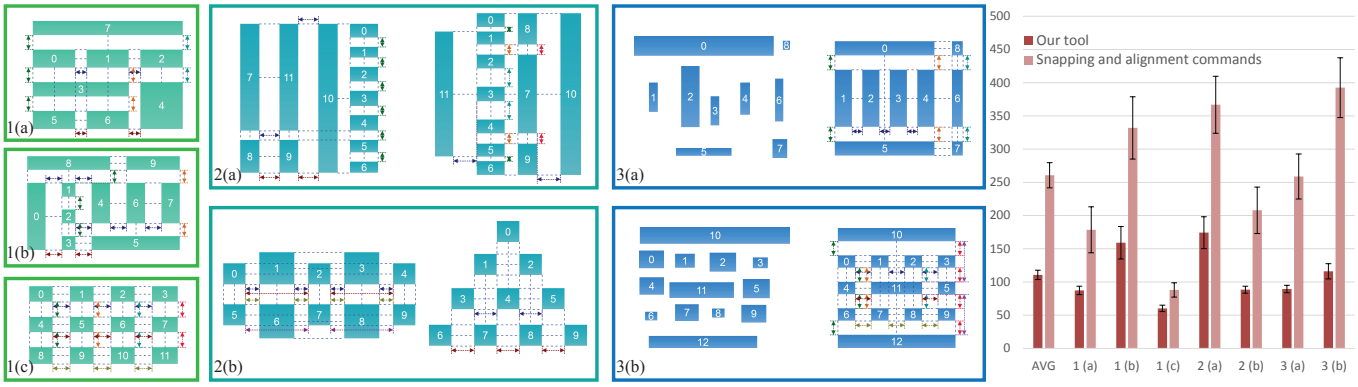

Figure 8. From left to right: 1(a)-(c): target layouts used in the layout creation scenario; 2(a)&(b): source layouts (left) and target layouts (right) in the layout editing scenario; 3(a)&(b): source (left) and target layouts (right) in the layout refinement scenario; average completion time (in seconds) of each task. Error bars represent standard error of the mean.

Performance measures. During the study, the following information was recorded for quantitative analysis: the completion time of individual trials, the time spent on element editing (selection, moving, resizing), the number of general operations (moving, resizing, duplicate, removal, undo), and the number of edits for edge-alignment constraints and equal-spacing constraints.

Results. Figure 6 plots the statistics of the core information captured in Study 1. The preview window significantly shortened the average completion time per task among the participants, from 151.7 seconds to 103.2 seconds. A significant difference was confirmed by repeated measures analysis of variance (repeated measures ANOVA): $F = 57.77$ and $p = 0.000126$. The benefit of our interface with the preview window was even clearer for Task 1, a more challenging task. The average completion time of Task 1 was 110.5 seconds with our interface, compared to 180.6 seconds without the preview window ($F = 41.59$, $p = 0.000351$).

Figure 6 also shows the average number of operations for editing constraints and the average number of general operations (moving, resizing, etc). While on average the number of general operations was less with our interface, the difference was not statistically significant. Repeated measures ANOVA confirmed a significant difference in the average of constraint edits for Task 1 ($F = 17.78$, $p = 0.003954$) but not for Task 2 ($F = 2.951$, $p = 0.12951$) between the beautifier with and without the preview window. For the more challenging Task 1, without the preview window, the users easily got distracted by intermediate beautification results, making them difficult to achieve desired layouts simply by direct repositioning and resizing of elements. Thus they had to resort to the tools for editing constraints more often. It is interesting to note that the standard errors here were relatively large, indicating that different participants might have difference preferences for the interface for editing constraints.

Study 2: Comparing to Snapping & Alignment Commands

We conducted a user study to evaluate the effectiveness of our technique, compared to snapping and alignment commands, which are arguably the most popular tools available in almost all commercial graphic editors like OmniGraffle, Visio, and InDesign. We implemented these standard tools in a way like in PowerPoint 2013. Specifically, snapping was activated only for the element being dragged. This element could be snapped to achieve edge/center alignment or equal spacing with other elements. The compared system supported 8 commands: align top/middle/bottom/left/center/right, and distribute horizontally/vertically. We chose the lead object for alignment similar to PowerPoint, e.g., the topmost element for top-align, and the outermost elements for even distribution. We deliberately excluded layout helper features beyond the standard snapping and alignment commands, since it is expected that such extra functions would similarly benefit our tool and the compared system.

Apparatus and participants. We used the same set of instruments as those in Study 1. Another 11 university students were recruited for the user study. Again all of them had used traditional alignment tools extensively. A handful of them were even good at vector graphics editing and were familiar with professional software like Adobe Illustrator.

Tasks. Besides layout refinement, we evaluated the performance of the traditional tools and ours in another two scenarios, i.e., creating and editing layouts. In the creation scenario, three tasks were tested. For each task, each participant was asked to create a layout from scratch towards a reference target layout (Figure 8 1(a)-(c)). The participants were allowed to draw, duplicate, or remove elements. The layout editing scenario comprises two tasks, each of which required the participants to significantly change an input layout towards a target layout (Figure 8 2(a)-(b)). Element creation and removal were disabled. Like Study 1, the participants were asked to complete two layout refinement tasks (Figure 8 3(a)-(b)). But this time we locked the aspect ratios of the elements, controlled by a toggle button. For the editing and refining scenarios, consistent correspondence numbers were provided.

In total there were seven tasks under the three scenarios. Each participant was asked to complete each of them twice, one with our tool and the other with the set of traditional alignment tools. That is, our experiment involved 11 (participants) $\times$ 2 (tools) $\times$ 7 (tasks) $=$ 154 trials. The tools in each task were (almost) counter-balanced and the order of the tasks in each scenario was random. The participants took breaks between different scenarios so that they were briefed on the

newly enabled or disabled operations before each scenario. For simplicity no break was allowed between tasks in each scenario.

Performance measures. Besides the measures used in Study 1, the following information was also recorded: the time spent on drawing the elements (for the creation scenario only), and the number of specific operations with the traditional tools (commands like edge alignment and equal spacing).

Results. Figure 8 shows the target layouts in each task, and the corresponding average completion times. Repeated measures ANOVA found a significant difference in the average completion time per task among the participants between our tool and the traditional tools ($F = 78.8$, $p < 4 \times 10^{-6}$). On average, significantly less time was needed to accomplish each task using our tool (110.7 seconds) compared to the traditional tools (260.7 seconds).

Our tool performed much better than the traditional tools for target layouts involving complex relationships such as Tasks 1(b) and 2(a). For easier tasks, such as Tasks 1(a) and 1(c), our tool was still faster. For the tasks in the refinement scenario, due to the locked aspect ratios of the elements, the editing freedom was seriously restricted, making such tasks more challenging to complete for the traditional tools. The performance of the traditional tools was severely affected by this restriction on editing freedom. More than half of the participants failed to reproduce the target layouts with the traditional tools, while all of them had no difficulty with our tool.

In Figure 9 we show more statistics from the user study. There was no statistically significant difference in the drawing time between the two tools ($p = 0.465$), though we observed that our tool was still slightly faster. Our tool required significantly fewer general operations ($p < 5 \times 10^{-5}$). The task success rates also reflects the effectiveness of our tool. For almost all the tasks, each participant could successfully reproduce the target layouts with our tool. In contrast, the completion rates for the traditional tools were much lower, despite they had been used by all the participants on a regular basis. This confirms the ease of use of our tool, even for the first-time users.

Figure 9 also shows the average number of edits on edge alignment and equal-spacing constraints for individual tasks. While the number of edits was small, constraint editing was performed for 5 out of 7 tasks. We speculate that the number of constraint edits needed is correlated with layout complexity, which is somewhat reflected by the (average) completion time. For example, the number of constraint edits used for Task 1(b) was much bigger than that for Task 1(a) and Task 1(c). A similar conclusion on the average completion time of these tasks could be reached.

It can be seen that the standard error of the mean (shown as error bars in Figures 8 and 9) of the completion time, editing time, and editing number is much larger with the traditional tools. This means that the performance of our tool was more consistent across all participants. In other words our

Figure 9. More statistics. The timings are in seconds. Error bars represent standard error of the mean.

tool was less dependent on the experience of the individual participants.

Our statistics show that on average each participant performed 0.97 edge-alignment commands and 1.90 equal-spacing commands per task. This is mainly because equal-spacing commands were more indispensable, while edge-alignment commands could be replaced with more intuitive snapping operations. This reiterates the fact that snapping has to be used together with command-based alignment tools in many cases.

We also got some interesting observations from the user study. With our tool, some participants tended to draw or edit elements carefully at the beginning despite being told that our tool can tolerate rough inputs. However, they quickly get used to our tool and performed the creation and editing operations more casually. Another observation was that, in the refining scenario, when using the traditional tools, some participants moved all the elements away before editing, though most of the input elements were already located very close to the target positions. Such behavior was never observed with our tool. This behavior was largely due to the not-aligned elements triggering often excessive and distracting suggestions from snap-dragging.

After completing the tasks, we asked each participant to provide feedback on the two tools in terms of their ease of use. All participants except one preferred our tool. They expressed that our tool enables them to disregard the order of specific operations and focus on the layout design. One participant who preferred the traditional tools liked the full user control and disliked rough inputs. This concern could be addressed by integrating snap-dragging into our framework.

DISCUSSIONS

To use our tool in existing graphic editors like PowerPoint, the user may first select a set of target elements and then enter a "layout beautification mode", for example, via an activation button, menu or gesture. Similar to the interface described

in our prototype, we may pop up a dialog to edit elements and/or constraints while previewing the beautified layout in the input panel. In the mode of element editing, our interface is completely compatible with existing layout helper features. All edits are confirmed after the user hits the "apply" button; otherwise they are canceled.

While the results of Study 1 imply the usefulness of having a preview of the refined layout in the experimental condition, rendering the original and refined layouts in separate windows might cause the problem of divided attention. To alleviate this problem we might use a smaller preview window instead of a full-scale copy of the design. We believe that it is not necessary to always activate the preview window especially when the input layout is undergoing dramatic changes, e.g., at the beginning of layout editing and creation tasks in Study 2.

To further solve the problem of divided attention and to save space required by an extra preview window, we could show the preview in place. For example we might overlay the beautified layout as a translucent layer on top of the original layout. Since this solution easily results in visual clutter, it might work well for only relatively simple layouts with fewer elements, but would likely not be able to scale to very complex diagrams. In the future it is worth evaluating whether such alternative solutions would be more effective than a simple preview window in our current prototype.

Since our main goal in this work is to find a better alternative to snapping and alignment commands, we focused only on a limited set of simple constraints, i.e., alignment and equal spacing. However, it is easy to incorporate new constraints, such as symmetry and length equality into our framework, as already suggested by Pavlidis and Van Wyk [22]. Similarly, although our current prototype concentrated on grid-like layouts, it is possible to handle more general layouts by introducing advanced pattern detectors, e.g., to detect elements distributed roughly along an arbitrary line, circle etc.

CONCLUSION AND FUTURE WORK

We presented a novel user interface for layout beautification. Our user studies confirmed that our technique is very effective and can even be used as a better replacement of the standard snapping and command-based alignment tools, given its faster performance and better ease of use. We speculate that the advantages of our tool would be even clearer to novice users who have little or no experience in creating precise layouts. It would be interesting to apply our tool to real-world examples like posters and to really integrate it with existing graphic editors to examine the interplay between our tool and other layout helper features. Our current implementation requires explicit mode switching to activate or deactivate the gesture interface for editing geometric constraints. In the future we will explore a modeless interface where the user could use the editing operations of both element and constraint without explicit mode switching. We are also interested in extending the proposed interface to 3D layout design, where the traditional tools are even more cumbersome.

Acknowledgements

We thank the reviewers for their constructive comments and the user study participants for their time. This work was partially supported by grants from the Research Grants Council of HKSAR, China (Project No. 113513, 619611, and 11204014) and the City University of Hong Kong (Project No. 7003058).

REFERENCES

1. Alvarado, C. Sketch recognition user interfaces: Guidelines for design and development. In *Proceedings of AAAI Fall Symposium on Intelligent Pen-based Interfaces*, vol. 1 (2004).
2. Arvo, J., and Novins, K. Fluid sketches: continuous recognition and morphing of simple hand-drawn shapes. In *UIST '00* (2000), 73–80.
3. Badros, G. J., Borning, A., and Stuckey, P. J. The cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI) 8*, 4 (2001), 267–306.
4. Baudisch, P., Cutrell, E., Hinckley, K., and Eversole, A. Snap-and-go: helping users align objects without the modality of traditional snapping. In *CHI* (2005), 301–310.
5. Bier, E. A., and Stone, M. C. Snap-dragging. In *ACM SIGGRAPH Computer Graphics*, vol. 20 (1986), 233–240.
6. Bolz, D. Some aspects of the user interface of a knowledge based beautifier for drawings. In *IUI '93* (1993), 45–52.
7. Cheema, S., Gulwani, S., and LaViola, J. Quickdraw: Improving drawing experience for geometric diagrams. In *CHI '12* (2012), 1037–1064.
8. Davis, T. Cholmod: a sparse supernodal cholesky factorization package., version 2.1.2, 2013. University of Florida, Available online at http://www.cise.ufl.edu/research/sparse/cholmod/.
9. Dwyer, T., Marriott, K., and Wybrow, M. Dunnart: A constraint-based network diagram authoring tool. In *Graph Drawing* (2009), 420–431.
10. Fischler, M. A., and Bolles, R. C. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM 24*, 6 (1981), 381–395.
11. Frisch, M., Kleinau, S., Langner, R., and Dachselt, R. Grids & guides: multi-touch layout and alignment tools. In *CHI '11* (2011), 1615–1618.
12. Galindo, D., and Faure, C. Perceptually-based representation of network diagrams. In *International Conference on Document Analysis and Recognition*, vol. 1 (1997), 352–356.
13. Gleicher, M., and Witkin, A. Drawing with constraints. *The Visual Computer 11*, 1 (1994), 39–51.

14. Hammond, T., and Davis, R. Automatically transforming symbolic shape descriptions for use in sketch recognition. In *AAAI '04*, AAAI'04, AAAI Press (2004), 450–456.

15. Hammond, T., and Davis, R. Interactive learning of structural shape descriptions from automatically generated near-miss examples. In *IUI '06* (2006), 210–217.

16. Hosobe, H. A modular geometric constraint solver for user interface applications. In *UIST '01* (2001), 91–100.

17. Igarashi, T., and Hughes, J. F. A suggestive interface for 3D drawing. In *UIST '01* (2001), 173–181.

18. Igarashi, T., Matsuoka, S., Kawachiya, S., and Tanaka, H. Interactive beautification: a technique for rapid geometric design. In *UIST '97* (1997), 105–114.

19. Marriott, K., and Chok, S. S. Qoca: A constraint solving toolkit for interactive graphical applications. *Constraints 7*, 3-4 (2002), 229–254.

20. Murugappan, S., Sellamani, S., and Ramani, K. Towards beautification of freehand sketches using suggestions. In *6th Eurographics Symposium on Sketch-Based Interfaces and Modeling* (2009), 69–76.

21. Paulson, B., and Hammond, T. PaleoSketch: accurate primitive sketch recognition and beautification. In *IUI '08* (2008), 1–10.

22. Pavlidis, T., and Van Wyk, C. J. An automatic beautifier for drawings and illustrations. In *ACM SIGGRAPH Computer Graphics*, vol. 19 (1985), 225–234.

23. Plimmer, B., and Grundy, J. Beautifying sketching-based design tool content: issues and experiences. In *Proceedings of the Sixth Australasian conference on User interface-Volume 40* (2005), 31–38.

24. Raisamo, R., and Räihä, K.-J. A new direct manipulation technique for aligning objects in drawing programs. In *UIST* (1996), 157–164.

25. Reinert, B., Ritschel, T., and Seidel, H.-P. Interactive by-example design of artistic packing layouts. *ACM Trans. Graph. 31*, 6 (2013).

26. Ryall, K., Marks, J., and Shieber, S. An interactive constraint-based system for drawing graphs. In *UIST '97* (1997), 97–104.

27. Schulte, C., Lagerkvist, M., and Tack, G. Gecode. *Software download and online material at the website: http://www. gecode. org* (2006).

28. Sutherland, I. E. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE design automation workshop* (1964), 6–329.

29. Tollis, I., Eades, P., Di Battista, G., and Tollis, L. *Graph drawing: algorithms for the visualization of graphs*, vol. 1. Prentice Hall New York, 1998.

30. Veselova, O., and Davis, R. Perceptually based learning of shape descriptions for sketch recognition. In *AAAI '04* (2004).

31. Wybrow, M., Marriott, K., Mciver, L., and Stuckey, P. J. Comparing usability of one-way and multi-way constraints for diagram editing. *ACM Transactions on Computer-Human Interaction (TOCHI) 14*, 4 (2008), 19.

32. Zeidler, C., Lutteroth, C., Sturzlinger, W., and Weber, G. The auckland layout editor: an improved gui layout specification process. In *UIST '13* (2013), 343–352.

33. Zeleznik, R. C., Bragdon, A., Liu, C.-C., and Forsberg, A. Lineogrammer: creating diagrams by drawing. In *UIST '08* (2008), 161–170.

34. Zitnick, C. L. Handwriting beautification using token means. *ACM Trans. Graph. 32*, 4 (2013), 53:1–53:8.

APPENDIX

We first describe how we estimate a (horizontal) line $\mathcal{P}_i$ from a set of inliers $\mathcal{S}_i$ at iteration i (Step 3). We estimate the vertical coordinate of $\mathcal{P}_i$ as the weighted mean of the vertical coordinates of all inliers in $\mathcal{S}_i$. Specifically, the weight of each inlier $s \in \mathcal{S}_i$ is defined as $w(s) = e^{-(y(\mathcal{P}_{i-1})-y(s))^2/l(s)^2}$, where $l(s)$ is the length of the inlier edge s, $y(s)$ and $y(\mathcal{P}_{i-1})$ are the vertical coordinates of s and the line estimated in the previous iteration $\mathcal{P}_{i-1}$, respectively. This Gaussian weighting scheme indicates that the contribution of an inlier to the new line is higher if the inlier is longer or closer to the previous line. The vertical coordinate of the new estimated line $\mathcal{P}_i$ is then computed as $\sum_{s\in\mathcal{S}_i} \hat{w}(s)y(s)$, where $\hat{w}(s)$ is the normalized weight.

Now we give the details of Step 2: how to get an updated set of inliers $\mathcal{S}_{i+1}$ given the line $\mathcal{P}_i$. We introduce an adaptive inlier range for robust inliers detection. Specifically, an edge t is added into $\mathcal{S}_{i+1}$ if and only if $y(t) \in [y(\mathcal{P}_i)-r, y(\mathcal{P}_i)+r]$, where $r = \min(\overline{r}(1+\sigma)h(t), 2\overline{r})$ is adaptively determined. $\overline{r}$ is fixed as the average size of the input elements, multiplied by a fixed factor (0.125 in our implementation). $\sigma = \sqrt{\sum_{s\in\mathcal{S}_i} \hat{w}(s)(y(\mathcal{P}_i)-y(s))^2}$ measures the variance of the vertical coordinate among the inliers in $\mathcal{S}_i$. This changes the range adaptively with respect to the degree of alignment of the previous set of inliers. The term $h(t)$ is motivated by the Gestalt law of proximity: objects that are near to one another are perceived as belonging together as a unit. We thus define $h(t) = e^{-\hat{d}_t^2}$, where $\hat{d}_t$ measures the *horizontal* distance between t and the inliers in $\mathcal{S}_i$, normalized by the average horizontal and vertical distance between neighboring elements in the input layout. When t is away from the previous set of inliers, it is less likely to be detected as an inlier.